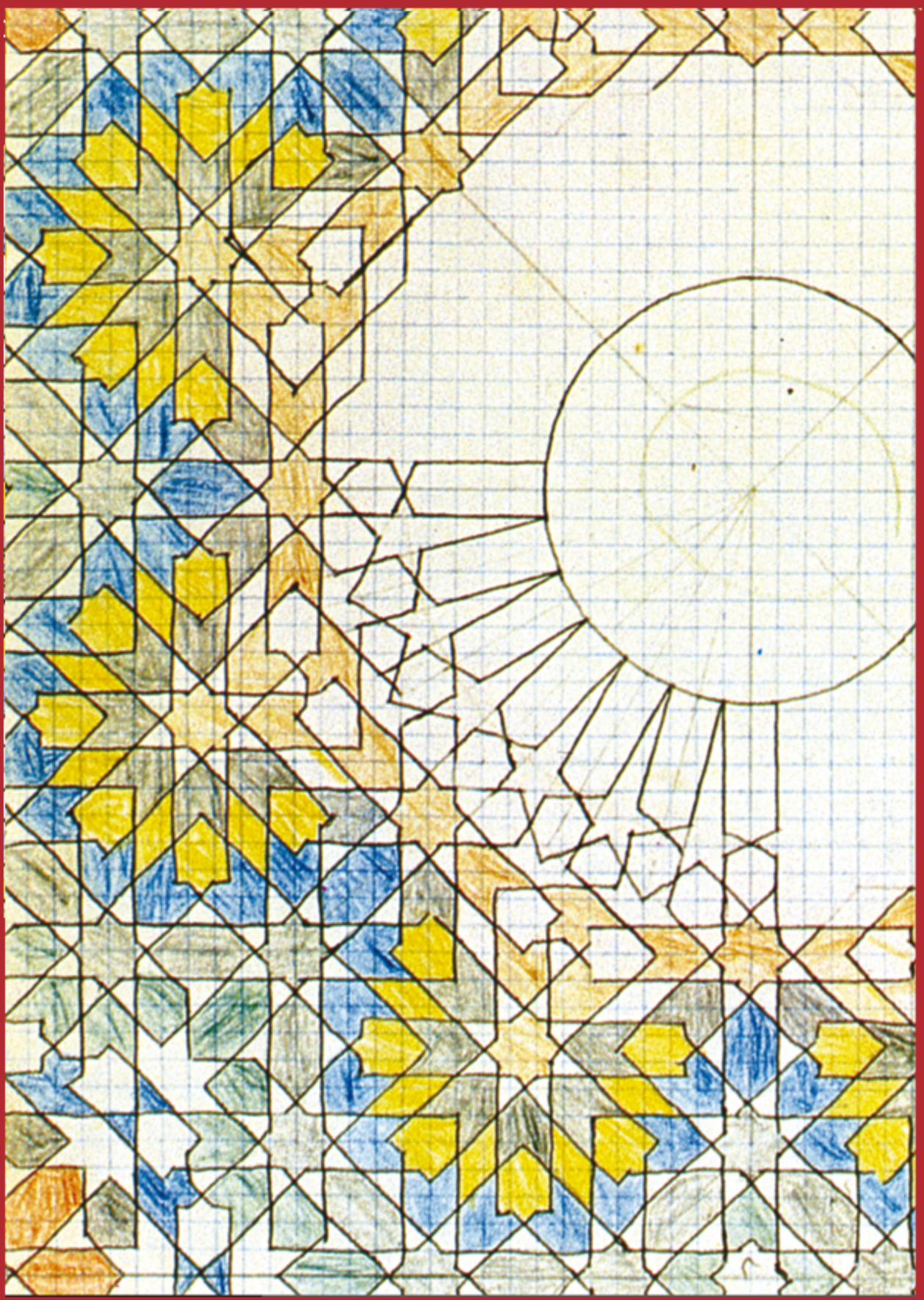
Jarka Arnold Tobias Kohn Aegidius Plüss

Programmierkonzepte

mit Python und der Lernumgebung TigerJython



Eine Online-Lernplattform

Inhalt

Version 2.4, September 2015

Autoren: Jarka Arnold, Tobias Kohn, Aegidius Plüss
Kontakt: help@tigerjython.com

Mit Unterstützung durch den SVIA/SSIE/SVIA
Schweizerischer Verein für Informatik in der Ausbildung

GELEITWORT

Zu Beginn der fünfziger Jahre des letzten Jahrhunderts hatte ich das Privileg, an der Eidgenössischen Technischen Hochschule (ETHZ) in Zürich mit Hilfe des ersten in der Schweiz verfügbaren programmierbaren Computers Zuse 4 eine Doktorarbeit zu verfassen. Die damaligen in unserem Lande ersten Schritte in den Computerwissenschaften, die später unter dem Namen „Informatik" zusammengefasst wurden, fanden aber bei den kantonalen Universitäten nur sehr zögerlich Aufnahme, insbesondere die Anerkennung der Informatik als eigene wissenschaftliche Disziplin. Selbst an der ETHZ erhielt erst 1974 die Gruppe der Informatikprofessoren ein eigenes Institut für Informatik und erst 1981 konnte ein Departement für Informatik organisiert werden.

Die rapide Entwicklung der Leistungsfähigkeit und Miniaturisierung der Computer trug wesentlich zu einem gewaltigen Anstieg der Datenproduktion bei und führte zu einem massiven Ausbau der Datenkommunikation, die wiederum nur mit einem weitgehenden Computereinsatz bewältigt werden kann. Dementsprechend müssen die Kommunikationstechniken ausgebaut und der Zugang zu diesen vermittelt werden.

Auf unserem Planeten, der durch ein zunehmendes Bevölkerungswachstum und steigende Ansprüche seiner menschlichen Bewohner bedroht wird, kann die Schweiz ihre Stellung als einer der wohlhabendsten und fortschrittlichsten Staaten mit vorzüglichem Lebensstandard und ihre direkte Demokratie nur bewahren, wenn sie über ein modernes, leistungsfähiges Bildungswesen und eine hochstehende Forschung verfügt. Dies erfordert aber nicht bloss einen optimalen Ausbau unserer Hochschulen zur Berücksichtigung der neuesten Entwicklungen im ICT-Bereich, sondern auch eine Neugestaltung der allgemeinen Grundausbildung, die in der Primar- und Sekundärstufe angeboten werden, sowie des Informatikunterrichts an den Maturitätsschulen. Die Vermittlung der drei Grundkompetenzen Lesen, Schreiben und Rechnen reicht heute bei weitem nicht mehr aus, um allen in den jetzigen persönlichen und beruflichen Lebensumständen, in denen Computer zentrale Aufgaben erfüllen, ein befriedigendes Auskommen zu sichern.

Mir war es als Direktor des Bundesamtes für Bildung und Wissenschaft ein Anliegen, den Informatikunterricht als eigenständiges Fach an den Maturitätsschulen zu etablieren. Dabei stellte sich konkret auch die Frage, ob die Integration eines ICT-Faches in die Unterrichtsprogramme genügt, oder ob ein umfassenderes Informatikwissen, das eine autonomere Nutzung der modernen Computertechnik gestattet, vermittelt werden sollte. Die Autoren der Lernplattform *TigerJython*, die darlegen, wie mit der Programmiersprache *Python* und einer didaktisch konzipierten Programmierumgebung die wichtigsten Informatikkonzepte auf eine einfache Art unterrichtet werden können, liefern zur Beantwortung dieser Frage eine ausgezeichnete Unterlage für ihre Empfehlung, ein Fach Informatik ab dem 6. Schuljahr einzuführen.

Die nachfolgende kürzlich erschienene schweizerische Pressemitteilung "*Die Schweizer E-Government-Angebote sind im internationalen Vergleich nur Mittelmass... Die Schweiz ist unter den europäischen Staaten gar auf den vorletzten Platz zurückgefallen*" zeigt meiner Meinung nach die Notwendigkeit eines baldigen Eingehens auf diesen Vorschlag, da dieser bedenkliche Rückfall nicht zuletzt auf eine ungenügende Verfügbarkeit von Informatikwissen in der für das Angebot verantwortlichen Institutionen zurück zu führen ist. In unserem reichen Land mit seiner grossen Computerdichte dürfte es nicht an den erforderlichen materiellen Voraussetzungen fehlen!

Prof. Dr. sc. math., Dr. h.c. Urs Hochstrasser, ehemaliger Direktor des Bundesamtes für Bildung und Wissenschaft (**http://hochstrasserurs.blogspot.ch**)

VORWORT DER AUTOREN

TigerJython besteht aus einem Online-Lehrmittel und einer speziell für den Unterricht entwickelten Entwicklungsumgebung. Das Online-Lehrmittel setzt bei der Turtlegrafik ein, führt aber weiter zu Themen wie der Programmierung von Lego-Robotern, Multimedia, Computerspielen, bis hin zu Datenbanken und stochastischen Simulationen. Zusammen mit dem modularen Aufbau und den zahlreichen Beispielen und Übungen eignet sich TigerJython sowohl für den Einsatz im Unterricht wie zum Selbststudium. Die ersten Kapitel können bereits in Informatik-Einführungskursen in der Volksschule (in der Schweiz S1) verwendet werden. Als Ganzes entspricht die Themenwahl und der Stoffumfang einem Grundlagenfach Informatik im gymnasialen Unterricht.

Die Autoren sind überzeugt, dass der Informatikunterricht Wesentliches zur geistigen Entwicklung von Jugendlichen beitragen kann. Er sollte unseres Erachtens bereits in der Grundschule spätestens im Alter von 12-13 Jahren einsetzen, damit bei Schülerinnen und Schülern frühzeitig Freude und Interesse am logisch-technischen Denken (computational thinking) geweckt wird.

Das Lehrmittel entstand in der ersten Fassung im Jahr 2013 und liegt nun in dieser zweiten überarbeiteten und korrigierten Fassung vor. Wir haben bei der Ausarbeitung unsere langjährigen Erfahrungen mit Schülerinnen und Schüler, sowie in der Ausbildung von Informatik-Lehrpersonen einfliessen lassen. Im Vordergrund stand dabei unsere Absicht, bei jungen Mädchen und Knaben Interesse und Freude am algorithmischen Problemlösen zu entwickeln und Lehrpersonen zu unterstützen.

Die Einstiegshürden ins Programmieren sind in diesem Lehrmittel bewusst sehr niedrig gehalten, wobei durchgängig die Programmierumgebung TigerJython und die Programmiersprache Python verwendet wird. Das Lehrmittel ist also sozusagen aus einem Guss entstanden. Viele Inhalte stammen aus dem täglichen Umfeld und aus Problemsituationen anderer Schulfächer. Damit lassen sich die Kenntnisse aus dem Informatikunterricht fächerübergreifend einsetzen.

Obschon Python bereits vor über 20 Jahren vom Niederländer Guido van Rossum entwickelt wurde, hat sich die Sprache in den Schulen erst in den letzten Jahren richtig entfaltet und erlebt in dieser Zeit an vielen Ausbildungsinstitutionen einen regelrechten Hype. Dies mag daran liegen, dass Python als interpretierte Sprache mit ihrem globalen Namensraum sehr einfach zu erlernen ist, aber auch, weil Python mit sehr wenig Computerressourcen auskommt und sogar auf Microsystemen lauffähig ist. Mit unserer Entwicklungsumgebung TigerJython steht zudem eine schülergerechte Umgebung zur Verfügung, die einen ausgewogenen Kompromiss zwischen Einfachheit und Professionalität aufweist und sich nach unserer Ansicht aus folgenden Gründen besonders gut für den Informatikunterricht eignet:

- Für die Installation auf Windows/Mac/Linux ist eine einzige Datei auf den Rechner zu kopieren. Dadurch können Lehrkräfte auch in Computerpools ohne Administratorrechte sofort mit dem Unterricht einsetzen
- Die IDE ist so einfach, dass keinerlei Einführungshilfe für deren Bedienung nötig ist. Es entfällt insbesondere das Erstellen von Projekten
- TigerJython führt eine präzise Fehleranalyse des Programms durch und schreibt für Programmieranfänger verständliche Fehlermeldungen aus
- TigerJython enthält zahlreiche Zusatzmodule, die speziell für den Unterricht zugeschnitten sind, wie Turtlegrafik, Koordinatengrafik, Robotik und Spielprogrammierung

Wir hoffen, dass wir mit TigerJython und dem Online-Lehrmittel etwas von unserer Begeisterung für die Informatikausbildung weitergeben können.

Verdankung:

Wir danken all jenen, die mit Anregungen und Rückmeldungen zum Gelingen von TigerJython und des Lehrmittels beigetragen haben, namentlich Walter Gander (ETH Zürich), Juraj Hromkovic (ETH Zürich), Theo Heußer (Gymnasium Hemsbach), Urs Hochstrasser (ehem. Bundesamt für Bildung und Wissenschaft, Bern).

Das Lehrmittel wird dank einer grosszügigen privaten Unterstützung gegenwärtig auf Englisch übersetzt. Die Übersetzung auf Französisch ist durch die Fachförderung 2015 der Schweizerischen Akademie der Technischen Wissenschaften (SATW) an die Schweizer Informatik Gesellschaft (SI) zugesichert. In diesem Zusammenhang danken wir dem Vorstand der SATW und Jürg Gutknecht (ETH Zürich, Präsident SI), sowie Beat Trachsler (KZO) für ihre Unterstützung.

Im Dezember 2014. Jarka Arnold, Tobias Kohn, Aegidius Plüss

kapitel eins

LERNUMGEBUNG

Lernziele

- Du kannst die TigerJython-Entwicklungsumgebung auf deinem Computer installieren.
- Du weisst, wie man mit TigerJython ein Programm editiert und ausführt.
- Du weisst, wie man Editor-Einstellungen vornimmt.
- Du weisst, wie man das Konsolenfenster für einfache Berechnungen verwendet.
- Du weisst, dass du TigerJython sogar auf dem Raspberry PI verwenden kannst.

"I think everybody in this country should learn how to program a computer because it teaches you how to think."

Steve Jobs, The lost interview

1.1 EINRICHTUNG

EINFÜHRUNG

Die Entwicklungsumgebung TigerJython eignet sich gut für Programmieranfänger und für Anwender, die in einer geschützten Umgebung (beispielsweise Computerpools ohne Administratorrechte) arbeiten. Die Distribution vom TigerJython umfasst eine einzige JAR-Datei, die kostenlos heruntergeladen werden kann.

Download TigerJython

Die Distribution enthält alle für die Programmierung notwendigen Komponenten, ausser dem Java Runtime Environment (**JRE**). TigerJython läuft sogar von einem externen Datenträger (USB-Stick, CD).

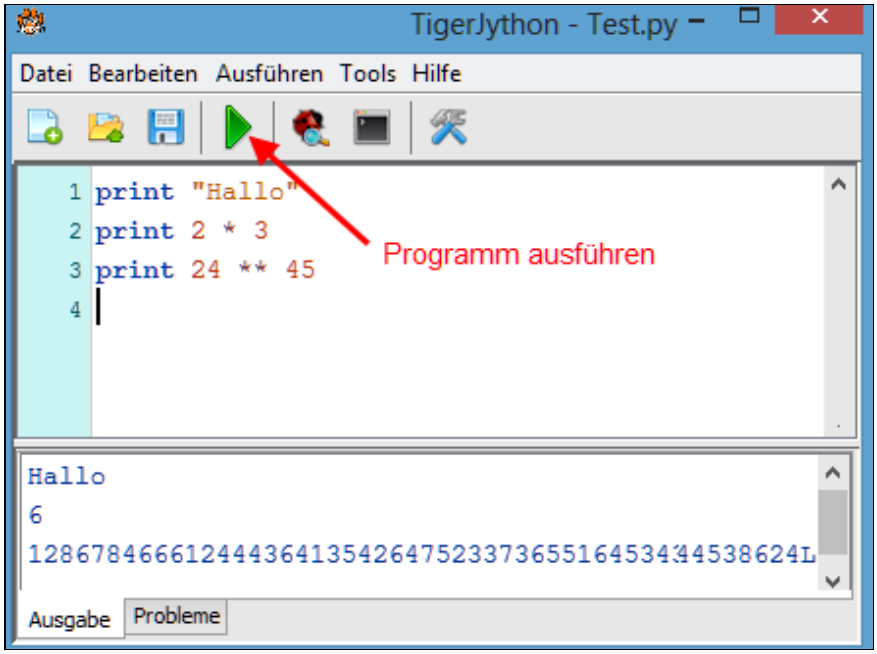
TigerJython ist **plattformunabhängig** und funktioniert problemlos unter Windows, Mac und Linux, sogar auch auf dem Raspberry Pi.

INSTALLATION

Lade die Datei tigerjython2.jar herunter. Speichere sie in einem beliebigen Verzeichnis auf der Festplatte und erstelle einen Link (Verknüpfung) auf diese Datei, damit du diese zum Beispiel vom Desktop starten kannst. Unter Linux musst du dieser Datei das Recht *Ausführen* (executable) geben. Im gleichen Verzeichnis kannst du auch alle deine Python-Programme speichern. Falls du dem Link eine passende Desktop-Ikone zuordnen möchtest, so kannst du diese für Windows von **hier** und für Mac/Linux von **hier** downloaden (Anleitung für Ubuntu siehe unten).

VERWENDUNG

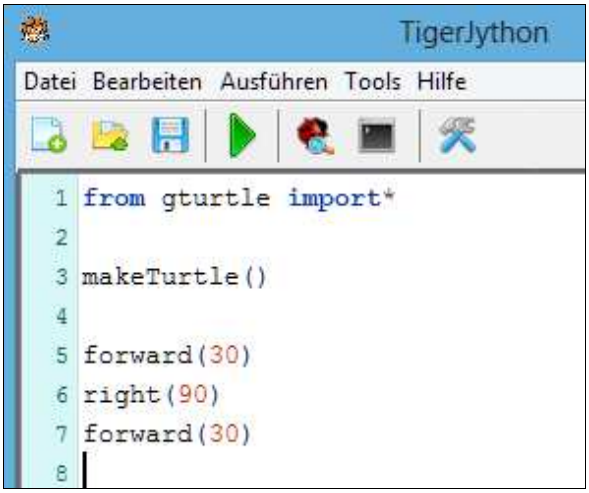
Starte den Editor mit Klick auf ***tigerjython2.jar***.



Die Bedienung des Editors ist einfach. Zur Verfügung stehen dir die Schaltflächen *Neues Dokument, Öffnen, Speichern, Programm ausführen, Debugger ein-/ausschalten, Konsole anzeigen* und *Einstellungen*. Teste die Funktionalität, in dem du einige print-Befehle eingibst und auf die grüne Schaltfläche *Programm ausführen* klickst. Im Unterschied zu den meisten anderen Programmiersprachen rechnet Python mit beliebig langen Zahlen.

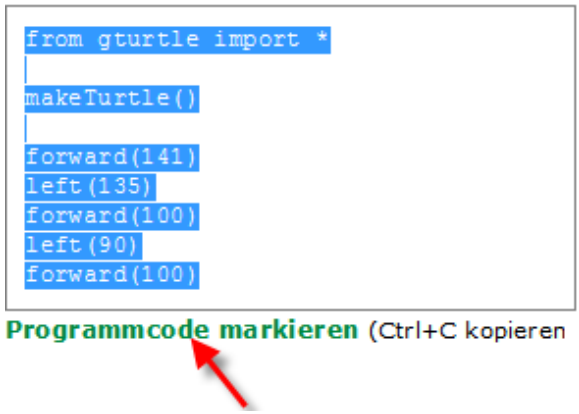
PROGRAMM EDITIEREN

Schreibe ein einfaches Turtlegrafik Programm.



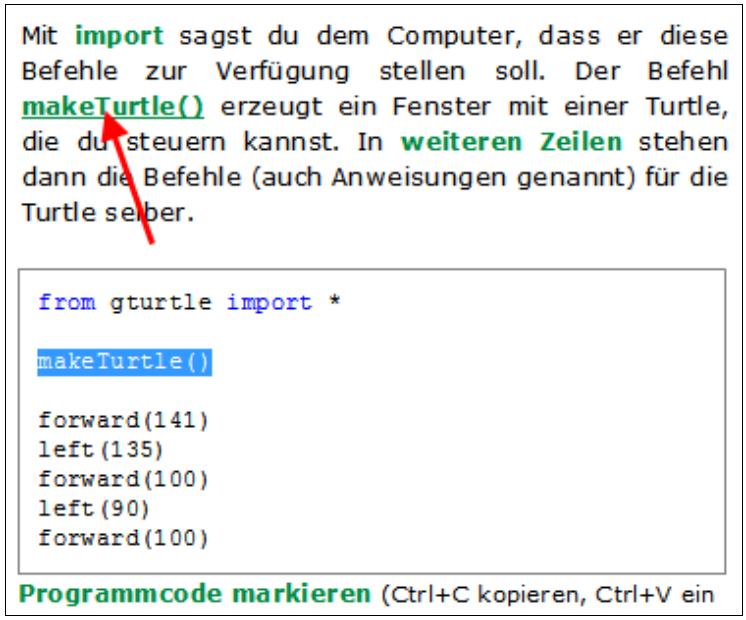
Tastenkürzel :

Ctrl+C	Kopieren
Ctrl+V	Einfügen
Ctrl+X	Ausschneiden
Ctrl+A	Alles markieren
Ctrl+Z	Rückgängig
Ctrl+S	Speichern
Ctrl+N	Neues Dokument
Ctrl+O	Öffnen
Ctrl+Y	Wiederholen
Ctrl+F	Suchen
Ctrl+H	Suchen und Ersetzen
Ctrl+Q	Markierte Zeilen auskommentieren Kommentarzeichen wegnehmen
Ctrl+D	Zeile löschen



Die Programmbeispiele im Lehrgang sind so aufbereitet, dass du sie sehr einfach als Programmvorlagen verwenden kannst.

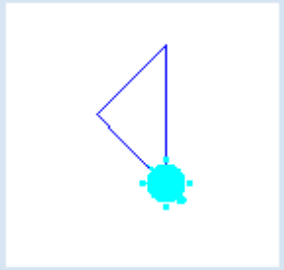
Mit Klick auf *Programmcode markieren* kannst du das ganze Programm auswählen. Mit der Maus kannst du aber auch nur einen Programmteil auswählen. Mit Ctrl+C kopierst du den markierten Programmcode in die Zwischenablage und mit Ctrl+V fügst du ihn in deinem TigerJython-Editorfenster ein.



Unser **Markierungstrick** hilft dir, die Erklärungen zum Programm zu verfolgen.

Mit Klick auf grün geschriebene Wörter wird die entsprechende Stelle im Programm markiert.

PROGRAMM AUSFÜHREN



Mit Klick auf den grünen Pfeil führst du das Programm aus.

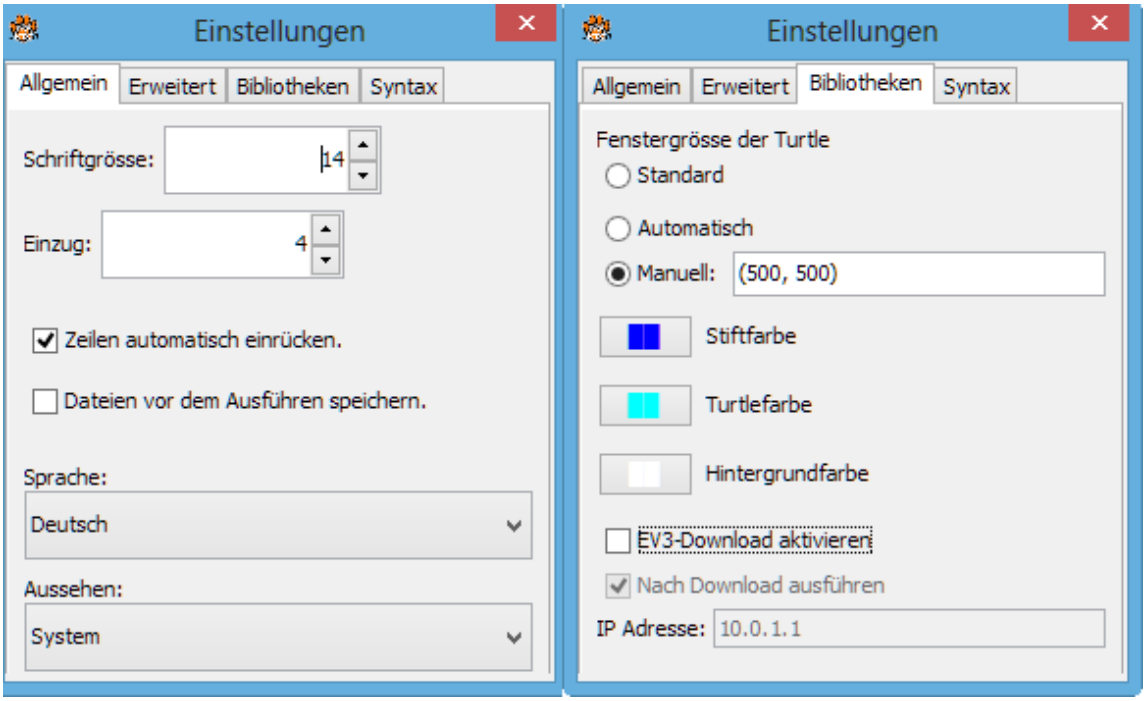
Die Grafik wird in einem neuen Grafikfenster angezeigt.

Falls das Programm nicht korrekt ist, erscheinen im Fenster *Probleme* die Fehlermeldungen.

EINSTELLUNGEN

Unter Einstellungen kannst du einige Anpassungen vornehmen:

- Schriftgrösse, Einzug und Schriftfarben des Editors
- Sprache (Deutsch, Englisch, Französisch)
- Grösse und Hintergrundfarbe des Turtlefensters, Stift- und Turtle-Farbe
- Zusätzliche Tools für EV3-Robotik aktivieren usw.



DOKUMENTATION

In TigerJython sind zusätzliche Module wie z. Bsp. Turtlegrafik integriert. Mit Klick auf *APLU Dokumentation* im Register *Hilfe* kannst du die Dokumentationen zu diesen Bibliotheken ansehen.

Wir schlagen dir vor, das Lehrmittel kapitelweise durchzuarbeiten und dabei die Beispielprogramme mit dem Link *Programmcode markieren* mit *Ctrl+C* und *Ctrl+V* einzeln in den Tigejython-Editor zu übernehmen, unter einem geeigneten Namen zu speichern und dann auszuführen. Du kannst aber auch **alle Programme** von **http://examples.tigerjython.ch downloaden**.

EINRICHTUNG IN COMPUTERPOOLS FÜR MEHRERE BENUTZER

TigerJython beschränkt sich auf einfache Konfigurationsdateien und lässt sich damit sehr einfach wieder von einem Computer entfernen. Es ist kein Installationsprozess notwendig und es werden keinerlei Einträge in eine Registry gemacht. Für benutzerdefinierbare Optionen wird eine Konfigurationsdatei *tigerjython2.cfg* verwendet, die im Normalfall automatisch im Homeverzeichnis von *tigerjython2.jar* angelegt wird. Für Einrichtungen in Computerpools kann diese Datei durch einen Systemadministrator verwaltet werden. Weitere Informationen findet man **hier**. Anmerkung: In seltenen Fällen sind in Computerpools für die Verwendung unter Java die JARs der APLU-Bibliotheken (z.B. aplu5.jar) in *<jrehome>/lib/ext* kopiert. Dies führt zu Konflikten mit TigerJython, das speziell konfigurierte APLU-Bibliotheken verwendet.

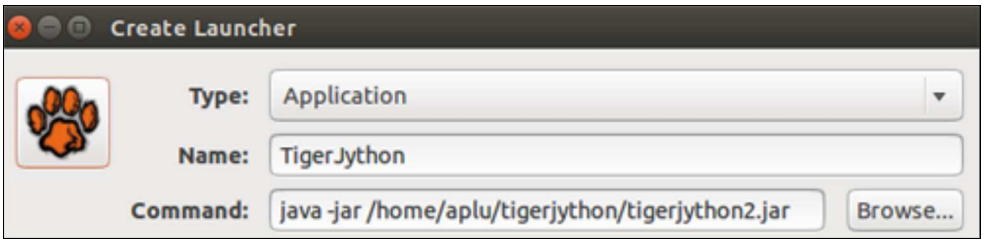
DESKTOP-LAUNCHER UNTER UBUNTU

Lade zuerst die Bilddatei tjlogo64.png von **hier** herunter und kopiere sie in das Verzeichnis, in dem sich tigerjython2.jar befindet. Für neuere Versionen von Ubuntu muss das gnome-panel installiert werden:

```
sudo apt-get install gnome-panel
```

Erzeugung der Launcher-Datei mit Alt-F2 und Eingabe von

```
gnome-desktop-item-edit --create-new ~/Desktop
```



Auf die Icone klicken und die heruntergeladene BildDatei tjlogo64.png angeben. Mit OK abschliessen.

PROGRAMM OHNE TIGERJYTHON IDE STARTEN

Da Python eine interpretierte Sprache ist, muss grundsätzlich immer ein Interpreter gestartet sein, damit ein Programm-Script ausgeführt werden kann. Unter Windows kann man ein Script mit der Kommandozeile
java -jar jython.jar <prog.py>
ausführen, vorausgesetzt dass man sich im Verzeichnis von jython.jar befindet.

Damit auch die zusätzlichen Module aus der APLU-Library automatisch mitgeladen werden, müssen sich diese in der JAR-Datei befinden. Von **hier** kann eine modifizierte Datei jython.jar mit dem Namen ajython.jar heruntergeladen werden, welche die Libraries enthält. Der Download enthält auch ein readme.txt mit weiteren Erklärungen sowie einige Testdateien. Achtung: Gewisse TigerJython-spezifische Konstrukte fehlen, insbesondere wird die repeat-Struktur nicht unterstützt und es fehlen die Input-Dialoge. Die Installation von Python oder Jython ist aber nicht nötig.

1.2 ERSTE SCHRITTE

EINFÜHRUNG

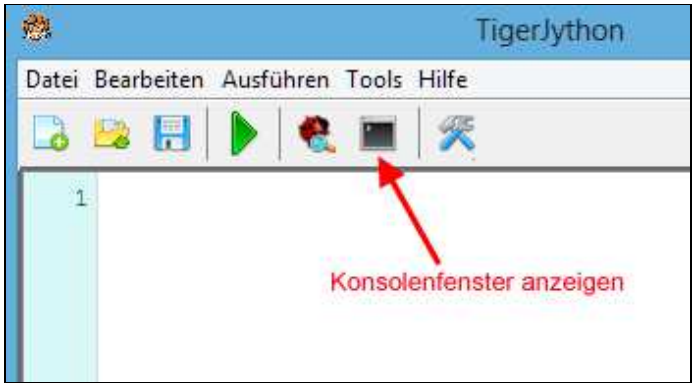
Ein Computerprogramm besteht in der Regel aus mehreren Anweisungen. In der Programmiersprache Python kannst du einzelne Anweisungen auch sofort ausführen. Dieses Vorgehen eignet sich besonders gut, um erste Erfahrungen mit Python zu machen oder etwas auszutesten. Dazu musst du das Konsolen-Symbol anklicken, wodurch das Konsolenfenster geöffnet wird. Auf der Befehlszeile, die mit >>> beginnt, tippst du die Anweisung ein und schliesst sie mit der ENTER-Taste (carriage return) ab. Wie in einem gewöhnlichen Editor kannst du dich mit den Cursortasten auf der Befehlszeile hin- und herbewegen und einzelne Zeichen löschen oder einfügen. Sobald du ENTER drückst, wird die Befehlszeile ausgeführt, ausser es handelt sich um einen mehrzeiligen Befehl. In diesem Fall wird der Befehl erst ausgeführt, wenn du mehrmals ENTER drückst.

Du kannst auch bereits verarbeitete Eingaben mit der Maus markieren und mit CTRL+C in die Zwischenablage kopieren. Befindest du dich auf der Befehlszeile, so kannst du mit CTRL+V den Inhalt der Zwischenablage dort einfügen.

Das Unterstreichungszeichen (Underline) ist ein Platzhalter für das Resultat einer vorgängigen Rechenoperation. Mit Cursor-Up kannst du die letzten Eingabezeilen zurückholen und mit Cursor-Left bzw. Cursor-Right editieren.

PYTHON KENNENLERNEN

Die folgenden Vorschläge kannst du ruhig auch ein wenig variieren, so wie du es gerade gut und lustig findest und je nachdem, was dich interessiert. Starte dazu *TigerJython* und wähle die Schaltfläche *Konsole*.



Tippe zum Kennenlernen der vier Grundrechenoperationen ein:

```
>>> 4 + 13
    17

>>> 2.5 - 5.7
    -3.2

>>> 1356 * 22345
    30299820

>>> 1 / 7
    0.14285714285714285
```

Wie du siehst, kannst du ganze Zahlen oder auch Dezimalbrüche verwenden. Die ganzen Zahlen nennt man *integer (int)* und die Dezimalbrüche *float*.

Du kannst mehrere Operationen auf eine Zeile schreiben. Dabei musst du den **Vorrang der Operationen** beachten, wobei gilt, dass * und / stärker binden als + und - und bei gleicher Rangordnung der Ausdruck von links nach rechts abgearbeitet wird. Du kannst

zusammengehörende Operation auch in ein rundes Klammerpaar schreiben. (Eckige und geschweifte Klammern haben aber eine andere Bedeutung):

```
>>> (66 - 12) * 5 / 6
    45.0

>>> 66 - 12 * 5 / 6
    56.0
```

Wichtig sind auch die **Ganzzahl-Division** und der **Divisionsrest** (Modulo-Operation):

```
>>> 5 // 3
    1

>>> 5 % 3
    2
```

Python kann mit langen Zahlen problemlos umgeben, beispielsweise bei der Verwendung des **Potenzoperators**:

```
>>> 45**123
    22138041353571795138171990088959838587798501812515796
    35495262099494113535880540560608088894435720496058262
    03407737866682728901508127084151522949268748976128137
    61281366450543228729941347410203889012336730957031 25L
```

Es gibt eine Reihe von eingebauten Funktionen, beispielsweise:

```
>>> abs(-9)
    9

>>> max(1, 5, 2, 4)
    5
```

Viele weitere Funktionen sind erst verfügbar, wenn du die entsprechenden Module importierst. Du kannst auf zwei Arten importieren. Bei der ersten Art musst du bei der Verwendung der Funktion (beim Funktionsaufruf) auch immer noch den Modulnamen mit einem Punkt voranstellen, bei der zweien Art kannst du die Funktion direkt aufrufen.

```
>>> import math
>>> math.pi
    3.141592653589793
>>> math.cos(pi)
    -1
```

```
>>> from math import *
>>> pi
    3.141592653589793
>>> sin(pi)
    1.2246467991473533e-16
```

Du siehst auch, dass ein Computerprogramm nie exakt rechnet, denn sin(pi) müsste ja exakt 0 sein.

Eine Aneinanderreihung von Buchstaben und Satzzeichen heisst ***String*** und du definierst ihn durch Verwendung von einfachen oder doppelten Anführungszeichen. Mit dem print-Befehl kannst du Strings und andere Werte in einem Ausgabefenster ausschreiben. Dabei verwendest du das Komma als Trennzeichen.

```
>>> print "Das Resultat ist", 2
```

Im Ausgabefenster erscheint: `Das Resultat ist 2`

Wie in der Mathematik kannst du Werte mit einem Variablennamen versehen. Dazu verwendest du einen Bezeichner aus einem oder mehreren Buchstaben. Nicht erlaubt sind aber Leerzeichen, Umlaute, Akzente und die meisten Spezialzeichen. Variablen dienen auch dazu, ein vorher errechnetes Resultat weiter zu verwenden. Es ist sehr praktisch, dass die bereits bekannten Variablen (und ihr Datentyp) im rechten Teil des Konsolenfensters aufgelistet sind.

```
>>> a = 2
```

```
>>> b = 3
>>> sum = a + b
>>> print "Die Summe von", a, "und", b, "ist", sum
```

Im Ausgabefenster erscheint: `Die Summe von 2 und 3 ist 5`

Eine eindimensionale Sammlung von beliebigen Daten nennt man eine ***Liste***. Listen sind in allen Programmiersprachen ein äusserst zweckmässiger und flexibler Datentyp. In Python schreibst du die Listenelemente einfach in eine eckige Klammer und kannst sie mit print im Ausgabefenster anzeigen lassen.

```
>>> li = [1, "Huhn", 3.14]
>>> print li
```

Im Ausgabefenster: `[1, "Huhn", 3.14]`

Listen und viele andere Objekte besitzen zugehörige Funktionen, die man Methoden nennt. Beispielsweise kannst du mit der ***Methode append()*** ein neues Element am Ende der Liste hinzufügen:

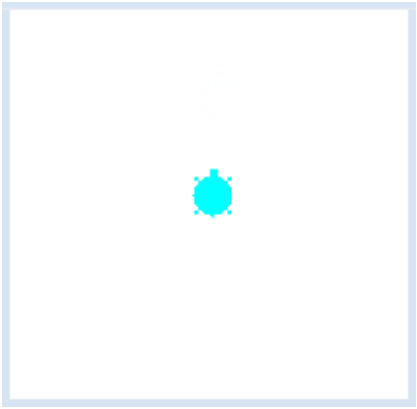
```
>>> li.append("Ei")
>>> print li
```

Im Ausgabefenster: `[1, 'Huhn', 3.14, 'Ei']`

Du kannst auch **eigene Funktionen definieren.** Dazu verwendest du das Schlüsselwort *def.* Werte gibst du mit return zurück. Nach der Definition kannst du deine Funktion wie jede andere eingebaute Funktion aufrufen:

```
>>> def sum(a, b):
>>>     return a + b
>>> sum(2, 3)

   5
```

DER TURTLE BEFEHLE ERTEILEN



Die Console ist sehr nützlich, um schnell ein paar Befehle oder Funktionen auszuprobieren. Willst du dich beispielsweise in die Turtlegrafik einarbeiten, so importierst du zuerst das Modul *gturtle* und erstellst mit dem Befehl *makeTurtle()* ein Turtlefenster mit einer Turtle.

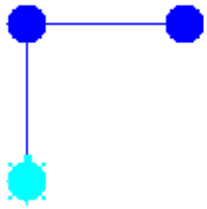
```
>>> from gturtle import *
>>> makeTurtle()
```

Danach stehen dir alle Befehle aus der Turtlegrafik direkt zur Verfügung. Beispielsweise:

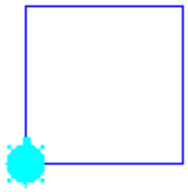
forward(100)	kurz: fd(100)	100 Schritte (Pixel) vorwärts bewegen
back(50)	kurz: bk(50)	50 Schritte rückwärts bewegen
left(90)	kurz: lt(90)	90° nach links drehen
right(90)	kurz: rt(90)	90° nach rechts drehen
clearScreen()	kurz: cs()	löscht alle Spuren und setzt die Turtle in die Mitte

Beispiel:

```
>>> fd(100)
>>> dot(20)
>>> rt(90)
>>> fd(100)
>>> dot(20)
>>> home()
```

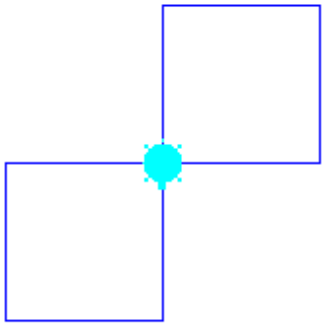


Mit dem Schlüsselwort *repeat* kannst du eine oder mehrere Anweisungen wiederholt ausführen. Wenn du mehrere Befehle als Befehlsblock wiederholen willst, so musst du sie gleich weit einrücken.



```
>>> repeat 4:
        fd(100)
        rt(90)
```

Wie oben gezeigt, kannst du mehrere zusammengehörige Anweisungen unter einem eigenen Namen zusammenfassen. Du erstellst eine eigene Funktionsdefinition. Der hauptsächliche Vorteil von Funktionen besteht darin, dass du Funktionen beiliebig oft einfach durch ihren Namen aufrufen kannst, statt den gesamten Code jedes Mal erneut aufzuschreiben.



```
>>> def zeichnequadrat():
      repeat 4:
         fd(100)
         rt(90)

>>> zeichnequadrat()
>>> rt(180)
>>> zeichnequadrat()
```

Es macht dir sicher Spass, weitere Turtlebefehle auszuprobieren. Eine Übersicht der Befehle findest du im Kapitel Turtlegrafik unter **Dokumentation**. In diesem Kapitel lernst du auch systematisch, wie du ganze Programme schreiben kannst.

1.3 HINWEISE FÜR LEHRPERSONEN

Das Lehrmittel besitzt einen inneren methodischen Aufbau "**vom Einfachen zum Komplizierten**" und verwendet in späteren Kapiteln grundsätzlich nur Kenntnisse und Begriffe, die in vorangehenden Kapiteln behandelt sind. Der gesamte Umfang deckt ein Grundlagenfach mit 4 - 5 Jahreslektionen ab. Es können aber je nach Schulstufe und Lektionenzahl auch nur ausgewählte Teile behandelt werden, wobei fehlende Begriffe nachzuholen sind. Da sich die Turtlegrafik hervorragend als Einstieg eignet, werden dort die meisten Grundlagen vermittelt.

Auf Grund unserer Unterrichtserfahrungen schlagen wir folgende **Minimalprogramm-Varianten** vor.

1. Turtlegrafik und Schülerprojekte (als Einführung in die Informatik im Sekundarschulbereich S1 und S2, 1 - 2 Jahresstunden)

2. Ausgewählte Themen der Turtlegrafik und des Kapitels Robotik (als Einführung in die Informatik oder für Informatik-Spezialwochen, falls Lego EV3 oder NXT-Roboter zur Verfügung stehen, mindestens 10 - 20 Lektionen)

3. Ausgewählte Themen der Turtlegrafik und Spielprogrammierung (1 - 2 Jahresstunden an höheren Schulen)

4. Ausgewählte Themen der Turtlegrafik, der Koordinatengrafik und Anwendungen in Schulfächern (als Einführung in die Informatik mit fächerübergreifenden Anwendungen, 1 - 2 Jahreslektionen)

5. Die ersten Themen der Turtlegrafik (als Einführung ins Programmieren in ICT-Kursen, 4 - 10 Lektionen)

Wir haben uns entschieden, in den Programmen durchwegs **englische Bezeichner** und Kommentare zu verwenden. Dies erleichtert einerseits die Übersetzung in andere Sprachen, entspricht aber auch dem allgemeinen Trend zur Internationalisierung von Programmcode.

Als Unterrichtshilfe für Lehrpersonen wird zu jedem Thema eine Stichwortliste mit den dort vorkommenden wichtigen **Informatikkonzepten** aufgeführt.

Lösungen der Aufgaben:

Sind Sie an einer Ausbildungsinstitution tätig, so können Sie die Lösungen der Aufgaben mit einer Email-Anfrage an help@tigerjython.com erhalten. Die Anfrage muss die folgenden überprüfbaren Angaben enthalten: Name, Adresse, Ausbildungsinstitution, Email-Adresse. Sie verpflichten sich dabei, die Lösungen nur für den persönlichen Gebrauch zu verwenden und nicht weiter zu geben.

1.4 RASPBERRY PI

EINFÜHRUNG

TigerJython kannst du auf dem Raspberry Pi verwenden, um die Programmiersprache Python zu erlernen oder auf das Soundsystem und den GPIO-Port zuzugreifen. TigerJython startet zwar etwas langsamer als das vorinstallierte Python mit IDLE, dafür steht dir eine ausgereifte grafische Entwicklungsumgebung mit vielen bereits integrierten Bibliotheksroutinen (Turtlegrafik, Robotik, Game-Entwicklung, usw.) zur Verfügung.

INSTALLATION

Am einfachsten lädst du von **http://www.raspberrypi.org/downloads** den Betriebssystem-Installer *NOOS* herunter, kopierst den Inhalt auf eine mindestens 8 GB grosse SD-Karte und wählst beim Start das Betriebssystem *Raspbian* (eine Linux Debian Variante). Da die Distribution bereits ein JRE enthält, brauchst du nur *tigerjython2.jar* in ein Verzeichnis, z.B. */home/pi/tigerjython* zu kopieren und der Datei mit dem File Manager unter *Dateieigenschaften* Ausführungsrechte zu geben.

Um TigerJython zu starten, tippst du in einem Terminal (Console) folgenden Befehl ein:

```
java -jar /home/pi/tigerjython/tigerjython2.jar
```

Für die Verwendung des GPIO-Moduls benötigt TigerJython Administratorrechte. Darum startest du TigerJython am besten immer mit vorgestelltem sudo:

```
sudo java -jar /home/pi/tigerjython/tigerjython2.jar
```

Statt jedesmal diese Befehlszeile einzutippen, kannst du im File Manager angeben, dass für Dateien von Dateityp .jar immer dieser Befehl auszuführen ist oder ein Shell-Skript erstellen. Noch einfacher ist es, einen Desktop-Link zu machen, wobei du dabei wie folgt vorgehst:

- Erstelle mit mit einem rechten Mausklick auf die Ikone *IDLE* durch Kopieren und Einfügen ein neues Link-Symbol auf dem Desktop.

- Um das zughörige Link-Skript zu editieren, klickst du mit der rechten Maustaste auf dieses Symbol und wählst *Leafpad*. Du kannst nun die Einträge entsprechend anpassen und sogar ein TigerJython-Logo angeben (**download**). Hier ein Beispiel:

Datei Bearbeiten Suchen Optionen Hilfe

```
[Desktop Entry]
Name=Tiger
Exec=sudo java -jar /home/pi/tigerjython/tigerjython2.jar
Icon=/usr/share/pixmaps/tjlogo1.png
Terminal=false
Type=Application
Categories=Application;Development;
StartupNotify=true
```

Nach dem Speichern kannst du TigerJython mit einem Klick auf die neue Ikone starten. Dabei brauchst du auf dem etwas schmalbrüstigen Raspberry Pi etwa 1 Minute Geduld, bis die IDE gestartet ist. Wie du feststellen wirst, ist aber Ausführung von Python-Programmen erstaunlich schnell.

DIGITALER INPUT-OUTPUT AM GPIO-PORT

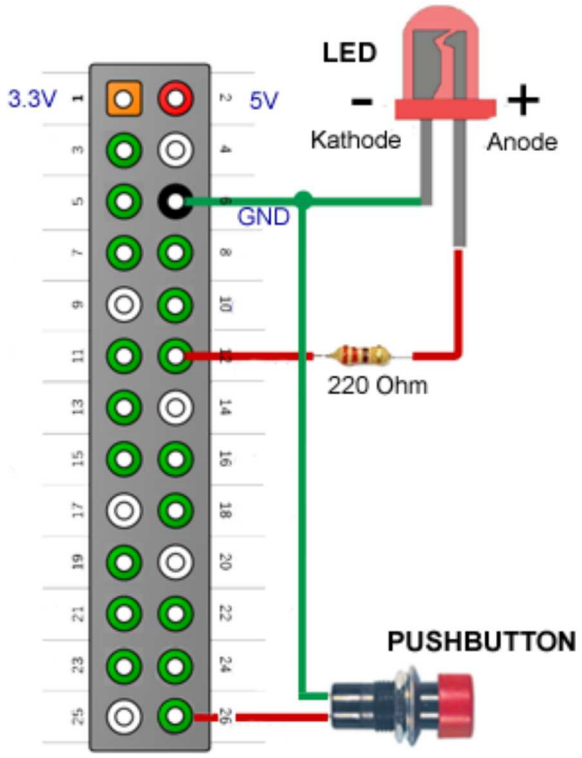
Raspberry Pi stellt dir 17 digitale Input-Output-Kanäle zur Verfügung, die sich an einer 26-poligen Steckerleiste (die neue Version mit dem 40 poligen Stecker wir auch unterstützt) abgreifen lassen. Jeder Kanal kann als Ausgang oder als Eingang mit einem internen Pull-up-Widerstand, mit einem Pull-down-Widerstand oder ohne Widerstand definiert werden. Auf der Steckerleiste sind auch 5V, 3.3V und Ground-Pins vorhanden. **Die Eingangsspannung darf in keinem Fall 3.3 V übersteigen, man darf also externe 5V-Logik-Ausgänge nicht direkt mit den Eingängen verbinden**.

In TigerJython steht dir die Klasse *GPIO* aus dem Modul *RPi_GPIO* zur Verfügung, mit dem du die IO-Ports sehr einfach ansprechen kannst. Das Modul verwendet die Bibliothek Pi4J von Robert Savage, entspricht aber weitgehend dem Modul ***RPi.GPIO***. Du benötigst zwei zusätzliche JAR-Dateien *raspi-gpio.jar und pi4j-core.jar*, die du von **hier** herunterlädst und sie in das Lib-Unterverzeichnis des Verzeichnisses kopierst, in dem sich *tigerjython2.jar* befindet.

Standardmässig werden die Pins des GPIO-Ports mit den Pinnummern 1..26 (neues Board: 1..40) angesprochen. Jeder Pin kann mit *GPIO.setup()* als Ein- oder Ausgangskanal (channel) definiert werden. Mit *GPIO.output(channel, state)* wird ein Ausgabewert gesetzt und mit *GPIO.input(channel)* der aktuelle Eingabewert gelesen und zurückgegeben.

Du findest die ausführliche Dokumentation unter der Menüoption *Hilfe > APLU Dokumentation*.

Zum Testen schliesst du am einfachsten eine LED mit einem Seriewiderstand zwischen Pin 6 (Ground) und Pin 12 an, wobei du die Polarität der LED ausprobieren musst (sie wird durch eine falsche Polarität nicht zerstört). Hast du einen Tastenschalter zur Verfügung, so schliesst du ihn zwischen Pin 6 (Ground) und Pin 26 an.



In deinem Programm definierst du zuerst Channel 12 als Ausgang und lässt die LED 10 mal pro Sekunde blinken:

```
from RPi_GPIO import GPIO

GPIO.setup(12, GPIO.OUT)

while True:
    GPIO.output(12, 1)
    GPIO.delay(100)
    GPIO.output(12, 0)
    GPIO.delay(100)
```

Zur Demonstration eines Eingangsports schliesst du einen Tastenschalter wie im Schema gezeigt an Pin 26 an und brauchst dann typisch einige Flags, damit du durch Drücken des Tasters das Blinken einschalten und durch nochmaliges Drücken das Blinken wieder ausschalten kannst.

```
from RPi_GPIO import GPIO

# pin numbers
switch = 26
led = 12

print "Press button turn blinking on/off"

GPIO.setup(led, GPIO.OUT)
GPIO.setup(switch, GPIO.IN, GPIO.PUD_UP)
buttonPressed = False
blinking = False
ledOn = False

while True:
    v = GPIO.input(switch)
    if not buttonPressed and v == GPIO.LOW:
        buttonPressed = True
        blinking = not blinking

    if buttonPressed and v == GPIO.HIGH:
        buttonPressed = False

    if blinking:
        if ledOn:
            ledOn = False
            GPIO.output(led, GPIO.LOW)
        else:
            ledOn = True
            GPIO.output(led, GPIO.HIGH)
    else:
        ledOn = False
        GPIO.output(led, GPIO.LOW)
    GPIO.delay(100)
```

Programmcode markieren
(Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

In der Schleife holst du zuerst den aktuellen Zustand des Tastenschalters (du "pollst" ihn). Beim gedrückten Zustand verbindet er GND mit dem Eingang auf Pin 26 und *input(26)* liefert GPIO.LOW oder 0. Lässt du die Taste los, so bewirkt der interne Pull-up-Widerstand, dass der Eingang auf logisch HIGH (3.3V) gesetzt wird, ohne dass du diese Spannung von Aussen anlegen musst.

Die Verwendung des Tastenschalters als Ein-/Ausschalter ist etwas trickreich, da du trotz dem ständigen Pollen das Ein-/Ausschalten in ein Ereignis umwandeln musst, das nur dann auftritt, wenn die Taste runter geht. Du verwendest dazu ein Flag *buttonPressed*, das du beim ersten Tastendruck auf *True* setzt. Nachfolgend durchläufst du den entsprechenden Codeteil nicht mehr, bist du die Taste wieder losgelassen hast und erneut drückst.

Da die Schleife auch für das Blinken verantwortlich ist, verwendest du ein flag *blinking*, dass sozusagen den Ein-/Ausschaltzustand repräsentiert. Schliesslich muss du dir noch mit dem Flag l*edOn* merken, ob bei einem Schleifendurchlauf die LED ein- oder ausgeschaltet werden soll.

Im Moment des Abschaltens bist du nicht sicher, ob die LED gerade leuchtet oder nicht, daher schaltest du sie im letzten *else* aus. Als kleine Unschönheit wird aber dieser Code nachfolgend immer wieder durchlaufen.

Die Elektroniker wissen, dass eine Taste beim Drücken meist nicht sofort definitiv Kontakt macht (sie "prellt"). Da aber die Schleife wegen des *delay(100)* erst nach 100 ms wieder durchlaufen wird, ist anzunehmen, dass das "Prellen" dann zu Ende ist.

ELEGANT MIT DEM EVENTMODELL PROGRAMMIEREN

Viel einfacher ist es, das Eventmodell zu verwenden. Dabei wird das Drücken bzw. Loslassen der Taste als ein Ereignis aufgefasst, wobei automatisch eine Funktion *onButtonPressed()* aufrufen wird. Dort brauchst du nur das Flag *blinking* umzukehren.

```
from RPi_GPIO import GPIO

def onButtonPressed(channel, state):
   global blinking
   blinking = not blinking

# pin numbers
switch = 26
led = 12

print "Press button to turn blinking on/off"
GPIO.setup(led, GPIO.OUT)
GPIO.setup(switch, GPIO.IN, GPIO.PUD_UP) # pull-up resistor
GPIO.add_event_detect(switch, GPIO.FALLING) # event on falling edge
GPIO.add_event_callback(switch, onButtonPressed) # register callback

blinking = False
while True:
    if blinking:
        GPIO.output(led, 1)
        GPIO.delay(100)
        GPIO.output(led, 0)
    GPIO.delay(100)
```

Programmcode markieren
(Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Um Events zu verwenden, muss du zuerst mit der Methode *add_event_detect()* angeben, ob du auf eine steigende oder fallende Flanke oder auf beide reagieren willst, also beim Übergang von LOW auf HIGH oder umgekehrt. Nachfolgend registrierst du mit *add_event_callback()* die Funktion, die beim Auftreten des Ereignisses aufgerufen werden soll.

kapitel zwei

TURTLE - GRAFIK

Lernziele

- Du kannst ein einfaches Programm schreiben und mit der Turtle Figuren auf den Bildschirm zeichnen.
- Du kannst die Farbe der Linien und Flächen und die Linienbreite einstellen.
- Du weisst, wie die Turtle Anweisungen mehrmals wiederholen kann.
- Du weisst, wie man Programmteile nur unter bestimmten Bedingungen ausführen kann.
- Du kannst eigene Befehle mit Parametern definieren.
- Du kennst die Bedeutung von Variablen und kannst diese in deinen Programmen verwenden.
- Du weisst, was Rekursionen sind und kannst einfache rekursive Programme schreiben.
- Du kannst Turtle-Objekte erzeugen und mehrere Turtles gleichzeitig verwenden.

"Eine Turtle befindet sich an einem bestimmten Ort und sie hat auch eine bestimmte Blickrichtung. Damit ist eine Turtle wie eine Person... Kinder können sich mit der Turtle identifizieren und übertragen ihr Wissen über ihren Körper in das Erlernen der Geometrie."

Seymour Papert

2.1 TURTLE BEWEGEN

EINFÜHRUNG

Programmieren heisst, einer Maschine Befehle zu erteilen und sie damit zu steuern. Die erste solche Maschine, die du steuerst, ist eine kleine Schildkröte auf dem Bildschirm: Die Turtle. Was kann diese Turtle und was musst du wissen, um sie zu steuern?

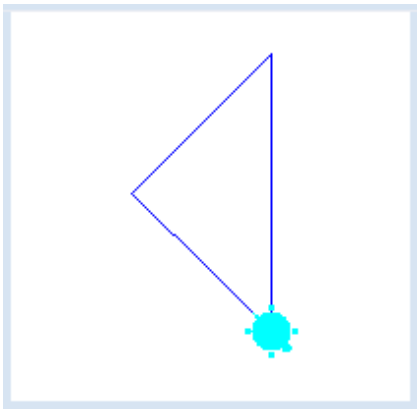
Turtlebefehle werden grundsätzlich Englisch geschrieben und enden immer mit einem Klammerpaar. Dieses enthält weitere Angaben zum Befehl. Selbst wenn keine weiteren Angaben nötig sind, muss ein leeres Klammerpaar vorhanden sein. Die Klein/Grossschreibung muss exakt eingehalten werden.

Die Turtle kann sich innerhalb ihres Fensters bewegen und dabei eine Spur zeichnen. Bevor die Turtle aber loslegt, musst du den Computer anweisen, dir eine solche Turtle zu erzeugen. Das machst du mit dem Befehl *makeTurtle().* Um die Turtle zu bewegen verwendest du die drei Befehle *forward(distanz)*, *left(winkel)* und *right (winkel)*.

PROGRAMMIERKONZEPTE: *Quellprogramm editieren, Programm ausführen, Programmsequenz*

DEIN ERSTES PROGRAMM

So sieht dein erstes Programm mit der Turtle aus. Klicke auf *Programmcode markieren*, kopiere und füge es in den TigerJython-Editor ein. Führe es aus, indem du auf den grünen Start-Knopf klickst. Die Turtle zeichnet ein rechtwinkliges Dreieck.



Die Turtlebefehle sind in einer Datei (einem sogenannten Modul) »gturtle« gespeichert.

Mit **import** sagst du dem Computer, dass er diese Befehle zur Verfügung stellen soll. Der Befehl **makeTurtle()** erzeugt ein Fenster mit einer Turtle, die du steuern kannst. In **weiteren Zeilen** stehen dann die Befehle (auch Anweisungen genannt) für die Turtle selber.

```
from gturtle import *

makeTurtle()

forward(141)
left(135)
forward(100)
left(90)
forward(100)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Am Anfang jedes Turtle-Programms musst du zuerst das Turtle-Modul laden und eine neue Turtle erzeugen:

```
from gturtle import *
makeTurtle()
```

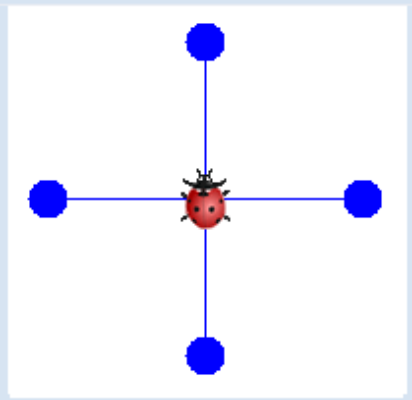
Danach kannst du der Turtle beliebig viele Befehle geben. Die drei Befehle, die die Turtle sicher versteht sind:

forward(s)	s (in pixel) vorwärts bewegen.
left(w)	Um den Winkel w (in Grad) nach links drehen.
right(w)	Um den Winkel w nach rechts drehen.

DEINE PERSÖNLICHE TURTLE

Du kannst bei **makeTurtle()** eine Datei angeben, welche als Turtlebild verwendet wird und so deinem Programm eine persönliche Note geben. Hier verwendest du die Datei *beetle.gif* aus dem Verzeichnis *sprites* der Tigerjython-Distribution. Beachte, dass du den Dateinamen in Anführungszeichen setzen musst.

Die Turtle zeichnet ein Kreuz mit gefüllten Kreisen in den Endpunkten.



```
from gturtle import *

makeTurtle("sprites/beetle.gif")

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

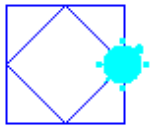
MEMO

Willst du ein anderes Symbol für die Turtle verwenden wie in obigem Beispiel, so musst du mit irgendeinem Bildeditor ein Bild erstellen. Übliche Turtlebilder haben eine Grösse von 32x32 Pixel auf einem transparenten Hintergrund im gif oder png-Format. Die Bilddatei sollte sich im Unterverzeichnis *sprites* des Verzeichnisses gespeichert sein, in dem sich dein Programm befindet [**mehr...**]

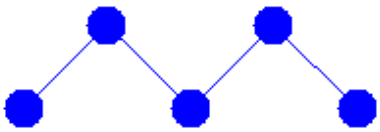
Im Programm verwendest du den neuen Befehl **back()**, mit dem sich die Turtle rückwärts bewegt, sowie **dot()**, mit dem die Turtle einen gefüllten Kreis zeichnet, dessen Radius (in Pixel) du angeben kannst.

AUFGABEN

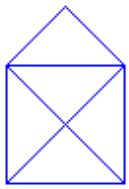
1. Zeichne mit der Turtle zwei Quadrate ineinander.



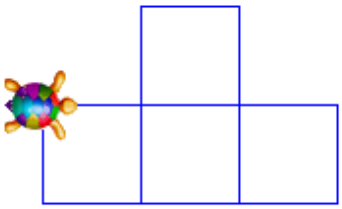
2. Verwende zusätzlich den Befehl dot(), um folgende Figur zu zeichnen:



3. Das »Haus vom Nikolaus« ist ein Zeichenspiel für Kinder. Ziel ist es, das besagte Haus in einem Linienzug aus genau 8 Strecken zu zeichnen, ohne dabei eine Strecke zweimal zu durchlaufen. Zeichne das Haus vom Nikolaus mithilfe der Turtle.



4*. Erstelle mit einem Bildeditor ein eigenes Turtlebild und zeichne damit nebenstehendes Bild. Die Seitenlänge der Quadrate ist 100. Es ist gleichgültig, wo die Turtle mit der Zeichnung beginnt/endet.



2.2 FARBEN VERWENDEN

EINFÜHRUNG

Um ihre Spur zu zeichnen, hat die Turtle einen Farbstift (engl. pen). Für diesen Farbstift kennt die Turtle weitere Anweisungen. Solange der Farbstift »unten« ist, zeichnet die Turtle eine Spur. Mit *penUp()* nimmt sie den Farbstift nach oben und bewegt sich nun, ohne eine Spur zu zeichnen. Mit *penDown()* wird der Farbstift wieder nach unten auf die Zeichenfläche gebracht, so dass eine Spur gezeichnet wird.

Über die Anweisung *setPenColor(farbe)* kannst du die Farbe des Stifts auswählen. Wichtig ist, dass du den Farbnamen in Gänsefüsschen setzt. Wie immer beim Programmieren kennt die Turtle nur englische Farbnamen. Die folgende Liste ist zwar nicht vollständig, aber doch ein erster Anhaltspunkt: *yellow, gold, orange, red, maroon, violet, magenta, purple, navy, blue, skyblue, cyan, turquoise, lightgreen, green, darkgreen, chocolate, brown, black, gray, white.*

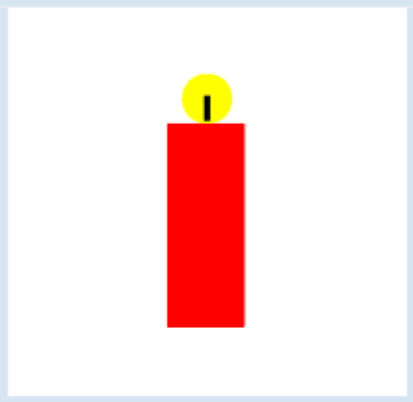
PROGRAMMIERKONZEPTE: *Programmgesteuertes Zeichnen*

FARBE UND BREITE DES STIFTS

Mit einer breiten **roten** Linie zeichnet die Turtle eine Kerze. Die Linienbreite in Pixel kannst du mit dem Befehl **setLineWidth()** einstellen.

Die **gelbe** Flamme kannst du mit dem Befehle dot() zeichnen. Dazwischen ist ein Programmteil, in dem sich die Turtle zwar bewegt, aber keine Linie zeichnet, weil der Stift mit **penUp()** gehoben wurde. Nach **penDown()** zeichnet die Turtle wieder.

hideTurtle() macht die Turtle unsichtbar.



```
from gturtle import *

makeTurtle()

setLineWidth(60)
setPenColor("red")
forward(100)
penUp()
forward(50)
penDown()
setPenColor("yellow")
dot(40)
setLineWidth(5)
setPenColor("black")
back(15)
hideTurtle()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Der Zeichenstift (pen) der Turtle kann mit **setPenColor(farbe)** die Farbe wechseln. Bei **penUp()** hört die Turtle auf, etwas zu zeichnen, bei **penDown()** zeichnet sie wieder weiter.
Die Breite der gezeichneten Linie kannst du mit der Anweisung **setLineWidth(breite)** steuern.

Die Turtle kennt die sogenannten ***X11-Farben***. Das sind einige dutzend Farbnamen, die du im Internet unter **http://cng.seas.rochester.edu/CNG/docs/x11color.html** finden kannst. Alle diese Farben kannst du mit setPenColor(farbe) wählen.

GEFÜLLTE FLÄCHEN

Du kannst mit der Turtle fast beliebige Flächen mit Farben füllen. Mit dem Befehl **startPath()** sagst du der Turtle, dass du die Absicht hast, eine Fläche zu füllen. Die Turtle merkt sich dabei die momentane Lage als Startpunkt eines Linienzugs. Du umfährst dann mit der Turtle die Fläche und befiehlst ihr zuletzt mit **fillPath()**, denn erreichten Endpunkt des Linienzugs mit dem Startpunkt zu verbinden und die so entstandene Fläche zu füllen. Die Füllfarbe kannst du mit **setFillColor(farbe)** einstellen.

Texte einer Zeile nach einer einleitenden Raute # sind **Kommentare**. Sie werden bei der Programmausführung nicht beachtet. Du kannst dir zum Beispiel notieren, unter welchen Programmnamen die Datei gespeichert ist oder erklärenden Text zum Programmcode schreiben.

```
from gturtle import *

makeTurtle()

setPenColor("sandybrown")
setFillColor("sandybrown")
startPath()
forward(100)
right(45)
forward(72)
right(90)
forward(72)
right(45)
forward(100)
fillPath()
hideTurtle()
```

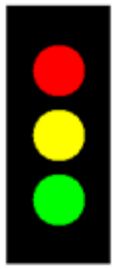
MEMO

Wenn du eine Fläche, die mit einem Linienzug beschränkt ist, füllen willst, beginnst du das Zeichnen mit dem Befehl **startPath()**. Mit **fillPath()** wird der Startpunkt und der Endpunkt verbunden und die geschlossene Fläche gefüllt. **Kommentare** werden mit dem Zeichnen # eingeleitet.

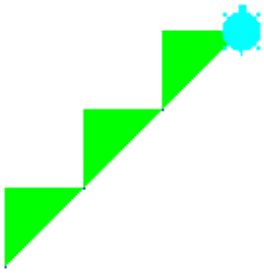
AUFGABEN

1. Zeichne mit der Turtle ein regelmässiges Sechseck (Hexagon) und wähle für jede Seite eine andere Farbe.

2. Zeichne eine Ampel. Das schwarze Rechteck kannst du mit der Stiftbreite 80 zeichnen, die Kreisflächen mit dot(40).



3. Die Turtle soll das nebenstehende Bild zeichnen.



2.3 WIEDERHOLUNG

EINFÜHRUNG

Computer sind besonders gut darin, die gleichen Anweisungen (also auch Turtle-Befehle) immer wieder zu wiederholen. Um ein Quadrat zu zeichnen, musst du also nicht viermal die Befehle *forward(100)* und *left(90)* eingeben. Es genügt auch, der Turtle zu sagen, sie soll diese zwei Anweisungen viermal wiederholen.

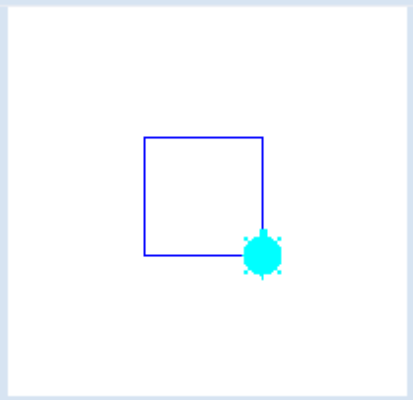
Mit der Anweisung *repeat* sagst du der Turtle, sie soll einige Befehle eine bestimmte Anzahl Mal wiederholen. Damit der Computer weiss, dass diese Befehle zusammengehören (einen Programmblock bilden), müssen diese gleich weit eingerückt sein. Wir verwenden für Einrückungen grundsätzlich drei Leerschläge.

PROGRAMMIERKONZEPTE: *Einfache Wiederholstruktur statt Codeduplikation*

REPEAT - STRUKTUR

Um ein Quadrat zu zeichnen, muss die Turtle vier mal geradeaus gehen und sich je um 90° drehen. Würdest du das alles untereinander schreiben, dann würde das Programm ziemlich lange.

Mit der Anweisung **repeat 4:** sagst du der Turtle, sie soll die **eingerückten Zeilen** viermal wiederholen. Achtung: Vergiss den Doppelpunkt nicht!



```
from gturtle import *

makeTurtle()
repeat 4:
    forward(100)
    left(90)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

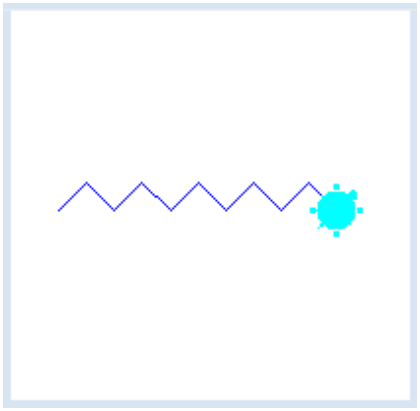
MEMO

Mit *repeat Anzahl:* sagst du dem Computer, er soll eine oder mehrere Anweisungen »Anzahl« Mal wiederholen, bevor er weitere Anweisungen ausführt. Alles, was wiederholt werden soll, muss unter repeat stehen und eingerückt sein.

```
repeat anzahl:
    Anweisungen, die
    wiederholt werden
    sollen
```

TÖNE WIEDERHOLEN

Ein typisches Beispiel für eine Wiederholung ist das Dah-Dih-Dah-Dih eines Feuerwehr-Autos. Mit der Turtle kannst du eine solche Tonfolge leicht erzeugen und gleichzeitig spasseshalber die Turtle eine Zickzack-Kurve zeichnen lassen. Einen reinen Ton erzeugst du mit **playTone()**, wobei du die Tönhöhe als Frequenzangabe (in Hertz) und die Dauer des Tons (in Millisekunden) angibst.



```
from gturtle import *

makeTurtle()

setPos(-200, 0)
right(45)

repeat 5:
    playTone(392, 400)
    forward(50)
    right(90)
    playTone(523, 400)
    forward(50)
    left(90)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

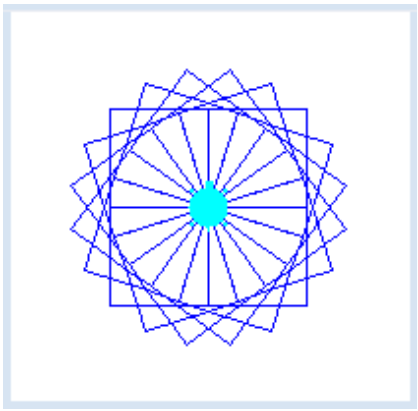
MEMO

Mit **setPos(x, y)** kannst du die Turtle auf eine bestimmte Position im Turtlefenster setzen, ohne dass sie dabei eine Spur zeichnet. Die beiden Zahlen x und y sind die Koordinaten, wobei der Nullpunkt in der Mitte des Fensters liegt. (Der Koordinatenbereich hängt von der Grösse des Fensters ab.)

Du kannst bei **playTone()** die Tonhöhen auch mit einem Buchstaben gemäss der Notenskala angeben, also mit c, d, usw. oder in der eingestrichenen Oktave mit c', d', usw. (oder mit zwei oder drei Apostrophen). Du muss die Tonhöhen aber in Anführungszeichen setzen. Wenn du willst, kannst du auch noch eine Angabe des zu verwendenden Musikinstruments machen (vorhanden sind: piano, guitar, harp, trumpet, xylophone, organ, violin, panflute, bird, seashore). Versuch es mal mit

Unterer Ton: playTone("g'", 400, instrument = "trumpet")
Oberer Ton: playTone("c''", 400, instrument = "trumpet")

VERSCHACHTELTES REPEAT



Ein Quadrat gelingt einfach mit einer vierfachen Wiederholung. Nun sollst du 20 Quadrate zeichnen und die Quadrate etwas gegeneinander verdrehen. Dazu musst du zwei repeat-Anweisungen ineinander schachteln. Im **inneren Programmblock** zeichnet die Turtle ein Quadrat und dreht anschliessend um 18 Grad nach rechts. Die **äussere repeat-Anweisung** wiederholt dies 20 mal. Beachte dabei die korrekte Einrückung bei den Anweisungen, die wiederholt werden sollen.

Falls du die Turtle mit **hideTurtle()** versteckst, erfolgt das Zeichnen schneller.

```
from gturtle import *

makeTurtle()

# hideTurtle()
repeat 20:
    repeat 4:
        forward(80)
        left(90)
    right(18)
```

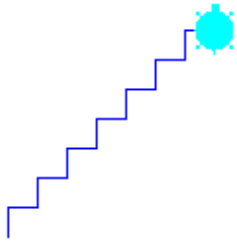
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

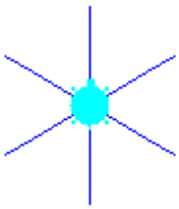
Die repeat-Befehle lassen sich verschachteln. Eine korrekte Einrückung bei den Anweisungen, die wiederholt werden sollen, ist wichtig.

AUFGABEN

1. Zeichne eine Treppe mit 7 Stufen.

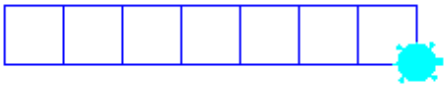


2. Zeichne einen Stern. Verwende dabei den Befehl *back()*.



3. Einen "richtigen" Stern kannst du mit den Drehwinkeln 140° und 80° zeichnen.

4. Zeichne folgende Figur mit zwei verschachtelten *repeat*-Anweisungen. Im inneren repeat-Block wird ein Quadrat gezeichnet.



5. Mit dem Befehl *leftArc(30, 180)* zeichnet die Turtle einen Linksbogen mit dem Radius 30 und Sektorwinkel *180°,* also einen Halbkreis.

Zeichne mit den Befehlen *leftArc(radius, angle)* und *rightArc(radius, angle)* die nebenstehende Wellen-Figur.

6. Zeichne mit Hilfe von zwei Viertelkreisen einen Vogel.

7..Zeichne 5 Treppenstufen und spiele dazu Töne mit den Frequenzen 264, 297, 330, 352 und 396 ab.

2.4 FUNKTIONEN

EINFÜHRUNG

In einem grösseren Bild kommen Figuren wie Dreiecke und Quadrate mehrmals vor. Die Turtle weiss aber nicht, was ein Dreieck oder ein Quadrat ist. Du musst also jedes Mal der Turtle mit einem vollständigen Programmcode erklären, wie sie die Figuren zeichnet. Geht das nicht auch einfacher?

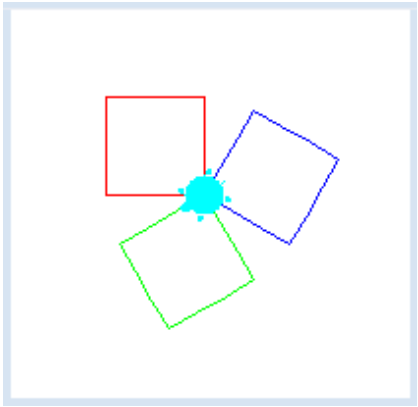
Es geht einfacher! Du kannst der Turtle nämlich neue Befehle beibringen, z.B.ein Quadrat oder Dreieck zu zeichnen, und musst ihr dann nur sagen, sie soll diesen Befehl ausführen, also ein Quadrat oder ein Dreieck zeichnen. Um einen neuen Befehl zu definieren, wählst du einen beliebigen Bezeichner, beispielsweide *square*, und schreibst *def square():* Danach notierst du alle Anweisungen, die zum neuen Befehl gehören. Damit der Computer weiss, was zum neuen Befehl gehört, müssen diese Anweisungen eingerückt sein.

PROGRAMMIERKONZEPTE: *Modulares Programmieren, Funktionsdefinition, Funktionsaufruf*

EIGENE BEFEHLE DEFINIEREN

In diesem Programm definierst du mit ***def*** den neuen Befehl **square()**. Die Turtle weiss danach, wie sie ein Quadrat zeichnen kann, sie hat aber noch keines gezeichnet.

Mit dem Befehl *square()* zeichnet die Turtle an jeder aktuellen Position ein **Quadrat** mit der Seitenlänge 100. In unserem Beispiel ist es ein rotes, ein blaues und ein grünes Quadrat.



```
from gturtle import *

def square():
    repeat 4:
        forward(100)
        left(90)

makeTurtle()
setPenColor("red")
square()
right(120)
setPenColor("blue")
square()
right(120)
setPenColor("green")
square()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

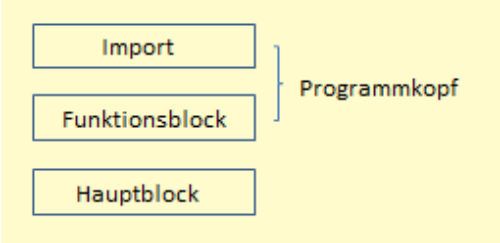
MEMO

Mit ***def bezeichner():*** definierst du einen neuen Befehl. Wähle einen Namen, der die Tätigkeit wiederspiegelt. Alle Anweisungen, die zum neuen Befehl gehören, müssen eingerückt sein.

```
def bezeichner():
    Anweisungen
```

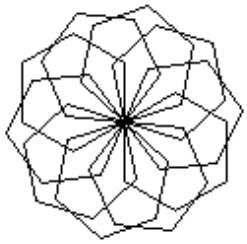
Vergiss die Klammern und den Doppelpunkt nach dem Bezeichner nicht! In Python nennt man neue Befehle auch *Funktionen*. Wenn du die Funktion *quadrat()* verwendest, sagt man auch, die Funktion werde "aufgerufen".

Wir gewöhnen uns daran, die Funktionsdefinitionen im Programmkopf anzuordnen, da diese vor ihrem Aufruf definiert sein müssen.

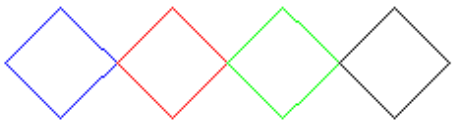


AUFGABEN

1. Definiere einen Befehl *hexagon()*, mit dem du ein Sechseck zeichnen kann. Verwende diesen Befehl, um die nebenstehende Figur zu zeichnen.



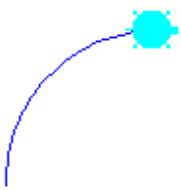
2a. Definiere einen Befehl für ein Quadrat, das auf der Spitze steht und zeichne damit die nebenstehende Figur.



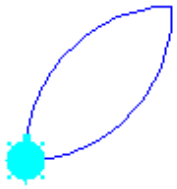
2b*. Du kannst gefüllte Quadrate zeichnen, indem du die Befehle *startPath()* und *fillPath()* verwendest.

3a. Du erlebst in dieser Aufgabe, wie du unter Verwendung von Funktionen eine Aufgabe schrittweise lösen kannst [**mehr...**].

Definiere eine Funktion *arc()*, mit dem die Turtle einen Bogen zeichnet und sich dabei insgesamt um 90 Grad nach rechts dreht. Mit *speed(-1)* kannst du die Turtlegeschwindigkeit auf den maximalen Wert setzen.

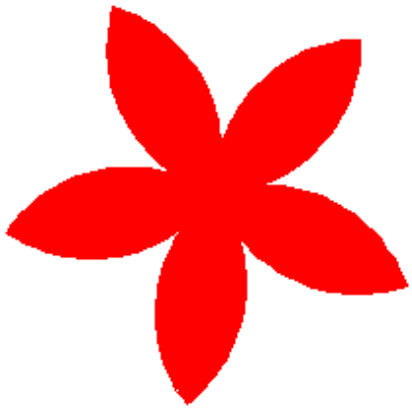


3b. Ergänze das Programm mit der Funktion *petal(),* welche zwei Bogen zeichnet. Die Turtle sollte am Ende aber wieder in Startrichtung stehen.

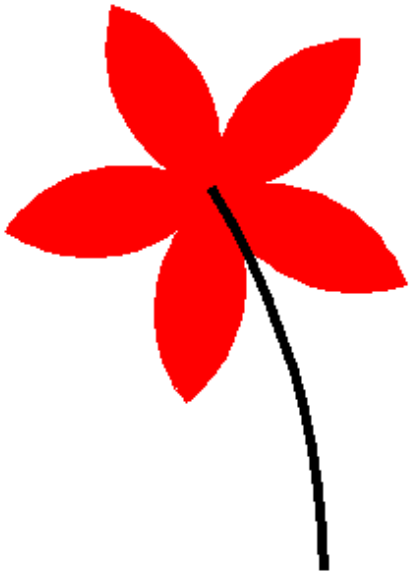


3c. Ergänze das Programm so, dass *petal()* ein rot gefülltes Blatt (ohne sichtbare Umrandungslinie) zeichnet.

3d. Erweitere das Programm mit der Funktion *flower()*, dass eine 5-blättrige Blume gezeichnet wird. Damit die Blume noch schneller entsteht, kannst mit *hideTurtle()* die Turtle bereits am Anfang unsichtbar machen.



3e*. Ergänze die Blume mit einem Stiel.



2.5 PARAMETER

EINFÜHRUNG

Beim Befehl *forward()* gibst du in Klammern an, um welche Strecke die Turtle vorwärts gehen soll. Dieser Wert in den Klammern gibt an, wie weit vorwärts gegangen wird. Er präzisiert den Befehl und heisst ein Parameter: Hier ist es eine Zahl, die bei jeder Verwendung von *forward()* anders sein kann. Im vorhergehenden Kapitel hast du einen eigenen Befehl *square()* definiert. Im Unterschied zu *forward()* ist die Seitenlänge dieses Quadrats aber immer 100 Pixel. Dabei wäre es doch in vielen Fällen praktisch, die Seitenlänge des Quadrats anpassen zu können. Wie geht das?

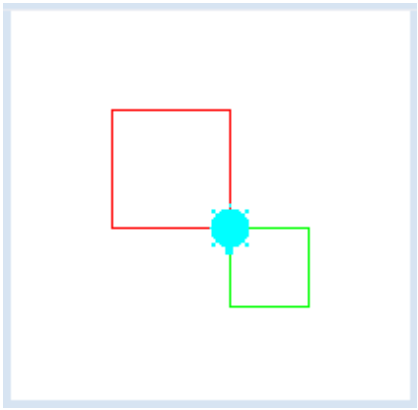
PROGRAMMIERKONZEPTE: *Parameter, Parameterübergabe*

BEFEHLE MIT PARAMETER

Auch in diesem Programm definieren wir ein Quadrat. An Stelle der leeren Parameterklammer bei der Definition der Funktion *square(),* setzen wir den Parameternamen *sidelength* ein und verwenden diesen beim Aufruf von **forward(sidelength)**.

Du kannst dadurch *square* mehrmals verwenden und bei jeder Verwendung eine Zahl für *seite* angeben.

Mit **square(80)** zeichnet die Turtle ein Quadrat mit der Seitenlänge von 80 Pixeln, mit **square(50)** eines mit der Seitenlänge von 50 Pixeln.



```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        left(90)

makeTurtle()
setPenColor("red")
square(80)
left(180)
setPenColor("green")
square(50)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Parameter sind Platzhalter für Werte, die jedes Mal anders sein können. Du gibst den Parameter bei der Definition eines Befehls hinter den Befehlsnamen in einem Klammerpaar an.

```
def befehlsname(parameter):
    Anweisungen, die
    parameter verwenden
```

Der Parametername ist frei wählbar, sollte aber seine Bedeutung wiederspiegeln. Bei der

Verwendung des Befehls gibst du wieder in Klammern den Wert an, den der Parameter haben soll.

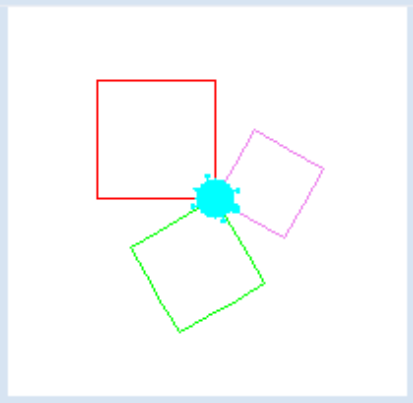
```
befehlsname(123)
```

Hier wird der Parameter im ganzen Befehl durch 123 ersetzt.

MEHRERE PARAMETER

Befehle können mehrere Parameter besitzen. Beim Quadrat kannst du zum Beispiel mit **def square(sidelength, color)** als Parameter seite und farbe wählen.

Du kannst dann *quadrat* viel flexibler verwenden.Mit **square(100, "red")** zeichnet die Turtle ein rotes Quadrat mit der Seitenlänge von 100 Pixeln, mit **square(80, "green")** ein grünes mit der Seitenlänge von 80 Pixeln.



```
from gturtle import *

def square(sidelength, color):
    setPenColor(color)
    repeat 4:
        forward(sidelength)
        left(90)

makeTurtle()
square(100, "red")
left(120)
square(80, "green")
left(120)
square(60, "violet")
```

MEMO

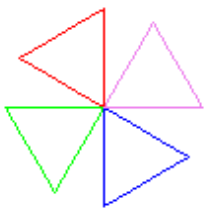
Befehle können mehrere Parameter besitzen. Diese werden in der Parameterklammer getrennt mit Komma eingegeben.

```
def befehlsname(parameter1, parameter2....):
   Anweisungen, die parameter1
   und parameter2 verwenden
```

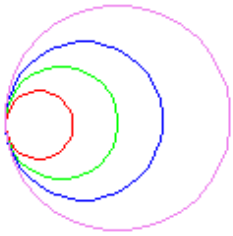
Die Reihenfolge der Parameter in der Parameterklammer bei der Definition des Befehls muss mit der Reihenfolge der Werte beim Aufruf des Befehls übereinstimmen.

AUFGABEN

1. Definiere einen Befehl *triangle(color)*, mit welchem die Turtle farbige Dreiecke zeichnen kann. Zeichne 4 Dreiecke in den Farben red, green, blue und violet



2. Definiere einen Befehl *colorCircle(radius, color)*, mit welchem die Turtle einen farbigen Kreis zeichnet. Du kannst dabei den Befehl *rightArc(radius, angle)* verwenden. Zeichne die nebenstehende Figur.



3. Das folgende Programm zeichnet leider 3 gleich grosse Fünfecke, aber nicht wie gewünscht verschieden grosse. Warum nicht? Korrigiere es.

```
from gturtle import *

def pentagon(sidelength, color):
   setPenColor(color)
   repeat 5:
      forward(90)
      left(72)

makeTurtle()
pentagon(100, "red")
left(120)
pentagon(80, "green")
left(120)
pentagon(60, "violet")
```

4. Du sagst der Turtle mit dem Befehl segment(), sich um eine bestimmte Strecke s vorwärts zu bewegen und sich um einen bestimmten Winkel w nach rechts zu drehen:

```
def segment(s, w):
   forward(s)
   right(w)
```

Schreibe ein Programm, das diesen Befehl 92 mal mit s = 300 und w = 151 ausführt. Mit *setPos(x, y)* kannst du die Turtle zu Beginn geeignet im Fenster positionieren.

5*. Die Turtle soll zwei, drei oder vier segment-Bewegungen ausführen. Schau dir die schönen Grafiken in folgenden Fällen an:

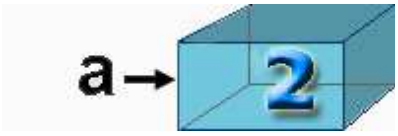
Anzahl Segmente	Werte	Anzahl Wiederholungen
2	forward(77) right(140.86) forward(310) right(112)	37
3	forward(15.4) right(140.86) forward(62) right(112) forwad(57.2) right(130)	46
4	forward(31) right(141) forward(112) right(87.19) forward(115.2) right(130) forward(186) right(121.43)	68

2.6 VARIABLEN

EINFÜHRUNG

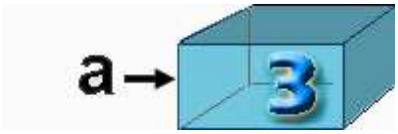
Im vorhergehenden Kapitel hast du Quadrate gezeichnet, deren Seitenlänge im Programm fest eingebaut waren. Manchmal möchtest du aber die Seitenlänge mit einem Eingabedialog einlesen. Dazu muss das Programm die eingegebene Zahl als **Variable** speichern. Du kannst eine Variable als einen Behälter (Container) auffassen, auf dessen Inhalt du mit einem Namen zugreifst. Kurz gesagt, hat eine Variable **einen Namen und einen Wert**. Den Namen der Variablen darfst du frei wählen. Nicht erlaubt sind Schüsselwörter und Namen mit Sonderzeichen. Zudem darf der Name nicht mit einer Zahl beginnen.

Mit der Schreibweise a = 2 erstellst du den Behälter, auf den du mit dem Namen a zugreifst und legst die Zahl 2 hinein. In Zukunft sagen wir, dass du damit eine Variable a **definierst** und ihr einen Wert **zuweist**.



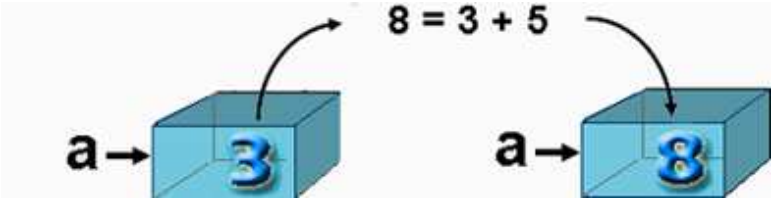
a = 2 : Variablendefinition (Zuweisung)

Du kannst in den Behälter nur **ein einziges** Objekt legen. Wenn du später unter dem Namen a die Zahl 3 speichern willst, so schreibst du a = 3 [**mehr...**].



a = 3 : neue Zuweisung

Was geschieht, wenn du nun **a = a + 5** schreibst? Du nimmst die Zahl, die sich gegenwärtig im Behälter befindet, auf den du mit a zugreifst, also die Zahl 3 und addierst dazu 5. Das Resultat 8 speicherst du wieder unter dem Namen a.

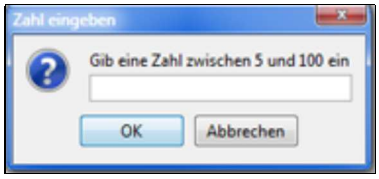


Das Gleichheitszeichen hat also in Computerprogrammen nicht dieselbe Bedeutung wie in der Mathematik. Es ist keine Gleichung, sondern eine Variablendefinition oder eine Zuweisung.

PROGRAMMIERKONZEPTE: *Variablendefinition, Zuweisung*

VARIABLENWERT EINLESEN UND VERÄNDERN

Im Programm kannst du mit Hilfe einer Dialogbox der **Variablen x** einen Wert zwischen 10 und 100 zuweisen. Diesen Wert **veränderst** du nachfolgend in der Wiederholstruktur und zeichnest dadurch eine Spirale.




```
from gturtle import *

makeTurtle()

x = inputInt("Enter a number between 5 and 100")
repeat 10:
    forward(x)
    left(120)
```

```
from gturtle import *

makeTurtle()

x = inputInt("Enter a number between 5 and 100")
repeat 10:
    forward(x)
    left(120)
    x = x + 20
```

MEMO

Mit *Variablen* kannst du Werte speichern, die du im Laufe des Programms lesen und verändern kannst. Jede Variable hat einen Namen und einen Wert. Mit dem Gleichheitszeichen definierst du eine Variable und weist ihr einen Wert zu [**mehr...**].

UNTERSCHEIDE ZWISCHEN VARIABLEN UND PARAMETER

Du solltest zwischen einer Variablen und einem Parameter unterscheiden. Parameter sind nur innerhalb einer Funktion gültig und transportieren Daten in eine Funktion, während Variablen überall möglich sind. Beim Aufruf erhält der Parameter einen Wert und kann im Innern der Funktion wie eine Variable verwendet werden. Um den Unterschied klar zu machen, verwendest du in deinem Programm in der Funktion *square()* den **Parameter sidelength**. Mit einem **Eingabedialog** liest du eine Zahl ein und speichert sie in der *Variablen* s. Beim Aufruf von **square()** übergibst du dem Parameter Seite den Variablenwert von s.


```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        right(90)

makeTurtle()
s = inputInt("Enter the side length")
square(s)
```

MEMO

Du musst zwischen der Variable *s* und dem Parameter *seite* unterscheiden. Parameter sind in der Funktionsdefinition Platzhalter und können beim Aufruf wie Variablen aufgefasst werden, die nur im Inneren der Funktion bekannt sind. Ruft man die Funktion mit einer Variablen auf, so wird der Variablenwert in der Funktion benutzt. *square(sidelength)* zeichnet somit ein Quadrat mit der Seitenlänge *sidelength*.

GLEICHE NAMEN FÜR VERSCHIEDENE DINGE

Wie du weisst, sollen Parameter- und Variablennamen ausdrücken, auf was sie sich beziehen. Sie sind aber eigentlich frei wählbar. Deshalb ist es üblich, bei gleichen Bezügen den gleichen Namen für **Parameter** und **Variablen** zu wählen.

Es ergeben sich dadurch im Programm keine Namenskonflikte, du musst aber zum Verständnis des Programms den Unterschied im Auge behalten.

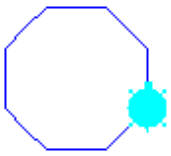
```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        right(90)

makeTurtle()
sidelength = inputInt("Enter the side length")
square(sidelength)
```

MEMO

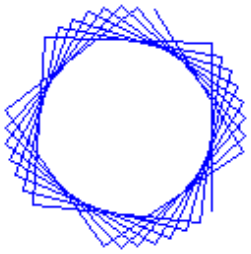
Auch wenn du für gewisse Parameter und Variablen den gleichen Namen verwendest, solltest du **Parameter** und **Variablen** begrifflich auseinander halten.

AUFGABEN

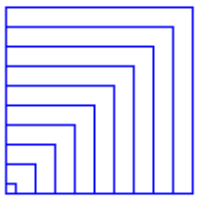
1. Nach Eingabe der Anzahl Ecken in einer Dialogbox soll die Turtle ein regelmässiges n-Eck zeichnen. Beispielsweise wird nach der Eingabe 8 ein 8-Eck gezeichnet. Den passenden Drehwinkel soll das Programm berechnen. Spiele dazu Turtle und überlege dir, wie weit du dich drehen musst, um die nächste Seite zu zeichnen. Erinnere dich an das Zeichnen eines gleichseitigen Dreiecks.



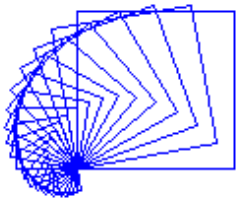
2. Nach der Eingabe eines Winkels in einer Dialogbox zeichnet die Turtle 30 Strecken der Länge 100, wobei sie nach jeder Strecke um den gegebenen Winkel nach links dreht. Experimentiere mit verschiedenen Winkeln und zeichne schöne Figuren. Mit *hideTurtle()* kannst du das Zeichnen beschleunigen.



3. Die Turtle soll 10 Quadrate zeichnen. Definiere zuerst einen Befehl *quadrat* mit dem Parameter *seite*. Die Seitenlänge des ersten Quadrats ist 8. Bei jedem nächsten Quadrat ist die Seitenlänge um 10 grösser.



4. Du kannst in einer Dialogbox die Seitenlänge des grössten Quadrats eingeben. Die Turtle zeichnet dann 20 Quadrate. Nach jedem Quadrat wird die Seitenlänge um den **Faktor** 0.9 kleiner und die Turtle dreht um den Winkel 10° nach links.



2.7 SELEKTION

EINFÜHRUNG

Was du im täglichen Leben unternimmst, hängt oft von gewissen Bedingungen ab. So entscheidest du dich je nach Wetter, wie du in die Schule fährst. Du sagst: "*Falls es regnet, fahre ich mit dem Tram, sonst mit dem Fahrrad*". Auch der Ablauf eines Programms kann von Bedingungen abhängig sein. Solche Programmverzweigungen auf Grund von bestimmten Bedingungen gehören zu den Grundstrukturen jeder Programmiersprache. Die Anweisungen nach *if* werden nur dann ausgeführt, wenn die Bedingung wahr ist, sonst werden die Anweisungen nach *else* ausgeführt.

PROGRAMMIERKONZEPTE: *Bedingung, Programmverzweigung, Selektion, If-Else-Struktur*

EINGABE ÜBERPRÜFEN

Nach der **Eingabe der Seitenlänge** in einer Dialogbox soll das Quadrat nur dann gezeichnet werden, wenn es im Fenster vollständig Platz hat.

Dazu prüfen wir den Wert von s. **Wenn s kleiner als 300 ist**, wird ein Quadrat mit der Seitenlänge s gezeichnet, **sonst** erscheint im unteren Teil des Tigerjython-Fensters eine Meldung. Diese Prüfung erfolgt in der Programmiersprache mit der *if*-Anweisung


```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        right(90)

makeTurtle()
s = inputInt("Enter the side length")
if s < 300:
    square(s)
else:
    print "The side length is too big"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Anweisungen nach ***if*** werden nur dann ausgeführt, wenn die Bedingung wahr ist, sonst werden die Anweisungen nach ***else*** ausgeführt. Manchmal kommt es vor, das nur etwas gemacht wird, wenn die Bedingung erfüllt ist; sonst aber nichts passiert. Für diese Fälle kannst du den *else*-Block weglassen. Achte auf die Doppelpunkte nach der *if*-Bedingung und nach *else*, sowie auf die korrekte Einrückung der beiden Programmblöcke.

MEHRFACHE AUSWAHL

Wir wollen farbige Quadrate zeichnen. In einer Dialogbox kannst du die gewünschte Farbe mit einer **Zahl eingegeben**. In einer **if-Struktur** wird diese Zahl überprüft und die entsprechende Füllfarbe gesetzt. Wir testen zuerst auf den Wert 1, dann mit **elif** auf 2 und dann auf **3**. Falls eine andere Zahl eingegeben wird, setzen wir mit **else** die Farbe auf schwarz.

Mit dem Befehl **fill(10, 10)** wird die geschlossene Fläche um den gegebene Punkt mit der Füllfarbe gefüllt. Da sich die Turtle nach dem Zeichnen des Quadrats wieder in der Fenstermitte (0, 0) befindet, wählen wir mit (10, 10) einen Punkt, der sicher im Inneren des Quadrats liegt.

```
from gturtle import *

def square():
    repeat 4:
        forward(100)
        right(90)

makeTurtle()
n = inputInt("Enter a number: 1:red 2:green 3:yellow")
if n == 1:
    setFillColor("red")
elif n == 2:
    setFillColor("green")
elif n == 3:
    setFillColor("yellow")
else:
    setFillColor("black")

square()
fill(10, 10)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Es können mehrere Bedingungen nacheinander überprüft werden. Falls die Bedingung bei *if* nicht erfüllt ist, wird die Bedingung bei *elif* überprüft. *elif* ist eine Abkürzung von *else if*. Falls keine der *elif*-Bedingungen erfüllt ist werden die Anweisungen nach *else* ausgeführt.

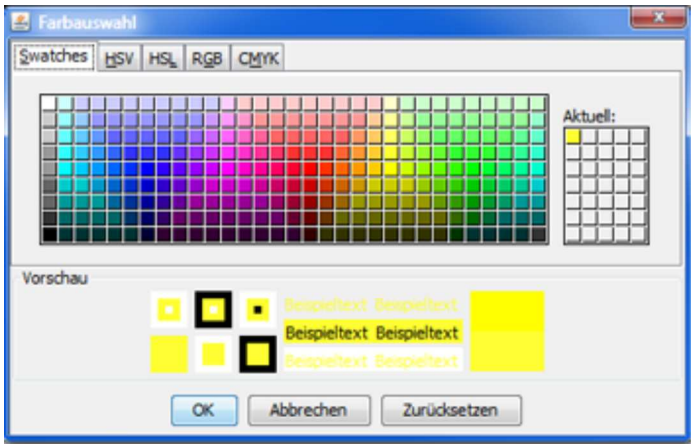
Es fällt dir sicher auf, dass das Prüfen auf Gleichheit in Python mit einem doppelten Gleichheitszeichen erfolgt. Dies ist etwas gewöhnungsbedürftig, aber nötig, da das einfache Gleichheitszeichen für Zuweisungen verwendet wird.

Beachte die Notation für Vergleichsoperatoren: >, >= , < , <= , == , != .

Mit dem Befehl *fill(x, y)* können geschlossene Figuren mit der Füllfarbe gefüllt werden. Der Punkt (x, y) muss sich aber im Inneren der Figur befinden.

FARBAUSWAHL, BOOLESCHE VARIABLEN

Das nachträgliche Füllen mit *fill()* setzt voraus, dass das Innere der Figur nicht bereits mit einer anderen Figur belegt ist. Du kennst von früher die **startPath()**/**fillPath()**-Kombination, mit der du auch neue Figuren, die über schon vorhandenen Figuren liegen, korrekt füllen kannst. Du verwendest im Programm durch Aufruf von **askColor()** ein elegantes Dialogfeld, mit welchem du die Farbe des Sterns auswählst.



Den Stern zeichnest du mit der Funktion **star()**, die neben der Grösse des Sterns auch einen Parameter hat, dessen Wert *True* oder *False* sein kann, und der bestimmt, ob der Stern gefüllt werden soll oder nicht.

```
from gturtle import *

makeTurtle()

def star(size, filled):
    if filled:
        startPath()
    repeat 9:
        forward(size)
        left(175)
        forward(size)
        left(225)
    if filled:
        fillPath()

clear("black")
repeat 5:
    color = askColor("Color selection", "yellow")
    if color == None:
        break
    setPenColor(color)
    setFillColor(color)
    setRandomPos(400, 400)
    back(100)
    star(100, True)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Funktion **askColor()** hat einen Parameter für den Text in der Titelzeile und für die Farbe, die als Standardwert ausgewählt ist. Die Funktion gibt beim Klicken des OK-Buttons die ausgewählte Farbe und beim Klicken des Abbrechen-Buttons den speziellen Wert *None* zurück. Du kannst mit einer if-Anweisung auf diesen Wert testen und die repeat-Schleife mit *break* abbrechen.

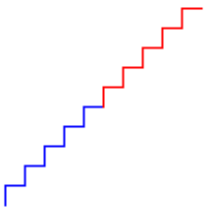
Eine Variable oder einen Parameter, der die Werte *True* oder *False* annehmen kann, nennt man eine boolesche Variable bzw. einen booleschen Parameter [**mehr...**]. Du kannst direkt mit **if filled:** auf den Wert testen, es ist also nicht nötig (und wenig elegant) *if filled == True:* zu schreiben.

AUFGABEN

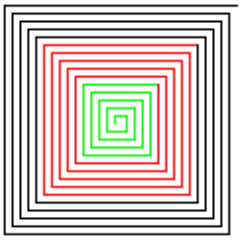
1. In einer Dialogbox fragst du den Benutzer, wie gross die Seitenlänge eines Quadrats sein soll. Falls sie kleiner ist als 50, wird ein rotes Quadrat mit dieser Seitenlänge gezeichnet, sonst ein grünes.

2. Die Turtle soll mir *repeat 10* eine Treppe mit 10 Stufen zeichnen, wobei die ersten 5 Stufen blau und die restlichen rot sind (Abbildung a).

(a)



(b)



Die Turtle soll eine Spirale zeichnen und dabei zuerst die grüne, dann die roten und am Schluss die schwarze Farbe verwenden (Abbildung b).

2.8 WHILE-SCHLEIFEN

EINFÜHRUNG

Du hast bereits den Befehl *repeat* kennengelernt, mit dem du einen Programmblock mehrmals wiederholen konnst. *repeat* kannst du so allerdings nur im TigerJython verwenden. Die *while*-Struktur ist aber überall einsetzbar.

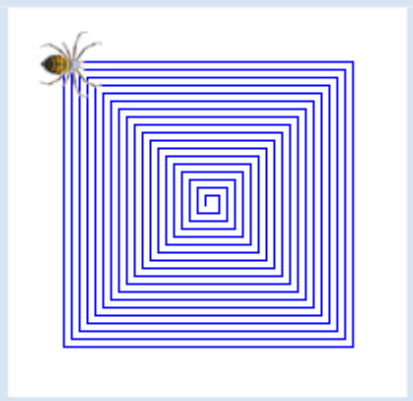
Die *while-Schleife* wird mit dem Schlüsselwort *while* eingeleitet, gefolgt von einer Schleifenbedingung. Die Anweisungen im Schleifenblock werden wiederholt, so lange die Bedingung erfüllt ist. Nach Ende der Wiederholungen wird das Programm mit der nächsten Anweisung nach dem Schleifenblock fortgesetzt.

PROGRAMMIERKONZEPTE: *Iteration, While-Struktur, Verknüpfte Bedingungen, Schleifenabbruch*

SPINNENNETZ

Mit Hilfe einer while-Schleife soll die Turtle eine reckteckige Spirale zeichnen. Dazu verwenden wir eine Variable a, die den **Startwert 5** erhält und bei jedem Schleifendurchlauf **um 2 vergrössert** wird. Solange die **Bedingung a < 200** wahr ist, werden die Anweisungen im Schleifenblock ausgeführt.

Zur Erhöhung des Spassfaktors nimmst du anstelle der Turtle eine Spinne.



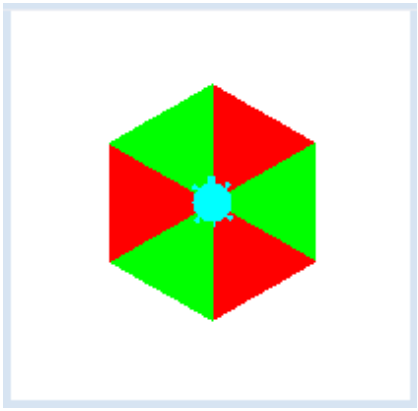
```
from gturtle import *

makeTurtle("sprites/spider.png")

a = 5
while a < 200:
    forward(a)
    right(90)
    a = a + 2
```

MEMO

Eine while-Schleife dient zur Wiederholung eines Programmblocks. Die Bedingung muss wahr sein, damit der Programmblock ausgeführt wird. Darum spricht man auch von einer*"Ausführungsbedingung"* oder *"Laufbedingung*". Fehlt im Schleifenblock die Wertänderung, bleibt die Laufbedingung immer wahr und das Programm bleibt endlos in der Schleife "hängen".

In unserer Lernumgebung kannst du ein hängendes Programm mit dem Stoppknopf oder mit Schliessen des Turtlefensters abbrechen. Im allgemeinen sind endlose Schleifen ohne Abbruch-Möglichkeit aber gefährlich, da im Extremfall ein Neustart des Computers nötig ist.

BEDINGUNGEN MIT OR VERKNÜPFEN



Die Turtle soll mit einer **while-Schleife** die nebenstehende Figur zeichnen. Wie du siehst, zeichnet sie abwechslungsweise rote und grüne Dreiecke.

Für den Farbwechsel kannst du folgenden Trick verwenden: Du testest die Schleifenvariable darauf, ob sie 0, 2 oder 4 ist und wählst die Stiftfarbe rot.

Mit dem Befehl **fillToPoint(0. 0)** kannst du eine Figur während des Zeichnens füllen. Dabei wird am Punkt (0, 0) sozusagen ein Gummiband befestigt, dessen anderes Ende die Turtle mitzieht. Dabei werden alle Punkte, über die sich das Gummiband bewegt, fortlaufend eingefärbt.

```
from gturtle import *

def triangle():
    repeat 3:
        forward(100)
        right(120)

makeTurtle()
i = 0
while i < 6:
    if i == 0 or i == 2 or i == 4:
         setPenColor("red")
    else:
         setPenColor("green")

    fillToPoint(0, 0)
    triangle()
    right(60)
    i = i + 1
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Bei Verwendung von mehreren Programmstrukturen musst du auf die korrekte Einrückung der einzelnen Schleifenblöcke achten.

Wie du siehst, kannst du zwei oder mehr Bedingungen mit *or* verknüpfen. Eine so verknüpfte Bedingung ist dann wahr, wenn die eine oder auch die andere Bedingung erfüllt ist (also auch, wenn beide Bedingungen erfüllt sind).

Mit dem Befehl *fillToPoint(x, y)* kannst du Figuren mit der Stiftfarbe während des Zeichnens füllen, im Gegensatz zum Befehl *fill()*, mit dem du bereits gezeichnete geschlossene Figuren füllen kannst.

BEDINGUNGEN MIT AND VERKNÜPFEN

Die Turtle soll mit einer while-Schleife 10 zusammengebaute Häuser zeichnen. Die Häuser sind von 1 bis 10 nummeriert. Die Häuser mit den Nummern 4 bis 7 sind gross, die übrigen klein.

In der *while*-Schleife wird die Hausnummer *nr* benutzt, um die Grösse der Häuser zu bestimmen. Wenn *nr* grösser als 3 und kleiner als 8 ist, sind die Häuser gross.

Zum Färben verwenden wir den Befehl **fillToHorizontal(0)**. Dadurch wird die Fläche zwischen der gezeichneten Figur und der waagrechten Linie y = 0 fortlaufend gefüllt.

```
from gturtle import *

makeTurtle()
setPos(-200, 30)
right(30)
fillToHorizontal(0)
setPenColor("sienna")

nr = 1
while nr <= 10:
    if nr > 3 and nr < 8:
        forward(60)
        right(120)
        forward(60)
        left(120)
    else:
        forward(30)
        right(120)
        forward(30)
        left(120)

    nr += 1
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

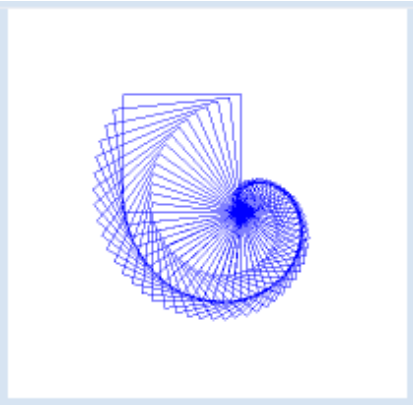
Zwei Bedingungen kannst du mit **and** verknüpfen. Eine solche Verknüpfung ist nur dann wahr, wenn beide Bedingungen erfüllt sind.

Mit dem Befehl **fillToHorizontal(y)** kannst du Figuren mit der Stiftfarbe während des Zeichnens füllen. Gefüllt wird die Fläche zwischen der gezeichneten Figur und der horizontalen Linie durch y.

nr += 1 kannst du lesen als: *nr* wird erhöht um 1. Es ist eine abgekürzte Scheibweise für die Zuweisung *nr = nr + 1*.

SCHLEIFEN MIT BREAK VERLASSEN

Eine Schleife, deren Bedingung immer wahr ist, wird endlos durchlaufen. Du kannst allerdings das Verlassen einer Schleife zu irgendeinem Zeitpunkt mit dem Schlüsselwort **break** erzwingen. Dein Programm zeichnet gedrehte Quadrate mit zunehmender Seitenlänge, bis die Seitenlänge 120 beträgt.



```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        left(90)

makeTurtle()
hideTurtle()

i = 0
while 1 == 1:
    if i > 120:
        break
    square(i)
    right(6)
    i += 2
print "i =", i
```

MEMO

Statt **while 1 == 1:** kannst du eleganter *while True*: einsetzen, da *True* immer wahr ist. (Im Gegensatz zum Wert *False*, der immer falsch ist.) Die Schleife wird in Zweierschritten durchlaufen. Statt i = i + 2 verwendest du die abgekürzte Schreibweise **i += 2** (i wird erhöht um 2). Mit dem **print**-Befehl kannst du etwas in die TigerJython-Console im unteren Bereich des Editors schreiben. Für Text verwendest du Anführungszeichen und trennst Zahlen mit einem Komma ab. Es wird automatisch ein Leerzeichen zwischen dem Text und der Zahl eingefügt. Verstehst du, warum i = 122 ausgegeben wird? Selten gebraucht wird das Schlüsselwort ***continue***, das bewirkt, dass der restliche Teil des Schleifenkörpers übersprungen wird, die Schleife danach aber weitergeführt wird.

EINGABE-VALIDIERUNG

Gibst du dem Benutzer die Möglichkeit mit einem Eingabedialog einen Wert in einem bestimmten Bereich einzugeben, so kannst du dich nicht darauf verlassen, dass er sich an deine Vorgaben hält. Ein "robustes" Programm überprüft die Eingabe und fängt eine fehlerhafte Eingabe mit einer Rückmeldung ab. Diese Eingabeprüfung kannst du am besten mit einer while-Schleife durchführen, die solange durchlaufen wird, bis sich der Eingabewert an die Vorgaben hält.

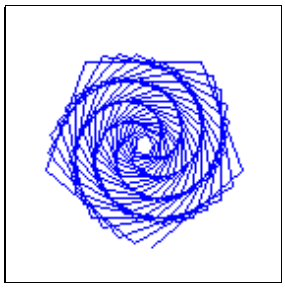
```
from gturtle import *

makeTurtle()

n = 0
while n < 1 or n > 3:
    n = inputInt("Enter 1, 2 or 3")
if n == 1:
    setPenColor("red")
elif n == 2:
    setPenColor("green")
else:
    setPenColor("yellow")
dot(200)
```

In deinem Programm wählt der Benutzer mit den Zahlen 1, 2, oder 3 die Farben rot, grün oder gelb des gezeichneten Kreises aus.

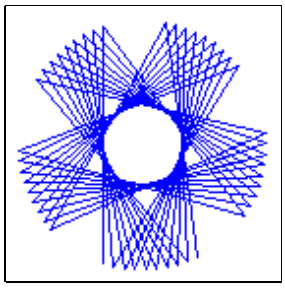
AUFGABEN

1. Die Turtle bewegt sich um eine Strecke 5 vorwärts, dreht sich um 70° nach rechts und vergrössert die Streckenlänge um 0.5. Diese Schritte wiederholt sie, solange die Streckenlänge kleiner als 150 ist.

   Versuche es auch mit dem Drehwinkel 89°!

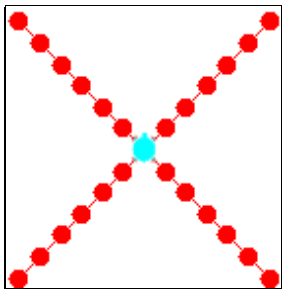


2. Wie du sicher gemerkt hast, beträgt die Drehung an der Spitze bei einem 5-er Stern 144°. Verändere diesen Drehwinkel ganz wenig, z. B. 143° und vergrössere die Anzahl der Wiederholungen. Dann erhältst du eine neue Figur.



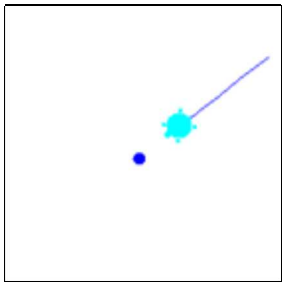
3. Die Turtle zeichnet ein Diagonalmuster mit gefüllten roten Kreisen, die die Distanz 40 haben. Der Abstand der Kreise vom der Mitte ist kleiner als 400.

   Verwende den Befehl *dot(25)*, um die Kreise zu zeichnen.



4. Die Turtle befindet sich an der Position (250, 200). Sie will sich mit Schritten der Länge 10 auf einer Geraden an die Homeposition laufen. Sie bewegt sich dabei so lange, bis der Abstand zur Home-Position kleiner als 1 ist.

   Verwende die Befehle *towards()* und *heading(degrees)* aus der Dokumentation.



5*. Du möchtest die Turtle präziser bei Home positionieren und wählst als Abstandskriterium einen 10-mal kleineren Wert. Es kann sein, dass die Turtle jetzt nicht mehr anhält. Begründe das Verhalten.

2.9 REKURSIONEN

EINFÜHRUNG

Aus deiner frühen Kindheit kennst du möglicherweise die etwas unheimliche Geschichte vom Mann mit dem hohlen Zahn:

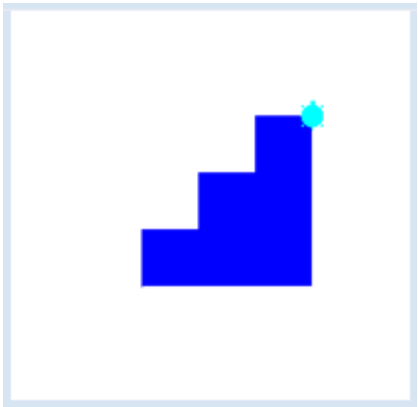
Äs isch ämal ä Ma gsi	Es war einmal ein Mann,
dä het ä hohle Zahn gha	der hatte einen hohlen Zahn,
u i däm hohle Zahn isch äs Schachteli gsi	in diesem hohlen Zahn befand sich eine Schachtel
u i däm Schachteli isch äs Briefli gsi	in der Schachtel war ein Brief
u i däm Briefli isch gstande:	in diesem Brief stand:
Äs isch ämal ä Ma gsi...	Es war einmal ein Mann...

Strukturen, die in ihrer Definition wieder sich selbst verwenden, nennt man ***rekursiv***. Du kennst sicher die russische Matrjoschka: Eine Matrjoschka enthält in sich eine (etwas kleinere) Matrjoschka, diese enthält in sich eine Matrjoschka, diese enthält...

PROGRAMMIERKONZEPTE: *Rekursion, Rekursionsverankerung, Indirekte Rekursion*

REKURSIVER TREPPENBAU

Man stellt dir die Aufgabe eine Treppe mit 3 Stufen zu bauen. Statt zu sagen, dass du dreimal eine Stufe legen musst, könntest du aber auch sagen, dass eine Treppe aus 3 Stufen aus einer Stufe und einer Treppe aus 2 Stufen besteht, dann wiederum dass eine Treppe aus 2 Stufen aus einer Stufe und einer Treppe aus 1 Stufe besteht und dass eine Treppe aus 1 Stufe aus einer Stufe und einer Treppe aus 0 Stufen besteht, eine Treppe aus 0 Stufen aber nichts ist.



Diese Bauanleitung nennt man rekursiv, da in der Definition der Treppe mit 3 oder allgemein n Stufen die Treppe mit 2 oder allgemein n-1 Stufen verwendet wird. Als Programmcode:

```
def treppe(n):
    stufe()
    treppe(n-1)
```

Du wirst gleich die Turtle damit beauftragen, mit stufe() einen Treppenblock zu zeichnen, um dann mit dem Aufruf treppe(3) eine 3 stufige Treppe zu bauen. Aber warte noch, es fehlt doch die Angabe, dass beim Bau einer Treppe aus 0 Stufen gar nichts geschehen soll. Dies kannst du aber leicht mit einer if-Bedingung einbauen:

```
if n == 0:
   return
```

Die Anweisung *return* besagt, dass sie aufhören und zur vorherigen Arbeit zurückkehren kann. Dein Programm sieht nun so aus:

```
from gturtle import *

def stairs(n):
    if  n == 0:
        return
    step()
    stairs(n - 1)

def step():
    forward(50)
    right(90)
    forward(50)
    left(90)

makeTurtle()
fillToHorizontal(0)
stairs(3)
```

MEMO

Unter Rekursionen versteht man ein fundamentales Lösungsverfahren in der Mathematik und Informatik, bei dem ein Problem derart gelöst wird, dass man es auf das gleiche, aber etwas vereinfachte Problem zurückführt. Wenn also f ein Funktion ist, die das Problem löst, so wird bei einer (direkten) Rekursion in der Definition von f wieder f verwendet. Auf den ersten Blick scheint es seltsam, dass man ein Problem derart lösen will, dass man die Lösung bereits voraussetzt. Dabei übersieht man aber einen wesentlichen Punkt: Es wird nicht genau dasselbe Problem zur Lösung verwendet, sondern eines, das **der Lösung näher liegt**. Dazu verwendet man einen meist ganzzahligen Ordnungsparameter n, den man f übergibt.

```
def f(n):
    ...
```

Bei der Wiederverwendung von f im Definitionsteil wird der Parameter verkleinert:

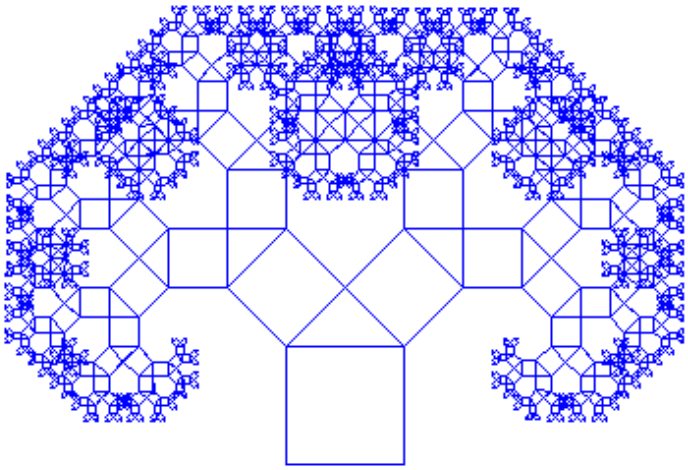
```
def f(n):
    ...
    f(n-1)
    ...
```

Eine so definierte Funktion würde sich aber **endlos** selbst aufrufen. Um dies zu verhindern, braucht man eine **Abbruchbedingung**, die man **Rekursionsverankerung** (oder Rekursionsbasis) nennt.

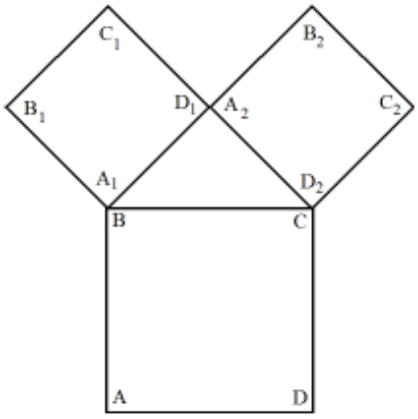
```
def f(n):
    if n == 0:
       return
    ...
    f(n - 1)
    ...
```

Mit dem Schüsselwort *return* wird die weitere Verarbeitung der Funktion abgebrochen. Man sagt auch, die Funktion kehre zurück.

DER PYTHAGORASBAUM



Mit rekursiven Algorithmen kannst du wunderbare Grafiken erzeugen. Du gehst von folgender Anleitung aus:



- Zeichne ausgehend von A ein Quadrat ABCD mit Basisseite AD
- Füge ein rechtwinkliges, gleichschenkliges BD_1C Dreieck an der Seite BC an
- Zeichne den Baum erneut ausgehend von den Quadraten A_1D_1 und A_2D_2 als Basisseiten

Es ist bekannt, dass die Umsetzung in ein rekursives Programm ungewohnt ist. Darum erhältst du hier eine ausführliche Anleitung, wie du vorgehen musst.

- Definiere einen Befehl *square(s),* mit dem die Turtle ein Quadrat mit der Seitenlänge *s* zeichnet und wieder in die Anfangsposition mit Anfangsblickrichtung zurückkehrt
- Definiere den Befehl *tree(s)*, welcher einen Baum ausgehend von einem Quadrat der Seitenlänge *s* zeichnet. In der Definition darfst du *tree()* wieder verwenden. Wichtig: **Nach dem Zeichnen des Baums ist die Turtle wieder in der Anfangsposition mit Anfangsblickrichtung**. Du überlegst schrittweise, als ob du die Turtle wärst (das neu Hinzugefügte ist grau unterlegt).

- Du zeichnest zuerst vom Punkt A aus ein Quadrat mit der Seitenlänge *s*:

```
def tree(s):
    square(s)
```

- Du fährst zur Ecke B des Quadrats, drehst 45 Grad nach links und betrachtest dies als Startpunkt eines neuen Baums mit verkleinertem Parameter s1. Es gilt nach dem Satz von Pythagoras:

$$s1 = \frac{s}{\sqrt{2}}$$

```
def tree(s):
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
    left(45)
    tree(s1)
```

- Da du ja voraussetzt, dass du nach dem Zeichnen des Baums wieder am Startpunkt mit der Startblickrichtung landest, befindest du dich wieder in B und schaust in Richtung B1. Du drehst dich um 90 Grad nach rechts und fährst die Strecke s1 vorwärts. Jetzt bist du im Punkt D1 und hast die Blickrichtung zu B2. Von hier aus zeichnest du den Baum erneut.

```
def tree(s):
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
    left(45)
    tree(s1)
    right(90)
    forward(s1)
    tree(s1)
```

- Jetzt musst du nur noch an den Anfangsort A mit der Anfangsblickrichtung zurückkehren. Dazu bewegst du dich um s1 rückwärts, drehst dich um 45 Grad nach links und fährst um s rückwärts.

```
def tree(s):
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
    left(45)
    tree(s1)
    right(90)
    forward(s1)
    tree(s1)
    back(s1)
    left(45)
    back(s)
```

```
from gturtle import *
import math

def tree(s):
    if s < 2:
        return
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
    left(45)
    tree(s1)
    right(90)
    forward(s1)
    tree(s1)
    back(s1)
    left(45)
    back(s)

def square(s):
    repeat 4:
        forward(s)
        right(90)

makeTurtle()
ht()
setPos(-50, -200)
tree(100)
```

MEMO

Bei vielen rekursiv definierten Figuren ist es wichtig, dass die Turtle wieder an ihren Anfangsort mit der Anfangsblickrichtung zurückkehrt.

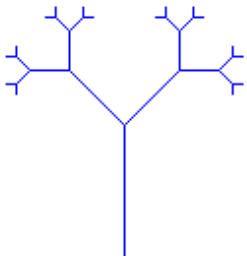
AUFGABEN

1. Wo liegt der wesentliche Unterschied der beiden Programme? Untersuche insbesondere Ort und Richtung der Turtle nach Programmende. Warum nennt man figA eine "*Last Line Recursion*" und figB eine "*First Line Recursion*"?

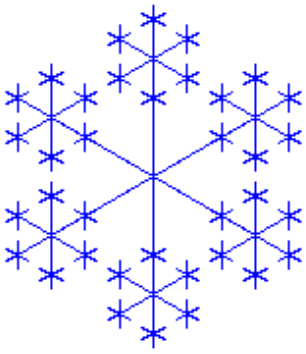
```
from gturtle import *

def  figA(s):
   if s > 200:
      return
   forward(s)
   right(90)
   figA(s + 10)

makeTurtle()
figA(100)
```

```
from gturtle import *

def  figB(s):
   if s > 200:
      return
   figB(s + 10)
   forward(s)
   right(90)

makeTurtle()
figB(100)
```

2. Ein bekannter Graf ist der vollständige binäre Baum. Er sieht für eine bestimmte Rekursionstiefe wie nebenstehend gezeigt aus. Schreibe eine rekursive Funktion tree(s), die den Baum mit der "Stammlänge" s zeichnet.

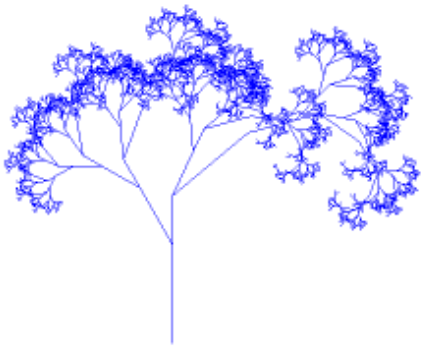


3. Zeichne die nebenstehende Sternfigur.
Definiere dazu die rekursive Funktion star(s), welche die Turtle einen Stern mit der "Dimension" s zeichnen lässt (s ist die Distanz vom Zentrum der Sternfigur zum Zentrum in der nächsten Generation). s wird von Generation zu Generation auf 1/3 reduziert. Rufe stern(180) auf und verankere die Rekursion, dass sie bei s < 20 abbricht. Wenn du hideTurtle() verwendest, wird die Turtle viel schneller zeichnen.



4*. Du wirst einen Baum zeichnen, der schon fast wie ein echter Baum aussieht. Definiere dazu die rekursive Funktion treeFractal(s), mit der "Stammlänge" s, die wie folgt aufgebaut ist:

- Verankere die Rekursion bei einer Stammlänge kleiner als 5.
- Speichere dir zuerst mit getX() und getY() die aktuelle x- und y-Koordinaten der Turtle, sowie mit heading() ihre Blickrichtung, damit du einfach zurückkehren kannst
- Jetzt fährst um s/3 nach vorne, drehst dich um 30 Grad nach links und zeichnest den Baum mit der Stammlänge 2*s/3
- Du drehst dich um 30 Grad nach rechts, fährst s/6 nach vorn und zeichnest den Baum mit der Stammlänge s/2
- Du drehst dich um weitere 25 Grad nach rechts, fährst um s/3 nach vorn, drehst 25° nach rechts und zeichnest den Baum noch einmal mit der Stammlänge s/2
- Du kehrst mit setPos() und heading() wieder in die Anfangslage mit der Anfangsposition zurück



2.10 EREIGNISSTEUERUNG

EINFÜHRUNG

Bisher bestand ein Programm aus einem einzigen Ablaufstrang, in dem eine Anweisung um die andere mit möglichen Verzweigungen und Wiederholungen ausgeführt wird. Klickst du eine Maustaste, so weisst du aber nicht, wo sich dein Programm eben gerade befindet. Um das Klicken im Programm zu erfassen, muss daher ein **neues Programmierkonzept** verwendet werden, die **Ereignissteuerung**. Hier das Prinzip:

Du definierst eine Funktion mit irgendeinem Namen, z.B. *onMouseHit()*, die im Programm nirgends explizit aufgerufen wird. Nun verlangst du vom Computer, dass **er** diese Funktion aufrufen soll, wenn die Maustaste geklickt wird. Du sagst also im Programm: *Wann immer die Maus geklickt wird, führe onMouseHit() aus*.

PROGRAMMIERKONZEPTE: *Ereignisgesteuertes Programm, Mausevent*

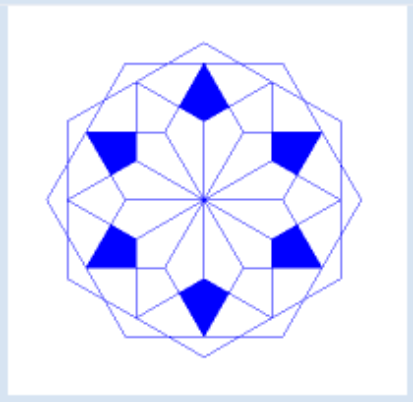
MAUSEVENTS

Das neue Konzept ist in Python sehr einfach umzusetzen. Im ersten ereignisgesteuerten Programm soll im Hauptteil die Turtle zuerst eine lustige Figur zeichnen. Nachher willst du diese noch verschönern, indem du bestimmte Bereiche mit einem Mausklick einfärbst.

Du schreibst dazu die Funktion **onMouseHit(x, y)**, die dir die x- und y-Koordinate des Mausklicks liefert, und führst darin mit **fill(x, y)** ein Floodfill (Füllen eines geschlossenen Bereichs) aus.

Das Wichtige dabei ist aber, das du dem **System mitteilst**, dass es die Funktion *onMouseHit()* aufrufen soll, wenn die Maustaste gedrückt wird. Dazu verwendest du beim Aufruf von *makeTurtle()* den Parameter mit dem festgelegten Namen **mouseHit** und übergibst ihm den Namen deiner Funktion.

Damit die Zeichnung rasch erstellt wird, kannst du die Turtle mit *hideTurtle()* verstecken.



```
from gturtle import *

def onMouseHit(x, y):
    fill(x, y)

makeTurtle(mouseHit = onMouseHit)
hideTurtle()
addStatusBar(30)
setStatusText("Click to fill a region!")

repeat 12:
    repeat 6:
        forward(80)
        right(60)
    left(30)
```

MEMO

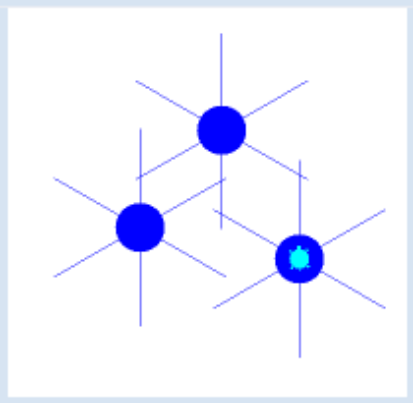
Programmtechnisch wird das Konzept der ereignisgesteuerten Programmierung bei unserer Turtle so umgesetzt, dass du eine Funktion schreibst, die beim Auftreten des Ereignisses aufgerufen werden soll. Du teilst dies dem System mit, indem du diesen Funktionsnamen als Parameter von *makeTurtle()* übergibst. Dabei verwendest du die Schreibweise *parameter_name = parameter_wert*.

Du kannst die Füllfarbe mit *setFillColor()* deinen Wünschen anpassen.

Wichtige Informationen für den Benutzer kannst du in einer Statuszeile ausschreiben, die mit **addStatusBar(n)** unten am Turtlefenster erscheint. Die Zahl *n* gibt die Höhe dieser Textzeile an (in Pixel).

ZEICHNEN PER MAUSKLICK

Die Turtle soll an der Stelle des Mausklicks einen Strahlenstern zeichnen. Du schreibst dazu die Funktion **onMouseHit(x, y)**, mit der du die Turtle anweist, wie sie den Stern zeichnen soll. Damit *onMouseHit()* beim Mausklick aufgerufen wird, übergibst du in *makeTurtle()* dem Parameternamen **mouseHit** diesen Funktionsnamen *onMouseHit*.



```
from gturtle import *

def onMouseHit(x, y):
    setPos(x, y)
    repeat 6:
        dot(40)
        forward(60)
        back(60)
        right(60)

makeTurtle(mouseHit = onMouseHit)
speed(-1)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

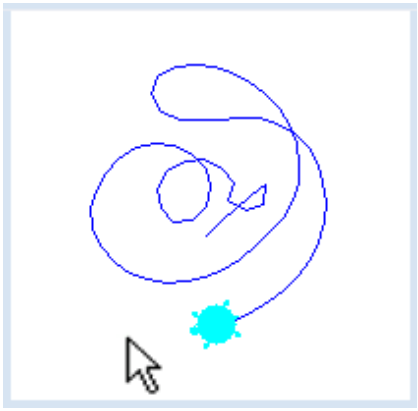
Das Programm hat einen Schönheitsfehler: Wenn du bereits wieder klickst, während die Turtle noch am Zeichnen eines Sterns ist, so zeichnet sie diesen Stern nicht fertig, sondern beginnt mit dem Zeichnen des neuen Sterns. Dabei führt sie aber die Befehle des alten Sterns auch noch weiter, wodurch der neue Stern falsch gezeichnet wird.

Dieses Falschverhalten ist offenbar darauf zurück zu führen, dass bei jedem Klick die Funktion *onMouseHit()* aufgerufen und ausgeführt wird, auch wenn die vorhergehende Ausführung noch nicht beendet ist. Um dies zu verhindern, verwendest du an Stelle des Parameters mit dem Namen *mouseHit* den Parameter mit dem Namen *mouseHitX*.

TURTLE VERFOLGT MAUS

Du möchtest, dass die Turtle ständig der Maus folgt. Dazu kannst du den Mausklick nicht gebrauchen, sondern musst die **Mausbewegung** als Ereignis betrachten. *makeTurtle()* kennt den Parameter *mouseMoved*, dem du eine Funktion übergeben kannst, die bei jeder Verschiebung der Maus aufgerufen wird.

Die Funktion **onMouseMoved(x, y)** erhält als Parameterwerte die aktuellen Cursorkoordinaten.



```
from gturtle import *

def onMouseMoved(x, y):
    setHeading(towards(x, y))
    forward(10)

makeTurtle(mouseMoved = onMouseMoved)
speed(-1)
```

MEMO

Neben *mouseHit* und *mouseHitX* stehen dir für das Erfassen von Mausevents in *makeTurtle()* weitere Parameter zur Verfügung.

mousePressed	Maustaste wird gedrückt
mouseReleased	Maustaste wird losgelassen
mouseClicked	Maustaste wird gedrückt und losgelassen
mouseDragged	Maus wird mit gedrückter Taste bewegt
mouseMoved	Maus wird bewegt
mouseEntered	Maus tritt in das Turtlefenster ein
mouseExited	Maus tritt aus dem Turtlefenster aus

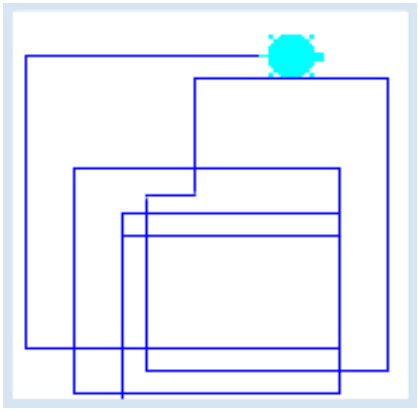
Du kannst auch mehrere Parameter gleichzeitig verwenden, also beispielsweise die zwei Funktionen *onMousePressed()* sowie *onMouseDragged()* angeben:

```
makeTurtle(mousePressed = onMousePressed, mouseDragged = onMouseDragged)
```

Mit *isLeftMouseButton()* bzw. *isRightMouseButton()* findest du heraus, welche Maustaste gedrückt wurde.

Es gibt bei diesen Events einen wichtigen Unterschied zu *mouseHit*: Die Bewegung der Turtle ist während der Ausführung der Funktion nicht sichtbar. Du solltest also entweder die Turtle mit *speed(-1)* auf hohe Geschwindigkeit setzen, mit *hideTurtle()* verstecken oder den Code für die Bewegung im Hauptteil des Programms durchführen.

TASTATUREVENTS



Auch das Drücken einer Taste auf der Tastatur löst einen Event aus.Um ihn "abzufangen", übergibst du wie vorhin mit Mausevents in *makeTurtle()* dem System den Namen der Funktion, hier *onKeyPressed()*, die es aufrufen soll, wenn das Ereignis eintritt. Dazu verwendest du den vordefinierten Parameternamen **keyPressed**. Beim Aufruf erhält deine Funktion eine Zahl *key*, aus der du die Taste bestimmen kannst, die gedrückt wurde. (Die Werte kannst du durch einige Versuche selbst herausfinden.) In deinem Programm bewegt sich die Turtle ständig um 10 Schritte vorwärts. Du kannst ihre Richtung mit den Cursortasten verändern. Damit die Turtle das Fenster nicht verlässt, verwendest du den Wrap-Modus.

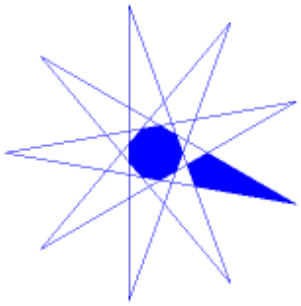
```
from gturtle import *

LEFT = 37
RIGHT = 39
UP = 38
DOWN = 40

def onKeyPressed(key):
   if key == LEFT:
      setHeading(-90)
   elif key == RIGHT:
      setHeading(90)
   elif key == UP:
      setHeading(0)
   elif key == DOWN:
      setHeading(180)

makeTurtle(keyPressed = onKeyPressed)
wrap()
while True:
    forward(10)
```

MEMO

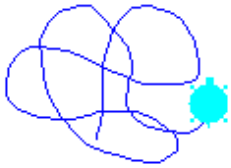
Das Hauptprogramm ist in einer Endlosschleife damit beschäftigt, die Turtle vorwärtszuschieben. Bei einem Tastaturevent wird es kurz unterbrochen und die Callbackfunktion gestartet, die sehr schnell zu Ende läuft. Das Hauptprogramm läuft dann weiter, wobei sich aber (eventuell) die Bewegungsrichtung der Turtle geändert hat.

AUFGABEN

1. Zeichne mit einer Wiederholstruktur den nebenstehenden Stern und fülle ihn mit Mausklicks nach deinem Geschmack aus.



2. Du kannst die Turtle dazu verwenden, ein Programm zum Freihandzeichen zu erstellen. Dabei wird der Zeichenstift mit mit dem Press-Event gesetzt mit dem Drag-Event bewegt.



3. Mit gedrückter linken Maustaste zeichnest du eine beliebige Figur. Mit einem Klick auf die rechte Maustaste kannst du ein Gebiet ausfärben.

4. Ergänze das Programm mit der Tastatursteuerung so, dass beim Drücken der Leertaste (Key Code 32) eine geschlossene Fläche, in der sich die Turtle gerade befindet, gefüllt wird. Zeichne damit einen gefüllten Buchstaben.

ZUSATZSTOFF

DEIN PERSÖNLICHES MAUSBILD

Du kannst das Bild des Mauscursors deinen eigenen Vorstellungen anpassen und damit dem Programm ein spezielles Aussehen geben. Dazu verwendest du den Befehl *setCursor()* und übergibst einen der Werte aus der untenstehenden Tabelle. Du kannst sogar ein eigenes Mausbild verwenden, wenn du **setCustomCursor()** den Pfad deiner Bilddatei übergibst. Übliche Mausikonen sind 32x32 Pixel gross und haben einem transparenten Hintergrund. Sie sollten im gif oder png-Format gespeichert sein.

Das oben gezeigte Verfolgungsprogramm kannst du jetzt mit deiner eigenen Mausfigur verschönern, wobei du auch mit *moveTo()* dafür sorgst, dass die Turtle sich immer bis zur Maus bewegt.

```
from gturtle import *

def onMouseMoved(x, y):
      moveTo(x, y)

makeTurtle(mouseMoved = onMouseMoved)
setCustomCursor("sprites/cutemouse.gif")
speed(-1)
```

MEMO

Mögliche Parameter von *setCursor()*:

Parameter	**Ikone**
Cursor.DEFAULT_CURSOR	Standard-Ikone
Cursor.CROSSHAIR_CURSOR	Fadenkreuz
Cursor.MOVE_CURSOR	Verschiebungs-Cursor (Kreuzpfeile)
Cursor.TEXT_CURSOR	Text-Cursor (vertikaler Strich)
Cursor.WAIT_CURSOR	Geduld-Cursor

Das Verzeichnis sprites in der Pfadangabe von *setCustomCursor()* ist im Verzeichnis, in dem sich dein Programm befindet

2.11 TURTLEOBJEKTE

EINFÜHRUNG

In der Natur ist eine Schildkröte ein Individuum mit seiner ganz spezifischen Identität. In einem Zoogehege könntest du jeder Schildkröte einen eigenen Namen geben, z.B. Pepe oder Maya. Schildkröten habe aber auch Gemeinsamkeiten: Sie sind Lebewesen aus der Tierklasse der Schildkröten. Solche Beschreibungen haben sich derart gut bewährt, dass sie in der Informatik als grundlegendes Konzept eingeführt wurden, das sich **Objektorientierte Programmierung** (**OOP**) nennt. Mit der Turtlegrafik ist es für dich spielerisch leicht, die Grundprinzipien der OOP kennen zu lernen.

PROGRAMMIERKONZEPTE: *Klasse, Objekt, Objektorientierte Programmierung, Konstruktor, Klone*

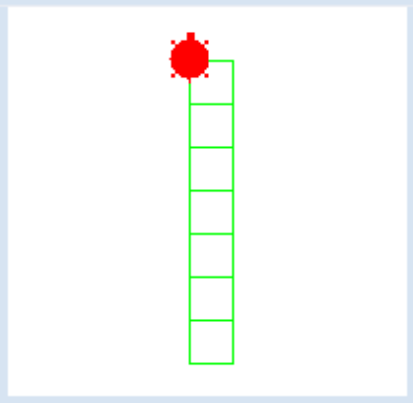
TURTLEOBJEKT ERZEUGEN

Bei der bisher verwendeten Turtle handelt es sich um ein anonymes Objekt, für das wir keinen Namen brauchten. Wenn du mehrere Turtles gleichzeitig verwenden willst, musst du aber jeder Turtle mit einem Namen eine eigene Identität geben. Den Namen kannst du als Variablennamen verwenden.

Mit der Anweisung **maya = Turtle()** erzeugst du eine Turtle mit dem Namen *maya*.
Mit der Anweisung *pepe = Turtle()* erzeugst du eine Turtle mit dem Namen *pepe*.

Du kannst die benannten Turtles mit den dir bekannten Befehlen steuern, aber du musst natürlich nun immer sagen, welche der Turtles du meinst. Dazu stellst du dem Befehl den Turtlenamen durch einen Punkt getrennt voran, zum Beispiel **maya.forward(100)**.

Im ersten Beispiel zeichnet *maya* eine Leiter. Die Zeile *makeTurtle()* brauchst du nicht mehr, da du ja die Turtle selbst erzeugst.



```
from gturtle import *

maya = Turtle()
maya.setColor("red")
maya.setPenColor("green")
maya.setPos(0, -200)

repeat 7:
    repeat 4:
        maya.forward(50)
        maya.right(90)
    maya.forward(50)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Gleichartige Objekte werden in Klassen zusammengefasst. Ein Objekt einer Klasse wird erzeugt, indem man den Klassennamen mit einer Parameterklammer verwendet. Wir nennen dies den **Konstruktor** der Klasse. Funktionen, die zu einer bestimmten Klasse gehören, nennen wir in Zukunft **Methoden**.

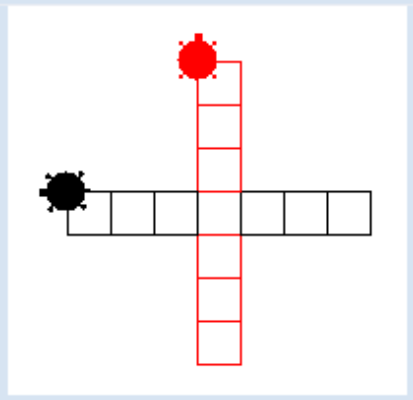
MEHRERE TURTLEOBJEKTE ERZEUGEN

Es liegt ja nun auf der Hand, dass du auf die beschriebene Art im gleichen Programm mehrere Turtles verwenden kannst. Willst du *maya* und *pepe* erzeugen, so schreibst du
maya = Turtle() und **pepe = Turtle()**

Allerdings befinden sich diese dann jeweils in ihrem eigenen Turtlefenster. Du kannst sie aber ins gleiche Turtlegehege setzen, indem du auch das Gehege als ein Objekt der Klasse *TurtleFrame* erzeugst:

tf = TurtleFrame()

und dieses bei der Erzeugung der Turtles angibst.
Während maya die gleiche Leiter wie vorhin baut, soll der schwarze pepe gleichzeitig eine horizontale schwarze Leiter bauen.

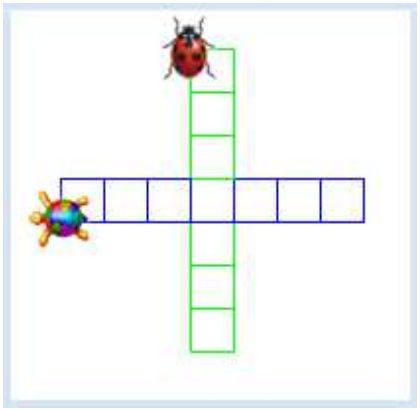


```
from gturtle import *

tf = TurtleFrame()

maya = Turtle(tf)
maya.setColor("red")
maya.setPenColor("red")
maya.setPos(0, -200)

pepe = Turtle(tf)
pepe.setColor("black")
pepe.setPenColor("black")
pepe.setPos(200, 0)
pepe.left(90)

repeat 7:
    repeat 4:
        maya.forward(50)
        maya.right(90)
        pepe.forward(50)
        pepe.left(90)
    maya.forward(50)
    pepe.forward(50)
```

MEMO

Wenn du mehrere Turtles in das gleiche Fenster setzen willst, musst du ein *TurtleFrame* erzeugen und dieses als Konstruktorparameter der Turtle angeben. Die Turtles kollidieren nicht miteinander, sondern bewegen sich sozusagen übereinander, wobei die eben sich bewegende Turtle immer über allen anderen zu liegen kommt.

TURTLEPARAMETER



Beide Turtles zeichnen eine gleichartige Leiter. Dafür wird derselbe Code verwendet. Daher ist es eleganter, dazu eine Funktion **step()** zu definieren, der man mitteilt, welche Turtle die Zeichnungen ausführen soll. Dazu wird die jeweilige turtle als Parameter an die Funktion step() übergeben.

Als Parameterbezeichner kannst du irgendeinen Namen verwenden, beispielsweise lediglich t, kurz für irgendeine Turtle. Du übergibst dann beim Aufruf das eine Mal **maya** und das andere Mal **pepe**.

```
from gturtle import *

def step(t):
    repeat 4:
        t.forward(50)
        t.right(90)
    t.forward(50)

tf = TurtleFrame()

maya = Turtle(tf, "sprites/beetle.gif")
maya.setPenColor("green")
maya.setPos(0, -150)
pepe = Turtle(tf, "sprites/cuteturtle.gif")
pepe.setPos(200, 0)
pepe.left(90)

repeat 7:
    step(maya)
    step(pepe)
```

MEMO

Du kannst für jede Turtle ein **eigenes Bild** verwenden, wenn du beim Erzeugen der Turtle den Pfad auf die Bilddatei angibst. Hier verwendest du die beiden Bilddateien beetle.gif und cuteturtle.gif, die sich in der Distribution von TigerJython befinden.

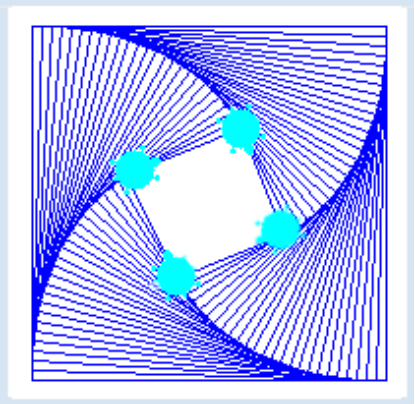
KÄFERPROBLEME MIT GEKLONTER TURTLE

Beim berühmten Käferproblem [**mehr...**] starten n Käfer in den Ecken eines regulären n-Ecks und verfolgen sich gegenseitig mit konstanter Geschwindigkeit. Dabei wird die Lage der Käfer in gleichen Zeitschritten fixiert und jeder Käfer in die Richtung zum Käfer an der nächsten Polygonecke gedreht. Nachher bewegen sich alle Käfer geradlinig um eine immer gleiche Schrittweite vorwärts.

Du kannst dieses Problem sehr elegant lösen, indem du zuerst mit der namenlosen (globalen) Turtle das Polygon zeichnest und an jeder Ecke ein geklontes Turtleobjekt erstellst. Du wählst hier ein Viereck und erstellst mit **clone()** die Turtleklons t1, t2, t3, t4. Ein Klone ist ein neues Turtle-Objekt mit identischen Eigenschaften.

Nachher stellst du in einer Endlosschleife mit **setHeading()** ihre Blickrichtung ein und schiebst sie um die Schrittweite 5 vorwärts. Die Zeichnung wird besonders schön, wenn du noch die Verbindungsgeraden zwischen den jeweils sich verfolgenden Turtles einzeichnest.

Am einfachsten definierst du dazu die Funktion **drawLine(a, b)**, mit welcher die Turtle a mit **moveTo()** eine Spur zur Turtle b zeichnet und wieder zurück springt.



```
from gturtle import *

s = 360

makeTurtle()
setPos(-s/2, -s/2)

def drawLine(a, b):
    ax = a.getX()
    ay = a.getY()
    ah = a.heading()
    a.moveTo(b.getX(), b.getY())
    a.setPos(ax, ay)
    a.heading(ah)

# generate Turtle clone
t1 = clone()
t1.speed(-1)
forward(s)
right(90)
t2 = clone()
t2.speed(-1)
forward(s)
right(90)
t3 = clone()
t3.speed(-1)
forward(s)
right(90)
t4 = clone()
t4.speed(-1)
forward(s)
right(90)
hideTurtle()

repeat:
    t1.setHeading(t1.towards(t2))
    t2.setHeading(t2.towards(t3))
    t3.setHeading(t3.towards(t4))
    t4.setHeading(t4.towards(t1))

    drawLine(t1, t2)
    drawLine(t2, t3)
    drawLine(t3, t4)
    drawLine(t4, t1)

    t1.forward(5)
    t2.forward(5)
    t3.forward(5)
    t4.forward(5)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Erzeugst du mit **clone()** aus der globalen Turtel eine neue Turtle, so hat diese die gleiche Position, die gleiche Blickrichtung und die gleiche Farbe (bei Verwendung von benutzerdefinierten Turtlebildern hat sie das gleiche Turtlebild) [**mehr...**] .

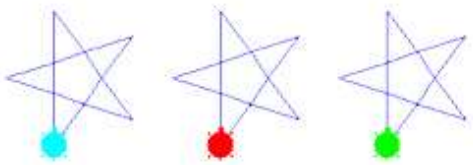
Die Funktion *drawLine()* kann vereinfacht werden, wenn man Position und Blickrichtung der Turtle mit *pushState()* abspeichert und den Zustand mit *popState()* wieder zurückholt:

```
def drawLine(a, b):
    a.pushState()
    a.moveTo(b.getX(), b.getY())
    a.popState()
```

Die Verfolgungskurve lässt sich mathematisch berechnen (**siehe**).

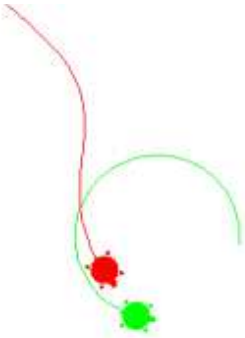
AUFGABEN

1. Drei Turtles sollen abwechlungsweise Zack-um-Zack einen fünfzackigen Stern zeichnen. Die Turtles haben die Farben cyan (Standardfarbe), rot und grün. Die Turtlefarbe kann als zusätzlicher Parameter des Konstruktors angegeben werden.



2. Eine grüne Mutterturtle bewegt sich mit grüner Stiftfarbe ständig auf einem Kreis. Eine rote Kindturtle ist zuerst weit von der Mutter entfernt und bewegt sich dann mit roter Stiftfarbe in Richtung zur Mutter.

   (Das Kind *child* kann mit *direction = child.towards(mother.getX(), mother.getY())* die Richtung zur Mutter bestimmen.)

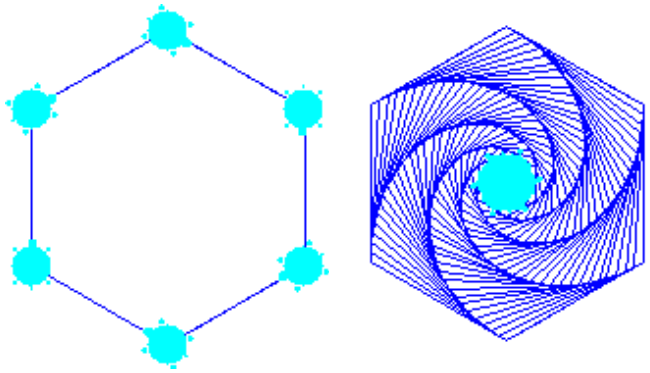


3. 

   Die Turtle laura zeichnet (nicht gefüllte) Quadrate. Nach jedem gezeichneten Quadrat springt eine zweite Turtle hinein und färbt es grün.

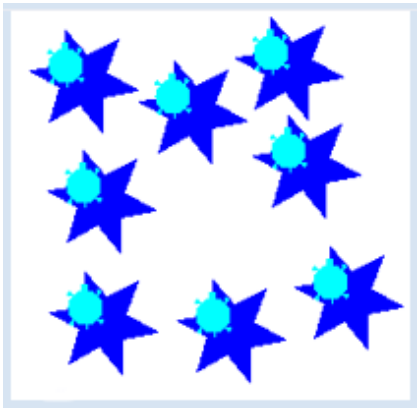
   Verwende für die beiden Turtles verschiedene Turtlebilder. Im *tigerjython2.jar* stehen die Bilder beetle.gif, beetle1.gif, beetle2.gif und spider.png zur Verfügung. Du kannst aber euch eigene Bilder verwenden. Du musst diese im Unterverzeichnis *sprites* des Verzeichnisses speichern, in dem sich dein Programm befindet.

4. Erstelle ähnlich wie im Beispiel "Käferprobleme" eine Verfolgungsgrafik für 6 Turtles, die in den Ecken eines regelmässigen 6-Eck die Verfolgung starten.



ZUSATZSTOFF

TURTLES MIT MAUSKLICK ERZEUGEN



Dein Programm erzeugt bei jedem Mausklick an der Stelle des Mauscursors eine neue Turtle, die unabhängig von den bereits vorhandenen Turtles einen Stern zeichnet. Dabei erlebst du die volle Tragweite und die Eleganz der Objektorientierten Programmierung sowie der Ereignissteuerung.

Um den Mausklick zu erfassen, definierst du eine Funktion **drawStar()**. Damit diese beim Drücken der linken Maustaste vom System aufgerufen wird, verwendest du im Konstruktor von TurtleFrame den benannten Parameter **mouseHit** und übergibst ihm den Namen dieser Funktion.

```
from gturtle import *

def drawStar(x, y):
    t = Turtle(tf)
    t.setPos(x, y)
    t.fillToPoint(x, y)
    for i in range(6):
        t.forward(40)
        t.right(140)
        t.forward(40)
        t.left(80)

tf = TurtleFrame(mouseHit = drawStar)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

In der OOP werden Objekte mit gleichen Fähigkeiten und gleichen Eigenschaften in Klassen zusammengefasst. Mit dem Konstruktor erzeugt man einzelne Objekte (**Instanzen**).

Um einen Mausklick zu verarbeiten, schreibst du eine Funktion mit beliebigem Namen (aber zwei Parametern x und y) und übergibt diesen Funktionsnamen im Konstruktor von *TurtleFrame* dem benannten Parameter *mouseHit*.

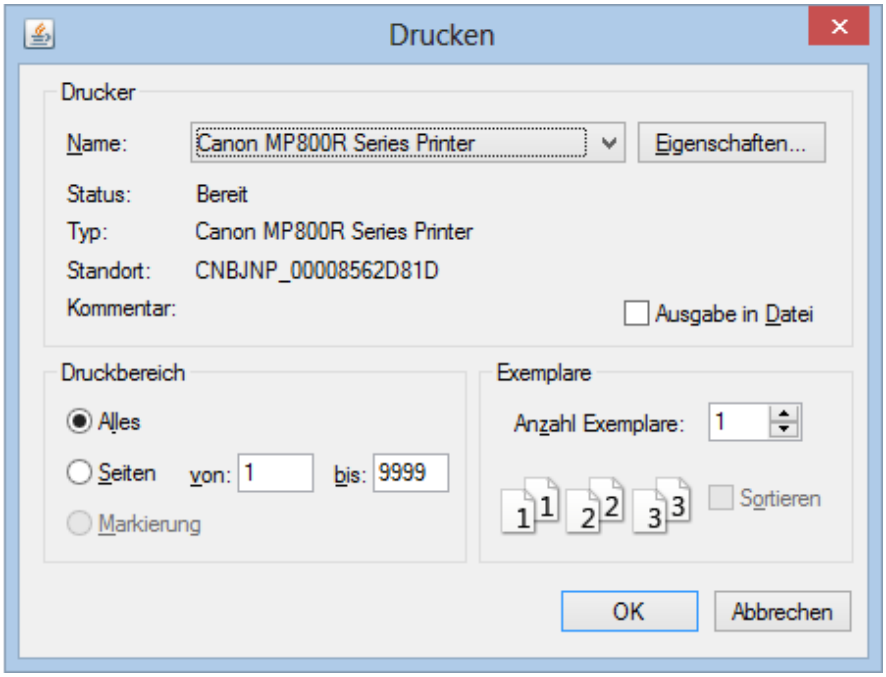
x und y liefern die Koordinaten des Mausklicks.

2.12 DRUCKEN

EINFÜHRUNG

Für exaktes Zeichnen eignet sich ein Drucker besser als der Bildschirm, da ein Drucker meist eine weit höhere Auflösung hat, beispielsweise 1200x1200 dpi gegenüber den üblichen Bildschirmauflösungen von ungefähr 100 dpi. Das Ausdrucken einer GPanel-Grafik erfolgt so, dass die Grafikoperationen statt auf dem Bildschirm auf dem Drucker dargestellt werden. Damit lassen sich hochauflösene Vektorgrafiken erstellen. Dazu definiert man eine Funktion mit irgend einem Namen, die alle Befehle zur Erstellung des Bildes enthält. Beim direkten Aufruf erscheint das Bild auf dem Bildschirm. Um es auszudrucken, ruft man *printerPlot()* mit dem Funktionsnamen auf.

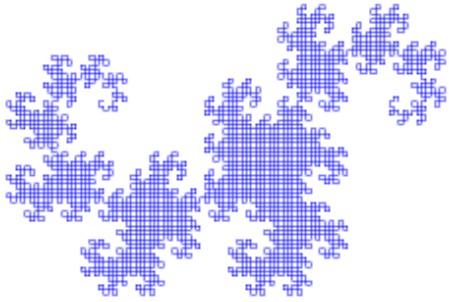
Es wird ein Druckerdialog angezeigt, in dem man den Drucker auswählen und seine Eigenschaften einstellen kann. Du kannst so auch auf virtuelle Drucker ausgeben, die eine Grafik-Datei in einem hochauflösenden Format (z.B. Tiff oder EPS) erstellen [**mehr...**].



PROGRAMMIERKONZEPTE: *Hochauflösende Grafik*

EIN NICHT FEUERSPEIENDER DRACHEN

Um dir Freude am hochauflösenden Drucken zu bereiten, lässt du die Turtle ein komplizierteres Bild zeichnen, nämlich eine Drachenkurve. Zwar kann man durch Falten eines Papierstreifens eine solche Kurve auch erzeugen, aber mit Computergrafik geht es viel einfacher. Allerdings ist die Umsetzung der Faltanleitung in einen Algorithmus langwierig nicht trivial. Mit Computergrafik geht es viel einfacher.



Es geht ja hier ums Drucken und deshalb verwendest du eine vorgegebene Funktion *figure(s, n, flag)*, um die Kurve zu zeichnen. Immerhin siehst du, dass die Kurve rekursiv definiert ist und

Funktion einen Parameter flag, der 1 oder -1 sein kann und bestimmt, in welcher Richtung gezeichnet wird. Um das Bild auszudrucken, erstellst du es in der Funktion *doIt()*, die keine Parameter haben darf. Wenn du *doIt()* ganz normal aufrufst, so erscheint die Zeichnung auf dem Bildschirm. Wenn du *printerPlot(doIt)* aufruftst, alsodem Befehl *printerPlot()* den Namen *doIt* übergibst, wird die Zeichnung (ohne die Turtle zu zeigen) ausgedruckt.

```
from gturtle import *
import math

nbGenerations = 12

def doIt():
    rt(90)
    figure(300, nbGenerations, 1)

def figure(s, n, flag):
    if n == 0:
        fd(s);
    else:
        alpha = 45
        if flag == 1:
            alpha = -alpha
            flag = -flag
        lt(alpha)
        figure(s / math.sqrt(2), n - 1, -flag)
        rt(2 * alpha)
        figure(s / math.sqrt(2), n - 1, flag)
        lt(alpha)

makeTurtle()
ht()
setPos(-100, 100) # screen
doIt()
setPos(100, 0)      # printer
printerPlot(doIt)
```

MEMO

Du musst mit *setPos()* die Zeichnung auf dem Blatt richtig positionieren. Je nach Grösse des Turtlefensters und dem verwendeten Drucker, wird sich diese Position ändern. Du kannst beim Aufruf von *printerPlot()* auch noch einen Skalierungsfaktor k angeben, also *printerPlot(doIt, k)*, der für k > 1 das Bild vergrössert und es für k < 1 verkleinert.

AUFGABEN

1. Joshua Goldstein schlägt vor, Paare von Move/Turn-Befehlen zur Erstellung von schönen Figuren zu verwenden. Ein Schritt besteht also aus den Befehlen
   forward(s)
   right(a)
   Zeichne und drucke die Goldstein-Figuren für folgende Fälle:
   a. 31 Schritte mit s = 300, a = 151°
   b. 142 Schritte mit s = 400, a = 159.72°
   Für die Positionierung musst du selbst sorgen.

2. Ein Schritt kann auch aus zwei Move/Turn-Paaren bestehen
   Zeichne und drucke die Goldstein-Figur für folgenden Fall:
   37 Schritte mit s = 77, a = 140.86° und s = 310, a = 112°

3. Zeichne und drucke die Goldstein-Figur mit drei Move/Turn-Paaren:
   47 Schritte mit s = 15.4, a = 140.86 ° und s = 62, a = 112° und s= 57.2, a = 130°

Dokumentation Turtlegrafik

Module import: from gturtle import *

Funktion	Aktion
makeTurtle()	erzeugt eine (globale) Turtle im neuen Grafikfenster
makeTurtle(color)	erzeugt eine Turtle mit angegebener Farbe
makeTurtle("sprites/turtle.gif")	erzeugt Turtle mit einem eigenen Turtle-Bild turtle.gif
t = Turtle()	erzeugt ein Turtleobjekt t
tf = TurtleFrame()	erzeugt ein Bildschirmfenster, in dem mehrere Turtles leben
t = Turtle(tf)	erzeugt ein Turtleobjekt t im TurtleFrame tf
clone()	erzeugt ein Turtleklon (gleiche Farbe, Position, Blickrichtung)
isDisposed()	gibt True zurück, falls das Turtlefenster geschlossen ist
putSleep()	hält den Programmablauf an, bis wakeUp() aufgerufen wird
wakeUp()	führt angehaltenen Programmablauf weiter

Bewegen

back(distance), bk(distance)	bewegt Turtle rückwärts
forward(distance), fd(distance)	bewegt Turtle vorwärts
hideTurtle(), ht()	macht Turtle unsichtbar (Turtle zeichnet schneller)
home()	setzt Turtle in die Mitte des Fensters mit Richtung nach oben
left(angle), lt(angle)	dreht Turtle nach links
penDown(), pd()	setzt Zeichenstift ab (Spur sichtbar)
penErase(), pe()	setzt die Stiftfarbe auf die Hintergrundfarbe
leftArc(radius, angle)	bewegt Turtle auf einem Bogen mit dem Sektor-Winkel *angle* nach links
leftCircle(radius)	bewegt Turtle auf einem Kreis nach links
penUp(), pu()	hebt den Zeichenstift (Spur unsichtbar)
penWidth(width)	setzt die Dicke des Stifts in Pixel
right(angle), rt(angle)	dreht Turtle nach rechts
rightArc(radius, angle)	bewegt Turtle auf einem Bogen mit dem Sektor-Winkel *angle* nach rechts
rightCircle(radius)	bewegt Turtle auf einem Kreis nach rechts
setCustomCursor(cursorImage)	wählt die Bilddatei des Mauscursors
setCustomCursor(cursorImage, Point(x, y))	wählt die Bilddatei des Mauscursors unter Angabe der Mausposition innerhalb des Bildes
setLineWidth(width)	setzt die Dicke des Stifts in Pixel
showTurtle(), st()	zeigt Turtle
speed(speed)	setzt Turtlegeschwindigkeit
delay(time)	hält das Programm während der Zeit time (in Millisekunden) an
wrap()	setzt Turtlepositionen ausserhalb des Fensters ins Fenster zurück
clip()	Turtles ausserhalb des Fensters sind nicht sichtbar
getPlaygroundWidth()	gibt die Breite m des Turtlefensters zurück (Turtlekoordinaten x = -m/2 ... m/2)
getPlaygroundHeight()	gibt die Höhe n des Turtlefensters zurück (Turtlekoordinaten y = -n/2 ... n/2)

Positionieren

distance(x, y)	gibt die Entfernung der Turtle zum Punkt(x, y) zurück

distance(coords)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
getPos()	gibt die Turtleposition zurück als Punkt
getX()	gibt die aktuelle x-Koordinate der Turtle zurück
getY()	gibt die aktuelle y-Koordinate der Turtle zurück
heading()	gibt die Richtung der Turtle zurück
heading(degrees)	setzt die Richtung der Turtle (0 ist gegen oben, im Uhrzeigersinn)
moveTo(x, y)	bewegt Turtle auf die Position (x, y)
moveTo(coords)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
setHeading(degrees), setH(degrees)	setzt die Richtung der Turtle (0 gegen oben, im Uhrzeigersinn)
setRandomHeading()	setzt die Richtung zufällig zwischen 0 und 360°
setPos(x, y)	setzt Turtle auf die Position (x, y)
setPos(coords)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
setX(x)	setzt Turtle auf x-Koordinate
setY(y)	setzt Turtle auf y-Koordinate
setRandomPos(width, height)	setzt Turtle auf einen zufälligen Punkt im Bereich 0..width, 0..height
setScreenPos(x, y)	setzt Turtle auf gegebene Pixelkoordinaten (x, y)
setScreenPos(Point(x, y))	setzt Turtle auf gegebene Pixelkoordinaten
towards(x, y)	gibt die Richtung (in Grad) zur Position (x, y) (auch list, tuple, complex)
towards(x, y)	gibt die Richtung (in Grad) zur Position (x, y) (auch list, tuple, complex)
towards(coord)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
toTurtlePos(x, y)	gibt die Turtlekoordinaten zu den gegebenen Pixelkoordinaten (x, y) zurück
toTurtlePos(Point(x, y))	gibt die Turtlekoordinaten zu den gegebenen Pixelkoordinaten zurück
pushState()	speichert den Turtlezustand in einem Stapelspeicher
popState()	holt den zuletzt gespeicherten Zustand vom Stapelspeicher
clearStates()	löscht den Stapelspeicher

Farben

askColor(title, defaultColor)	zeigt ein Fenster zur Farbwahl und gibt die gewählte Farbe zurück, (None, wenn Abbrechen gedrückt wurde)title: Fenstertitel an, defaultColor: Farbvorschlag
clean()	löscht die Zeichnung und versteckt alle Turtles, Turtles bleiben am Ort
clean(color)	löscht die Zeichnung, versteckt alle Turtles, färbt den Hintergrund, Turtles bleiben am Ort
clear()	löscht die Zeichnung, Turtles bleiben am Ort
clear(color)	löscht die Zeichnung und färbt den Hintergrund, Turtles bleiben am Ort
clearScreen(), cs()	löscht die Zeichnung und setzt die Turtle an die Homeposition
dot(diameter)	zeichnet einen mit Stiftfarbe gefüllten Kreis
openDot(diameter)	zeichnet einen nicht gefüllten Kreis
fill()	füllt die geschlossene Figur, in der sich die Turtle befindet mit der Füllfarbe
fill(x , y)	füllt eine geschlossene Figur um den inneren Punkt (x, y) mit der Füllfarbe
fill(coords)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
fillToPoint()	füllt fortlaufend die gezeichnete Figur von der aktuellen Turtleposition mit der Stiftfarbe
fillToPoint(x , y)	füllt fortlaufend die gezeichnete Figur vom Punkt (x, y) mit der Stiftfarbe (auch list, tuple, complex)

fillToHorizontal(y)	füllt fortlaufend die Fläche zwischen der Figur und der horizontalen Linie in Höhe y mit der Stiftfarbe
fillToVertical(x)	füllt fortlaufend die Fläche zwischen der Figur und der vertikalen Linie am Wert xmit der Stiftfarbe
fillOff()	beendet den fortlaufenden Füllmodus
getColor()	gibt die Turtlefarbe zurück
getColorStr()	gibt die X11-Turtlefarbe zurück
getFillColor()	gibt die Füllfarbe zurück
getFillColorStr()	gibt die X11-Füllfarbe zurück
getPixelColor()	gibt die Farbe des Pixels an der Turtlekoordinate zurück
getPixelColorStr()	gibt die Farbe des Pixels an der Turtlekoordinate als X11-Farbe zurück
makeColor()	gibt eine Farbreferenz von value zurück. Werte-Beispiele: ("7FFED4"), ("Aqua-Marine"), (0x7FFED4), (8388564), (0.5, 1.0, 0.83), (128, 255, 212), ("rainbow", n) mit n = 0..1, Lichtspektrum
setRandomX11Color()	gibt eine zufällige X11-Farbe zurück
setColor(color)	legt Turtlefarbe fest
setPenColor(color)	legt Stiftfarbe fest
setFillColor(color)	legt Füllfarbe fest
startPath()	startet die Aufzeichnung der Turtlebewegung zum nachträglichen Füllen
fillPath()	verbindet die aktuelle Turtleposition mit dem Startpunkt und füllt die geschlossene Figur mit der Füllfarbe
stampTurtle()	erzeugt ein Turtlebild an der aktuellen Turtleposition
stampTurtle(color)	erzeugt ein Turtlebild mit angegebener Farbe an der aktuellen Turtleposition

Callbacks

makeTurtle(mouseNNN = onMouseNNN) auch mehrere, durch Komma getrennt	registriert die Callbackfunktion onMouseNNN(x, y), die beim Mausevent aufgerufen wird. Werte für NNN: Pressed, Released, Clicked, Dragged, Moved, Entered, Exited, SingleClicked, DoubleClicked, Hit: Aufruf im eigenen Thread, HitX: Dasselbe, aber nachfolgende Events ignoriert, bis Callback zurückkehrt
isLeftMouseButton(), isRightMouseButton()	gibt True zurück, falls beim Event die linke bzw. rechte Maustaste verwendet wurde
makeTurtle(keyNNN = onKeyNNN)	registriert die Callbackfunktion onKeyNNN(keyCode), die beim Drücken einer Tastaturtaste aufgerufen wird. Werte für NNN: Pressed, Hit: Aufruf im eigenen Thread, HitX: Dasselbe, aber nachfolgende Events ignoriert, bis Callback zurückkehrt. keyCode ist ein für die Taste eindeutiger integer Code
getKeyModifiers()	liefert nach einem Tastaturevent einen Code für Spezialtasten (Shift, Ctrl, usw.)
makeTurtle(closeClicked = onCloseClicked)	registriert die Callbackfunktion onCloseClicked(), die beim Klick des Close-Buttons des Turtlefensters aufgerufen wird. Das Fenster wird mit dispose() geschlossen
makeTurtle(turtleHit=onTurtleHit)	registriert die Callbackfunktion onTurtleHit(x, y), die aufgerufen wird, wenn auf das Turtlebild geklickt wird
t = Turtle(turtleHit = onTurtleHit)	registriert die Callbackfunktion onTurtleHit(t, x, y), die aufgerufen wird, wenn auf das Bild der Turtle t geklickt wird

Texte, Bilder und Sound

addStatusBar(20)	fügt eine Statusbar mit der Höhe 20 Pixel hinzu
beep()	erzeugt einen Ton

playTone(freq)	spielt Ton mit gegebener Frequenz (in Hz) 1000 ms (blockierende Funktion)
playTone(freq, blocking = False)	nicht blockierende Funktion (um mehrere Töne gleichzeitig abzuspielen)
playTone(freq, duration)	spielt Ton mit gegebener Frequenz und Dauer
playTone([f1, f2, f3 ...])	spielt hintereinander mehrere Töne mit geg. Frequenzen
playTone([(f1, d1),(f2, d2), (f3, d3)...])	spielt hintereinander mehrere Töne mit geg. Frequenzen und Dauer
playTone([("c", 700),("e", 1500)...]	spielt hintereinander mehrere Töne mit geg. Tonbezeichnungen und geg. Dauer. Erlaubt sind: grosse Oktave , ein- , zwei- und dreigestrichene Oktave also im Bereich c, c#, ...h"')
playTone([("c", 700),("e", 1500)...], instrument="piano")	wie vorher, aber mit gewähltem Instrument (piano, quitar, harp, trumpet, organ, panfluete, seashore, violin, xylophone... (gemäss MIDI Spezifikation)).
playTone([("c", 700),("e", 1500...)], instrument="piano", volumen = 10)	wie vorher, aber mit gewählten Lautstärke (0....100)
label(text)	schreibt Text an der aktuellen Position
printerPlot(draw)	druckt die mit der Funktion draw erstellte Zeichnung
setFont(Font font)	legt Schriftart fest
setFontSize(size)	legt Schriftgrösse fest
setStatusText("Press any key!")	schreibt eine Mitteilung in die Statusbar
setTitel("Text")	schreibt den Text in die Titelzeile
img = getImage(path)	lädt ein Bild (im png- gif, jpg-Format) vom lokalen Filesystem oder von einer URL und gibt eine Referenz darauf zurück. Für path = sprites/nnn werden auch Bilder von der TigerJython-Distribution geladen (Help/Bilderbibliothek zeigt die vorhanden Bilder)
drawImage(img)	stelt das Bild img an der Position der Turtle mit ihrer Blickrichtung dar

Dialoge

msgDlg(message)	öffnet einen modalen Dialog mit einem OK-Button und gegebenem Mitteilungstext
msgDlg(message, title = title_text)	dasselbe mit Titelangabe
inputInt(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt den eingegebenen Integer zurück (falls kein Integer, wird Dialog neu angezeigt). Abbrechen od. Schliessen beendet das Programm
inputInt(prompt, False)	dasselbe, aber Abbrechen beendet das Programm nicht, gibt None zurück
inputFloat(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt den eingegebenen Float zurück (falls kein Float, wird Dialog neu angezeigt). Abbrechen od. Schliessen beendet das Programm
inputFloat(prompt, False)	dasselbe, aber Abbrechen beendet das Programm nicht, gibt None zurück
inputString(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt den eingegeben String zurück. Abbrechen od. Schliessen beendet das Programm
inputString(prompt, False)	dasselbe, aber Abbrechen beendet das Programm nicht, gibt None zurück
input(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt Eingabe als Integer, Float oder String zurück. Abbrechen beendet das Programm
input(prompt, False)	dasselbe, aber Abbrechen beendet das Programm nicht, gibt None zurück
askYesNo(prompt)	öffnet einen modalen Dialog mit Ja/Nein-Buttons. Ja gibt True, Nein gibt False zurück. Schliessen beendet das Programm
askYesNo(prompt, False)	dasselbe, aber Schliessen beendet das Programm nicht, gibt None zurück

kapitel drei

GRAFIK & BILDER

Lernziele

- Du kannst eine einfache 2-D-Grafik mit geometrischen Formen erstellen.
- Du weisst, wie man Tastatur- und Mauseingaben im Grafikfenster verwendet.
- Du kannst einfache Funktionsgrafen y = f(x) im Grafikfenster darstellen.
- Du weisst, dass ein digitalisiertes Bild aus Farbpixeln besteht, die als Zahlen gespeichert werden.
- Du kannst mit einem eigenen Programm ein digitalisiertes Bild einlesen, gezielt verändern, auf dem Bildschirm darstellen und abspeichern.
- Du kennst einige Verfahren der Bilderkennung und kannst sie in einem eigenen Programm anwenden.
- Du kannst Tastatur - und Mausgesteuerte Programme schreiben.
- Du weisst, wie man Zufallszahlen erzeugen kann und kannst diese bei Zufallsexperimenten einsetzen.
- Du kannst in deinen Programmen Eingabefelder, Buttons und Menüs verwenden.

"Ein Bild sagt mehr als tausend Worte."

Altes Sprichwort

3.1 KOORDINATEN

EINFÜHRUNG

Du hast mit der Turtle bereits erste Erfahrung mit dem Zeichnen auf dem Computer gemacht. Allerdings hat die Turtlegrafik ihre Grenzen und so wirst du hier flexiblere Möglichkeiten für die Grafikausgabe kennenlernen.

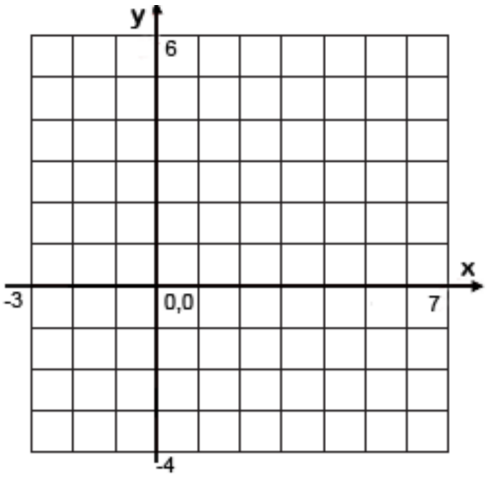
PROGRAMMIERKONZEPTE: *Koordinatengrafik, Kartesisches Koordinatensystem*

DAS GRAFIKFENSTER ÖFFNEN

Die Bibliothek (bzw. das Fenster für die Grafikausgabe) heisst GPanel. In TigerJython ist diese Bibliothek zwar bereits installiert, du musst aber trotzdem angeben, dass du das GPanel verwenden möchtest und beginnst daher dein Programm mit einem import. Danach erzeugst du mit **makeGPanel()** ein neues Grafikfenster:

```
from gpanel import *
makeGPanel(-3, 7, -4, 6)
```

Soweit macht das Programm noch nichts spannendes, sondern zeigt lediglich ein leeres Fenster, das du wieder schliessen kannst. Dein GPanel-Fenster ist immer quadratisch und verwendet ein x-y-Koordinatensystems, so wie du es aus der Mathematik kennst:

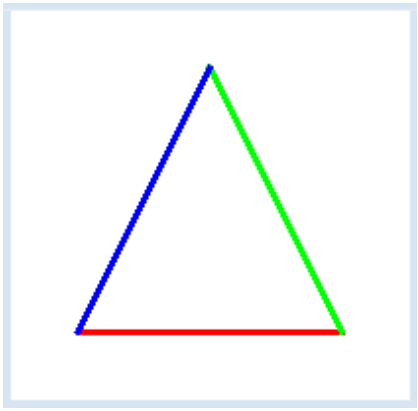


Mit den vier Zahlen -3, 7 -4, 6 wählst du den x- und y-Koordinatenbereich. -3 ist die x-Koordinate am linken Rand, 7 ist die x-Koordinate am rechten Rand, -4 ist die y-Koordinate am unteren Rand und 6 ist die y-Koordinate am oberen Rand.

MEMO

Mit **makeGPanel()** wird ein Fenster erzeugt. Dabei gibst du mit vier Zahlen den gewünschten Bereich des Koordinatensystems an:
makeGPanel(xmin, xmax, ymin, ymax)
Du kannst als ersten Parameter auch zusätzlich einen Fenster-Titel angeben:
makeGPanel(title, xmin, xmax, ymin, ymax)

LINIEN ZEICHNEN



Nachdem das Fenster offen ist, kannst du nun nach Belieben darin zeichnen. Dafür gibt es eine Reihe von nützlichen Funktionen. Mit **line()** lässt sich z. B. eine Linie zeichnen, mit **setColor()** änderst du die Farbe. Damit kannst du beispielsweise ein buntes Dreieck zeichnen.

Bei jeder Linie gibst du zuerst x- und y-Koordinaten des Startpunkts an, dann die x- und y- Koordinaten des Endpunkts. Die Eckpunkte sollen folgende Koordinaten haben: (1, -1) (5, -1) (3, 3). Du siehst das Dreieck aber natürlich nur, wenn du das Koordinatensystem passend wählst. Unverzerrte Zeichnungen ergeben sich aber nur, wenn das Koordinatensystem in x- und y-Richtung gleich lang ist.

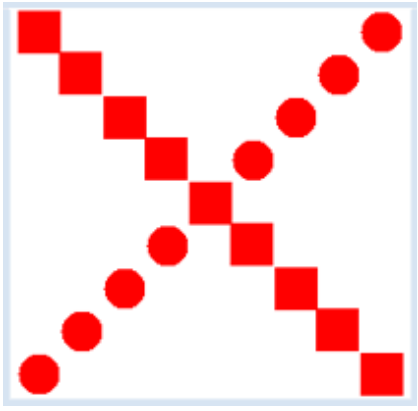
```
from gpanel import *

makeGPanel("Mein Grafikfenster", 0, 6, -2, 4)

lineWidth(3)
setColor("red")
line(1, -1, 5, -1)
setColor("green")
line(5, -1, 3, 3)
setColor("blue")
line(3, 3, 1, -1)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Breite der Linie stellst du mit der Funktion **lineWidth()** ein, wobei du die Breite in Pixeln angibst.

KREISE UND RECHTECKE



GPanel kann nicht nur Linien, sondern auch Kreise, Ellipsen, Rechtecke, Dreiecke und Kreisbögen zeichnen und sogar Texte ausschreiben. Mit dem Befehl *fillCircle(radius)* kannst du einen gefüllten Kreis zeichnen. Bevor du aber einen Kreis zeichnest, musst du den Grafikcursor mit *move(x, y)* positionieren, um den Mittelpunkt festzulegen.

fillRectangle(länge, breite) zeichnet ein Rechteck, dessen Mittelpunkt an der Position des Grafikcursors liegt. In unserem Beispiel zeichnen wir mit einer while-Schleife mehrere Quadrate und Kreise.

```
from gpanel import *

makeGPanel(0, 20, 0, 20)

setColor("red")
x = 2
y = 2
while y < 20:
    move(x, y)
    fillCircle(1)
    move(x, 20 - y)
    fillRectangle(2, 2)
    x = x + 2
    y = y + 2
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Mit GPanel kannst du verschiedene Figuren zeichnen. Hier die wichtigsten Befehle:

point(x, y)	Ein Punkt
line(x1, y1, x2, y2)	Eine Linie
rectangle(width, height)	Ein Rechteckt (Breite, Höhe)
fillRectangle(width, height)	Ein ausgefülltes Rechteck
rectangle(x1, y1, x2, y2)	Ein Rechteckt (Eckpunkte)
fillRectangle(x1, y1, x2, y2)	Ein ausgefülltes Rechteck
fillTriangle(x1, y1, ..., y3)	Eine Dreieck (Eckpunkte)
circle(r)	Ein Kreis mit Radius r
fillCircle(r)	Ein ausgefüllter Kreis
ellipse(a, b)	Eine Ellipse mit den Achsen a, b
fillEllipse(a, b)	Eine ausgefüllte Ellipse
arc(r, a, b)	Ein Kreisbogen
text("t")	Den Text t schreiben
move(x, y)	Postition festlegen

Für Kreise, Kreisbögen, Ellipsen, Texte und Rechtecke, die durch Länge und Breite festgelegt sind, musst du zuerst die Position der Figur mit *move()* festlegen.

GPanel kennt die sogenannten ***X11-Farben***. Das sind einige dutzend Farbnamen, die du im Internet unter **http://cng.seas.rochester.edu/CNG/docs/x11color.html** finden kannst. Alle diese Farben kannst du mit ***setColor(farbe)*** wählen.

AUFGABEN

1. Zeichne eine ähnliche Figur:

2. Wie sehen eigentlich Regenbogen aus? Lass dir vom Computer einen Regenbogen zeichnen. Verwende dazu die Funktion *circle(r)* so, dass nur der obere Teil des Kreises sichtbar ist.

3.2 FOR - SCHLEIFEN

EINFÜHRUNG

Oft musst du während der Wiederholung mitzählen lassen. Dazu benötigst du in einem Wiederholblock eine Variable, die sich in jedem Schleifendurchgang um einen bestimmten Wert ändert. Statt einer while-Struktur kannst du dies einfacher mit einer *for-Struktur* erledigen. Dazu muss du zuerst die *range()-Funktion* verstehen. Im einfachsten Fall hat *range()* einen einzigen Parameter (auch *Stop-Wert* genannt) und liefert eine Folge (Liste) von natürlichen Zahlen, die mit 0 beginnt und bei der letzten Zahl vor dem Stop-Wert endet.

Du kannst dies an ein paar Beispielen selbst ausprobieren. Wenn du beispielsweise ein Programm mit der einzigen Anweisung
print range(10) ausführst, werden die Zahlen [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
in das Ausgabefenster geschrieben. Versuche es mit ein paar anderen Parametern. Wie du siehst, ist der Stop-Wert 10 nicht mehr in der Liste enthalten. Er gibt aber an, wie viele Listenelemente es hat.

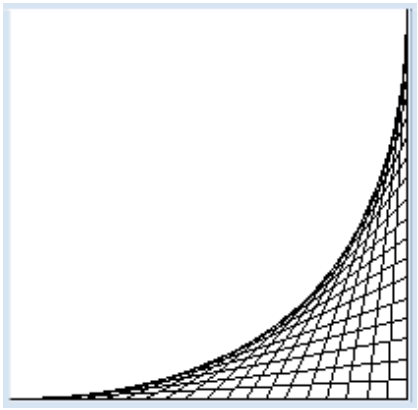
PROGRAMMIERKONZEPTE: *Iteration, For-Struktur, Verschachtelung von For-Schleifen*

LINIENSCHAR

Du kannst mit dieser **for-Struktur** eine hübsche Linienschar mit 20 Linien zeichnen.

```
from gpanel import *

makeGPanel(0, 20, 0, 20)

for i in range(21):
    line(i, 0, 20, i)
```

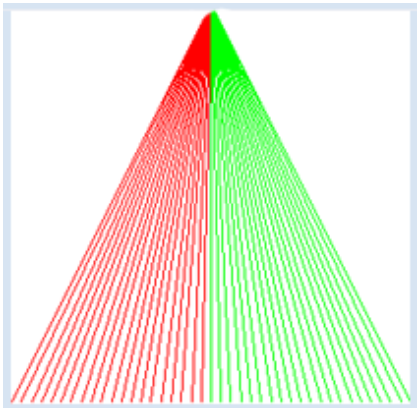
Programmcode markieren (Ctrl+C copy, Ctrl+V paste)



MEMO

for i in range(n) durchläuft die Zahlen von 0 bis n-1, also insgesamt n Zahlen. Die Verdichtungsstellen der Linien bilden eine *quadratische Bézierkurve*.

RANGE() MIT ZWEI PARAMETERN



Die range-Funktion kann auch **zwei Parameter** haben. In diesem Fall ist der erste Parameter der *Start-Wert* der Liste und der zweite der *Stop-Wert,* der allerdings nicht mehr in der Liste enthalten ist.

Wenn du beispielsweise *print range(2, 9)* schreibst, werden die Zahlen [2, 3, 4, 5, 6, 7, 8] in das Ausgabefenster geschrieben. Versuche es mit ein paar anderen Parametern.

Mit dem folgenden Programm zeichnest du Linien in zwei Farben mit den Startpunkten auf der x-Achse bei den Koordinaten -20 bis 20. Der Endpunkt ist bei allen Linien der Punkt (0, 40).

```
from gpanel import *

makeGPanel(-20, 20, 0, 40)

for i in range(-20, 21):
    if i < 0:
        setColor("red")
    else:
        setColor("green")
    line(i, 0, 0, 40)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Schleife **for i in range(start, stop)** mit ganzzahligem Start- und *Stop-Wert* beginnt bei i = start und endet bei i = stop - 1, wobei der Schleifenzähler i bei jedem Schleifendurchgang um 1 vergrössert wird. Dabei musst du *start* kleiner als *stop* wählen, sonst wird Schleife nie durchlaufen.

RANGE() MIT DREI PARAMETERN

Du kannst die range-Funktion sogar mit **drei Parametern** aufrufen. In diesem Fall ist der erste Parameter der Start-Wert der Liste und der zweite der *Stop-Wert*, der dritte die Wertveränderung von einem Element zum nächsten. Damit kannst du die Schrittweite, die bisher immer 1 war, der Situation anpassen.

Wenn du beispielsweise *print range(2, 15, 3)* schreibst, werden die Zahlen [2, 5, 8, 11, 14] in das Ausgabefenster geschrieben.

In der nebenstehenden Grafik zeichnest du mit gefüllten Rechtecken eine auf der Spitze stehende Pyramide. Das kleinste Rechteck hat die Breite 2, das grösste 40.

```
from gpanel import *

makeGPanel(0, 40, 0, 40)

setColor("red")
y = 1
for i in range(2, 41, 2):
    move(20, y)
    fillRectangle(i, 1.5)
    y = y + 2
```

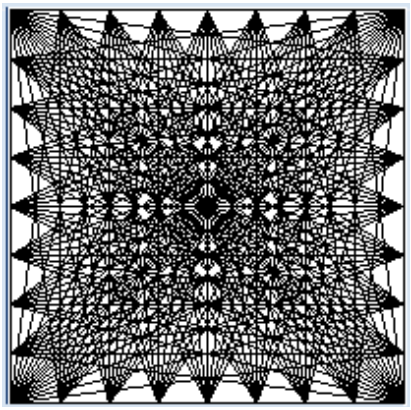
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Schleife **for i in range(start, stop, step)** beginnt bei *i* = *start* und endet bei einem Wert, der kleiner ist als *stop*. *i* wird bei jedem Schleifendurchgang um *step* vergrössert. Für die Werte *start, stop* und *step* kannst du auch negativen Zahlen wählen. Bei negativen *step* wird i bei jedem Durchlauf um *step* verkleinert; der letzte Wert ist grösser als *stop*.

VERSCHACHTELTE FOR-SCHLEIFEN (Moiré)

Eng übereinander gezeichnete Linien können einen optischen Effekt erzeugen, den man Moiré-Muster nennt. Du zeichnest in einem Quadrat von 10 Punkten des unteren Rands je eine Linie zu 10 Punkten des oberen Rands. Dann machst du das gleiche vom linken zum rechten Rand.



```
from gpanel import *

makeGPanel(0, 10, 0, 10)

for i in range(0, 11):
    for k in range (0, 11):
        line(i, 0, k, 10)
        delay(100)
for i in range(0, 11):
    for k in range (0, 11):
        line(0, i, 10, k)
        delay(100)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

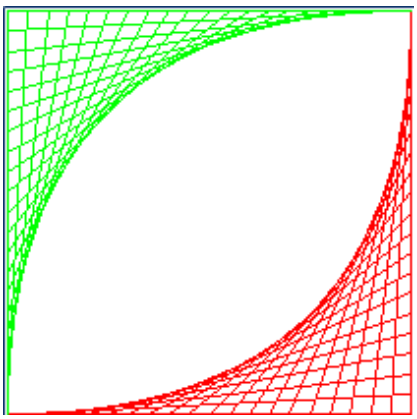
MEMO

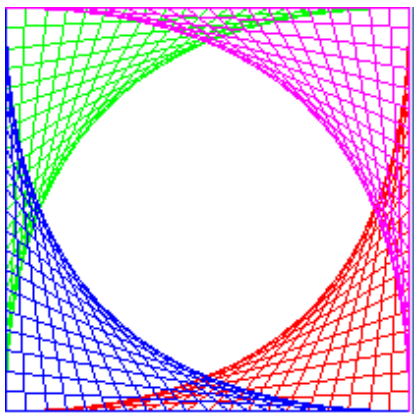
Das Programm ist nicht ganz leicht zu verstehen, aber wichtig. Am besten gehst du davon aus, dass die Schleifenvariable i der äusseren Schleife einen konstanten Wert hat (zuerst 0). Mit diesem Wert wird die innere Schleife durchlaufen, also für die Werte k = 0 bis einschliesslich 10. Danach wird i auf 1 gesetzt und die innere Schleife mit diesem Wert von i erneut durchlaufen usw.

Durch den Befehl **delay**(millisec) wartet das Programm zwischendurch diese Anzahl Millisekunden, so dass du beobachten kannst, wie das Muster entsteht.

AUFGABEN

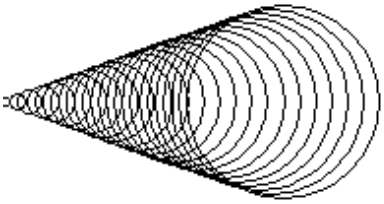
1. Eine noch schönere Grafik als im Beispiel 1 erhältst du, wenn du Farben verwendest. Zeichne auch eine zweite Linienschar mit einer anderen Farbe (Abbildung a).

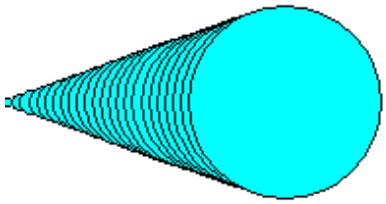
(a) 

(b) 

Die blaue Linienschar (Abbildung b) ist mit *line(i, 0, 0, 20 - i)* gezeichnet. Kannst du auch die violette programmieren?

2. Zeichne eine Kreisschar.





Die farbige Kreischar kannst du mit folgendem Vorgehen zeichnen: Du zeichnest zuerst einen gefüllten Kreis mit dem Radius y, dann wählst du schwarze Farbe und zeichnest einen Kreis mit dem gleichen Radius:

```
setColor("cyan")
fillCircle(y)
setColor("black")
circle(y)
```

3. Im Beispiel 3 haben wir eine Pyramide gezeichnet, die auf dem Kopf steht. Zeichne eine "richtige" Pyramide mit drei Farben. Dabei kannst du eine for-Schleife mit Rückwärtszählen verwenden.

3.3 STRUKTURIERTES PROGRAMMIEREN

EINFÜHRUNG

Für das Programmieren ist das Konzept der Variablen sehr wichtig. Daher musst du dir besondere Mühe geben, es möglichst vollständig zu verstehen. Du weisst bereits, dass eine Variable ein Speicherplatz ist, der mit einem Namen angesprochen wird und in dem sich ein Wert befindet. Du weisst auch, dass Parameter als "flüchtige" Speicherplätze aufgefasst werden können, die beim Aufruf der Funktion einem Wert erhalten, auf den die Funktion während ihrer Abarbeitung zugreifen kann.

PROGRAMMIERKONZEPTE: *Konstanten, Prozedurale Programmierung, Wiederverwendbarkeit*

EIN MOSAIK AUS 10x10 STEINEN

Du hast die Aufgabe, ein schönes farbiges Mosaik mit quadratischen Steinen zu erstellen. Du erhältst verschiedenfarbige Mosaiksteine mit der Seitenlänge 10 und sollst diese auf einen Boden mit der Grösse 400x400 exakt aneinander legen. Du bist etwas faul und überlässt dem Computer diese Aufgabe, schreibst ihm aber vor, dass er die Steine mit **zufälligen Farben** zeilenweise legen soll. Damit du ihm beim Legen zusehen kannst, baust du mit **delay(1)** eine kurze Arbeitspause nach jedem gelegten Stein ein.

```
from gpanel import *
makeGPanel(0, 400, 0, 400)

for y in range(0, 400, 10):
    for x in range(0, 400, 10):
        setColor(getRandomX11Color())
        move(x + 5, y + 5)
        fillRectangle(10, 10)
        delay(1)
```

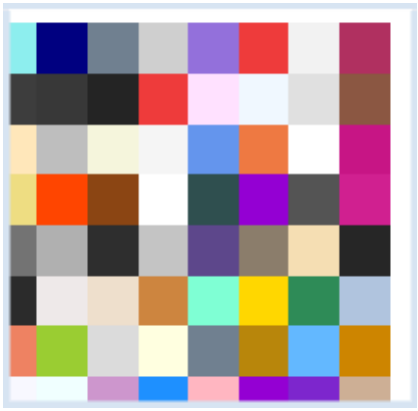
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Immer wenn ein Gitter durchlaufen wird, eignen sich zwei ineinander geschachtelte for-Schleifen. Überlege dir genau, warum tatsächlich die Steine zeilenweise von unten nach oben gelegt werden. Du benötigst bei *move()* eine Verschiebung, weil die Steine mittig verankert werden.

Die Methode **getRandomX11Color()** gibt dir (als Wort) eine der Farben aus der X11-Farbpalette zurück, die du dann *setColor()* übergeben kannst. Du lässt zuerst die Funktion *getRandomX11Color()* und dann die Funktion *setColor()* ausführen.

MAGISCHE ZAHLEN



Zwei Wochen später erhältst du eine Lieferung von fünf mal so grossen Steinen, die eine Seitenlänge von 50 aufweisen. Wieder soll sie der Computer auf den gleich grossen Boden legen. Was musst du am Programm ändern? Du siehst dir den früheren Code an, weisst aber natürlich nicht mehr so genau, was die einzelnen Zeilen bedeuten.

Du denkst: Es wird sicher so sein, dass überall, wo eine 10 stand, diese Zahl in eine 50 verändert werden muss. Machst du das so, so hast du falsch gedacht. Das Mosaik bedeckt leider nicht mehr den ganzen Boden.

Jetzt musst du dir wieder den ganzen Programmcode durchgehen und Zeile und Zeile verstehen, um den Fehler zu finden. Wenn das nötig ist, um ein Programm an eine neue Situation anzupassen (**Wiederverwendung**), so war dein Programm zwar richtig, aber schlecht geschrieben. Du solltest dir einen **guten Programmierstil** angewöhnen, damit du Programme leicht an neue Situationen anpassen kannst.

Wie musst du vorgehen? Statt die Steingrösse als **fixe Zahl** in den Code zu schreiben, definierst du eine Variable *size*, die du überall, wo die Steingrösse eine Rolle spielt, an Stelle der fixen Zahl einsetzt. Um das Programm noch besser zu **strukturieren**, schreibst du zudem noch für das Legen eines Steins eine eigene Funktion **drawStone()**.

```
from gpanel import *

def drawStone(x, y):
    setColor(getRandomX11Color())
    move(x + size/2, y + size/2)
    fillRectangle(size, size)

makeGPanel(0, 400, 0, 400)

size = 50

for x in range(0, 400, size):
    for y in range(0, 400, size):
        drawStone(x, y)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Verwendung fixer Zahlenwerte, die über das Programm verteilt eingesetzt werden, führt zu schlecht wiederverwendbaren Programmen. Statt dessen sollte man Variablen definieren und diese an Stelle der Zahlen verwenden. Um anzudeuten, dass diese nirgends verändert werden dürfen, schreibt man solche Variablen manchmal in **Grossbuchstaben** und nennt sie **Konstanten**. Auf eine Variable, die im Hauptblock definiert wird, kann auch in jeder Funktion lesend zugegriffen werden. Wir nennen eine solche Variable darum auch eine **globale Variable**.

Den Code für längere in sich abgeschlossene Aktionen sollte man in eine eigene Funktion verpacken. Dies hat mehrere Vorteile: Zum einen erkennt man am Funktionsnamen, was die Funktion tun soll, zum zweiten kann man sie mehrmals aufrufen, ohne den Code neu schreiben zu müssen und zum dritten wird das Programm damit übersichtlicher und verständlicher. Diese Art der Programmierung nennt man **strukturierte Programmierung** (oder auch **prozedurale Programmierung**) und sie ist ein wichtiges Merkmal eines guten Programmierstils.

AUFGABEN

1. Wie du mit dem Satz von Pythagoras herausfinden kannst, zeichnet der Befehl

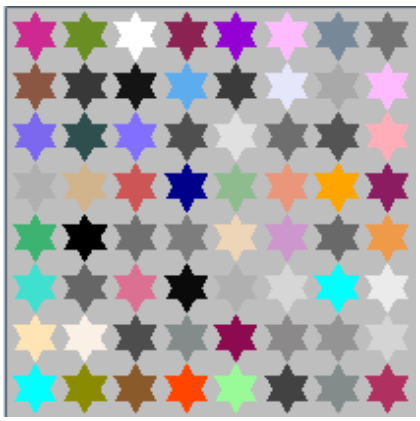
   ```
   fillTriangle(x - math.sqrt(3)/2 * r, y - r/2,
                x + math.sqrt(3)/2 * r, y - r/2,
                x, y + r);
   ```

   ein gleichseitiges Dreieck mit dem Umkreismittelpunkt bei (x, y) und dem Umkreisradius r. Verifiziere dies in einem GPanel mit dem Koordinatensystem -1..1 für beide Achsen, das ein Dreieck mit dem Umkreismittelpunkt im Ursprung zeichnet.

2. Verwende den Code in von Aufgabe 1 und definiere eine Funktion stern(x, y, r), die mit zwei gleichseitigen Dreiecken einen Stern mit Mittelpunkt (x, y) und der Grösse r zeichnet. Zeichne damit einige Sterne.

3. Erweitere die Funktion stern() mit einem Parameter, der die Farbe des Sterns festlegt. Zeichne damit auf einem grauen Hintergrund ein Sternmosaik mit 50x50 Sternen. Verwende für die Sterngrösse eine Konstante.

   Sorge dafür, dass keine Sterne mit der Hintergrundfarbe gezeichnet werden.



3.4 FUNKTIONEN MIT RÜCKGABEWERT

EINFÜHRUNG

Du weisst bereits, wie man eine Funktion mit oder ohne Parameter definiert und sie aufruft. Aus der Mathematik weisst du wahrscheinlich, dass dort Funktionen etwas anders verstanden werden. Eine Funktion y = f(x) hat in der Mathe eine unabhängige Variable x. Zu jedem Wert von x liefert die Funktion einen Wert der abhängigen Variablen y. Ein Beispiel ist die quadratische Funktion
$y = x^2$. Für x = 0, 1, 2, 3 ergeben sich die Quadratzahlen 0, 1, 4, 9.

Auch in Python kann man Funktionen definieren, die einen Wert berechnen und diesen wie eine Variable "zurückgeben".

PROGRAMMIERKONZEPTE: *Rückgabewert einer Funktion, Diskretisierung*

DAS SCHLÜSSELWORT RETURN

Du kannst eine Funktion *quadratzahl(x)* definieren, die wie in der Mathematik zu einem Wert des Parameters x die Quadratzahl x * x berechnet und sie zurück gibt. Die Rückgabe erfolgt mit dem Schlüsselwort *return*. In einem GPanel zeichnest du dann den Funktionsgraf. In der Grafik verwendest du am einfachsten *draw(x, y),* das eine Strecke von der letzten Position des Grafikcursors zu (x, y) zeichnet und den Grafikcursor auf (x, y) setzt. Nach Erscheinen des GPanel-Fenster steht der Grafikcusor bei (0, 0). Du musst ihn zuerst mit *move()* an den Startpunkt der Grafik setzen, sonst siehst du eine fehlerhafte Anfangslinie.

```
from gpanel import *
makeGPanel(-25, 25, -25, 25)

def quadratzahl(x):
    y = x * x
    return y

for x in range(-5, 6):
    y = quadratzahl(x)
    if x == -5:
        move(x, y)
    else:
        draw(x, y)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

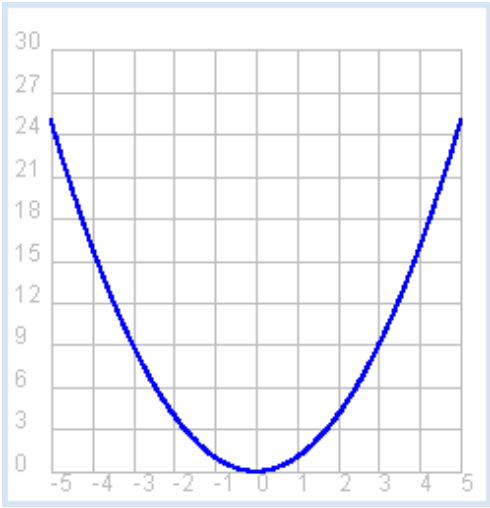
Mit **return** gibt eine Funktion einen Wert an den Aufrufer zurück und hört mit der weiteren Abarbeitung auf. Eine Funktion kann wie in der Mathematik nicht mehrere Werte zurückgeben [**mehr...**].

Wie du gesehen hast, gibt es aber im Gegensatz zur Mathematik in der Informatik auch Funktionen, die keinen Wert zurückgeben, aber doch etwas bewirken. Funktionen können sogar beides tun, etwas bewirken und etwas zurückgeben [**mehr...**].

Diese graphische Darstellung der Quadratfunktion ist noch nicht sehr schön. Neben dem fehlenden Koordinatensystem fällt auch der "eckige" Verlauf unangenehm auf, der darauf zurückzuführen ist, dass du die Funktion nur an wenigen ganzzahligen Stützpunkten berechnest, die du mit Geradenstücken verbindest. Hier zeigt sich eine wesentliche Schwäche der Informatik gegenüber der Mathematik: Obschon die Funktion für jeden Wert der x-Achse (für jede reelle Zahl) einen y-Wert liefert, können wir sie in der Informatik nur an endlich vielen Punkten berechnen. Wir sprechen davon, dass die kontinuierliche x-Achse in **diskrete Punkte aufgelöst** wird.

DEZIMALZAHLEN (FLOATS)

Immerhin können wir die Darstellung etwas schöner machen, wenn wir die Berechnungspunkte auf der x-Achse eng nebeneinander wählen. Du kannst beispielsweise den Bereich -5 bis 5 in Hundertstel-Schritten durchlaufen. Als weitere Verbesserung zeichnest du noch ein Koordinatengitter ein.



Leider kannst du in Python eine for-Schleife nur mit ganzzahligen Werten durchlaufen. Brauchst du eine feinere Auflösung, so benötigst du eine while-Schleife. Dabei wird zur x-Koordinate bei jedem Schritt 0.01 dazu addiert. Python fasst x nun nicht mehr als ganze Zahl, sondern als Dezimalzahl (float) auf.

```
from gpanel import *
makeGPanel(-6, 6, -3, 33)
setColor("gray")
drawGrid(-5, 5, 0, 30)

def quadratzahl(x):
    y = x * x
    return y

setColor("blue")
lineWidth(2)
x  = -5
while x < 5:
    y = quadratzahl(x)
    if x == -5:
        move(x, y)
    else:
        draw(x, y)
    x = x + 0.01
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

In Python werden Dezimalzahlen **float** genannt. Im Unterschied zu den Dezimalzahlen in der Mathematik haben sie in der Informatik allerdings immer eine bestimmte (endliche) Ziffernzahl. In Python sind es rund 14 Ziffern (solche Zahlen werden in anderen Programmiersprachen **double** genannt.) Man kann in einem Computerprogramm beispielsweise die Zahl π, die ja ein unendlicher Dezimalbruch ist, nie genau angeben, sondern nur mit einem float auf rund 14 Ziffern genau.

Falls du ein Koordinatengitter brauchst, so gehst du wie folgt vor:

- Du vergrösserst den Koordinatenbereich links und rechts, sowie oben und unten um 10% (statt -5 bis 5 nimmst du -6 bis 6, statt 0 bis 30 nimmst du -3 bis 33)
- Du rufst *drawGrid()* mit 4 Parameterwerten auf, die dem tatsächlich verwendeten Koordinatenbereich entsprechen. Dabei entstehen 10 Koordinatenfelder.

AUFGABEN

1. Definiere die Funktion mean(a, b), welche das arithmetische Mittel der zwei Parameterwerte zurückgibt. Teste sie mit der Console aus.

2. Untersuche das Verhalten der Funktion y = cos(x). Wie unterscheidet sie sich von y = sin(x).

3. Stelle in einem GPanel den Graf der Funktion y = sin(5x) im Bereich 0 bis 2π mit einer Auflösung 0.01 dar (π kannst du mit math.pi erhalten).
   Nehme in der Sinusfunktion statt 5 einen anderen Wert. Welchen Zusammenhang gibt es zwischen dieser Zahl und dem Graf?

4. Definiere die Funktion f(x) = 1 / sin(x) und stelle sie mit GPanel im Bereich -5 .... 5 (für beide Achsen) mit einer Auflösung von 0.001 dar. Zeichne mit einer anderen Farbe auch die Koordinatenachsen ein. Was stellst du Interessantes fest?

3.5 GLOBALE VARIABLEN, ANIMATIONEN

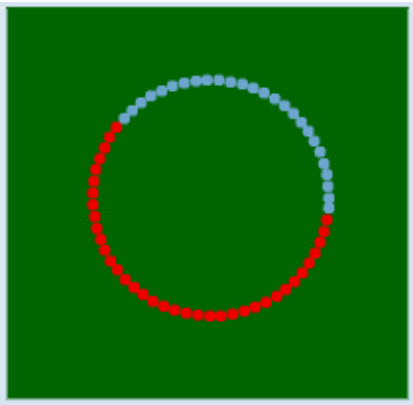
EINFÜHRUNG

Die Computergrafik wird vielfach zur Darstellung von zeitlich veränderlichen Inhalten verwendet. Damit kannst du einen Ablauf in der Physik oder Biologie simulieren oder ein Computergame erstellen. Man nennt ein solches Programm allgemein eine Animation. Um den zeitlichen Ablauf sichtbar zu achten, wird immer nach einem gleich grossen Zeitschritt, den man Animationsschritt nennt, ein neues Bild gezeichnet.

PROGRAMMIERKONZEPTE: *Globale Variablen, Nebenwirkungen (Seiteneffekte), Doppelpufferung*

VERÄNDERUNG GLOBALER VARIABLEN

Du willst einen Ball darstellen, der sich auf einem Kreis bewegt. Eine Kreisbewegung mit dem Radius 1 kriegst du, indem du die x-Koordinate mit einem ansteigenden Parameter t, welcher der fortlaufenden Zeit entspricht, mit der Cosinusfunktion und die y-Koordinate mit der Sinusfunktion berechnest, also x = cos(t) und y = sin(t) . Willst du einen anderen Radius, so musst du die beiden Werte noch mit dem Radius multiplizieren.



In der Funktion **step()** wird die Situation zu jedem Animationsschritt gezeichnet. Wenn der Ball den Kreis einmal umrundet hat, soll sich seine Farbe verändern.

Es ist üblich, im Hauptprogramm eine endlose **Animationsschleife** einzuführen, die *step()* immer wieder aufruft. Durch Einbau einer Verzögerung, kann man die Geschwindigkeit der Animation verändern. In *step()* soll die **globale Variable t** in jedem Schritt erhöht und beim Erreichen von 2P wieder auf Null gesetzt und die Farbe verändert werden.

```
import math
from gpanel import *

def step():
    global t
    x = r * math.cos(t)
    y = r * math.sin(t)
    t = t + 0.1
    if t > 6.28:
        t = 0
        setColor(getRandomX11Color())
    move(x, y)
    fillCircle(10)

makeGPanel(-500, 500, -500, 500)
bgColor("darkgreen")

t = 0
r = 200

while True:
    step()
    delay(10)
```

MEMO

Python verbietet, in Funktionen den Wert von globalen Variablen zu verändern. Wir können dieses Verbot aber umgehen, indem wir in der Funktion die Variable mit dem Schlüsselwort **global** versehen.

Der Bezeichner *global* birgt Gefahren: Jede beliebige Funktion kann nicht nur eine als global bezeichnete Variable verändern, sondern sie sogar erzeugen, wie folgendes Beispiel zeigt:

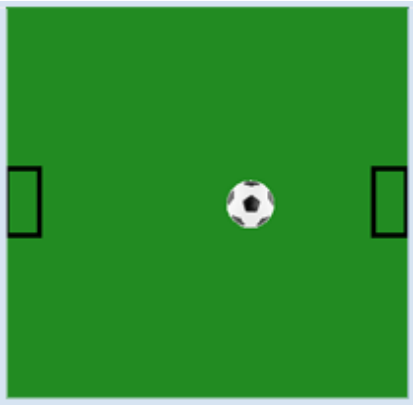
```
def set():
    global a
    a = 2

def get():
    print "a =", a
set()
get()
```

Da *set()* eine Variable *a* erzeugt, die im ganzen Programm sichtbar ist, sagt man, dass die Funktion *set()* **Nebenwirkungen** (auch *Seiteneffekte* gennant) habe. Beachte auch, wie man elegant mit der print-Anweisung durch Kommatrennung mehrere Grössen hintereinander in die Console schreiben kann. Beim Ausschreiben wird das Komma durch einen Leerschlag ersetzt.

DER TRICK MIT DEM EXAKTEN TICK

Die Animationsschleife sollte in möglichst in gleichen Zeitticks, d.h. mit der gewünschten Animationsperiode, durchlaufen werden, da sonst die Bewegung ruckelt. In **step()** wird jeweils der neue Animationszustand aufgebaut und dies kann unterschiedlich viel Zeit in Anspruch nehmen, da eventuell nicht immer gleiche Codeteile durchlaufen werden, aber auch deswegen, weil der Computer im Hintergrund noch mit anderen Aufgaben beschäftigt ist, was die Ausführung des Python-Codes verzögert. Um das verschieden lange dauernde *step()* **auszugleichen**, wird daher folgender Trick angewendet, auf den du auch selbst gekommen wärst: Man merkt sich in der Variable *startTime* vor dem **Aufruf von step()** die aktuelle Uhrzeit. Nach der Rückkehr von *step()* bildet wartet man in einer Warteschleife so lange, bis die **Differenz** der neuen Uhrzeit und der Startzeit die Animationsperiode erreicht.

Das Programm bewegt einen Fussball von Tor zu Tor. Dabei verwendest du ein Bild **football.gif**, das sich im Verzeichnis *_sprites* der TigerJython-Distribution befindet. Du kannst aber auch ein eigenes Bild nehmen, indem du die Datei in ein entsprechendes Verzeichnis auf deinem Computer kopierst und den Dateipfad als Parameter in i*mage()* angibst (absolut oder relativ zum Verzeichnis, in dem sich dein Programm befindet).



```
from gpanel import *
import time

def step():
    global x
    global v
    clear()
    lineWidth(5)
    move(25, 300)
```

```
    rectangle(50, 100)
    move(575, 300)
    rectangle(50, 100)
    x = x + v
    image("_sprites/football.gif", x, 275)
    if x > 500 or x < 50:
        v = -v

makeGPanel(0, 600, 0, 600)
bgColor("forestgreen")
enableRepaint(False)

x = 300
v = 10

while True:
    startTime = time.clock()
    step()
    repaint()
    while (time.clock() - startTime)  < 0.020:
        delay(1)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du kannst mit *time.clock()* die aktuelle Zeit als Dezimalzahl erhalten. Der erhaltene Wert ist zwar computerabhängig (Prozessorzeit oder Zeit seit dem ersten Aufruf von *clock()*). Da du aber nur die Zeitdifferenz benötigst, spielt dies keine Rolle. Du speicherst die Zeit vor dem Aufruf von *step()* und wartest am Ende der Animationsschleife mit einem *delay(1)* so lange, bis die Zeitdifferenz zwischen aktueller Zeit und Startzeit die Animationsperiode (in Sekunden) erreicht. Merke dir diesen Trick, denn du wirst ihn für viele Prozesse, die möglichst periodisch ablaufen sollen, anwenden.

Im GPanel ist jeder Grafikbefehl im Bildschirmfenster sofort sichtbar. Das Löschen mit **clear()** bei Animationen zeigt daher auch kurzzeitig ein leeres Grafikfenster, was zu einem Flackereffekt führen kann. Um dies zu vermeiden, sollte bei Animationen die **Doppelpufferung** angewendet werden.

Man erreicht dies durch den Befehl **enableRepaint(False)**, der bewirkt, dass die Grafikbefehle nur noch in einem Hintergrund-Puffer (offscreen buffer) ausgeführt werden und nicht mehr automatisch im Grafikfenster sichtbar sind. Auch *clear()* löscht dann nur noch den Hintergrund-Puffer. Das Anzeigen des Grafikpuffers auf dem Bildschirm (**Rendern** genannt) musst du dann selbst im richtigen Moment durch Aufruf von **repaint()** auslösen.

Auch in diesem Programm musst du in der Funktion *step()* die Variablen x und v mit global kennzeichnen, da sie in der Funktion verändert werden.

AUFGABEN

1. Wenn man die x- und y-Koordinate nicht wie in deinem ersten Programm mit einer gewöhnlichen Cosinus- bzw. Sinusfunktion bewegt, sondern unterschiedlich schnell, so entstehen interessante Kurvenmuster, die man Lissajoux-Figuren nennt. Zeichne solche Figuren mit einer Auflösung von 1/1000 im Bereich t = 0 bis 2π mit der Wahl

   x = cos(4.5 * t) und y = sin(6.3 * t)

2. Verwende statt fixe Zahlen die Variablen omega_x und omega_y und zeichne die Figur für folgende Werte:

omega_x	omega_y
3	5
3	7
5	7

Erkennst du einen Zusammenhang zwischen der Figur und den Werten von omega_x und omega_y?

3. Zeichne die Lissajoux-Figur mit omega_x = 2 und omega_y = 7 im Bereich t = 0 bis 2π mit einer Auflösung von 1/100 in einem GPanel mit den Koordinaten -2 bis 2 (beide Achsen). Statt die Punkte mit Linien zu verbinden, zeichnest du nun im jedem Punkt einen Kreis mit dem Radius 0.2. So entstehen "federartige" Figuren. Wie du im Bild siehst, kannst du die Kreise einfarbig oder mit *getRandomX11Color()* füllen. Spiele ein wenig damit.

3.6 TASTATURSTEUERUNG

EINFÜHRUNG

Programme erhalten eine zusätzliche Interaktivität, wenn der Benutzer den Programmablauf durch Drücken von Tastaturtasten steuern kann. Obschon Tastaturbetätigungen eigentlich Ereignisse sind, die jederzeit unabhängig vom Programmablauf auftreten, werden sie im GPanel durch Abfragefunktionen erfasst.

PROGRAMMIERKONZEPTE: *Boolescher Datentyp, Spielzustand, Animation*

TASTATURSTEUERUNG

Mit dem Befehl **getKeyCodeWait()** wird das Programm angehalten, bis du eine Taste drückst. Dann liefert dir die Funktion als Rückgabewert den Tastaturcode der Taste zurück. Mit Ausnahme von einigen Spezialtasten hat jede Taste auf der Tastatur einen eigenen Zahlencode.

Du kannst mit einem einfachen Testprogramm die Tastencodes selbst herausfinden. Die Zahlencodes werden im Consolenfenster **ausgeschrieben**.

```
from gpanel import *
makeGPanel(0, 10, 0, 10)

text(1, 5, "Drücke irgendeine Taste.")
while True:
    key = getKeyCodeWait()
    print key,
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)


```
83 70 72 37 40 38 39
```

MEMO

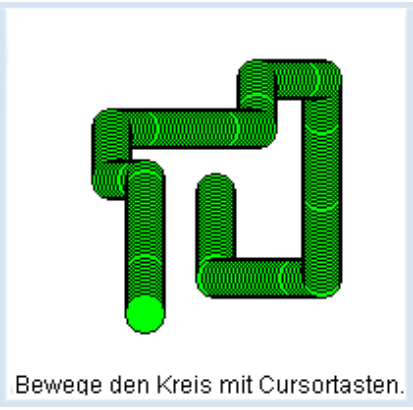
Für Tastatureingaben kannst du den Befehl **getKeyCodeWait()** verwenden. Der Computer wartet, bis du eine Taste drückst und gibt den Tastaturcode zurück.

Beachte aber, dass das GPanel-Fenster aktiv sein muss. Man sagt auch, dass es den Fokus besiten muss. Du musst in das Fenster hineinklicken, falls es den Fokus verloren hat. Nur das momentan aktive Fenster erhält die Tastaturereignisse.

FIGUREN STEUERN

Mit der Tastatur kannst du Grafikobjekte bewegen. Das Programm steuert den grünen Kreis mit Cursortasten nach links, rechts, oben oder nach unten. In einer sogenannten **Event-Loop** wartet das Programm auf einen Tastendruck und verarbeitet den erhaltenen Tastaturcode in einer verschachtelten if-else-Struktur.

Da das Zeichnen des Kreises immer wieder verwendet wird, ist es im Sinn der Strukturierten Programmierung klar, dass du dies in einer Funktion **drawCircle()**, die mehrmals aufgerufen wird, ausführst.




```
from gpanel import *

KEY_LEFT = 37
KEY_RIGHT = 39
KEY_UP = 38
KEY_DOWN = 40

def drawCircle():
    move(x, y)
    setColor("green")
    fillCircle(5)
    setColor("black")
    circle(5)

makeGPanel(0, 100, 0, 100)
text("Bewege den Kreis mit Cursortasten.")
x = 50
y = 50
step = 2
drawCircle()

while True:
    key = getKeyCodeWait()
    if key == KEY_LEFT:
        x -= step
        drawCircle()
    elif key == KEY_RIGHT:
        x += step
        drawCircle()
    elif key == KEY_UP:
        y += step
        drawCircle()
    elif key == KEY_DOWN:
        y -= step
        drawCircle()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

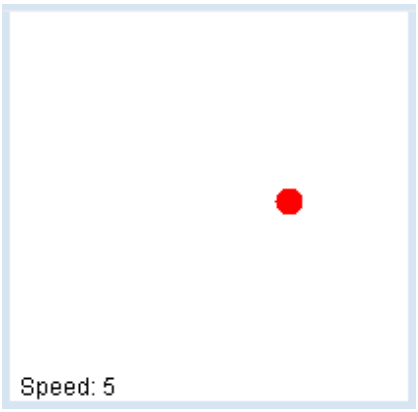
MEMO

Damit das Programm besser lesbar ist, kannst du für die Tastaturcodes der Cursortasten **Konstanten** einführen. Um sie besonders herauszuheben, sind sie sogar im Programmkopf platziert und mit Grossbuchstaben bezeichnet.

NICHTBLOCKIERENDE TASTATURABFRAGE

Wie dir sicher bekannt ist, wird die Tastatur bei Computergames häufig zur Steuerung des Spielablaufs verwendet. Hier kannst du natürlich die blockierende Funktion *getKeyCodeWait()* nicht verwenden, da sonst ja das Game anhalten würde. Du brauchst vielmehr eine Funktion, welche dir die Information liefert, ob eine Taste gedrückt wurde, aber sofort zurückkehrt.

Hast du tatsächlich eine Taste bedient, so verarbeitest du dieses Ereignis, sonst wird das Game normal weitergeführt.



Du willst die Geschwindigkeit eines sich hin- und her bewegenden Balls mit der Buchstabentaste 's' (für slow) verkleinern und mit 'f' (für fast) vergrössern, allerdings nur bis zu einem maximalen Wert. Dein Augenmerk musst du wieder auf die Event-Loop richten, in der alles Wesentliche passiert. Dort wird mit *kbhit()* periodisch abgefragt, ob eine Taste gedrückt wurde. Ist dies der Fall, liefert **kbhit()** True zurück und du kannst mit **getKeyCode()** den Tastaturcode holen.

```
from gpanel import *
import time

KEY_S = 83
KEY_F = 70

makeGPanel(0, 600, 0, 600)
title("Key 'f': faster, 's': slower")

enableRepaint(False)
x = 300
y = 300
v = 10
vmax = 50
isAhead = True

while True:
    startTime = time.clock()

    if kbhit():
        key = getKeyCode()
        if key == KEY_F:
            if v < vmax:
              v += 1
        elif key == KEY_S:
            if v > 0:
              v -= 1

    clear()
    setColor("black")
    text(10, 10, "Speed: " + str(v))
    if isAhead:
        x = x + v
    else:
        x = x - v
    move(x, y)
    setColor("red")
    fillCircle(20)
    repaint()
    if x > 600 or x < 0:
        isAhead = not isAhead
    while (time.clock() - startTime)  < 0.010:
        delay(1)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Da es sich um eine Animation handelt, müssen wir wieder einen **Animationstimer** verwenden, um einen möglichst periodischen Durchlauf der Event-Loop zu erhalten. In der Schleife wird der nächst folgende Spielzustand erstellt und dann mit **repaint()** im Bildschirmfenster angezeigt.

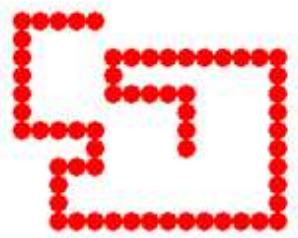
kbhit() liefert einen Wahrheitswert zurück, den wir auch als **boolean** bezeichnen. Wurde seit dem letztem Aufruf eine Taste gedrückt, so wird *True* zurückgegeben, sonst *False*.

Um den Ball nach rechts (vorwärts) zu bewegen, wird seine x-Koordinate bei jedem Durchgang der Event-Loop um v (Mass für die Geschwindigkeit) erhöht. Für die Bewegung nach links muss die Koordinate um v verkleinert werden. Vorwärts- bzw. Rückwärtsbewegung fassen wir als **Spielzustand** auf, den wir in der Variablen *isAhead* speichern.

In Python kannst du einem Wort ein zweites Wort mit dem + Zeichen anfügen, also beispielsweise ergibt "Hans" + "Peter" das Wort "HansPeter". Wenn du aber einem Wort eine Zahl zufügen willst, so muss du die Zahl zuerst mit der **str()-Funktion** konvertieren.

AUFGABEN

1. Mit den Cursortasten UP, DOWN, LEFT und RIGHT sollst du eine Schlangenlinie mit kleinen roten Kreisen zeichnen, die aneinander liegen.



2. Als Erweiterung definierst du folgende Tasten für die Farbwahl: Wenn du die Buchstabentaste *g* drückst, sollen die Kreise anschliessend grün, bei b blau und bei r wieder rot sein.

3. Erweitere dein Programm mit dem hin-und herlaufenden Ball so, dass mit den Cursor UP- und DOWN-Tasten der Ball nach oben bzw. nach unten verschoben wird.

3.7 MAUSEVENTS

EINFÜHRUNG

Bisher verstehst du den Computer so, dass er Anweisung um Anweisung ausführt. Er kann auch auf Grund von Bedingungen den Ablauf verändern und Wiederholschleifen durchlaufen. Die entsprechenden Programmstrukturen heissen **Sequenz, Selektion und Iteration.** Bereits 1966 habe Böhm und Jacopini in einem berühmten **Artikel** bewiesen, dass sich alle Rechenverfahren (Algorithmen) mit diesen drei Strukturen programmieren lassen.

Dies gilt allerdings nur, solange man keine äusseren Einflüsse mit einbezieht. Beispielsweise kann man ein Programm auch abbrechen, indem man in irgend einem Moment mit der Maus auf die Schaltfläche "Schliessen" (close-Button) klickt. Solche Vorgänge brauchen ein neues Programmierkonzept: die **Ereignissteuerung** (event handling). Das Grundprinzip hast du bereits im Kapitel Turtlegrafik/Ereignissteuerung kennengelernt. Es handelt sich dabei um Abläufe der Art:

"Wann immer das Ereignis E auftritt, führe die Funktion f aus".

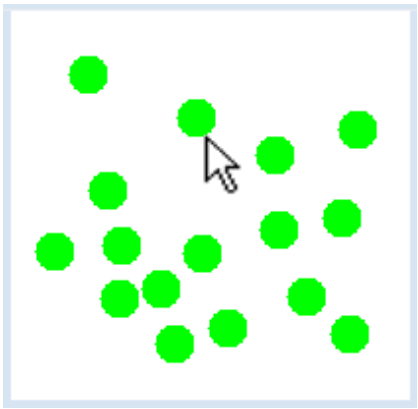
Die Implementierung ist einfach und seit den Anfängen der Computertechnik in den Fünfzigerjahren des letzten Jahrhunderts bekannt. Man definiert eine Funktion f (damals Interruptroutine genannt), welche vom eigenen Programm gar nie aufgerufen wird. Sie schläft sozusagen, bis ein bestimmtes Ereignis E auftritt und sie durch diesen äusseren Einfluss vom System automatisch aufgerufen wird. Heute nennt man eine solche Funktion **Callback** und sagt anschaulich, dass der **Callback f durch den Event E "gefeuert" wird**. Oft werden Callbacks mit Parameterwerten aufgerufen, die wichtige Informationen über den Event enthalten, beispielsweise, welcher Mausknopf gedrückt wurde oder wo sich die Maus befindet.

PROGRAMMIERKONZEPTE: *Ereignisgesteuertes Programm, Callback, Registrieren von Callbacks*

AUF EINEN MAUSEVENT REAGIEREN

Wie in der Turtlegrafik kannst du auch im GPanel Mausevents verwenden. Im ersten Beispiel wird beim Drücken der linken oder rechten Maustaste an der aktuellen Mausposition ein grüner Kreis gezeichnet. Du gehst wie folgt vor:

Zuerst definierst du in einer Funktion mit beliebig gewähltem Namen, was beim Drücken einer Maustaste geschehen soll. Du wählst einen Namen, hier z.B. **onMousePressed()**, der möglichst gut ausdrückt, was die Funktion macht. Der Callback erhält beim Aufruf durch das System als Parameterwerte die aktuellen Koordinaten des Mauscursors.



Als nächstes musst du dem System bekannt geben, dass es deinen Callback aufrufen soll, wenn der Mauskbutton gedrückt wird. Man nennt diesen Vorgang **Registrieren des Callbacks**.

Um deinen Callback zu registrieren, verwendest du einen benannten Parameter von *makeGPanel()*, der **mousePressed** heisst.

```
from gpanel import *

def onMousePressed(x, y):
    move(x, y)
    fillCircle(0.02)

makeGPanel(mousePressed = onMousePressed)
setColor("green")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

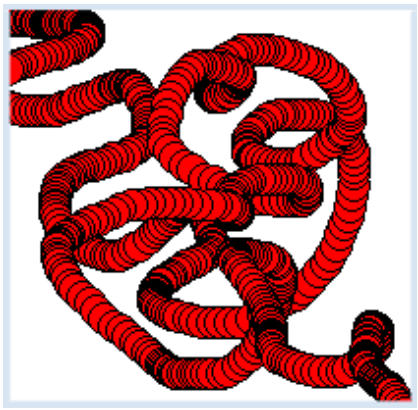
MEMO

Ein Callback wird **nicht** durch das eigene Programm aufgerufen, sondern automatisch, wenn der Event ausgelöst wird. Die Registrierung des Callbacks erfolgt durch einen benannten Parameter.

Das Drücken einer Maustaste kannst du mit zwei verschiedenen Callbacks erfassen: einem Click-Event oder einem Press-Event. Der Click-Event wird erst ausgelöst, nachdem die Taste wieder losgelassen wird, der Press-Event bereits beim Drücken der Taste.

MAUSBEWEGUNG ERFASSEN

Auch die Mausbewegung kann man als Event auffassen, der bei der Bewegung der Maus in rascher Abfolge ausgelöst wird. Der benannte Parameter heisst **mouseMoved**. Dein Programm zeichnet bei jedem Aufruf des Callbacks einen rot gefüllten Kreis mit einer schwarzen Umrandung, wodurch sich lustige röhrenartige Bilder zeichnen lassen.



```
from gpanel import *

def onMouseMoved(x, y):
    move(x, y)
    setColor("red")
    fillCircle(.04)
    setColor("black")
    circle(.04)

makeGPanel(mouseMoved = onMouseMoved)
```

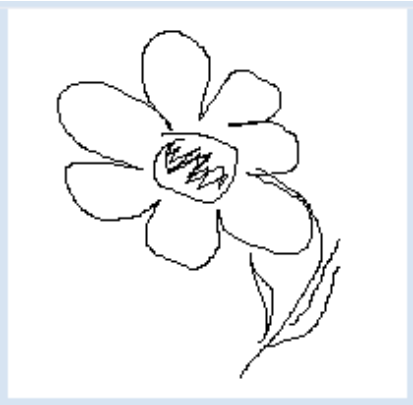
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Der Callback **onMouseMove(x, y)** wird durch einen benannten Parameter *mouseMoved* registriert.

FREIHANDZEICHNEN MIT GEDRÜCKTER MAUSTASTE

Jetzt bist du bereits in der Lage, ein einfaches Zeichnungsprogramm zu schreiben, mit dem du mit der Maus freihändig eine Figur zeichnen kann. Dazu brauchst du noch den Drag-Event, der dann in rascher Abfolge ausgelöst wird, wenn du die Maus mit **gedrückter** Taste bewegst. Die Programmlogik ist ganz einfach: Du setzt den Grafikcursor beim **Press-Event** auf den aktuellen Ort und zeichnest im **Drag-Event** mit *draw()* eine Linie.



```
from gpanel import *

def onMousePressed(x, y):
    move(x, y)

def onMouseDragged(x, y):
    draw(x, y)

makeGPanel(mousePressed = onMousePressed,
           mouseDragged = onMouseDragged)
```

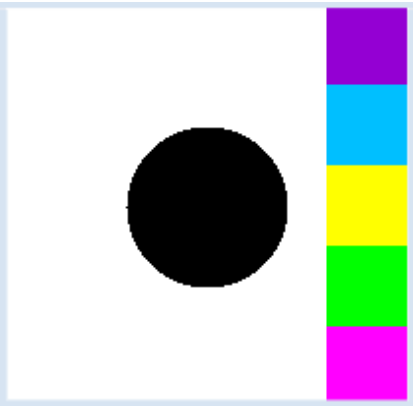
MEMO

Man kann **mehrere Callbacks** mit benannten Parameter gleichzeitig registrieren. Die Reihenfolge der Parameter ist unwesentlich.

LINKE UND RECHTE MAUSTASTE

Wie du sicher gemerkt hast, werden die Mausevents sowohl mit der linken als auch rechten Maustaste ausgelöst. Willst du zwischen der linken und rechten Maustaste unterscheiden, verwendest du Funktionen **isLeftMouseButton()** und **isRightMouseButton()**. Diese geben *True* zurück, wenn die linke bzw. rechte Maustaste beteiligt ist.

Das Programm öffnet beim Drücken der **rechten Maustaste** ein Farbpalette. Mit der **linken Maustaste** kannst du die Füllfarbe des Kreises auswählen.



```
from gpanel import *

def onMousePressed(x, y):
  if isLeftMouseButton():
       pixColor = getPixelColor(x, y)
       if pixColor == makeColor("white"):
           return
       clear()
       setColor(pixColor)
       move(5, 5)
       fillCircle(2)

  if isRightMouseButton():
```

```
        for i in range(5):
            move(9, 2 * i + 1)
            if i == 0:
                setColor("deep pink")
            if i == 1:
                setColor("green")
            if i == 2:
                setColor("yellow")
            if i == 3:
                setColor("deep sky blue")
            if i == 4:
                setColor("dark violet")
            fillRectangle(2, 2)

makeGPanel(0, 10, 0, 10, mousePressed = onMousePressed)
move(5, 5)
fillCircle(2)
```

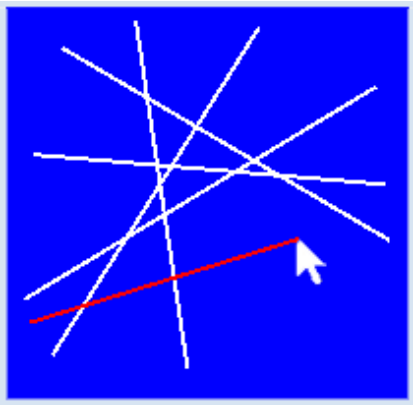
MEMO

Die registrierten Maus-Callbacks werden mit der linken und rechten Maustaste ausgelöst. Du kannst mit **isLeftMouseButton()** bzw. **isRightMouseButton()** herausfinden, welche Taste es war.

GUMMIBANDLINIEN

Wenn du mit einem Zeichnungsprogramm Linien zeichnen möchtest, so markierst du durch Drücken der Maustaste den Anfangspunkt. Dann legst du beim Ziehen der Maus eine provisorische Linie wie ein Gummiband, das am Anfangspunkt festgemacht ist. Erst wenn du mit der Lage der Linie zufrieden bist, lässt du die Maustaste los und die Linie wird definitiv gezeichnet.

Du benötigst hier also drei Callback: **onMousePresssed**, **onMouseDragged** und **onMouseReleased**.



Es gibt aber ein besonderes Problem: Bewegt sich das Gummiband über die Zeichnungsfläche, muss es in seiner alten Lage immer wieder gelöscht und an der neuen Lage gezeichnet werden, ohne dass dabei die bereits vorhandene Zeichnung verunstaltet wird. Würdest du die Linie löschen, indem du sie mit der Hintergrundfarbe überschreibst, so ergäben sich in der bestehenden Zeichnung bei den Überschneidungspunkten Lücken.

Um dieses Problem zu lösen, muss man im Press-Callback die bereits vorhandene Zeichnung abspeichern (man sagt auch "retten"). Das Löschen der temporären Gummibandlinie geschieht jetzt so, dass man diese "alte" Zeichnung wiederherstellt. Du kannst die Zeichnung mit **storeGraphics()** abspeichern und mit **recallGraphics()** wiederherstellen.

```
from gpanel import *

def onMousePressed(x, y):
    global x1, y1, x2, y2
    storeGraphics()
    x1 = x
    y1 = y
    x2 = x1
    y2 = y1
    setColor("red")
```

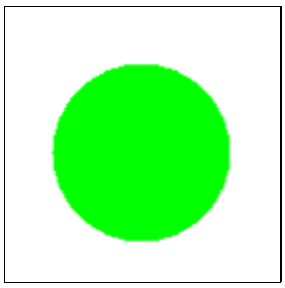
```
def onMouseDragged(x, y):
    global x2, y2
    recallGraphics()
    x2 = x
    y2 = y
    line(x1, y1, x2, y2)

def onMouseReleased(x, y):
    setColor("white")
    if not (x1 == x2 and y1 == y2):
        line(x1, y1, x2, y2)

x1 = 0
y1 = 0
x2 = 0
y2 = 0

makeGPanel(mousePressed = onMousePressed,
              mouseDragged = onMouseDragged,
              mouseReleased = onMouseReleased)
title("Press And Drag To Draw Lines")
bgColor("blue")
setColor("white")
lineWidth(2)
```

MEMO

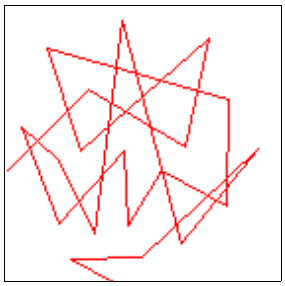
Merke dir das Prinzip zum Zeichnen von Gummibandlinien:
Beim **Press-Event** werden die Endpunkte der Linie initialisiert und die Grafik gespeichert.
Beim **Drag-Event** wird die gespeicherte Grafik wiederhergestellt, die temporäre Linie mit dem neuen Endpunkt gezeichnet und der neue Endpunkt gespeichert.
Beim **Release-Event** wird die Linie definitiv gezeichnet, allerdings nur, falls die Maus bewegt wurde.

AUFGABEN

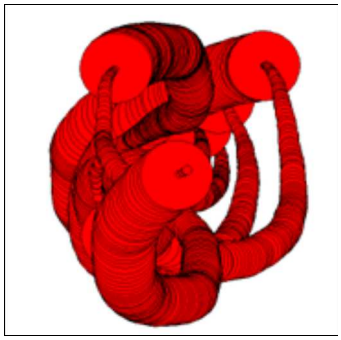
1. Zeichne einen grün gefüllten Kreis. Wenn du die Maus in den Kreis bewegst, soll die Füllfarbe auf rot wechseln. Beim Herausfahren wird sie wieder grün.Es ist vorteilhaft, die Koordinaten so zu wählen, dass der 0-Punkt in der Mitte des Fensters ist:
   makeGPanel(-10, 10, -10, 10,
   mouseMoved = onMouseMoved)



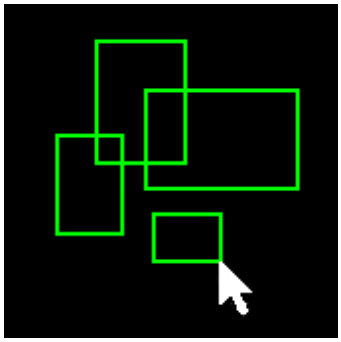
2. Dein Programm soll beim jedem Mausklick ein Linienstück zeichnen.



3. Dein Programm soll bei der Bewegung mit **gedrückter** Maustaste eine röhrenartige Figur zeichnen, wobei die Röhre bei der Bewegung von einer Anfangsdicke von 0.01 bis zu einer Dicke von 0.1 anschwellen soll, um dann wieder zur Anfangsdicke zurückzukehren.



4. Schreibe ein Programm, mit dem du auf schwarzem Hintergrund grüne Rechtecke zeichnen kannst. Dabei sollst du durch Drücken und Ziehen der Maus zuerst eine provisorisches "Gummirechteck" platzieren können, bevor es dann beim Loslassen der Maus definitiv dargestellt wird. Verwende dazu die Rechteck-Funktionen, die mit den Koordinaten von zwei gegenüberliegenden Eckpunkten des Rechtecks aufgerufen weden (rectangle(x1, y1, x2, y2).



ZUSATZSTOFF

CALLBACKS REGISTRIEREN MIT DECORATORS

Statt benannte Parameter von *makeGPanel()* zu verwenden, um einen Callback zu registrieren, kann eine beliebig genannte Funktion mit zwei Parameter x und y durch eine vorangestellte Zeile, die mit einem At-Symbol @ eingeleitet wird, so "dekoriert" werden, dass sie von TigerJython automatisch als Callback registriert und beim Eintreten des entsprechenden Events aufgerufen wird. Als Decorators stehen zur Verfügung:

@onMousePressed	Maustaste wird gedrückt
@onMouseReleased	Maustaste wird losgelassen
@onMouseClicked	Maustaste wird gedrückt und losgelassen
@onMouseDragged	Maus wird mit gedrückter Taste bewegt
@onMouseMoved	Maus wird mit nicht gedrückter Taste bewegt
@onMouseEntered	Maus tritt in das Grafikfenster ein
@onMouseExited	Maus tritt aus dem Grafikfenster aus

Das oben gezeigte Programm, welches beim Drücken einer Maustaste eine Kreis zeichnet, kannst du unter Verwendung eines Decorators auch so schreiben:

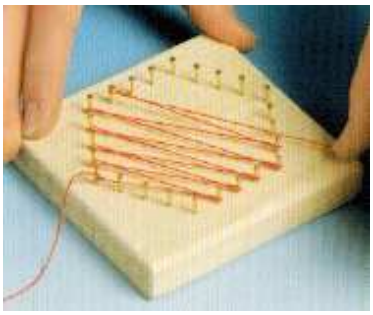
```
from gpanel import *

@onMousePressed
def doIt(x, y):
    move(x, y)
    fillCircle(0.02)

makeGPanel()
setColor("green")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

3.8 FADENGRAFIKEN

EINFÜHRUNG

Schon im Kindergarten hast du wahrscheinlich mit Fadengrafiken gespielt. Dabei musstest du gemäss einer Bastel-Anleitung entlang einer bestimmten Figur in ein Holz- oder Kartonbrett Nägel einschlagen oder Nadeln einstecken. Meist waren diese in gleichen Abständen angeordnet. Mit Fäden hast du sie dann miteinander verbunden. Wenn du genügend viele Fäden legst, erscheinen bei den Fadenverdichtungen interessante Kurven, die man in der Mathematik **Enveloppe** (auch Hüllkurve) nennt, da die Fäden Tangenten an diese Kurven sind.



Aus Täubner, Walz: Fadengrafik

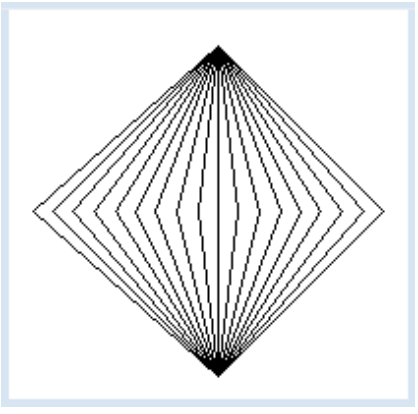
Statt die Fadengrafik selbst zu erzeugen, könntest du auch eine Maschine damit beauftragen. Diese müsste die Anleitung verstehen und dann in eine Aktion umsetzen, beispielsweise mit einem Roboterarm die Fäden ziehen oder die Fäden auf einem Bildschirm aufzeichnen. Eine solche Anleitung für eine Maschine nennt man auch **Algorithmus**. Man kann den Algorithmus zwar wie eine Bastel-Anleitung zuerst allgemein verständlich in einer Umgangssprache formulieren. Da es aber erwünscht ist, dass die Maschine bei jedem Durchlauf genau dasselbe Muster erzeugt, muss der Algorithmus so exakt formuliert sein, dass die Maschine bei jedem Schritt eindeutig weiss, was zu tun ist. Dazu hat man die Programmiersprachen erfunden und darum lernst du programmieren, denn in der natürlichen Sprache gibt es keine solche Eindeutigkeit.

PROGRAMMIERKONZEPTE: *Algorithmus, Datenstruktur, Modell, Programmeleganz, Liste, Index*

PUNKTE ALS LISTEN

Statt mit Brett, Nägeln und Fäden zu arbeiten, kannst du den Vorgang auf den Computer übertragen. Dabei machst du dir ein **Abbild der Natur**, du **modellierst** das Brett als Bildschirmfenster, die Nägel als Bildschirmpunkte und die Fäden als Linien.

Bei der Übertragung des Algorithmus in eine Programmiersprache ist es wichtig, eine möglichst enge Beziehung zur Wirklichkeit herzustellen. Nägel bzw. geometrische Punkte stellen für dich handfeste Objekte dar und so sollte es im Programm auch sein.



In der Geometrie schreibst du für einen Punkt P(x, y), wo x und y die Koordinaten sind. Im Programm können wir die zwei Zahlen x und y in eine Datenstruktur verpacken, die wir eine **Liste** nennen. Wir schreiben p = [x, y]. Der geometrische Punkt P(0, 8) wird also durch die Liste **p = [0, 8]** modelliert.

Auf die einzelnen Komponenten einer Liste kannst du mit einem **Index** zugreifen, wobei die Zählung bei 0 beginnt. Du schreibst den Index in eine eckiges Klammerpaar, also für die x-Koordinate **p[0]** und für die y-Koordinate p[1].

Das Schöne daran ist, dass alle Grafikfunktionen von GPanel "listenbewusst" sind, denn sie

funktionieren statt mit x-y-Koordinaten auch mit Punkt-Listen. Dein Programm modelliert das Ziehen der Fäden von einem Nagel A über 19 Nägel bei den Koordinaten auf der x-Achse zum Nagel B und wieder zurück. Du kannst sogar ein *delay()* einbauen, das bewirkt, dass das Fadenziehen tatsächlich eine für Menschen erfassbare Zeit lang dauert.

```
from gpanel import *

DELAY = 100

def step(x):
    p1 = [x, 0]
    draw(p1)
    delay(DELAY)
    draw(pB)
    delay(DELAY)
    p2 = [x + 1, 0]
    draw(p2)
    delay(DELAY)
    draw(pA)
    delay(DELAY)

makeGPanel(-10, 10, -10, 10)
pA = [0, 8]
pB = [0, -8]
move(pA)
for x in range(-9, 9, 2):
    step(x)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Bei der Implementierung eines Algorithmus müssen auch die Daten günstig strukturiert werden. Bei uns werden geometrische Punkte als Listen mit zwei Elementen (x- und y-Koordinaten) modelliert. Die Wahl der **Datenstruktur** beeinflusst das Programm ganz wesentlich. Niklaus Wirth, ein berühmter Informatikprofessor an der ETH Zürich, sagte treffend: **Programm = Algorithmus + Datenstruktur** [**Lit.**]
Listen können mehrere Werte, genannt Listenelemente speichern. Listen werden mit eckigen Klammern definiert. Man kann mit einem Listenindex die einzelnen Elemente lesen und ihnen neue **Werte zuweisen**.

Alle Grafikbefehle von GPanel funktionieren auch mit Punkten, die als Listen mit x- und y-Koordinaten modelliert sind.

PROGRAMMIEREN IST EINE KUNST

Du merkst sicher, dass du die eben erstellte Fadengrafik viel einfacher erzeugen kannst, wenn du die Linien unabhängig davon zeichnet, wie der Faden tatsächlich von Hand gezogen würde. Du brauchst ja nur die Punkte A und B mit Strecken zu verbinden.

```
from gpanel import *

makeGPanel(-10, 10, -10, 10)
pA = [0, 8]
pB = [0, -8]

for x in range(-9, 10, 1):
    pX = [x, 0]
    line(pA, pX)
    line(pB, pX)
```

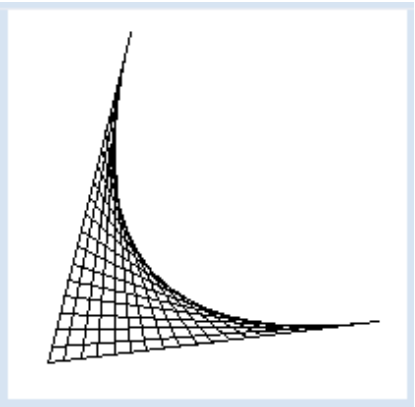
MEMO

Ein Algorithmus kann auf verschiedene Arten implementiert werden, die sich in der Länge des Codes und in der Laufzeit des Programms unterscheiden. Man spricht auch von eleganten und weniger eleganten Programmen. Merke dir, dass es nicht genügt, dass ein Programm ein richtiges Resultat hervorbringt, sondern dass es auch **elegant** geschrieben ist. Fasse darum Programmieren als eine Kunst auf.

ELEGANTE FADENGRAFIK-ALGORITHMEN

Für Fadengrafiken benötigst du oft Teilungspunkte (dividing points) einer Strecke. Dazu gibt es in GPanel eine einfache Funktion **getDividingPoint(pA, pB, r)**, der du die zwei Endpunkte pA und pB der Strecke und den Teilungsfaktor r übergibst. Sie liefert dir den Teilungspunkt als Liste zurück [**mehr...**].

Du modellierst nun eine Fadengrafik mit Nägeln auf den Seiten AB und AC mit einem besonders eleganten Programm.



```
from gpanel import *

makeGPanel(0, 100, 0, 100)

pA = [10, 10]
pB = [90, 20]
pC = [30, 90]

line(pA, pB)
line(pA, pC)

r = 0
while r <= 1:
    pX1 = getDividingPoint(pA, pB, r)
    pX2 = getDividingPoint(pA, pC, 1 - r)
    line(pX1, pX2)
    r += 0.05
    delay(300)
```

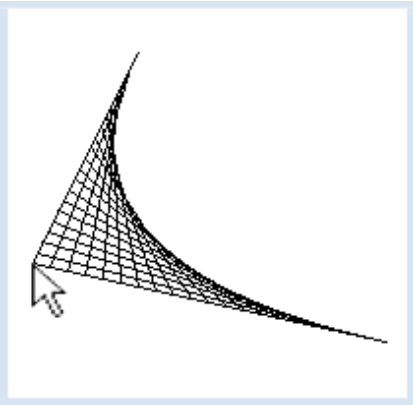
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Bibliotheksfunktionen, wie hier *getDividingPoint()*, können ein Programm stark vereinfachen. Für bestimmte wohldefinierte Teilaufgaben solltest du vorhandene Bibliotheksfunktionen verwenden, die du von deiner Programmiererfahrung her kennst, aus Dokumentationen entnimmst oder mit Web-Suchmaschinen findest.

Mathematisch betrachtet ist die entstehende Kurve eine **quadratische Bézierkurve**. Du kannst sie mit der Funktion *quadraticBezier(pB, pA, pC)* zeichnen (pB und pC sind die Endpunkte, pA der Kontrollpunkt der Kurve).

MAUSGESTEUERTE FADENGRAFIKEN



Das Modellieren von natürlichen Abläufen mit dem Computer ist nicht nur ein Spiel, sondern hat vielseitige Anwendungen. Mit dem Computer kannst du in viel kürzerer Zeit und mit viel weniger Aufwand verschiedene Situationen durchtesten, bis du eine gefunden hast, die du dann in die Praxis umsetzen willst. Dein Programm ist dann besonders attraktiv, wenn du mit der Maus Veränderungen vornehmen kannst, die sich sofort auswirken. Durch die Verwendung von Callbacks ist dies in Python mit wenig zusätzlichem Aufwand verbunden.

Du kannst in deinem Programm den Eckpunkt A mit einer Mausbewegung verschieben und die Fadengrafik wird unmittelbar neu erstellt.

Dazu verwendest du zum Erstellen der Grafik in die Funktion **updateGraphics()**, die von den **Maus-Callbacks** aufgerufen wird. Dabei löschst du jeweils das ganze Grafikfenster und erstellst es neu mit dem Punkt A an der aktuellen Lage des Mauscursors

```
from gpanel import *

def updateGraphics():
    clear()
    line(pA, pB)
    line(pA, pC)
    r = 0
    while r <= 1:
        pX1 = getDividingPoint(pA, pB, r)
        pX2 = getDividingPoint(pA, pC, 1 - r)
        line(pX1, pX2)
        r += 0.05

def myCallback(e):
    pA[0] = toWindowX(e.getX())
    pA[1] = toWindowY(e.getY())
    updateGraphics()

makeGPanel(0, 100, 0, 100,
               mousePressed = myCallback,
               mouseDragged = myCallback)

pA = [10, 10]
pB = [90, 20]
pC = [30, 90]
updateGraphics()
```

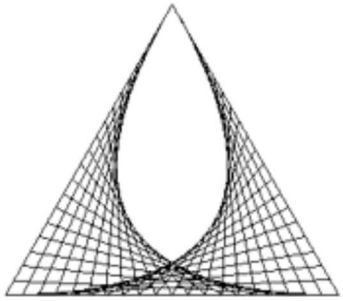
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

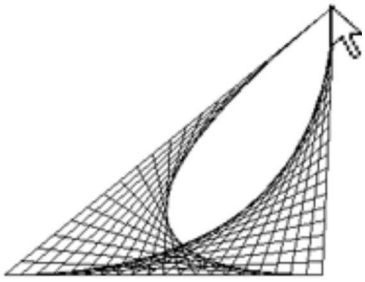
Du kannst auch zwei verschiedene Events, hier den Press-Event und den Drag-Event mit **demselben Callback** abhandeln.

AUFGABEN

1. Erstelle die nebenstehende Fadengrafik



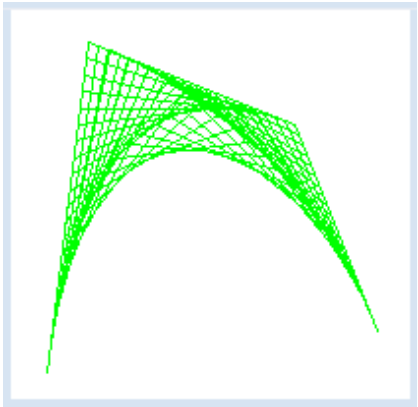
2. Ergänze die Fadengrafik von Aufgabe 1 derart, dass du mit einem Maus-Drag die Dreiecksspitze ziehen kannst und die Grafik dabei immer wieder neu gezeichnet wird.



ZUSATZSTOFF

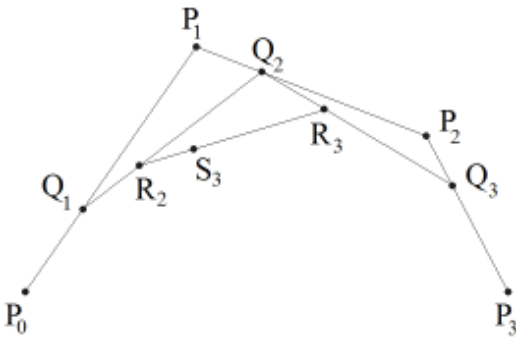
BÉZIER-KURVEN

Diese Kurven wurden in den sechziger Jahren des letzten Jahrhunderts von Pierre Bézier, damals Ingenieur beim Automobilkonzern Renault, erfunden, um ästhetisch wohlgeformte Kurven für den Design von Industrieprodukten zu erzeugen.



Du kannst eine kubische Bézier-Kurve mit dem Algorithmus von de Casteljau als Fadengrafik erzeugen.

Der Algorithmus lautet wie folgt:



- Du gibst dir 4 Punkte P0, P1, P2, P3 vor. (P0 und P3 werden die Endpunkte der Kurve sein, P1 und P2 nennt man die Kontrollpunkte).
- Verbinde P0P1, P1P2, P2P3
- Teile die Strecken P0P1, P1P2, P2P3 in äquidistante Teilungspunkte ein. Für ein bestimmtes Teilverhältnis gibt dies die Teilungspunkte Q1, Q2, Q3.
- Verbinde Q1Q2, Q2Q3
- Teile die Strecken Q1Q2, Q2Q3 im selben Teilverhältnis. Dies gibt dies die Teilungspunkte R2 und R3
- Verbinde R2R3

Du kannst den Algorithmus einfach in einem Programm implementieren, wenn du Punkte als Listen implementierst und die Funktion **getDividingPoint()** mehrmals aufrufst.

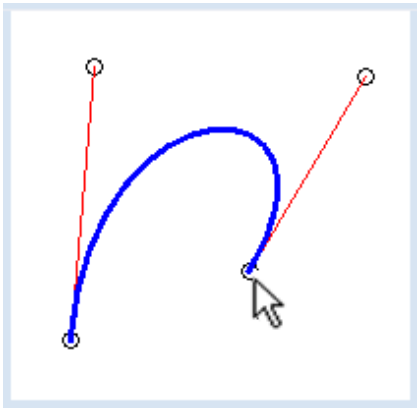
```
from gpanel import *

makeGPanel(0, 100, 0, 100)

pt1 = [10, 10]
pc1 = [20, 90]
pc2 = [70, 70]
pt2 = [90, 20]

setColor("green")

line(pt1, pc1)
line(pt2, pc2)
line(pc1, pc2)

r = 0
while r <= 1:
    q1 = getDividingPoint(pt1, pc1, r)
    q2 = getDividingPoint(pc1, pc2, r)
    q3 = getDividingPoint(pc2, pt2, r)
    line(q1, q2)
    line(q2, q3)
    r2 = getDividingPoint(q1, q2, r)
    r3 = getDividingPoint(q2, q3, r)
    line(r2, r3)
    r += 0.05

setColor("black")
#cubicBezier(pt1, pc1, pc2, pt2)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Eine kubische Bézier-Kurve ist durch 4 Punkte bestimmt. In GPanel kannst du sie mit der Funktion **cubicBezier()** zeichnen. Es werden die aktuelle Zeichnungsfarbe und Liniendicke verwendet.

INTERAKTIVER KURVENDESIGN



Kombinierst du deine Kenntnisse, so kannst du bereits ein ziemlich professionelles Programm schreiben, mit dem du ein Bézierkurve erzeugen und durch Ziehen mit der Maus interaktiv verändern kannst. Das Programm merkt sogar, wenn du mit dem Mauscursor in der Nähe eines der 4 Punkte bist und färbt diesen. Mit einem Press-Event kannst du dann diesen Punkt packen und ziehen.

Die vier Punkte müssen im Programm mehrmals durchlaufen werden. Es ist darum zweckmässig, dass du sie auch in eine Liste mit dem Namen **points** steckst, die du dann mit einer for-Struktur bearbeiten kannst.

Wichtig ist auch, dass du weisst, welchen der Punkte du gerade gepackt hast. Diese Information speicherst du in einer Variable **active** : sie hat den Wert -1, falls keiner der Punkte gepackt ist, sonst entspricht ihr Wert dem Index des betreffenden Punkts.

```
from gpanel import *

def updateGraphics():
    # erase all
    clear()

    # draw points
    lineWidth(1)
    for i in range(4):
        move(points[i])
        if active == i:
            setColor("green")
            fillCircle(2)
        setColor("black")
        circle(2)

    # draw tangents
    setColor("red")
    line(points[0], points[1])
    line(points[3], points[2])

    # draw Bezier curve
    setColor("blue")
    lineWidth(3)
    cubicBezier(points[0], points[1], points[2], points[3])

def onMouseDragged(e):
    if active == -1:
        return
    points[active][0] = toWindowX(e.getX())
    points[active][1] = toWindowY(e.getY())
    updateGraphics()

def onMouseReleased(e):
    active = -1
    updateGraphics()

def onMouseMoved(e):
    global active
    x = toWindowX(e.getX())
    y = toWindowY(e.getY())
    active = near(x, y)
    updateGraphics()

def near(x, y):
    for i in range(4):
        rsquare = (x - points[i][0]) * (x - points[i][0]) +
                  (y - points[i][1]) * (y - points[i][1])
        if rsquare < 4:
            return i
    return -1

pt1 = [20, 20]
pc1 = [10, 80]
pc2 = [90, 80]
pt2 = [80, 20]
points = [pt1, pc1, pc2, pt2]
active = -1

makeGPanel(0, 100, 0, 100,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased,
    mouseMoved = onMouseMoved)
updateGraphics()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

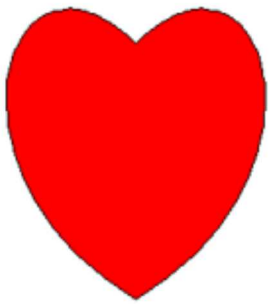
MEMO

Es gibt auch kompliziertere Datenstrukturen, wie beispielsweise eine Liste, deren Elemente wieder Listen sind. Willst du beispielsweise die x-Koordinate von P1 ansprechen, so verwendest du **points[1][0]**, also ein **doppeltes Klammerpaar**.

Heute sind die Bézierkurven ein wichtiges Designhilfsmittel im CAD-Bereich. [**Lit**.]

AUFGABEN

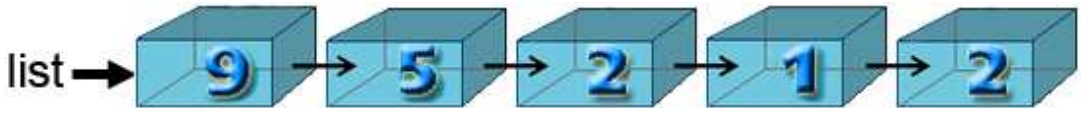
1. Das Herz besteht aus zwei kubischen Bézier-Kurven mit gleichen Anfangs- und Endpunkten sowie symmetrischen Kontrollpunkten. Skizziere auf einem Blatt Papier, wo diese Punkte etwa liegen könnten und erstelle dann die Grafik. Das Füllen erfolgt mit der Funktion *fill(punkt, alte_farbe, neue_farbe),* wo punkt ein innerer Punkt eines umrandeten Gebiets ist.



3.9 WACHSEN UND SCHRUMPFEN

EINFÜHRUNG

Manchmal musst du Werte speichern, die zusammen gehören, aber deren genaue Anzahl du beim Erstellen des Programms nicht kennst. Du brauchst also eine Datenstruktur, in der du mehrere Werte speichern kannst. Die Struktur sollte so flexibel sein, dass sie auch die **Reihenfolge** der hinzugefügten Werte berücksichtigt. Es ist naheliegend, eine Aneinanderreihung von einfachen Behältern dafür einzusetzen, von der du bereits gehört hast, nämlich eine **Liste**. Hier erfährst du nun ausführlich, wie man mit Listen umgeht.



Eine Liste mit 5 Elementen

Eine Liste besteht aus einzelnen Elementen, die nacheinander angeordnet sind. Im Gegensatz zu einer unstrukturierten Menge von Elementen gibt es also ein **erstes** und ein **letztes** Element und alle anderen Elemente haben einen **Vorgänger** und einen **Nachfolger**.

Listen (und ähnliche Container) sind für die Programmierung enorm wichtig. Die möglichen Operationen mit Listen sind sehr anschaulich. Die wichtigsten sind:

- Elemente hinzufügen (am Ende, am Anfang, irgendwo dazwischen)
- Element lesen
- Elemente verändern
- Elemente entfernen
- Alle Elemente durchlaufen
- Elemente sortieren
- Nach Elementen suchen

In Python kannst du in Listen beliebige Daten speichern, also nicht nur Zahlen. Die einzelnen Elemente können sogar einen unterschiedlichen Typ haben, du kannst also beispielsweise Zahlen und Buchstaben in der gleichen Liste speichern.

PROGRAMMIERKONZEPTE: *Container, Liste, Vorgänger, Nachfolger, Referenzvariable*

NOTENLISTE

Eine Liste kannst du wie eine Variable auffassen. Sie hat also auch einen Namen und einen Wert, nämlich ihre Elemente. Du erzeugst sie mit einem eckigen Klammerpaar, z.B. erzeugt *list* = [1, 2, 3] eine Liste mit den Elementen 1, 2 und 3. Eine Liste kann auch leer sein. Du definierst eine leere Liste mit *list = []*.

Eine typische Verwendung von Listen ist ein Notenbüchlein, wo du die Noten in einem bestimmten Schulfach eingetragen hast, sagen wir einmal Biologienoten. Zu Beginn des Semesters hast du eine leere Liste, in Python ausgedrückt **bioNoten = []**. Das Hineinschreiben von Noten entspricht dem Hinzufügen von Listenelementen, was du in Python mit dem Befehl *append()* machst, für eine Note 5 also: **bioNoten.append(5)**. Du kannst die Liste jederzeit mit einem print-Befehl ansehen, du schreibst einfach *print bioNoten*.

Willst du den Notendurchschnitt berechnen, so musst du die Liste **durchlaufen**. Das kannst du sehr einfach und elegant mit einer for-Schleife machen, denn

for note in biionoten:

kopiert der Reihe nach jeden Listenwert in die Variable *note* und du kannst diese im Schleifenkörper verwenden.

```
bioNoten = []
bioNoten.append(5.5)
print bioNoten
bioNoten.append(5)
print bioNoten
bioNoten.append(5.5)
print bioNoten
bioNoten.append(6)
print bioNoten
sum = 0
for note in bioNoten:
    sum += note
print "Schnitt: " + str(sum / len(bioNoten))
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Mit der Methode *append()* fügst du neue Elemente am Ende der Liste an.

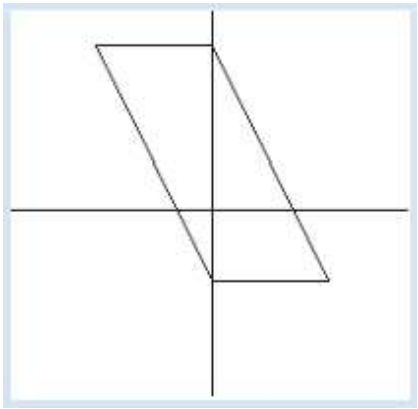
Die eingebaute Funktion **len()** gibt dir die aktuelle Länge der Liste zurück. Beachte den interessanten Trick mit der Variable **sum**, wie man die Summe bildet, um dann den Durchschnitt zu berechnen. Du kannst aber auch mit der eingebauten Funktion *sum(bioNoten)* die Summe direkt erhalten.

LISTE MIT FESTER ANZAHL VON ELEMENTEN

Oft ist dir bereits bei der Erstellung des Programms bekannt, wie lang ein Listenbehälter sein muss und dass alle Elemente denselben Datentyp haben. In vielen Programmiersprachen nennt man eine solche Datenstruktur einen **Array**. Auf die einzelnen Elemente greift man üblicherweise über ihren Index zu. In Python gibt es diesen Datentyp nicht und man verwendet dazu eine Liste.

Das Programm definiert ein Viereck als eine Liste mit 4 Eckpunkten (diese werden ebenfalls als eine Liste mit 2 Koordinaten definiert). Damit du von Anfang an mit Indizes auf die Viereckliste zugreifen kannst, erzeugst du eine Liste mit 4 Nullen *viereck = [0, 0, 0, 0].* Man kann dazu die Kurzschreibweise *viereck = [0] * 4* verwenden.

Nachher kopierst du die 4 Eckpunkte hinein. DAbei werden die Nullen durch Punktlisten ersetzt. Mit einer for-Schleife stellst du das Viereck dar.



```
from gpanel import *

pA = [0, -3]
pB = [5, -3]
pC = [0, 7]
pD = [-5, 7]

makeGPanel(-10, 10, -10, 10)
```

```
line(-10, 0, 10, 0)
line(0, -10, 0, 10)

viereck = [0] * 4  # list with 4 elements, initialized with 0
viereck[0] = pA
viereck[1] = pB
viereck[2] = pC
viereck[3] = pD

for i in range(4):
    k = i + 1
    if k == 4:
        k = 0
    line(viereck[i], viereck[k])
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

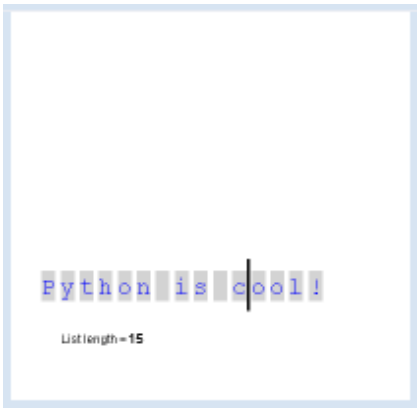
Wenn du bei der Erstellung des Programms die Listenlänge bereits kennst, erzeugst du eine Liste mit den Initialisierungswerten 0 und greifst dann über den Index auf die Elemente zu.

ELEMENTE EINFÜGEN UND LÖSCHEN

Das Programm zeigt, wie eine Textverarbeitung funktioniert. Die eingegebenen Buchstaben werden in eine Buchstabenliste eingefügt. Es ist klar, dass man zu Beginn nicht weiss, wieviele Buchstaben du eingeben wirst, also ist eine Liste die ideale Datenstruktur. Zudem siehst du einen Textcursor, der mit einem Mausklick an irgend eine Stelle des Text gesetzt werden kann.

Tippst du eine Buchstabentaste, so wird der Buchstaben rechts vom Cursors eingefügt und die Liste wächst, tippst die die Backspace-Taste, so wird der links neben dem Cursor stehende Buchstaben gelöscht und die Liste schrumpft.

Um das Ganze anschaulich zu gestalten, schreibst du die Buchstaben als Text mit einer Text- und Hintergrundfarbe in ein GPanel. Dabei durchläufst du die Liste mit einem Listenindex i.




```
from gpanel import *
from java.awt import Font

NO_KEY = 65535
BS_KEY = 8

def updateGraphics():
    clear()
    for i in range(len(letterList)):
        text(i, 2, letterList[i], Font("Courier", Font.PLAIN, 24),
             "blue", "light gray")
    line(cursorPos - 0.2, 1.7, cursorPos - 0.2, 2.7)
    text(1, 1, "List length = " + str(len(letterList)))
    if keyCode != NO_KEY:
        text(1, 0.5, "Last key code = " + str(keyCode))

def onMousePressed(e):
    x = toWindowX(e.getX())
```

```
    y = toWindowY(e.getY())
    setCursor(x)
    updateGraphics()

def setCursor(x):
    global cursorPos
    cursorPos = int(x + 0.7)

makeGPanel(-1, 30, 0, 12, mousePressed = onMousePressed)

letterList = []
cursorPos = 0
keyCode = NO_KEY
title("Enter Text. Backspace to delete. Mouse to set cursor.")
lineWidth(3)
updateGraphics()

while True:
    delay(10)
    key = getKey()
    keyCode = ord(key)
    if keyCode == NO_KEY:
        continue
    elif keyCode == BS_KEY:  # backspace
        if cursorPos > 0:
            cursorPos -= 1
            letterList.pop(cursorPos)
    else:
        letterList.insert(cursorPos, key)
        cursorPos += 1
    updateGraphics()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du hast bereits früher gelernt, dass du auf einzelne Elemente einer Liste über den Listenindex zugreifen kannst, der bei Null beginnt. Du verwendest dazu die eckigen Klammern, also **letterList[i]**. Der Index muss immer im Bereich 0 und Listenlänge - 1 liegen. Verwendest du eine **for in range()**-Struktur, so ist der Stoppwert gerade die Länge der Liste.

Mit dem Index darfst du aber nie auf ein Element zugreifen, dass es gar nicht gibt. Fehler mit ungültigen Listenindizes gehören zu den häufigsten Programmierfehlern. Passt du nicht auf, so kriegst du Programme, die unter Umständen manchmal funktionieren und manchmal absterben.

Um zu testen, welche Taste gedrückt wurde, kannst du **getKey()** verwenden. Diese Funktion kehrt nach dem Aufruf sofort zurück und liefert den Buchstaben der zuletzt gedrückten Taste oder den Wert 65535 (die grösste mit 16 bit darstellbare Ganzzahl), falls keine Taste gedrückt wurde.

BEREITS EIN PROFIPROGRAMM

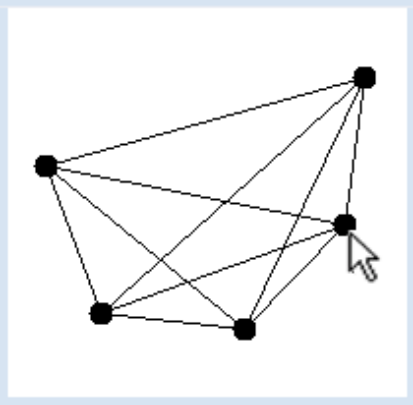
Mit deinem Wissen bist nun bereits in der Lage, einen Grafen zu visualisieren [**mehr...**]

Die Aufgabenstellung (auch "Pflichtenheft" genannt) ist die Folgende:

In einem Grafikfenster kannst du mit **rechtem Mausklick** gefüllte Kreise erzeugen, die als Knoten eines Grafen aufgefasst werden, wo die Knoten untereinander mit Strecken, Kanten genannt, verbunden sind. Fährst du mit der Maus auf einen Knoten, so kannst du diesen mit einem **linken Maus-Press** und nachfolgendem **Drag** herumziehen und der Graf wird ständig

aktualisiert. Klickst du mit mit einem rechten Mausklick auf einen bereits bestehenden Knoten, so wird dieser entfernt.

Es ist gescheit, dass du komplexere Aufgaben so löst, indem du zuerst eine **Teilaufgabe** erledigst, die noch nicht das ganze Pflichtenheft erfüllt. Du schreibst beispielsweise zuerst ein Programm, mit dem du per Mausklick Knoten erzeugen kannst. Sie sollen zwar mit allen bereits vorhandenen Knoten verbunden werden aber du kannst sie noch nicht verschieben.



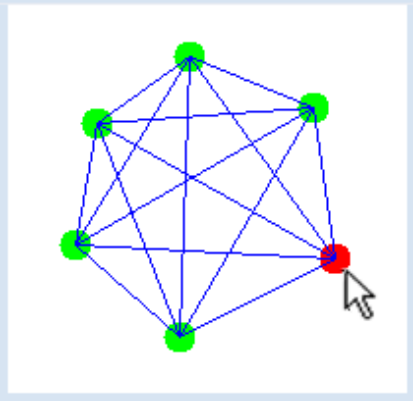
Es liegt auf der Hand, den Graf mit einer **Liste graph** zu modellieren, in der du die Knotenpunkte speicherst.

Die Knoten selbst sind Punkte mit zwei Koordinaten P(x, y), die du - wie bereits früher - mit einer **Punktliste [x, y]** modellierst. Es liegt also eine Liste vor, die als Elemente wiederum Listen (allerdings mit fester Länge 2) enthält. Das Verbinden der Knoten erreichst du über eine **doppelte for-Schleife**, wobei du acht geben musst, dass die Knoten nur einmal verbunden werden.

```
from gpanel import *

def drawGraph():
    clear()
    for pt in graph:
        move(pt)
        fillCircle(2)

    for i in range(len(graph)):
        for k in range(i, len(graph)):
            line(graph[i], graph[k])

def onMousePressed(e):
    x = toWindowX(e.getX())
    y = toWindowY(e.getY())
    pt = [x, y]
    graph.append(pt)
    drawGraph()

graph = []
makeGPanel(0, 100, 0, 100, mousePressed = onMousePressed)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Als nächstes baust du nun das Ziehen und Löschen der Knoten ein. Dazu brauchst du auch die rechte Maustaste. Beim Ziehen ist es wichtig zu wissen, welcher Knoten gerade gezogen wird. Diesen kannst du dir über seinen Index **iNode** in der Liste *graph* merken. Wird kein Knoten gezogen, ist **iNode = -1**. Du berechnest mit dem Satz von Pythagoras in der Funktion **near(x, y)** den Abstand des Punktes P(x,y) zu allen Knoten. Sobald eines der Abstandsquadrate kleiner als 10 ist, brichst du die Berechnung ab und gibst den Index des Knotens zurück. Dabei siehst du, dass man eine Funktion mit *return* auch mitten im Ablauf verlassen kann [**mehr...**].



Alles andere ist schöne Programmierarbeit, die du mit

deinem bisherigen Wissen auch allein zu Stande bringen würdest.

```
from gpanel import *

def drawGraph():
    clear()
    for i in range(len(graph)):
        move(graph[i])
        if i == iNode:
            setColor("red")
        else:
            setColor("green")
        fillCircle(2)

    setColor("blue")
    for i in range(len(graph)):
        for k in range(i, len(graph)):
            line(graph[i], graph[k])

def onMousePressed(e):
    global iNode
    x = toWindowX(e.getX())
    y = toWindowY(e.getY())
    if SwingUtilities.isLeftMouseButton(e):
        iNode = near(x, y)
    if isRightMouseButton(e):
        index = near(x, y)
        if index != -1:
            graph.pop(index)
            iNode = -1
        else:
            pt = [x, y]
            graph.append(pt)
    drawGraph()

def onMouseDragged(e):
    if isLeftMouseButton(e):
        if iNode == -1:
            return
        x = toWindowX(e.getX())
        y = toWindowY(e.getY())
        graph[iNode] = [x, y]
        drawGraph()

def onMouseReleased(e):
    global iNode
    if isLeftMouseButton(e):
        iNode = -1
        drawGraph()

def near(x, y):
    for i in range(len(graph)):
        p = graph[i]
        d = (p[0] - x) * (p[0] - x) + (p[1] - y) * (p[1] - y)
        if  d < 10:
            return i
    return -1

graph = []
iNode = -1
makeGPanel(0, 100, 0, 100,
               mousePressed = onMousePressed,
               mouseDragged = onMouseDragged,
               mouseReleased = onMouseReleased)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Das Programm ist voll **eventgesteuert**. Der Hauptblock definiert lediglich zwei globale Variablen und initialisiert das Grafikfenster. Bei jeder Aktion wird das ganze Grafikfenster gelöscht und mit der aktuellen Situation des Grafen neu aufgebaut.

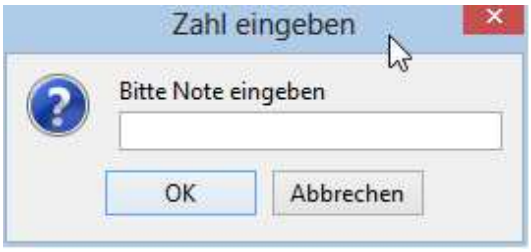
Die wichtigsten Operationen mit Listen

li = [1, 2, 3, 4]	Liste mit den Zahlen 1, 2, 3, 4 definieren
li = [1, "a", [7 , 5]]	Liste mit unterschiedlichen Datentypen definieren
li[i]	Auf Listenelement mit Index i zugreifen
li[start:end]	Neue Teilliste mit Elementen start bis end, aber ohne end
li[start:end:step]	Neue Teilliste mit Elementen start bis end mit Schritt step
li[start:]	Neue Teilliste mit allen Elementen von start an
li[:end]	Neue Teilliste von ersten Element bis end, aber ohne end
li.append(element)	Anfügen ans Ende
li.insert(i, element)	Einfügen an Stelle i (Element i rutscht nach rechts)
li.extend(li2)	Elemente aus li2 anfügen (Konkatenation)
li.index(element)	Sucht das erste Vorkommen und gibt dessen Index zurück
li.pop(i)	Entfernt das Element mit Index i und gibt es zuück
pop()	Entfernt das letzte Element und gibt es zurück
li1 + li2	Gibt die Konkatenation von li1 und li2 in neuer Liste zurück
li1 += li2	Ersetzt li1 durch die Konkatenation von li1 und li2
li * 4	Neue Liste mit Elementen von li viermal wiederholt
[0] * 4	Neue Liste mit der Länge 4 (jedes Element Zahl 0)
len(li)	Gibt die Anzahl Listenelemente zurück
del li[i]	Entfernt das Element mit Index i
del li[start:end]	Entfernt alle Elemente von start bis end, aber ohne end
del li[:]	Entfernt alle Elemente, es bleibt eine leere Liste
li.reverse()	Kehrt die Liste um (letztes Element wird das erste)
li.sort()	Sortiert die Liste (Vergleich mit Standardverfahren)
x in li	Gibt True zurück, falls x in der Liste enthalten ist
x not in li	Gibt True zurück, falls x nicht in der Liste enthalten ist

Die Notation mit den eckigen Klammern nennt man **Slice-Operation**. *start* und *end* sind Indizes der Liste. Die Slice-Operation funktioniert ähnlich wie die Parameter von range().

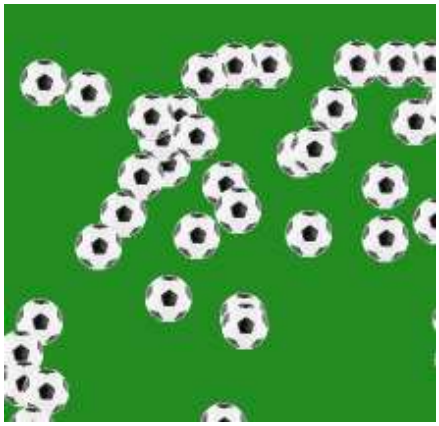
AUFGABEN

1. Lese mit *inputFloat("prompt", False)* eine beliebige Anzahl von Noten ein. Drückst du den *Abbrechen-Knopf*, so wird der Durchschnitt in der Console ausgeschrieben. Beachte, dass du den Parameterwert *False* verwenden musst, damit das Programm nicht beendet wird, wenn du *Abbrechen* klickst. Beim Abbruch wird der spezielle Wert *None* zurückgegeben, auf den du wie üblich mit *if* testen kannst.



2. Erweitere das oben stehende Editor-Programm unter Verwendung der Slice-Notation so, dass bei jedem rechten Mausklick der Beginn des Satzes bis und mit dem ersten Leerschlag weggeschnitten wird.

3. Bei jedem Mausklick soll ein neues Bild eines Fussballs (*football.gif*) an der Stelle des Mauscursors erscheinen. Alle Fussbälle bewegen sich ständig auf dem Bildschirm hin und her. Orientiere dich am Fussball-Beispiel im Kapitel *Animationen*. Du kannst das Programm etwas optimieren, indem du das Fussbaldbild nur einmal mit *img = getImage("_sprites/football.gif")* lädst und zum Zeichnen *img* der Funktion *image()* übergibst.



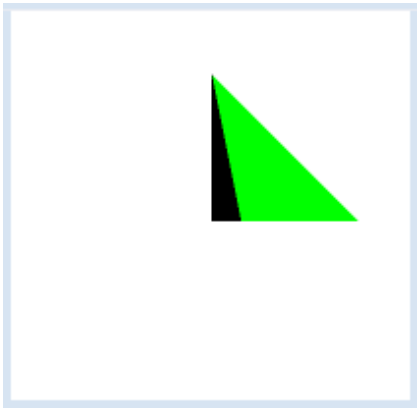
ZUSATZSTOFF:

VERÄNDERLICHE UND UNVERÄNDERLICHE DATENTYPEN

In Python werden alle Datentypen als Objekte gespeichert, also auch die numerischen Typen (integer, float, long, complex). Wie du weisst, greifst du auf ein Objekt über seinen Namen zu. Man sagt auch, dass der Name auf das Objekt **verweist** oder an das Objekt **gebunden** wird. Eine solche Variable heisst darum auch **Referenzvariable**.

Auf ein bestimmtes Objekt kann mehr als ein Name verweisen. Man nennt einen weiteren Namen auch ein **Alias**. Wie du damit umgehst, zeigt das folgende Beispiel.

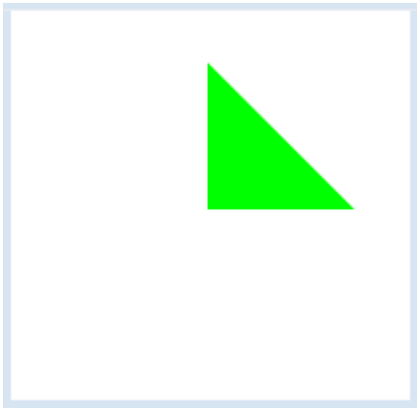
Ein Dreieck wird durch drei Eckpunktlisten a, b, c festgelegt. Mit der Anweisung *a_alias = a* erzeugst du einen Alias von a, so dass a und a_alias auf die gleiche Liste verweisen. Verändert man mit dem Namen a die Eckpunktliste, so wird logischerweise beim Zugriff mit dem Namen a_alias die Änderung auch sichtbar, da a und a_alias auf dieselbe Liste verweisen.



```
from gpanel import *

makeGPanel(-10, 10, -10, 10)

a = [0, 0]
a_alias = a
b = [0, 5]
c = [5, 0]

fillTriangle(a, b, c)
a[0] = 1
setColor("green")
fillTriangle(a_alias, b, c)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Wenn du für die Eckpunkte Zahlenkoordinaten verwendest, so verändert sich allerdings der Wert von xA_alias nicht, wenn du xA veränderst, obschon Zahlen auch Objekte sind.



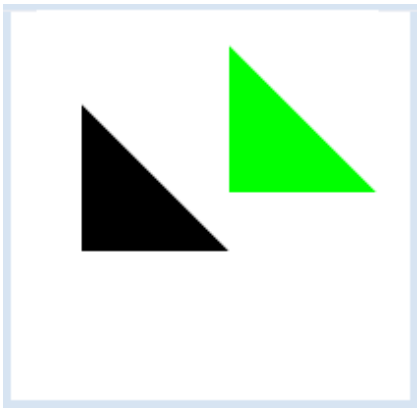
```
from gpanel import *

makeGPanel(-10, 10, -10, 10)

xA = 0
yA = 0
xA_prime = xA
yA_prime = yA
xB = 0
yB = 5
xC = 5
yC = 0

fillTriangle(xA, yA, xB, yB, xC, yC)
xA = 1
setColor("green")
fillTriangle(xA_prime, yA_prime, xB, yB, xC, yC)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Was ist die Erklärung dafür? Der Grund liegt darin, dass Zahlen **unveränderliche Objekte** sind und die Anweisung xA = 1 eine **neue Zahl erzeugt**. xA_alias ist also immer noch 0.

Der Unterschied zwischen unveränderlichen und veränderlichen Datentypen zeigt sich auch bei der Parameterübergabe an Funktionen. Wird ein unveränderliches Objekt übergeben, so kann im Innern der Funktion das übergebene Objekt nicht verändert werden. Wird hingegeben ein veränderliches Objekt übergeben, so kann die Funktion das Objekt verändern. Eine solche Veränderung nennt man einen **Seiteneffekt**. Es gehört zum guten Programmierstil, Seiteneffekte nur sparsam einzusetzen, weil sie zu schwer auffindbaren Fehlverhalten führen können.

Im folgenden Beispiel verändert die Funktion *translate()* die übergebenen Eckpunktlisten.



```
from gpanel import *

def translate(pA, pB, pC):
  pA[0] = pA[0] + 5
  pA[1] = pA[1] + 2
  pB[0] = pB[0] + 5
  pB[1] = pB[1] + 2
  pC[0] = pC[0] + 5
  pC[1] = pC[1] + 2

makeGPanel(-10, 10, -10, 10)

a = [0, 0]
b = [0, 5]
c = [5, 0]
fillTriangle(a, b, c)
translate(a, b, c)
setColor("green")
fillTriangle(a, b, c)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

In Python werden alle Daten als Objekte gespeichert, aber gewisse Objekte gelten als **unveränderlich** (immutable). Es sind dies: numerische Datentypen, String, byte, tuple.

Alle anderen Datentypen sind **veränderlich** (mutable). Weist man einem unveränderlichen Datentyp einen neuen Wert zu, so wird ein neues Objekt erzeugt.

Werden einer Funktion veränderliche Objekte übergeben, so kann die Funktion die Objekte verändern, unveränderliche Objekte sind aber vor solchen Veränderungen geschützt.

3.10 ZUFALL

EINFÜHRUNG

Im täglichen Leben spielen Zufälle eine grosse Rolle. Man versteht darunter Ereignisse, die nicht voraussehbar sind. Bittet man dich etwa, aus den Farben rot, grün und blau eine Farbe zu wählen, so kann niemand voraussagen, für welche du dich entscheidest, die Farbe ist also zufällig. Auch bei Games spielt der Zufall eine grosse Rolle: Wirft du einen Würfel, so ist die Augenzahl zwischen 1 und 6 zufällig.

Obschon die Welt vom Zufall regiert wird [**mehr...**] handelt es sich nicht um ein Chaos, sondern es gibt auch beim Zufall Gesetzmässigkeiten, welche gewisse Voraussagen ermöglichen. Diese gelten aber nur "im Mittel" oder anders ausgedrückt, wenn man oftmals in der gleichen Situation ist. Um die Gesetzmässigkeiten des Zufalls zu untersuchen, führt man **Zufallsexperimente** durch, bei denen man die Anfangsbedingungen fest vorgibt, wo aber der Ablauf durch Zufallszahlen gesteuert wird.

Für Zufallsexperimente ist der Computer hervorragend geeignet, da es sehr einfach ist, eine grosse Anzahl von Versuchen durchzuführen. Dazu muss der Computer eine Reihe von Zufallszahlen erzeugen, die möglichst voneinander unabhängig sind. Man verwendet meist Ganzzahlen in einem gewissen vorgegebenen Bereich, z.B. zwischen 1 und 6, oder Dezimalzahlen zwischen 0 und 1. Einen Algorithmus, der eine Reihe von Zufallszahlen berechnet, nennt man einen **Zufallszahlengenerator**. Es ist wichtig, dass die Zahlen mit gleicher Häufigkeit auftreten, wie man es von einem (nicht gezinkten) Würfel her gewohnt ist. Man nennt solche Zahlen **gleichverteilt** [**mehr...**].

PROGRAMMIERKONZEPTE: *Zufallszahl, Zufallsexperiment, Häufigkeit, Wahrscheinlichkeit*

ZUFÄLLIGES MALEN

Auf eine Leinwand kleckst du 20 farbige Ellipsen mit zufälliger Grösse, zufälliger Lage und zufälliger Farbe. Ob du dies als Malerei, ja gar als Kunst auffassen willst, bleibt dir überlassen. Lustig sind die Figuren alleweil. Für die Position und Grösse der Ellipsen verwendest du die Methode **random()** aus dem **Modul random**, die bei jedem Aufruf eine neue Zufallszahl zwischen 0 und 1 liefert. Um die zufälligen Farben zu erhalten, brauchst du drei **Zufallszahlen zwischen 0 und 255**, welche die Anteile der roten, grünen und blauenFarbe festlegen.

```
from gpanel import *
import random

def randomColor():
    r = random.randint(0, 255)
    g = random.randint(0, 255)
    b = random.randint(0, 255)
    return makeColor(r, g, b)

makeGPanel()
bgColor(randomColor())
```

```
for i in range(20):
    setColor(randomColor())
    move(random.random(), random.random())
    a = random.random() / 2
    b = random.random() / 2
    fillEllipse(a, b)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

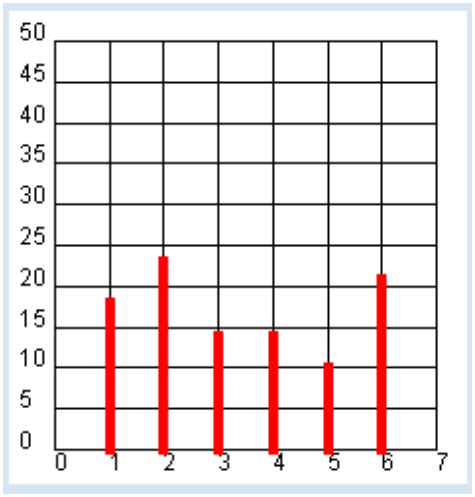
random.random() liefert gleichverteilte Zufallszahlen als Float zwischen 0 (eingeschlossen) und 1 (ausgeschlossen). Du musst das Modul random **importieren**, um darauf zugreifen zu können. Farben werden durch ihren Rot-, Grün- und Blauanteil (RGB) festgelegt. Die Werte sind Ganzahlen zwischen 0 und 255.

Mit **randint(start:end)** erhält man eine ganzzahlige Zufallszahl zwischen *start* und *end* (beide eingeschlossen). Die Funktion *makeColor()* liefert aus den 3 Farbwerten für Rot, Grün und Blau ein Farbobjekt.

HÄUFIGKEIT VON WÜRFELZAHLEN

Das Zufallsexperiment besteht darin, dass du 100 Mal würfelst und dann herausfinden willst, wie oft die Augenzahlen 1, 2, ...6, vorgekommen sind.

Mit einem Computerprogramm kannst du das Experiment viel schneller durchführen. Anstelle des Würfelns verwendest du **Zufallszahlen von 1 bis 6**. Die Häufigkeitsverteilung kannst du in einem GPanel grafisch darstellen.




```
from gpanel import *
import random

NB_ROLLS = 100

makeGPanel(-1, 8, -0.1 * NB_ROLLS / 2, 1.1 * NB_ROLLS / 2)
title("# Rolls: " + str(NB_ROLLS))
drawGrid(0, 7, 0, NB_ROLLS // 2, 7, 10)
setColor("red")

histo = [0, 0, 0, 0, 0, 0, 0]
# hist = [0] * 7  # short form

for i in range(NB_ROLLS):
    pip = random.randint(1, 6)
    histo[pip] += 1

lineWidth(5)
for n in range(1, 7):
    line(n, 0, n, histo[n])
```

MEMO

Die Häufigkeit, mit der die einzelnen Augenzahlen (pip) vorkommen, müssen gespeichert werden. Du verwendest dazu die Liste **histo**, in der du die Vorkommnisse beim entsprechenden Index aufsummierst. Du benötigst eine Liste mit 7 Elementen, da der Index von 1 bis 6 läuft.

Wie du durch einige Experimente feststellen kannst, sind die Häufigkeiten bei grösser werdenden Wurfzahlen **NB_ROLLS** immer besser ausgeglichen und erreichen immer genauer 1/6 der Wurfzahl. Diese Tatsache drückt man so aus: **Die Wahrscheinlichkeit, beim Würfelwerfen eine der Augenzahlen zu erhalten, ist 1/6**.

Für das Koordinatengitter rufst du *drawGrid(xmin, xmax, ymin, ymax, xticks, yticks)* mit 6 Parametern auf. Die zwei letzten Parameter bestimmen die Anzahl der Unterteilungen. Ist *xmax* oder ymax ein Float so erfolgt die Achsenbeschriftung ebenfalls in Floats, sonst sind es Integer.

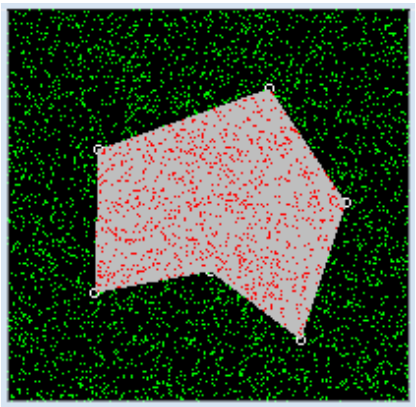
MONTE-CARLO-SIMULATION

Das Fürstentum Monaco ist durch sein Casino im Stadtteil Monte-Carlo weltberühmt. Das Casino hat nicht nur seit 150 Jahren eine magische Anziehungskraft auf Berühmtheiten, sondern auch auf Mathematiker, die versuchen, die Spiele zu analysieren und Gewinnstrategien zu entwickeln. Zum Test der Strategien eignet sich der Computer besser als das reale Spiel, da man bei Computerexperimenten kein Geld verliert.

Beim folgenden "Spiel" wirfst du Punkte auf eine quadratische Fläche, auf der ein Polygon liegt. Die Punkte kannst du anschaulich auch als Regentropfen auffassen. Wie dies beim Regen üblich ist, fallen in einer gewissen Zeit ungefähr immer gleich viel Tropen auf jede Flächeneinheit. Die Tropfen sind also **gleichverteilt**. Du lässt eine bestimmte totale Zahl von Regentropfen fallen und zählst, wie viele davon auf die Polygonfläche fallen. Es ist offensichtlich, dass diese Zahl mit zunehmendem Flächeninhalt des Polygons zunimmt und im Mittel proportional zum Flächeninhalt ist. Beispielsweise fallen auf ein Polygon mit einem Flächeninhalt von ¼ des Flächeninhalts des umgebenden Quadrats ja wohl (im Mittel) ¼ aller Regentropfen. Aus dieser Erkenntnis kannst du nun umgekehrt durch Zählen der Tropfen den Flächeninhalt herausfinden. Ist das nicht elegant?

Das Programm ist modern und benutzerfreundlich gestaltet. Du kannst zu Laufzeit mit einem **linken Mausklicks** die Eckpunkte des Polygons erzeugen. Klickst du dann mit der **rechten Maustaste** in Fläche, die du berechnen willst, so wird das Polygon gezeichnet und es beginnt zu regnen.

Das Ergebnis wird in der Titelleiste angezeigt.



```
from gpanel import *
import random

NB_DROPS = 10000

def onMousePressed(e):
    if isLeftMouseButton(e):
        x = toWindowX(e.getX())
```

```
        y = toWindowY(e.getY())
        pt = [x, y]
        move(pt)
        circle(0.5)
        corners.append(pt)
    if isRightMouseButton(e):
        wakeUp()

def go():
    global nbHit
    setColor("gray")
    fillPolygon(corners)
    title("Working. Please wait...")
    for i in range(NB_DROPS):
        pt = [100 * random.random(), 100 * random.random()]
        color = getPixelColorStr(pt)
        if color == "black":
            setColor("green")
            point(pt)
        if color == "gray" or color == "red":
            nbHit += 1
            setColor("red")
            point(pt)
    title("All done. #hits: " + str(nbHit) + " of " + str(NB_DROPS))

makeGPanel(0, 100, 0, 100, mousePressed = onMousePressed)
title("Select corners with left button. Start dropping with right button")
bgColor("black")
setColor("white")
corners = []
nbHit = 0
putSleep()
go()
```

MEMO

Beim Klicken der **linken Maustaste** speicherst du die Eckpunkte des Polygons in einer Liste *corners* und zeichnest als Markierung kleine Kreise.

Die eigentliche Regensimulation wird in der Funktion **go()** durchgeführt. Sie beginnt beim Klicken der rechten Maustaste und dauerst eine gewisse Zeit. Dabei machst du mit verschieden farbigen Punkten die fallenden Regentropfen sichtbar. Rufst du, wie es eigentlich auf der Hand liegt, *go()* im *pressCallback()* direkt auf, so siehst du nichts, bis die Simulation zu Ende ist. Das System unterbindet nämlich aus systeminternen Gründen das Auffrischen der Grafik in einem Maus-Callback. Willst du also in einem Callback eine länger dauernde Aktion sichtbar machen, so muss dies in einem anderen Teil des Programms geschehen. Oft wird der Hauptblock des Programms dazu verwendet. Die Ausführung wird mit **putSleep()** vorübergehend angehalten. Der Press-Event weckt das schlafende Hauptprogramm mit **wakeUp()** auf und dieses führt nun die Simulation mit dem Aufruf von *go()* aus.

Um Schwierigkeiten zu vermeiden, solltest du dich in Zukunft unbedingt an das folgende Prinzip halten: **Callbacks müssen immer rasch zurückkehren. Es dürfen darin keine lange dauernde Aktionen ausgeführt werden.**

Um herauszufinden, ob ein Regentropfen auf die grau gefärbte Polygonfläche gefallen ist, wendest du folgenden Trick an: Du holt dir mit **getPixelColorStr()** die Farbe der Auftreffstelle. Ist die Farbe grau (oder rot, wenn bereits ein anderer Tropf dort hingefallen ist), so erhöhst du **nbHit um 1** und färbst die Stelle rot.

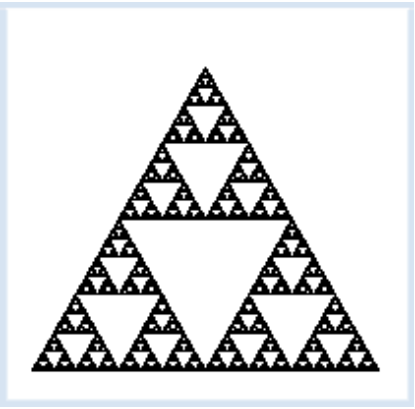
Du kannst das Verfahren testen, indem du einige einfache Polygone (z.B. Rechtecke, Dreiecke) erzeugst und mit dem Massstab den Bildschirm ausmisst. Du erkennst dann, dass es sehr viele Regentropfen braucht, um ein einigermassen exaktes Resultat zu erhalten [**mehr...**].

CHAOS-SPIEL

Es ist auf den ersten Blick erstaunlich, dass sich mit Zufallsexperimenten regelmässige Muster erzeugen lassen. Dies hängt damit zusammen, dass sich die statistischen Schwankungen bei grossen Zahlen ausgleichen. [**mehr...**]. Michael Barnsley hat 1988 im Rahmen der Chaos-Theorie den folgenden Algorithmus erfunden, der auf einer zufälligen Auswahl der Eckpunkte eines Dreiecks aufbaut:

1. Konstruiere ein gleichseitiges Dreieck mit den Ecken A, B, C
2. Wähle einen Punkt P im Innern
3. Wähle zufällig einen der Eckpunkte
4. Halbiere die Verbindungsstrecke von P zum Eckpunkt. Dies gibt den neuen Punkt P
5. Zeichne den Punkt P
6. Wiederhole Schritt 3, 4, 5

Eine solche Formulierung ist umgangssprachlich üblich, lässt sich aber nicht direkt in Programmcode übersetzen, da der Punkt 6 verlangt, dass man wieder zum zu Punkt 3 springen soll. In vielen modernen Programmiersprachen, so auch in Python, gibt es aber keine Sprung-Struktur (**kein goto**). Sprünge müssen mit einer der Wiederholstrukturen implementiert werden. [**mehr...**].



```
from gpanel import *
import random

MAXITERATIONS = 100000
makeGPanel(0, 10, 0, 10)

pA = [1, 1]
pB = [9, 1]
pC = [5, 8]

triangle(pA, pB, pC)
corners = [pA, pB, pC]
pt = [2, 2]

title("Working...")
for iter in range(MAXITERATIONS):
    i = random.randint(0, 2)
    pRand = corners[i]
    pt = getDividingPoint(pRand, pt, 0.5)
    point(pt)
title("Working...Done")
```

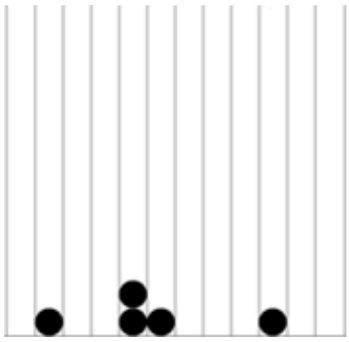
MEMO

Benötigst du ein zufälliges Objekt, so fügst du alle Objekte in eine **Liste** und holst mit einem zufälligen Index eines davon heraus. Es ist erstaunlich, dass du mit zufällig gewählten Punkten eine regelmässige Figur, nämlich ein **Sierpinski-Fraktal**, erzeugen kannst.

AUFGABEN

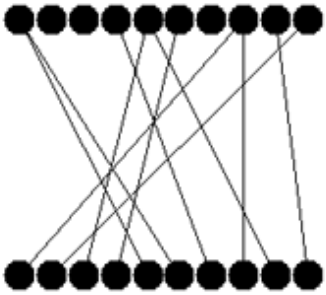
1. 5 Kinder treffen sich auf dem Pausenhof und fragen sich nach dem Monat, in welchem sie Geburtstag haben. Es ist erstaunlich, dass die Wahrscheinlichkeit, dass mindestens zwei den gleichen Geburtstagsmonat haben, relativ gross ist.

   Erstelle eine Simulation mit 100 Zufallsversuchen, um diese Wahrscheinlichkeit experimentell zu bestimmen. Zeige zur Veranschaulichung für jeden Versuch in einem GPanel zwölf rechteckige Monatsbehälter und zeichne die Kinder durch Kugeln dargestellt ein. Das Resultat der Versuchsreihe kann in der Titelzeile ausgeschrieben werden.



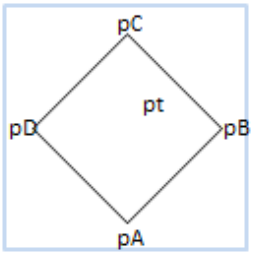
2. Beim Ballspiel werfen 10 Kinder einer ersten Mannschaft gleichzeitig den Ball auf 10 Kinder der zweiten Mannschaft und treffen immer ein Kind. (Die Bälle sollen sich nicht gegenseitig beeinflussen.) Die getroffenen müssen ausscheiden. Wie viele der zweiten Mannschaft bleiben im Mittel im Spiel?

   Erstelle eine Simulation mit 100 Zufallsversuchen, um diese Zahl experimentell zu bestimmen. Zeige zur Veranschaulichung für jeden Versuch in einem GPanel die beiden Mannschaften als gefüllte Kreise und zeichne die Ballrichtungen als Strecken ein. Das Resultat der Versuchsreihe kann in der Titelzeile ausgeschrieben werden.



3. Du kannst mit der Monte-Carlo-Simulation sogar die Fläche von beliebigen Figuren zu bestimmen. Mit gedrückter linker Maustaste zeichnest du die Umrandung als Freihandlinie. Durch Klicken mit der rechten Maustaste auf einen Punkt im Inneren wird die Fläche gefüllt und die Simulation durchgeführt.

4. Führe das Chaos-Spiel mit einem Quadrat aus. Wähle die Eckpunkte pA(0, -10), pB(10, 0), pC(0, 10), pD(-10, 0) und einen beliebigen Punkt pt im Innern. Teile die Strecken von einem beliebig gewählten Eckpunkt zu pt mit dem Teilungsfaktor 0.45
   (*pt = getDividingPoint(Corner, pt, 0.45)*).



3.11 BILDBEARBEITUNG

EINFÜHRUNG

Wir fassen ein Bild als ein ebene, rechteckförmige Fläche auf, auf der sich farbige Formen befinden. In der Druck- und Computertechnik beschreibt man ein Bild durch eine gitterartige Anordnung von farbigen Bildpunkten, auch Pixels genannt. Die Anzahl Bildpunkte pro Flächeneinheit wird **Bildauflösung** genannt und oft in *dots per inch* (dpi) angegeben.

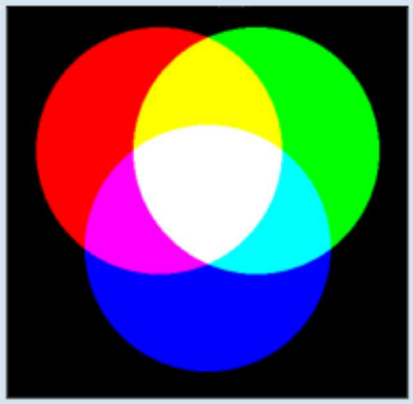
Um ein Bild auf dem Computer zu speichern und zu verarbeiten, muss die Farbe als Zahl definiert werden. Es gibt dazu mehrere Möglichkeiten, die man **Farbmetriken** oder **Farbmodelle** nennt. Eines der bekanntesten Modelle ist das RGB-Farbmodell, wo die Intensitäten der drei Farbkomponenten für Rot, Grün und Blau als Zahlen zwischen 0 (dunkel) und 255 (hell) angegeben werden [**mehr...**]. Im ARGB-Modell wird in einer weiteren Zahl zwischen 0 und 255 ein Mass für die Transparenz (Alpha-Wert) mitgeliefert [**mehr...**].

Zusammengefasst: Ein Computerbild besteht aus einer rechteckigen Anordnung von Pixels, die als Farben codiert sind. Oft spricht man von einer **Bitmap**.

PROGRAMMIERKONZEPTE: *Bilddigitalisierung, Bildauflösung, Farbmodell, Bitmap, Bildformat*

FARBMISCHUNG IM RGB-MODELL

TigerJython stellt dir Objekte vom Typ *GBitmap* zur Verfügung, um dir die Arbeit mit Bitmaps zu erleichtern. Du erzeugst mit **bm = GBitmap(width, height)** eine Bitmap mit der gewünschten Anzahl horizontaler und vertikaler Bildpunkten. Nachher kannst du mit den Methoden **setPixelColor(x, y, color)** die Farbe einzelner Pixels setzen und sie mit *getPixelColor(x, y)* lesen. Mit der Methode *image()* stellst du schliesslich deine Bitmap in einem GPanel dar. Dein Programm zeichnet die berühmten 3 Farbkreise der additiven Farbmischung, indem du mit einer verschachtelten for-Schleife die Bitmap durchläufst.



```
from gpanel import *

xRed = 200
yRed = 200
xGreen = 300
yGreen = 200
xBlue = 250
yBlue = 300

makeGPanel(Size(501, 501))
window(0, 501, 501, 0)    # y axis downwards
bm = GBitmap(500, 500)
for x in range(500):
  for y in range(500):
      red = green = blue = 0
      if (x - xRed) * (x - xRed) + (y - yRed) * (y - yRed) < 16000:
         red = 255
      if (x - xGreen) * (x - xGreen) + (y - yGreen) * (y - yGreen) < 16000:
```

```
            green = 255
        if (x - xBlue) * (x - xBlue) + (y - yBlue) * (y - yBlue) < 16000:
            blue = 255
        bm.setPixelColor(x, y, makeColor(red, green, blue))

image(bm, 0, 500)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Farben werden mit ihrem Rot-, Grün- und Blauanteil definiert. *makeColor(red, green, blue)* setzt diese Farbanteile zu einer Farbe (einem Farbobjekt) zusammen.

Für Bilder wird meist ein Integer-Koordinatensystem mit dem Ursprung in der oberen linke Ecke und nach unten zeigender positiver y-Achse verwendet [**mehr...**].

GRAUSTUFENBILD SELBST GEMACHT

Du hast dich vielleicht manchmal gefragt, wie deine Bildverarbeitungs-Software (wie Photoshop, o.ä.) funktioniert. Hier lernst du einige einfache Verfahren kennen. Dein Programm macht aus einem Farbbild ein Graustufenbild, indem du den Mittelwert des Rot-, Grün- und Blauanteils bestimmst und diesen zur Definition des Grauwerts verwendest.

```
from gpanel import *

size = 300

makeGPanel(Size(2 * size, size))
window(0, size, size, 0)    # y axis downwards
img = getImage("sprites/colorfrog.png")
w = img.getWidth()
h = img.getHeight()
image(img, 0, size)
for x in range(w):
    for y in range(h):
        col = img.getPixelColor(x, y)
        red = col.getRed()
        green = col.getGreen()
        blue = col.getBlue()
        intensity = (red + green + blue) // 3
        gray = makeColor(intensity, intensity, intensity)
        img.setPixelColor(x, y, gray)
image(img, size / 2, size)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Aus einem Farbobjekt kannst du mit den Methoden **getRed()**, *getGreen(), getBlue()* die Farbwerte als Integer bestimmen.

Der Hintergrund muss weiss und nicht etwa transparent sein. Willst du die Transparenz berücksichtigen, so kannst du mit *alpha = getAlpha()* den Transparenzwert bestimmen und diesen in *makeColor(red, green, blue, alpha)* verwenden.

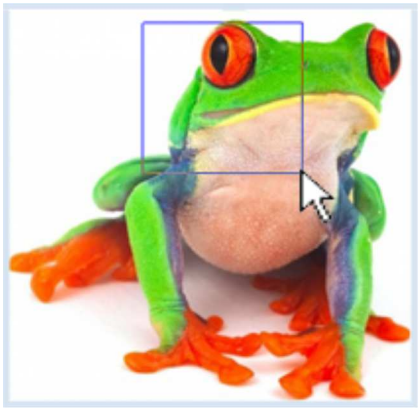
WIEDERVERWENDBARKEIT

Bei vielen Bildbearbeitungen muss der Benutzer einen Teilbereich des Bildes auswählen. Dazu zieht er mit der Maus ein temporäres Rechteck ("Gummiband-Rechteck"). Beim Loslassen der Maustaste wird der rechteckige Bereich definitiv auswählt. Es ist ratsam, zuerst dieses Teilproblem zu lösen, da dieser Code später in vielen Bildbearbeitungsprogrammen wieder verwendet werden kann. Wiederverwendbarkeit ist ein Gütezeichen für alle Software-Entwicklungen.

Wie du früher gesehen hast, kann man das Zeichnen von Gummiband-Linien als Animation auffassen. Dabei muss aber bei jeder Bewegung das ganze Bild immer wieder neu aufgebaut werden. Ein eleganter Trick, um dies zu vermeiden, ist der **XOR-Zeichnungsmodus**. In diesem Modus wird eine neue Figur mit der darunter liegenden so verschmolzen, dass durch erneutes Darüberzeichnen die Figur wieder gelöscht wird, ohne dass sich das darunterliegende Bild verändert. Der Nachteil besteht darin, dass beim Zeichnen der Figur die Farben verändert werden. Dies kann man aber im Zusammenhang mit Gummiband-Rechtecken meist in Kauf nehmen.

Das Programmgerüst soll nach der Rechteckwahl lediglich die Funktion **doIt()** aufrufen und die Koordinaten der oberen linken Ecke *ulx (upper left x)*, *uly (upper left y)* und der unteren rechten Ecke *lrx (lower right x), lry (lower right y)* ausschreiben. Später wirst du deinen Code zur Bildbearbeitung in *doIt()* einklinken.

Du verstehst den Code auf Grund deiner Kenntnisse im Kapitel Maus-Events ohne grössere Probleme.



```
from gpanel import *

size = 300

def onMousePressed(e):
    global x1, y1
    global x2, y2
    setColor("blue")
    setXORMode(Color.white) # set XOR paint mode
    x1 = x2 = e.getX()
    y1 = y2 = e.getY()

def onMouseDragged(e):
    global x2, y2
    rectangle(x1, y1, x2, y2) # erase old
    x2 = e.getX()
    y2 = e.getY()
    rectangle(x1, y1, x2, y2) # draw new
```

```
def onMouseReleased(e):
    rectangle(x1, y1, x2, y2) # erase old
    setPaintMode() # establish normal paint mode
    ulx = min(x1, x2)
    lrx = max(x1, x2)
    uly = min(y1, y2)
    lry = max(y1, y2)
    doIt(ulx, uly, lrx, lry)

def doIt(ulx, uly, lrx, lry):
    print "ulx = ", ulx, "uly = ", uly
    print "lrx = ", lrx, "lry = ", lry

x1 = y1 = 0
x2 = y2 = 0

makeGPanel(Size(size, size),
    mousePressed = onMousePressed,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased)
window(0, size, size, 0)    # y axis downwards

img = getImage("sprites/colorfrog.png")
image(img, 0, size)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du holst dir die Bitmap für eine Bild, das du auf dem Computer gespeichert hast mit **getImage()**, wobei du den vollständig qualifizierten Dateinamen oder auch nur einen Teil des Pfades relativ zum Verzeichnis, in dem sich dein Programm befindet, angeben musst. Für Bilder, die sich in der Distribution befinden, verwendest du den Verzeichnisnamen *sprites*.

Beim Press-Event setzst du das System in den **XOR-Modus**, damit du beim Drag-Event mit zweimaligem Zeichnen zuerst das alte Rechteck löschen und dann das neue zeichnen kannst. Dazu musst du die Eckpunkte in den globalen Werten *x1, y1, x2, y2* abspeichern. Wenn du beim Release-Event das Gummiband-Rechteck nochmals zeichnest, bevor du in den Paint-Mode wechselst, so wird das Rechteck verschwinden. Wenn du aber zuerst in den Paint-Mode wechselt, bleibt es erhalten.

Das Programm funktioniert unabhängig davon, wie du das Rechteck ziehst. Es liefert immer die richtigen Werte für *ulx,uly* und *lrx, lry* (immer *ulx* < *lrx, uly* < *lry*). Beachte, dass du die Mauskoordinaten nicht auf Window-Koordinaten umrechnen musst, da beide gleich gross sind, wenn du bei der Fenstergrösse mit Size() und dem Koordinatensystem mit window() dieselbe Werte verwendest.

Du kriegst auch Drag-Events, wenn du die Maus aus dem Fenster ziehst. Du musst aufpassen, was du mit solchen Koordinaten machst, sonst kann das Programm unerwartet crashen.

ROTAUGEN-EFFEKT

Die Bildbearbeitung spielt für die Nachbearbeitung von digitalen Fotos eine zentrale Rolle. Es gibt im Internet zahlreiche Nachbearbeitungsprogramme. Du brauchst aber keinen allzu grossen Respekt vor dieser Software zu haben, denn du kannst nun mit Python und einem gesunden Mass an Fantasie und Durchhaltewillen eigene Programme schreiben, die deinen Bedürfnissen besser angepasst sind. Deine Aufgabe besteht nachfolgend darin, ein Programm zu schreiben, dass den Rotaugen-Effekt beheben kann. Dieser tritt auf, wenn bei der Aufnahme das Blitzlicht am Augenhintergrund (fundus) reflektiert wird. Als Bild verwendest du hier ein Froschbild, da dieses auch noch andere rote Stellen aufweist.

```
from gpanel import *

size = 300

def onMousePressed(e):
    global x1, y1
    global x2, y2
    setColor("blue")
    setXORMode("white")
    x1 = x2 = e.getX()
    y1 = y2 = e.getY()

def onMouseDragged(e):
    global x2, y2
    rectangle(x1, y1, x2, y2) # erase old
    x2 = e.getX()
    y2 = e.getY()
    rectangle(x1, y1, x2, y2) # draw new

def onMouseReleased(e):
    rectangle(x1, y1, x2, y2) # erase old
    setPaintMode()
    ulx = min(x1, x2)
    lrx = max(x1, x2)
    uly = min(y1, y2)
    lry = max(y1, y2)
    doIt(ulx, uly, lrx, lry)

def doIt(ulx, uly, lrx, lry):
    for x in range(ulx, lrx):
        for y in range(uly, lry):
            col = img.getPixelColor(x, y)
            red = col.getRed()
            green = col.getGreen()
            blue = col. getBlue()
            col1 = makeColor(3 * red // 4, green, blue)
            img.setPixelColor(x, y, col1)
    image(img, 0, size)
x1 = y1 = 0
x2 = y2 = 0

makeGPanel(Size(size, size),
    mousePressed = onMousePressed,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased)
window(0, size, size, 0)    # y axis downwards

img = getImage("sprites/colorfrog.png")
image(img, 0, size)
```

MEMO

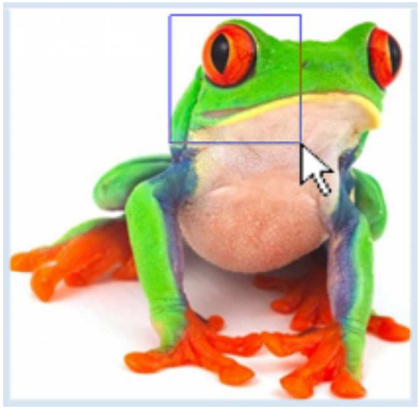
Der Code zur Bildbearbeitung wird in die Funktion **doIt()** eingeklinkt. Alles andere übernimmst du unverändert vom vorhergehenden Programm. Du kannst den Abschwächungsgrad für die rote Farbe anpassen. Hier wird die **Rotintensität auf 75%** hinuntergesetzt. Beachte den doppelten Bruchstrich, der eine Integer-Division durchführt (der Divisionsrest wird vernachlässigt). Das Resultat ist wieder ein Integer, wie es auch sein muss.

Das Programm zeigt noch einige Fehlerhalten, die du leicht beheben kannst. Zum einen verfärbt es auch nicht rote Gebiete, zum anderen schmiert es ab, wenn du das Gummiband-Rechteck aus dem Fenster ziehst.

Es wäre natürlich nett, wenn das Programm die roten Augen selbst finden würde. Dazu müsste es aber das Bild analysieren und Bildinhalte automatisch erkennen, was ein besonders schwieriges Problem der Informatik ist [**mehr...**].

BILDER AUSSCHNEIDEN UND ABSPEICHERN

Das Ausschneiden von Bildteilen gehört ebenfalls zu den Grundfunktionen von Bildverarbeitungs- programmen. Dein Programm kann nicht nur einen Teil, den du mit einem Gummiband-Rechteck auswählst, in ein anderes Fenster kopieren, sondern dieses Bild auch als JPEG-Datei zur späteren Wiederverwendung abspeichern.



```
from gpanel import *

size = 300

def onMousePressed(e):
    global x1, y1
    global x2, y2
    setColor("blue")
    setXORMode("white")
    x1 = x2 = e.getX()
    y1 = y2 = e.getY()

def onMouseDragged(e):
    global x2, y2
    rectangle(x1, y1, x2, y2) # erase old
    x2 = e.getX()
    y2 = e.getY()
    rectangle(x1, y1, x2, y2) # draw new

def onMouseReleased(e):
    rectangle(x1, y1, x2, y2) # erase old
    setPaintMode()
    ulx = min(x1, x2)
    lrx = max(x1, x2)
    uly = min(y1, y2)
    lry = max(y1, y2)
    doIt(ulx, uly, lrx, lry)
```

```
def doIt(ulx, uly, lrx, lry):
    width = lrx - ulx
    height = lry - uly
    if ulx < 0 or uly < 0 or lrx > size or lry > size:
        return
    if width < 20 or height < 20:
        return

    cropped = GBitmap.crop(img, ulx, uly, lrx, lry)
    p = GPanel(Size(width, height))  # another GPanel
    p.window(0, width, 0, height)
    p.image(cropped, 0, 0)
    rc = GBitmap.save(cropped, "mypict.jpg", "jpg")
    if rc:
        p.title("Saving OK")
    else:
        p.title("Saving Failed")


x1 = y1 = 0
x2 = y2 = 0

makeGPanel(Size(size, size),
    mousePressed = onMousePressed,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased)
window(0, size, size, 0)    # y axis downwards

img = getImage("sprites/colorfrog.png")
image(img, 0, size)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du kannst bei Bedarf mehrere GPanel-Fenster anzeigen, indem du GPanel-Objekte erzeugst. Zum Zeichnen verwendest du die Grafikbefehle, die du mit dem Punktoperator aufrufst.

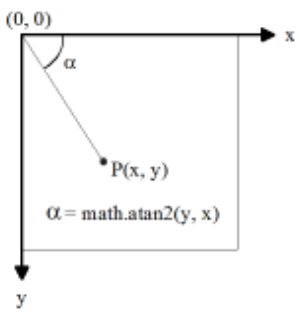
Falls der gewählte Ausschnitt zu klein ist (insbesondere wenn man ohne zu Ziehen mit der Maus klickt), so kehrt *doIt()* mit einem *return* unverrichteter Dinge zurück, ebenfalls wenn die Eckpunkte nicht im Bildbereich liegen.

Zum Abspeichern wird die Methode **GBitmap.save()** verwendet, die als letzten Parameter das Bildformat festlegt. Erlaubt sind die Werte: "bmp", "gif", "jpg", "png".

AUFGABEN

1. Schreibe ein Programm, das die Rot- und Grünanteile des Bildes *frog.png* vertauscht.

2. Schreibe ein Programm, wo du durch Ziehen mit der Maus das Bild drehen kannst. Verwende die Funktion *atan2(y, x)*, die dir den Winkel a zum Punkt *P(x, y)* liefert. Du musst diesen noch mit *math.degrees()* in Grad umwandeln, bevor du mit *GBitmap.scale()* das Bild drehst.



Als Testbild kannst du wieder *frog.png* nehmen.

3. Schreibe ein Retouchier-Programm, dass bei einem Mausklick die Farbe des Pixels speichert (color pick). Nachfolgendes Ziehen soll mit so gefüllten Farbkreisen in das Bild hineinzeichnen. Du musst hier den Press-, Drag- und Click-Event verwenden. Als Testbild kannst du wieder *frog.png* verwenden. Schreibe die 3 Farbkomponenten der "gepickten" Farbe in die Titelzeile des Fensters.

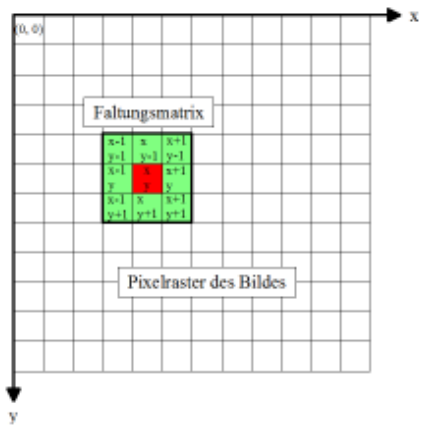
ZUSATZSTOFF: BILDFILTERUNG DURCH FALTUNG

Dir ist sicher bekannt, dass man in den bekannten Bildverarbeitungsprogrammen ein Bild durch verschiedenartige Filter verändern kann. Es gibt beispielsweise Glättungsfilter, Schärfungsfilter, Weichzeichnungsfilter, usw. Dabei kommt das wichtige Prinzip der **Faltung** zur Anwendung, das du hier kennen lernst [**mehr...**]. Man verändert dabei die Farbwerte jedes Bildpixels, indem man gemäss einer Filterregel einen neuen Wert aus ihm und seinen nächsten 8 Nachbarn berechnet.

Im Detail funktioniert dies wie folgt: Gehe der Einfachheit halber von einem Graustufenbild aus, wo jeder Pixel im RGB-Farbmodell einen Grauwert v zwischen 0 und 255 besitzt. Die Filterregel wird durch 9 Zahlen festgelegt, die sich in einem Quadrat anordnen lassen:

m00 m01 m02
m10 m11 m12
m20 m21 m22

Diese Darstellung nennt sich **Faltungsmatrix** (auch Maske genannt). In Python implementieren wir sie zeilenweise in einer Liste:
mask = [[0, -1, 0], [-1, 5, 1], [0, -1, 0]]



Mit dieser Datenstruktur kannst du leicht mit Doppelindizes auf die einzelnen Werte zugreifen, beispielsweise ist m12 = mask[1][2] = 1. Diese neun Zahlen sind Gewichtsfaktoren für einen Bildpunkt und seine 8 Nachbarn. Man berechnet nun den neuen Grauwert vnew eines Bildpunkts an der Stelle x, y aus den vorhandenen 9 Werten v(x, y) wie folgt:

```
vnew(x, y) =   m00 * v(x - 1, y -1)   + m01 * v(x, y - 1)   + m02 * v(x + 1, y - 1) +
               m10 * v(x - 1, y)      + m11 * v(x, y)       + m12 * v(x + 1, y) +
               m20 * v(x - 1, y + 1)  + m21 * v(x , y + 1)  + m22 * v(x + 1, y + 1)
```

Anschaulich könnte man sagen, dass man die Faltungsmatrix für die Neuberechnung über den Bildpunkt legt, ihre Werte mit den darunter liegenden Grauwerten multipliziert und aufsummiert.

Das Programm führt diese Faltungsoperation auf alle Bildpunkte (ausser den Randpunkten) aus und speichert die neuen Grauwerte in einer neuen Bitmap, die dann dargestellt wird. Dazu bewegst du sozusagen mit einer for-Struktur die Faltungsmatrix zeilenweise von links nach

rechts und von oben nach unten über das Bild. Du verwendest hier die Faltungsmatrixwerte eines Scharfzeichnungsfilters und das Graustufenbild *frogbw.png* des Frosches.

```
from gpanel import *

size = 300

makeGPanel(Size(2 * size, size))
window(0, size, size, 0)    # y axis downwards

bmIn = getImage("sprites/frogbw.png")
image(bmIn, 0, size)
w = bmIn.getWidth()
h = bmIn.getHeight()
bmOut = GBitmap(w, h)

#mask = [[1/9, 1/9, 1/9], [1/9, 1/9, 1/9], [1/9, 1/9, 1/9]]  # smoothing
mask = [[ 0, -1,  0], [-1,  5, -1], [0,  -1,  0]] #sharpening
#mask = [[-1, -2, -1], [ 0,  0,  0], [ 1,  2,  1]] #horizontal edge extraction
#mask = [[-1,  0,  1], [-2,  0,  2], [-1,  0,  1]] #vertical edge extraction

for x in range(0, w):
    for y in range(0, h):
        if x > 0 and x < w - 1 and y > 0 and y < h - 1:
            vnew = 0
            for k in range(3):
                for i in range(3):
                    c = bmIn.getPixelColor(x - 1 + i, y - 1 + k)
                    v = c.getRed()
                    vnew +=  v * mask[k][i]
            # Make int in 0..255
            vnew = int(vnew)
            vnew = max(vnew, 0)
            vnew = min(vnew, 255)
            gray = Color(vnew, vnew, vnew)
        else:
            c = bmIn.getPixelColor(x, y)
            v = c.getRed()
            gray = Color(v, v, v)

        bmOut.setPixelColor(x, y, gray)

image(bmOut, size / 2, size)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Bei der Faltung wird jeder Bildpunkt durch ein gewichtetes Mittel aus sich und den Nachbarpunkten ersetzt. Die Filterart wird durch die Faltungsmatrix festgelegt.

Du kannst mit den folgenden bekannten Faltungsmatrizen experimentieren, aber auch eigene erfinden.

Filterart	Faltungsmatrix
Scharfzeichnungsfilter	$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$
Weichzeichnungsfilter	$\begin{pmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix}$
Kantenfilter (horizontal)	$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$
Kantenfilter (vertikal)	$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$

3.12 BILDER DRUCKEN

EINFÜHRUNG

Du hast bereits im Kapitel Turtlegrafik die Turtle auf einem Drucker hochauflösend zeichnen lassen. Gleichartig kannst du ein Bild von GPanel auf dem Drucker rendern. Du kannst auch einen virtuellen Drucker verwenden, der eine Grafik-Datei in einem hochauflösenden Format (z.B. Tiff oder EPS) erstellt [**mehr...**]. Dazu definierst du eine parameterlose Funktion mit irgend einem Namen, beispielsweise *doIt()*, die alle Befehle zur Erstellung des Bildes enthält. Beim direkten Aufruf erscheint das Bild auf dem Bildschirm. Um es auszudrucken, rufst du *printerPlot(doIt)* auf. Du kannst auch noch einen Skalierungsfaktor k angeben, also *printerPlot(doIt, k)* aufrufen. Für k < 1 ergibt sich eine Verkleinerung, für k > 1 eine Vergrösserung.

PROGRAMMIERKONZEPTE: *Hochauflösende Grafik*

ROSETTEN

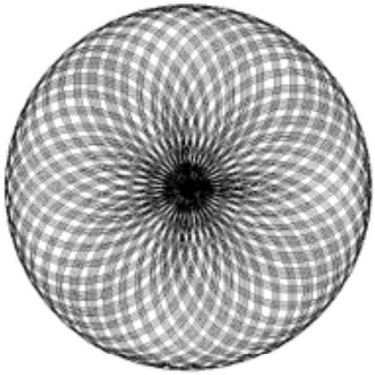
Die rosenartigen Kurven gehen auf dem Mathematiker Guido Grandi aus dem 18. Jahrhundert zurück [**mehr...**]
Die erzeugende Funktion lässt sich am einfachsten in Polarkoordinaten (ρ, φ) ausdrücken. Sie besitzt einen Parameter n

$$\rho = \sin(n\varphi)$$

Die kartesischen Koordinaten erhält man wie üblich aus

$$x = \rho \cos(\varphi)$$
$$y = \rho \sin(\varphi)$$

Eine hübsche Rosette erhältst du mit $n = \sqrt{2}$. Auf einem Drucker sieht sie aber noch viel schöner aus als auf dem Bildschirm.



```
from gpanel import *
import math

def rho(phi):
    return math.sin(n * phi)

def doIt():
    phi = 0
    while phi < nbTurns * math.pi:
        r = rho(phi)
        x = r * math.cos(phi)
        y = r * math.sin(phi)
        if phi == 0:
          move(x, y)
        else:
          draw(x, y)
        phi += dphi

n = math.sqrt(2)
dphi = 0.01
nbTurns = 100
```

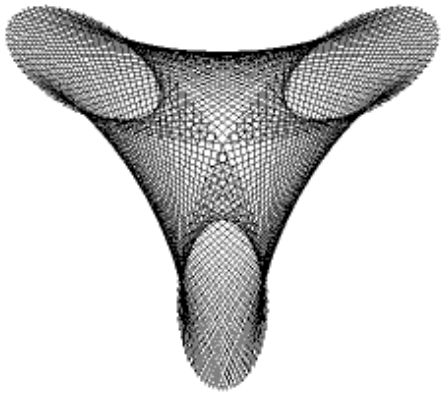
```
makeGPanel(-1.2, 1.2, -1.2, 1.2)
doIt()
printerPlot(doIt)
```

MEMO

Je nach der Wahl des Parameters n kannst du ganz verschiedenartige Kurven erzeugen. Versuche es mit natürlichen Zahlen, mit rationalen Zahlen (Brüchen) und mit irrationalen Zahlen (π, e).

MAURER-ROSEN

Der Mathematiker Peter Maurer hat diese Kurven 1987 in seinem Artikel "A Rose is a Rose..." eingeführt. Sie verwenden die Rosetten als "Leitlinie". Auf dieser Leitlinie wählst immer nach einem bestimmten Drehwinkel d insgesamt 360 Punkte. Nachher verbindest du diese Punkt mit Geradenstücken.



Je nach Wahl von n und d ergeben sich ganz unterschiedliche Kurvengebilde. Ausgedruckt wirken diese noch viel schöner (hier für n = 3 und d = 47 Grad).

```
from gpanel import *
import math

def sin(x):
    return math.sin(math.radians(x))

def cos(x):
    return math.cos(math.radians(x))

def cartesian(polar):
    return [polar[0] * cos(polar[1]), polar[0] * sin(polar[1])]

def rho(phi):
    return sin(n * phi)

def doIt():
    for i in range(361):
        k = i * d
        pt = [rho(k), k]
        corners.append(pt)

    move(cartesian(corners[0]))
    for pt in corners:
        draw(cartesian(pt))

corners = []
n = 3
d = 47
makeGPanel(-1.2, 1.2, -1.2, 1.2)
doIt()
printerPlot(doIt)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Im Programm verwendest du Grad- und nicht Bogenmass. Darum ist es günstig, deine eigenen Funktionen für Sinus und Cosinus zu definieren, die mit Grad rechnen. Dies vereinfacht auch die Schreibweise, da du nicht immer *math.* davor schreiben musst.

Ebenso ist es günstig, eine Umrechnung von Polar- auf kartesische Koordinaten in der Funktion *cartesian()* vorzunehmen, wo die Koordinatenpaare als Liste verpackt sind.
Die Polarkoordinaten der 361 Punkte, die du auf der Leitlinie auswählst, speicherst du in der Liste *corners*. Am Ende durchläufst du diese und ziehst mit *draw()* Linien von Punkt zu Punkt.

Bekannte andere Maurer-Rosen kannst du mit folgenden Parametern zeichnen:

n	d
2	39
2	31
6	71

AUFGABEN

1. Zeichne mit der Funktion *wave(center, wavelength)* 50 konzentrische Kreise mit *center* als Mittelpunkt und *wavelength* als Radiusinkrement. Man kann das Bild als Wellenberge einer Kreiswelle auffassen. Zeichne die Welle mit wenig verschobenen Mittelpunkt und beobachte auf einem Ausdruck das entstehende Interferenzbild. Welche aus der Geometrie bekannte Kurve erkennst du?

3.13 WIDGETS

EINFÜHRUNG

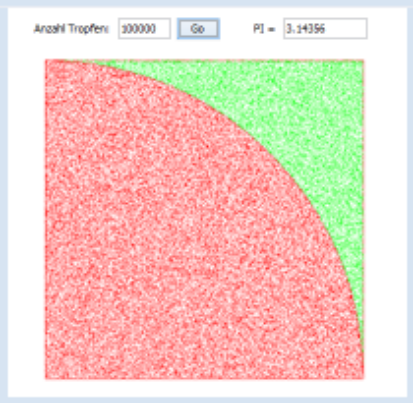
Dir bekannte Programme besitzen meist eine grafische Benutzeroberfläche, ein GUI (Graphics User Interface). Du erkennst gewöhnlich eine Menüleiste, Eingabefelder und Schaltflächen (Buttons). GUI-Komponenten, auch Widgets genannt, werden als Objekte aufgefasst, so wie du sie aus dem Kapitel *Turtleobjekte* bereits kennst. Willst du Programme mit einer modernen Benutzeroberfläche schreiben, so ist es darum unumgänglich, dass du die grundlegenden Konzepte der Objektorientierten Programmierung (OOP) kennst. Die Widgets werden in verschiedene Klassen gemäss folgende Liste eingeteilt:

Widget	**Klasse**
Schaltflächen (Buttons)	JButton
Beschriftungen (Labels)	JLabel
Eingabefelder (Textfelder)	JTextField
Menüleiste (MenuBar)	JMenuBar
Menüeintrag	JMenuItem
Menü mit Menüeinträgen	JMenu

So wie du eine Turtle mit dem Aufruf des Konstruktors der Klasse *Turtle* erzeugt hast, musst du eine GUI-Komponente durch den Aufruf des entsprechenden Klassenkonstruktors erzeugen. Oft besitzen die Konstruktoren Parameter, mit denen du bestimmte Eigenschaften des Widgets festlegen kannst. Beispielsweise erzeugst du ein Eingabefeld der Länge 10 Zeichen mit *tf = JTextField(10)*. Beim Aufruf des Konstruktors wir auch eine Variable definiert, mit der du nachher auf das Objekts zugreifst. Beispielsweise liefert *tf.getText()* den Text, der im Textfeld steht. Um ein Widget in einem GPanel sichtbar zu machen, verwendest du die Funktion *addComponent()* und übergibst ihr die Objektvariable. Dabei werden die Widgets automatisch in der Reihenfolge der Aufrufe im oberen Teil des GPanel-Fensters plaziert.

PROGRAMMIERKONZEPTE: *Grafische Benutzeroberfläche, GUI-Komponente, GUI-Callback*

PI MIT DER REGENTROPFEN-SIMULATION



Du hast bereits gelernt, wie man mit der Monte-Carlo-Simulation eine Fläche bestimmen kann. Stell dir vor, dass du in einem Quadrat mit Seitenlänge 1 einen Viertelkreis mit Radius 1 einzeichnest. Wenn du nun n Regentropfen gleichmässig auf das Quadrat fallen lässt, so kannst du dir leicht überlegen, wie viele davon in Mittel auf den Viertelkreis fallen. Da die Viertelkreisfläche

$$S = \frac{1}{4} r^2 \pi = \frac{\pi}{4}$$

ist und das Quadrat die Fläche 1 besitzt, sind es nämlich

$$k = n * \frac{\pi}{4}$$

Tropfen. Lässt du also in einer Computersimulation n Tropfen fallen und zählst k, so kriegst du einen Näherungswert für Pi aus

$$\pi = 4 * k / n .$$

Das GUI besteht aus zwei Labels, zwei Textfeldern und einem Button. Nach ihrer Erzeugung fügst du sie mit **addComponent()** ins GPanel ein. Es versteht sich von selbst, dass das Klicken auf den OK-Button als Event aufgefasst wird. Der Callback wird über den benannten Parameter **actionListener** im Konstruktor von *JButton* registriert. Du hast sicher nicht vergessen, dass du in einem Callback keinen lang dauernden Code ausführen solltest. Darum rufst du im Callback lediglich **wakeUp()** auf, wodurch das mit **putSleep()** in der while-Schleife angehaltene Programm aufgeweckt wird und es die Simulation ausführt.

```
from gpanel import *
import random
from javax.swing import *

def actionCallback(e):
    wakeUp()

def createGUI():
    addComponent(lbl1)
    addComponent(tf1)
    addComponent(btn1)
    addComponent(lbl2)
    addComponent(tf2)
    validate()

def init():
    tf2.setText("")
    clear()
    move(0.5, 0.5)
    rectangle(1, 1)
    move(0, 0)
    arc(1, 0, 90)

def doIt(n):
    hits = 0
    for i in range(n):
        zx = random.random()
        zy = random.random()
        if zx * zx + zy * zy < 1:
            hits = hits + 1
            setColor("red")
        else:
            setColor("green")
        point(zx, zy)
    return hits

lbl1 = JLabel("Number of drops: ")
lbl2 = JLabel("             PI = ")
tf1 = JTextField(6)
tf2 = JTextField(10)
btn1 = JButton("OK", actionListener = actionCallback)

makeGPanel("Monte Carlo Simulation", -0.1, 1.1, -0.1, 1.1)
createGUI()
tf1.setText("10000")
init()
while True:
    putSleep()
    init()
    n = int(tf1.getText())
    k = doIt(n)
    pi =  4 * k / n
    tf2.setText(str(pi))
```

MEMO

Widgets sind Objekte der Swing-Klassenbibliothek. Sie werden mit dem Konstruktor, der den Namen der Klasse hat, erzeugt. Beim Aufruf des Konstruktors wird eine Variable definiert, mit der du auf das Objekt zugreifen kannst. Um das Widget im GPanel anzuzeigen, rufst du die Funktion **addComponent()** auf und übergibst ihr die Widget-Variable. Nachdem du alle Widgets zum GPanel hinzugefügt hast, solltest du **validate()** aufrufen, damit das Fenster mit Sicherheit mit den eingefügten Widgets neu aufgebaut wird. Button-Callbacks kannst du mit dem benannten Parameter **actionListener** registrieren. Denke daran, dass ein Callback nie lange dauernden Code ausführen sollte.

MENÜS (Nichts zum Essen)

Viele Bildschirmfester besitzen eine Menüleiste mit mehreren Menüeinträgen (Items). Beim Klicken auf einen Menüeintrag, kann auch ein Untermenü geöffnet werden, das wiederum Menüeinträge enthält. Menüs und Menüobjekte werden ebenfalls als Objekte aufgefasst. Die Auswahl einer Menüoption löst einen Event aus, der über einen Callback behandelt wird.

Ein Menü baust du so auf, dass du ein Objekt von **JMenuBar()** erzeugst und diesem mit *add()* Objekte von *JMenuItem* hinzufügst. Du kannst aber auch ein Untermenü hinzufügen. Dazu erstellst du ein Objekt von *JMenu*, und fügst ihm Objekte von *JMenuItem* hinzu. Ein Menü ist also hierarchisch aufgebaut.

Um den Code etwas zu vereinfachen, kannst du für alle Menüoptionen den gleichen Callback *actionCallback()* verwenden. Du registriert ihn bei jedem Konstruktor von **JMenuItem** mit dem benannten Parameter **actionPerformed**. Im Callback kannst du dann mit **getSource()** bestimmen, durch welche Menüoption der Callback ausgelöst wurde.

```
from gpanel import *
from javax.swing import *

def actionCallback(e):
    if e.getSource() == goItem:
        wakeUp()
    if e.getSource() == exitItem:
        dispose()
    if e.getSource() == aboutItem:
        msgDlg("Pyramides Version 1.0")

def doIt():
    clear()
    for i in range(1, 30):
        setColor(getRandomX11Color())
        fillRectangle(i/2, i - 0.35, 30 - i/2, i + 0.35)

fileMenu = JMenu("File")
goItem = JMenuItem("Go", actionPerformed = actionCallback)
exitItem = JMenuItem("Exit", actionPerformed = actionCallback)
fileMenu.add(goItem)
fileMenu.add(exitItem)

aboutItem = JMenuItem("About", actionPerformed = actionCallback)

menuBar = JMenuBar()
menuBar.add(fileMenu)
```

```
menuBar.add(aboutItem)

makeGPanel(menuBar, 0, 30, 0, 30)

while not isDisposed():
    putSleep()
    if not isDisposed():
        doIt()
```

MEMO

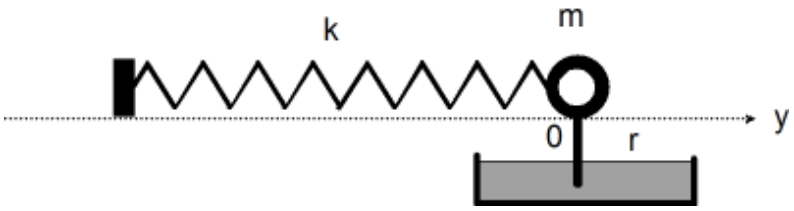
Du hältst dich an die Regel, dass in Callbacks kein lange dauernder Code ausgeführt werden soll. Du führst das Zeichnen also im Hauptblock durch. Damit das Programm mit Sicherheit beendet wird, wenn du den Close-Button des Fensters oder die Exit-Option klickst, testest du mit *isDisposed()*, ob das Fenster geschlossen wurde.

EIN SEPARATES DIALOGFENSTER BENUTZEN

TigerJython stellt dir Werkzeuge zur Verfügung, mit denen sich eigenständige Dialogfenster ohne viel Aufwand realisieren lassen. Dabei werden die klassischen Bedienungselemente wie Textfelder, Schaltflächen (Buttons), Markierungsfelder (Checkboxen), Optionsfelder (Radiobuttons) und Schieberegler (Sliders) als Python-Objekte modelliert, die sich in Bereichen (Panes) eines Bildschirmfensters befinden, das während der Programmausführung ständig sichtbar bleibt. Ein solches Fenster nennt man auch einen nicht-modalen Dialog.

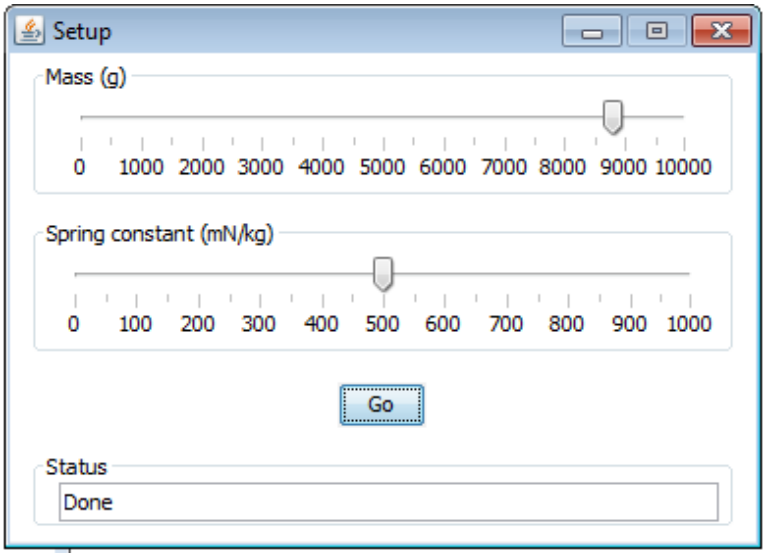
Das Dialogfenster wird durch ein Objekt der Klasse *EntryDialog()* erzeugt, wo die einzelnen Bereiche als Objekte der Klasse *EntryPane()* in der Reihenfolge der Parameter untereinander eingefügt sind. Die EntryPanes enthalten die Bedienungselemente als Objekte der Klassen *ButtonEntry, RadioEntry, CheckEntry, IntEntry, FloatEntry, LongEntry, StringEntry und SliderEntry*. (Du kannst dich in der APLU-Dokumentation genauer orientieren.)

Dein Programm simuliert die Bewegung einer Masse, die an einer Feder befestigt ist und eine geschwindigkeitsproportionale Reibung erfährt. Es könnte sich um die Aufhängung eines Autorads handeln, das mit einem Stossdämpfer versehen ist.



Gemäss dem Newtonschen Bewegungsgesetz berechnet sich die Beschleunigung aus a = F/m, wo F die Kraft und m die Masse sind. Das Feder- und das Reibungsgesetz liefern F = -k*x - r*v, wo k die Federkonstante und r der Reibungskoeffizient sind. Die Lösung erfolgt iterativ in kleinen Zeitschritten dt: Die neue Geschwindigkeit ist v = v + a*dt und die neue y-Koordinate y = y + v*dt.

Du verwendest ein Dialogfenster mit einem Schieberegler für den Reibungskoeffizenten, einer Statuszeile, um Rückmeldungen an den Benutzer zu geben, und einem Start-Button, um die Simulation zu starten. Es werden drei Objekte friction, status und btn erzeugt, die du je in eine *EntryPane pane1*, *pane2*, *pane3* fügst. Diese übergibst du dem EntryDialog, der sie in der Parameterreihenfolge darstellt.




```
from gpanel import *
import math

def doIt():
    clear()
    drawGrid(0, 200, -100, 100, "seagreen")
    m = mass.getValue() / 1000 # Mass (kg)
    k = spring.getValue() / 1000 # Spring const (N/kg)
    t = 0; y = 50; v = 0 # Initial conditions
    r = 0.7       # Coefficient of friction (N/m/s)
    d = 200; dt = 0.01
    status.setValue("m = %6.2f kg, k = %6.2f N/kg" % (m, k))
    move(t, y)  # Initial cursor position
    while t < d:
        draw(t, y)      # Draw segment
        F = -k*y - r*v  # Force
        a = F/m         # Acceleration
        v = v + a*dt    # New velocity
        y = y + v*dt    # New position
        t = t + dt      # New time
    T = 2 * math.pi * math.sqrt(m / k)
    status.setValue("Done")

mass = SliderEntry(0, 10000, 10000, 1000, 500)
pane1 = EntryPane("Mass (g)", mass)
spring = SliderEntry(0, 1000, 500, 100, 50)
pane2 = EntryPane("Spring constant (mN/kg)", spring)
goBtn = ButtonEntry("Go")
pane3 = EntryPane(goBtn)
status = StringEntry("")
pane4 = EntryPane("Status", status)
dlg = EntryDialog(850, 150,
        pane1, pane2, pane3, pane4)

makeGPanel()
window(-20, 220, -110, 110)
drawGrid(0, 200, -100, 100, "seagreen")
title("Harmonic Oscillation")
status.setValue("Press Go to start")
while not dlg.isDisposed():
    if isDisposed():
        dlg.dispose()
        break
    if goBtn.isTouched():
        doIt()
dispose()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

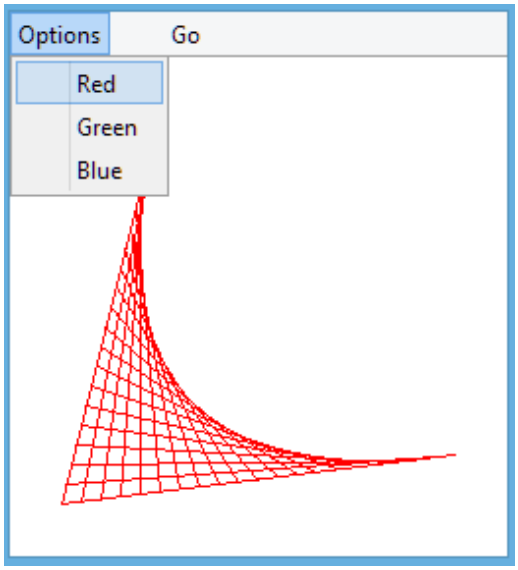
Damit das Programm bei Klicken auf den Close-Button der Titelleiste beendet wird, verwendest du in einer while-Schleife die Bedingung *isDisposed()*, die wahr wird, sobald der Close-Button gedrückt wird. In *doIt()* frägst die den Zustand der Bedienungselemente mit *getValue()* ab. Mit *isTouched()* findest du heraus, ob du mit der Maus auf ein Bedienungselement geklickt hast, hier zum Beispiel auf den Start-Button.

Durch einige Versuche findest du heraus, dass sich die Schwingungsdauer mit zunehmender Dämpfung wenig ändert und dass es einen Grenzfall gibt, wo keine Schwingung mehr auftritt. Dieser asymptotische Grenzfall wird bei Stossdämpfern von Autos angewendet, damit das Auto möglichst wenig schwingt.

AUFGABEN

1. Gehe vom Programm Moiré im Kapitel 3.2 aus und füge ein Textlabel, ein Eingabefeld für die Verzögerungszeit und einen OK-Button hinzu. Beim Klicken des OK-Buttons wird die Grafik mit der eingegebenen Verzögerungszeit (in Millisekunden) immer wieder neu erstellt.

2. Gehe vom Programm unter "Elegante Fadengrafik-Algorithmen" im Kapitel 3.8 aus und füge ihm folgendes Menü hinzu: Der Menüeintrag "Options" soll ein Untermenü mit den Einträgen "Red", "Green" und "Blue" enthalten. Der Menüeintrag "Go" soll die Fadengrafik mit der unter Options ausgewählten Farbe zeichnen. Ist noch keine Farbe gewählt, so wird mit schwarzer Farbe gezeichnet.



3*. Nehme eines deiner Lieblingsprogramme zur GPanel-Grafik und füge ihm einige sinnvolle Widgets hinzu.

Dokumentation GPanel

Module import: from gpanel import *

Befehl	Aktion
makeGPanel()	erzeugt ein (globales) GPanel-Grafikfenster mit Koordinaten (0, 1, 0, 1). Cursor auf (0, 0)
makeGPanel(xmin, xmax, ymin, ymax)	erzeugt ein (globales) GPanel-Grafikfenster mit angegebenen Koordinaten.
makeGPanel(xmin, xmax, ymin, ymax, isVisible)	wie oben, aber mit Wahl es zu verbergen, bis gewisse andere Aktionen ausgeführt sind
makeGPanel(Size(width, height))	wie makeGPanel(), aber Fenstergrösse wählbar (in Pixels)
getScreenWidth()	gibt Bildschirmbreite zurück (in Pixel)
getScreenHeight()	gibt Bildschirmhöhe zurück (in Pixel)
window(xmin, xmax, ymin, ymax)	setzt ein neues Koordinatensystem
drawGrid(x, y)	zeichnet Gitter mit 10 Ticks im Bereich 0..x, 0..y
drawGrid(x, y, color)	dasselbe mit Angabe der Farbe
drawGrid(x1, x2, y1, y2)	dasselbe im Bereich x1..x2, y1..y2
drawGrid(x1, x2, y1, y2, color)	dasselbe mit Angabe der Farbe
drawGrid(x1, x2, y1, y2, x3, y3)	dasselbe mit Angabe der Anzahl Ticks x3, y3 in x- und y-Richtung
drawGrid(x1,x2,y1,y2,x3,y3,color)	dasselbe mit Angabe der Farbe
drawGrid(p, ...)	wie drawGrid(...) mit Angabe des GPanel-Referenz (für mehere GPanels)
visible(isVisible)	macht Fenster sichtbar/unsichtbar
resizeable(isResizeable)	aktiviert, deaktiviert das Zoomen des Fensters
dispose()	schliesst das Fenster
isDisposed()	gibt True zurück, falls das Fenster geschlossen ist
bgColor(color)	setzt die Hintergrundfarbe (X11-Farbstring oder Color type)
title(text)	setzt den Text in der Titelzeile
makeColor(colorStr)	gibt Farbe als Color type zum gegebenen X11-Farbstring zurück
windowPosition(ulx, uly)	setzt das Fenster an die gegebene Bildschirmposition
windowCenter()	setzt das Fenster in Bildschirmmitte
storeGraphis()	speichert die Grafik in einem internen Buffer
recallGraphics()	rendert die Grafik aus dem internen Buffer
clearStore(color)	löscht den internen Buffer (überschreibt mit Farbe)
delay(time)	Programm um Zeit time (Millisekunden) anhalten
getDividingPoint(pt1, pt2, ratio)	Teilpunkt der Strecke mit Punktlisten pt1, pt2, der sie im Teilverhältnis ratio teilt
getDividingPoint(c1, c2, ratio)	Teilpunkt der Strecke mit complex c1, c2, der sie im Teilverhältnis ratio teilt
clear()	löscht Fensterinhalt (füllt mit Hintergrundfarbe) und setzt Cursor auf (0, 0)
erase()	löscht Fensterinhalt (füllt mit Hintergrundfarbe) ohne Veränderung des Cursors
putSleep()	hält den Programmablauf an, bis wakeUp() aufgerufen wird
wakeUp()	führt angehaltenen Programmablauf weiter
linfit(X, Y)	führt eine lineare Regression y = a*x + b mit Daten in X- und Y Liste aus und gibt (a, b) zurück

Zeichnen

lineWidth(width)	setzt die Stiftdicke (in Pixel)
setColor(color)	setzt die Stiftfarbe (X11-Farbstring oder Colortype)
move(x, y)	setzt Cursor auf (x, y) ohne zu zeichnen
move(liste)	setzt Cursor auf Punktliste = [x, y] ohne zu zeichnen
move(c)	setzt Cursor auf complex(x, y) ohne zu zeichnen
getPosX()	liefert x-Koordinate des Cursors
getPosY()	liefert y-Koordinate des Cursors
getPos()	liefert Cursorposition als Punktliste
draw(x, y)	zeichnet Linie zu neuem Punkt (x, y) und ändert Cursor
draw(liste)	zeichnet Linie zu Punktliste = [x, y] und ändert Cursor
draw(c)	zeichnet Linie zu complex(x, y) und ändert Cursor
line(x1, y1, x2, y2)	zeichnet Linie von (x1, y1) zu (x2, y2) ohne Änderung des Cursors
line(pt1, pt2)	zeichnet Linie von pt1 = [x1, y1] zu pt2 = [x2, y2]
line(c1, c2)	zeichnet Linie von complex(x1, y1) zu complex(x2, y2)
circle(radius)	zeichnet Kreis mit Mitte bei Cursor und gegebenem Radius
fillCircle(radius)	zeichnet gefüllten Kreis (Füllfarbe = Stiftfarbe)
ellipse(a, b)	zeichnet Ellipse mit Mitte bei Cursor und Halbachsen
fillEllipse(a, b)	zeichnet gefüllte Ellipse (Füllfarbe = Stiftfarbe)
rectangle(a, b)	zeichnet Rechteck mit Zentrum bei Cursor und Seiten
rectangle(x1, y1, x2, y2)	zeichnet Rechteck mit gegenüberliegenden Eckpunkten
rectangle(pt1, pt2)	zeichnet Rechteck mit gegenüberliegenden Punktlisten
rectangle(c1, c2)	zeichnet Rechteck mit gegenüberliegenden complex
fillRectangle(a, b)	zeichnet gefülltes Rechteck (Füllfarbe = Stiftfarbe)
fillRrectangle(x1, y1, x2, y2)	zeichnet gefülltes Rechteck (Füllfarbe = Stiftfarbe)
fillRectangle(pt1, pt2)	zeichnet gefülltes Rechteck (Füllfarbe = Stiftfarbe)
fillRrectangle(c1, c2)	zeichnet gefülltes Rechteck (Füllfarbe = Stiftfarbe)
arc(radius, startAngle, extendAngle)	zeichnet Kreisbogen mit Zentrum bei Cursor, Radius und Start- und Sektorwinkel (0 nach Osten, positiv im Gegenuhrzeigersinn)
fillArc(radius, startAngle, extendAngle)	zeichnet gefüllten Kreisbogen (Füllfarbe = Stiftfarbe)
polygon(x-li, y-li)	zeichnet Polygon mit Eckpunkten mit x-Koordinaten aus der Liste x-li und y-Koordinaten aus y-li
polygon((li[pt1, pt2,..)	zeichnet Polygon mit Liste aus Eckpunktlisten pt1, pt2,...
polygon(li[c1, c2, c3,...])	zeichnet Polygon mit Liste aus Eckpunktcomplex c1, c2,...
fillPolygon(x-li, y-li)	zeichnet gefülltes Polygon (Füllfarbe = Stiftfarbe)
fillPolygon((li[pt1, pt2,..)	zeichnet gefülltes Polygon (Füllfarbe = Stiftfarbe)
fillPolygon(li[c1, c2, c3,...])	zeichnet gefülltes Polygon (Füllfarbe = Stiftfarbe)
quadraticBezier(x1, y1, xc, yc, x1, y2)	zeichnet quadratische Bezier-Kurve mit 2 Punkten (x1, y1), (x2, y2) und Kontrollpunkt (xc, yc)
quadraticBezier(pt1, pc, pt2)	zeichnet quadratische Bezier-Kurve mit 3 Punktlisten
quadraticBezier(c1, cc, c2)	zeichnet quadratische Bezier-Kurve mit 3 complex
cubicBezier(x1, y1, xc1, yc1, xc2, yc2, x2, y2)	zeichnet kubische Bezier-Kurve mit 2 Punkten (x1, y1), (x2, y2) und 2 Kontrollpunkten (xc1, yc1), (xc2, yc2)
cubicBezier(pt1, ptc1, ptc2, pt2)	zeichnet kubische Bezier-Kurve mit 2 Punktlisten und 2 Kontrollpunktlisten
cubicBezier(c1, cc1, cc2, c2)	zeichnet kubische Bezier-Kurve mit 4 complex

triangle(x1, y1, x2, y2, x3, y3)	zeichnet Dreieck mit Eckpunkten
triangle(pt1, pt2, pt3)	zeichnet Dreieck mit Eckpunktlisten
triangle(li[pt1, pt2, pt3])	zeichnet Dreieck mit Liste der Eckpunkte
triangle(c1, c2, c3)	zeichnet Dreieck mit complex
fillTriangle(x1, y1, x2, y2, x3, y3)	zeichnet gefülltes Dreieck (Füllfarbe = Stiftfarbe)
fillTriangle(pt1, pt2, pt3)	zeichnet gefülltes Dreieck (Füllfarbe = Stiftfarbe)
fillTriangle(li[pt2, pt2, pt3])	zeichnet gefülltes Dreieck (Füllfarbe = Stiftfarbe)
fillTriangle(c1, c2, c3)	zeichnet gefülltes Dreieck (Füllfarbe = Stiftfarbe)
point(x, y)	zeichnet 1-Pixel-Punkt bei (x, y)
point(pt)	zeichnet 1-Pixel-Punkt bei Punktliste pt = [x,y]
point(complex)	zeichnet 1-Pixel-Punkt bei complex(x, y)
fill(x, y, color, replacementColor)	füllt geschlossenes Gebiet um Punkt (x, y). color wird durch replacementColor ersetzt (floodfill)
fill(pt, color,replacementColor)	füllt geschlossenes Gebiet um Punkt pt. color wird durch replacementColor ersetzt (floodfill)
fill(complex, color,replacementColor)	füllt geschlossenes Gebiet um complex(x, y). color wird durch replacementColor ersetzt (floodfill)
image(path, x, y)	fügt eine Bild im GIF- , PNG- oder JPG-Format mit unterer linker Ecke bei (x, y) ein. Pfad zur Bilddatei: Relativ zum Verzeichnis von TigerJython, im Distributions-JAR (Verzeichnis *sprites*) oder mit http:// vom Internet.
image(path, pt)	dasselbe mit Punktliste
image(path, complex)	dasselbe mit complex
imageHeighpath)	gibt die Bildhöhe zurück
imageWidth(path)	gibt die Bildbreite zurück
enableRepaint(boolean)	aktiviert/deaktiviert das automatische Rendern des Offscreen-Buffers (Standard: aktiviert)
repaint()	rendert den Offscreen-Buffer (nötig, wenn das Rendern deaktiviert ist)
setPaintMode()	wählt den normalen Zeichnungsmodus
setXORMode(color)	wählt den XOR-Zeichnungsmodus mit geg.Farbe
getPixelColor(x, y)	gibt Farbe am Punkt (x, y) als Color type zurück
getPixelColor(pt)	gibt Farbe an Punktliste pt = [x, y] als Color type zurück
getPixelColor(complex)	gibt Farbe an complex(x, y) als Color type zurück
getPixelColorStr(x, y)	gibt Farbe am Punkt (x, y) als X11-Farbstring zurück
getPixelColorStr(pt)	gibt Farbe an Punktliste als X11-Farbstring zurück
getPixelColorStr(complex)	gibt Farbe an Punktliste als X11-Farbstring zurück

Text

text(string)	schreibt Text mit Beginn beim aktuellen Cursor
text(x, y, string)	schreibt Text mit Beginn bei Punkt (x, y)
text(pt, string)	schreibt Text mit Beginn bei Punktliste pt = [x, y]
text(complex, string)	schreibt Text mit Beginn bei complex(x, y)
text(x, y, string, font, textColor, bgColor)	schreibt Text mit Beginn bei (x, y) mit Font, Textfarbe und Hintergrundfarbe
text(pt, string, font, textColor, bgColor)	schreibt Text mit Beginn bei Punkliste mit Font, Textfarbe und Hintergrundfarbe

text(complex,string, font, textColor, bgColor)	schreibt Text mit Beginn bei complex(x, y) mit Font, Textfarbe und Hintergrundfarbe
font(font)	setzt ein neues Standardfont
style	einer der Konstanten: Font.PLAIN, Font.BOLD, Font.ITALIC
size	ein Integer mit einer auf dem System verfügbaren Font-Grösse (in pixels)

Callbacks

makeGPanel(mouseNNN = onMouseNNN) auch mehrere, durch Komma getrennt	registriert die Callbackfunktion onMouseNNN(x, y), die beim Mausevent aufgerufen wird. Werte für NNN: Pressed, Released, Clicked, Dragged, Moved, Entered, Exited, SingleClicked, DoubleClicked
isLeftMouseButton(), isRightMouseButton()	gibt True zurück, falls beim Event die linke bzw. rechte Maustaste verwendet wurde
makeGPanel(keyPressed = onKeyPressed)	registriert die Callbackfunktion onKeyPressed(keyCode), die beim Drücken einer Tastaturtaste aufgerufen wird. keyCode ist ein für die Taste eindeutiger integer Code
getKeyModifiers()	liefert nach einem Tastaturevent einen Code für Spezialtasten (Shift, Ctrl, usw., auch kombiniert)
makeGPanel(closeClicked = onCloseClicked)	registriert die Callbackfunktion onCloseClicked(), die beim Klick des Close-Buttons der Titelzeile aufgerufen wird. Das Fenster muss mit dispose() geschlossen werden

Tastatur

getKey()	holt den letzten Tastendruck ab und liefert String zurück
getKeyCode()	holt den letzten Tastendruck ab und liefert Code zurück
getKeyWait()	wartet bis Taste gedrückt und liefert String zurück
getKeyCodeWait()	wartet bis Taste gedrückt und liefert Code zurück
kbhit()	liefert True, falls ein Tastendruck noch nicht abgeholt ist

GUI-Komponenten

add(component)	fügt component in einer Zeile am oberen Rand hinzu
validate()	baut das Fenster mit hinzugefügten Komponenten neu auf

Statusbar

addStatusBar(height)	fügt eine Statusbar mit gegebener Höhe in Pixels hinzu
setStatusText(text)	schreibt Text in Statusbar
setStatusText(text, font, color)	schreibt Text mit Font und Textfarbe in Statusbar

Koordinatenumrechung (Window-Koordinaten in xmin, xmax, ymin, ymax; User-Koordinaten: Systemnahe: Pixelkoordinaten. Bezeichnungen etwas ungewöhnlich)

toWindowX(userX)	gibt x-Koordinate (in xmin, xmax) von geg. x-Pixelkoordinate zurück
toWindowY(userY)	gibt y-Koordinate (in ymin, ymax) von geg. y-Pixelkoordinate zurück
toWindowHeigh(userHeight)	gibt dx-Koordinatenzunahme (in xmin, xmax) von geg. dx-Pixelzunahme zurück
toWindowWidth(userWidth)	gibt dy-Koordinatenzunahme (in ymin, ymax) von geg. dy-Pixelzunahme zurück
toUserX(windowX)	gibt x-Pixel-Koordiante zurück
toUserY(windowY)	gibt y-Pixel-Koordiante zurück
toUserHeight(windowHeight)	gibt dx Pixel-Koordinatenzunahme zurück
toUserWidth(windowWidth)	gibt dy Pixel-Koordinatenzunahme zurück

Dialoge

msgDlg(message)	öffnet einen modalen Dialog mit einem OK-Button und gegebenem Mitteilungstext
msgDlg(message, title = title_text)	dasselbe mit Titelangabe
inputInt(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt den eingegebenen Integer zurück (falls kein Integer, wird Dialog neu angezeigt). Abbrechen od. Schliessen beendet das Programm
inputInt(prompt, False)	dasselbe, aber Abbrechen/Schliessen beendet das Programm nicht, sondern gibt None zurück
inputFloat(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt den eingegebenen Float zurück (falls kein Float, wird Dialog neu angezeigt). Abbrechen od. Schliessen beendet das Programm
inputFloat(prompt, False)	dasselbe, aber Abbrechen/Schliessen beendet das Programm nicht, sondern gibt None zurück
inputString(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt den eingegeben String zurück. Abbrechen od. Schliessen beendet das Programm
inputString(prompt, False)	dasselbe, aber Abbrechen/Schliessen beendet das Programm nicht, sondern gibt None zurück
input(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt Eingabe als Integer, Float oder String zurück. Abbrechen od. Schliessen beendet das Programm
input(prompt, False)	dasselbe, aber Abbrechen/Schliessen beendet das Programm nicht, sondern gibt None zurück
askYesNo(prompt)	öffnet einen modalen Dialog mit Ja/Nein-Buttons. Ja gibt True, Nein gibt False zurück. Schliessen beendet das Programm
askYesNo(prompt, False)	dasselbe, aber Schliessen beendet das Programm nicht, sondern gibt None zurück

kapitel vier

SOUND

Lernziele

- Du weisst, wie Sound digitalisiert und aufbewahrt wird.
- Du kennst den Begriff Abtastrate und seine Auswirkungen.
- Du kannst mit einem eigenen Programm einen Sound aufzeichnen, gezielt verändern, abspielen und abspeichern.

"As the skills that constitute literacy evolve to accommodate digital media, computer science education finds itself in a sorry state. While students are more in need of computational skills than ever, computer science suffers dramatically low retention rates and a declining percentage of women and minorities. Studies of the problem point to the over-emphasis in computer science classes on abstraction over application, technical details instead of usability, and the stereotypical view of programmers as loners lacking creativity. Media Computation, teaches programming and computation in the context of media creation and manipulation."

In Forte, Guzdial, Not Calculation: Media as a Motivation and Context for Learning

4.1 SOUND ABSPIELEN

EINFÜHRUNG

Um ein Soundsignal im Computer zu verarbeiten, muss es zuerst digitalisiert werden. Dazu tastet man es in äquidistanten Zeitschritten ab und wandelt den Wert des Signals mit einem Analog-Digital-Wandler in jedem Abtastzeitpunkt in eine Zahl um. Aus dem Soundsignal ergibt sich danach eine Zahlenfolge, die im Computer gespeichert und verarbeitet werden kann. Unter der Abtastfrequenz (oder Abtastrate) versteht man die Anzahl Abtastwerte pro Sekunde. Sie ist für das WAV Audioformat standardisiert und kann folgende Werte annehmen: 8000, 11025, 16000, 22050 and 44100 Hertz. Je höher die Abtastfrequenz, je genauer kann nach einer Digital-Analog-Wandlung der Sound wiederhergestellt werden. Für die Qualität ist ebenfalls der Wertebereich der Abtastwerte wichtig. In der TigerJython Soundbibliothek werden die Werte immer als Integer-Zahlen im 16-bit Bereich (-32768 und 32767) in einer Liste abgespeichert.

Wir müssen unterscheiden, ob es sich um einen monauralen oder binauralen Sound handelt. Je nach dem, werden ein oder zwei Kanäle verwenden. Bei zwei Kanälen (Stereo) sind die Werte für den linken und rechte Kanal als aufeinanderfolgende Zahlen abgespeichert.

In diesem Kapitel benötigst du einen Computer mit einer Soundkarte und der Möglichkeit, Sound über einen Lautsprecher oder einen Kopfhörer anzuhören und mit einem Mikrofon aufzunehmen.

PROGRAMMIERKONZEPTE: *Sounddigitalisierung, Audiosignal, Sample, Abtastrate*

SOUND ANHÖREN

Beschaffe dir einen lustigen Soundclip im WAV Format von kurzer Dauer (ungefähr 2 bis 5 Sekunden lang). Du findest solche auf dem Internet. Kopiere die Sounddatei unter dem Namen *mysound.wav* ins gleiche Verzeichnis, in dem sich *dein Programm* befindet.

Zuerst importierst du alle Funktionen der **Soundbibliothek**. Nun kopierst du die Soundsamples in die Liste **samples** und schreibst die Information der Sounddatei in das **Konsolenfenster**, denn du benötigst die Abtastrate. In hier gezeigten Beispiel beträgt ihr Wert **22050 Hz**. Mit **openMonoPlayer** hast du die Möglichkeit den Sound abzuspielen. Übergibst du eine falsche Abtastrate, so wird der Sound mit anderer Geschwindigkeit und daher mit anderen Frequenzen abgespielt.

```
from soundsystem import *

samples = getWavMono("mysound.wav")
print getWavInfo("mysound.wav")

openMonoPlayer(samples, 22050)
play()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Funktion **getWavMono()** liefert die Soundsamples in einer Python Liste. Jeder Wert ist ein Integer im Bereich -32768 und 32767. Mit **openMonoPlayer()** wird ein Soundplayer zur Verfügung gestellt, um den Sound mit *play()* abzuspielen.

Da Listen nur eine bestimmte maximale Grösse haben dürfen, die von der Speicherfähigkeit deines Computer abhängt, können nur relativ kurze Soundclips mit *getWavMono()* eingelesen werden.

SOUNDKURVE

Es ist interessant, die Soundsamples auch grafisch darzustellen. Dazu verwendest du am einfachsten ein **GPanel**-Fenster und durchläufst in einer **for-Schleife** die Liste.

```
from soundsystem import *

samples = getWavMono("mysound.wav")
print getWavInfo("mysound.wav")

openMonoPlayer(samples, 44100)
play()

from gpanel import *

makeGPanel(0, len(samples), -33000, 33000)
for i in range(len(samples)):
    draw(i, samples[i])
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Wir müssen das Koordinatensystem des GPanels günstig wählen. In der x-Richtung liegen die Werte zwischen 0 und der Anzahl der Abtastwerte, die gleich der Länge der Samples-Liste ist. In der y-Richtung liegen die Werte zwischen -32768 und 32767. Darum verwenden wird einen Bereich von +-33000.

ES GEHT NOCH EINFACHER

Falls du eine Sounddatei nur abspielen willst, so benötigst du nur drei Zeilen Code. Damit kannst du sogar auch lange Sounds, z.B. deine Lieblingssongs abspielen.

```
from soundsystem import *

openSoundPlayer("meinlieblingssong.wav")
play()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du kannst auch einige Soundclips verwenden, die sich in der Distribution von TigerJython befinden. Dazu wählst du einen der folgenden Dateinamen:

Sounddatei	Beschreibung
wav/bird.wav	Vogelzwitschern
wav/boing.wav	Aufschlag
wav/cat.wav	Katzenmiauen
wav/click.wav	Klick
wav/dummy.wav	Leerer Sound
wav/explode.wav	Explosion
wav/frog.wav	Froschquaken
wav/mmm.wav	Esslaut
wav/notify.wav	Notifikationsclip
wav/ping.wav	Ping-Laut

(Die Liste wird laufend ergänzt. Falls sich in deinem eigenen Unterverzeichnis wav eine Sounddatei mit gleichem Namen befindet, wird diese verwendet.)

Der SoundPlayer kennt wie ein professioneller Musik-Player viele Steuerungsbefehle. Man kann beispielsweise den Song mit *pause()* anhalten und mit *play()* an derselben Stelle weiterspielen lassen.

Die Dauer des Sounds ist bei diesen Funktionen nicht begrenzt, da der Sound nur in kleinen Paketen eingelesen und abgespielt wird (Streaming Player).

play()	Spielt Sound von der aktuellen Position ab und kehrt sofort zurück
blockingPlay()	Spielt Sound von der aktuellen Position und wartet, bis er fertig gespielt ist
advanceFrames(n)	Verschiebt die aktuelle Position um die Anzahl Abtastwerte nach vorne
advanceTime(t)	Verschiebt die aktuelle Position um die angegebene Zeit nach vorne
getCurrentPos()	Gibt die aktuelle Position zurück
getCurrentTime()	Gibt die aktuelle Spielzeit zurück
pause()	Hält das Abspielen an. Mit play() wird weitergespielt
rewindFrames(n)	Schiebt die aktuelle Position um die Anzahl Abtastwerte zurück
rewindTime()	Schiebt die aktuelle Position um die angegebene Zeit zurück
stop()	Hält das Abspielen an. Die Abspielposition wird an den Anfang gesetzt
setVolume(v)	Stellt die Lautstärke ein (Werte zwischen 0 und 1000)

MP3 SOUNDDATEIEN ABSPIELEN

Zum Abspielen von Sounds im MP3 Format, werden zusätzliche Bibliotheksdateien benötigt, die du separat von **hier** herunterlädst und auspackst. Im Verzeichnis, in dem sich *tigerjython2.jar* befindet, erstellst du dann das Unterverzeichnis *Lib* (falls es noch nicht vorhanden ist) und kopierst die ausgepackten Dateien hinein.

Statt *openSoundPlayer()*, *openMonoPlayer()* und *openStereoPlayer()* verwendest du für MP3-Dateien dann **openSoundPlayerMP3()**, *openMonoPlayerMP3()* und *openStereoPlayerMP3()* und gibst den Pfad zur Sounddatei an. Zum Abspielen stehen dir die gleichen oben angegebenen Funktionen zur Verfügung.

```
from soundsystem import *

openSoundPlayerMP3("song.mp3")
play()
```

MEMO

Um MP3-Dateien abzuspielen, benötigst du zusätzliche JAR-Bibliotheksdateien, die sich im Unterverzeichnis Lib des Homeverzeichnisses von tigerjython2.jar befinden müssen.

AUFGABEN

1. Erkläre, warum die Tonfrequenzen verändert werden, falls man die Abtastrate beim Abspielen verändert.

2. Zeige im GPanel nur den kurzen Bereich von 0.1 Sekunden beginnend bei 1 s Spielzeit an. Erkläre das Bild.

3. Erstelle einen Soundplayer mit einem GPanel, wo mit der Tastatur die folgenden Befehle ausgeführt werden können:

Taste	Aktion
Cursor up	play
Cursor down	pause
Cursor left	rewind um 10 s
Cursor right	adwance um 10 s
Buchstabe s	stop

   Schreibe die Befehlsliste als Text in das Fenster. Bei jedem Tastendruck soll die Aktion in der Titelzeile ausgeschrieben werden.

4.2 SOUND BEARBEITEN

EINFÜHRUNG

Wie du weisst, werden die Soundsamples (Abtastwerte eines Sounds) in einer Liste abgespeichert und können mit dieser Liste wieder abgespielt werden. Willst du den Sound bearbeiten, so kannst du also ganz einfach diese Liste entsprechend verändern.

PROGRAMMIERKONZEPTE: *Rechteckschwingung, Ganzzahldivision, Modulo-Operation*

LAUTSTÄRKE VERÄNDERN

Das Programm soll die Lautstärke des Sounds auf einen Viertel reduzieren. Dazu kopierst du die Soundliste in eine **andere Liste**, wobei jedes Listenelement auf **¼** seines ursprünglichen Werts gesetzt wird.

```
from soundsystem import *

samples = getWavMono("mysound.wav")
soundlist = []
for item in samples:
    soundlist.append(item // 4)

openMonoPlayer(soundlist, 22010)
play()
```

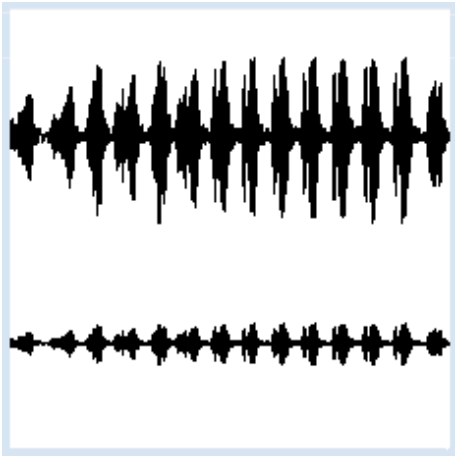
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Um eine Liste umzukopieren, erstellst du zuerst eine **leere Liste** und füllst sie dann mit **append()**. Damit du wieder eine Liste mit Integers erhältst, musst duwir die **Ganzzahldivision** (mit verdoppeltem Bruchstrich) verwenden.

VERWENDEN DES LISTENINDEX

Im folgenden Beispiel durchläufst du die Liste über ihren **Listenindex** und **veränderst die Listenelemente** ohne eine neue Liste zu erstellen. Du stellst den Sound vor und nach der Veränderung grafisch dar.



```
from soundsystem import *
from gpanel import *

samples = getWavMono("mysound.wav")

makeGPanel(0, len(samples), -33000, 33000)
for i in range(len(samples)):
    if i == 0:
        move(i, samples[i] + 10000)
    else:
        draw(i, samples[i] + 10000)

for i in range(len(samples)):
    samples[i] = samples[i] // 4

for i in range(len(samples)):
    if i == 0:
        move(i, samples[i] - 10000)
    else:
        draw(i, samples[i] - 10000)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Als Listenindex verwendet man oft den Variablennamen i. Wenn du mit einer for-Struktur

```
for i in range(10):
```

den Schleifenblock durchläufst, nennt man i auch einen ***Stepper***.

SOUNDS GENERIEREN

Es ist spannend, eigene Sounds zu erzeugen, indem du die Soundliste nicht von einer Sounddatei einliest, sondern die Listenelemente selbst erzeugst. Für einen "Rechtecksound" speicherst du über einen gewissen Indexbereich die Werte 5000 und nachfolgend über den gleich langen Bereich -5000 in die Liste.

```
from soundsystem import *

samples = []
for i in range(4 * 5000):
   value = 5000
   if i % 10 == 0:
       value = -value
   samples.append(value)

openMonoPlayer(samples, 5000)
play()
```

MEMO

Die Abtastrate 10000 Hz entspricht einem Soundsample alle 0.1 ms. Wir wollen immer nach 10 Werten, also alle 1 ms das Vorzeichen ändern. Dies entspricht einer Rechteckperiode von 2 ms, was einen Klang von 500 Hz ergibt. Wir verwenden den Modulo-Operator %, der den Rest bei der Ganzzahldivision liefert. Die Bedingung **i % 10 == 0** ist also wahr bei i = 0, 10, 20, 30, usw.

AUFGABEN

1. Verwende die Listenoperation *reverse()*, um einen Sound von hinten nach vorne abzuspielen, z.B. einen gesprochenen Text.

2. Mit der Slice-Notation liste[start: end] kann man Liste erstellen, die nur die Elemente mit dem Index *start* bis *end* (ohne das letzte Element) enthält. Entferne damit einen Teil eines dir vorliegenden Sounds.

3. Lese einen Soundclip ein und bestimme den maximalen Amplitudenwert. Schreibe ihn in die Titelzeile des GPanels und stelle den Sound grafisch dar. Erhöhe nun alle Soundsamples so, dass der maximale Amplitudenwert 32767 beträgt (maximale Lautstärke) und stelle den Clip wieder dar. (Wichtige Funktion von bekannten Soundeditoren).

4*. Erzeuge mit der Abtastrate von 10000Hz einen Sinuston von ungefähr 500 Hz, indem du die Sinusfunktion math.sin(x) verwendest, die immer nach x = 2π = 6.28 neu startet. Um Zugriff auf die Sinusfunktion zu erhalten, musst du *import math* einfügen.

5*. Überlagere zwei Sinustöne mit benachbarten Frequenzen. Was stellst du beim Zuhören fest?

4.3 SOUND AUFZEICHNEN

EINFÜHRUNG

Mit dem Soundsystem kann man auch Sounds aufnehmen und abspeichern. Dazu musst du den Soundkarten-Eingang mit einer externen Quelle, z.B. einem Mikrofon oder einem Abspielgerät verbinden. Notebooks haben meist ein eingebautes Mikrofon.

PROGRAMMIERKONZEPTE: *Blockierende und nichtblockierende Funktion*

SOUNDRECORDER

Vor der Aufzeichnung rufst du **openMonoRecorder()** auf, um das Aufnahmesystem vorzubereiten, wobei du die Abtastrate als Parameter angeben musst. Die Aufzeichnung startest du mit **capture()**. Diese Funktion ist nicht blockierend und kehrt sofort zurück. Mit **stopCapture()** musst du später die Aufnahme beenden. Die aufgenommenen Soundsamples werden in eine Liste kopiert, die du mit **getCapturedSound()** abholst. Du führst eine Aufnahme von 5 Sekunden Dauer durch und spielst dann den Sound ab.

```
from soundsystem import *

openMonoRecorder(22050)
print("Recording...");
capture()
delay(5000)
stopCapture()
print("Stopped");
sound = getCapturedSound()

openMonoPlayer(sound, 22050)
play()
```

MEMO

Ein Befehl wie **capture()**, der eine Aktion auslöst und sofort zurückkehrt, nennt man auch eine **nicht-blockierende Funktion**. Damit hast du die Möglichkeit, im weiteren Programmverlauf die im Hintergrund weiter laufende Aktion zu steuern, beispielsweise abzubrechen.

AUFGENOMMENEN SOUND DARSTELLEN

Natürlich interessiert uns oft auch die grafische Darstellung des aufgenommenen Sounds. Du weisst ja bereits, wie man dies mit dem GPanel macht.

Die nebenstehende Grafik stellt die Aufnahme der Wörter
"one two three for five six seven eight nine ten" dar.

```
from soundsystem import *

openMonoRecorder(22050)
print("Recording...");
capture()
delay(5000)
stopCapture()
print("Stopped");
sound = getCapturedSound()

from gpanel import *
makeGPanel(0, len(sound), -33000, 33000)
for i in range(len(sound)):
    draw(i, sound[i])
```

MEMO

Die Anzahl Abtastwerte erhältst du aus der Länge der Soundliste. Für die grafische Darstellung verwendest du am einfachsten eine for-Struktur. Spiele etwas mit verschiedenen aufgenommenen Sounds und überlege dir, ob du die Soundkurve verstehst.

WAV-DATEIEN SPEICHERN

Du kannst den aufgenommenen Sound mit **writeWavFile()** auch als WAV-Datei abspeichern.

```
from soundsystem import *

openMonoRecorder(22050)
print("Recording...");
capture()
delay(5000)
stopCapture()
print("Stopped");
sound = getCapturedSound()

writeWavFile(sound, "mysound.wav")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Nach dem Abspeichern kannst du die Sounddatei entweder mit Python oder mit irgend einem auf dem Computer installierten Soundplayer anhören.

AUFGABEN

1. Zeichne einzelne Wörter auf.

2. Setze diese Wörter zu einem Satz zusammen.

3*. Lass den Computer eine als Text gegebene Telefonnummer in Einzelziffern sprechen.

4.4 SPRACHSYNTHESE

EINFÜHRUNG

Bei der Sprachsynthese wird die menschliche Stimme durch den Computer erzeugt. Ein Text-To-Speech-System (TTS) wandelt dabei geschriebenen Text in eine Sprachausgabe um. Die maschinelle Erzeugung der menschlichen Sprache ist kompliziert, aber es wurden in den letzten Jahren grosse Fortschritte erzielt. Gegenüber der Wiedergabe von vorgefertigten Sprachaufnahmen hat eine TTS den Vorteil, sehr flexibel beliebige Texte zu sprechen. Die Sprachsynthese ist Teil der Computerlinguistik. Bei der Entwicklung eines TTS ist darum eine enge Zusammenarbeit zwischen Sprachwissenschaftlern und Informatiker notwendig.

Die in TigerJython eingesetzte Sprachsynthese-Software heisst *MaryTTS* und wurde in der Fachrichtung "Allgemeine Linguistik" der Universität des Saarlandes in Deutschland entwickelt.

Das System verwendet grosse Bibliotheksdateien, die du separat von **hier** herunterlädst und auspackst. Im Verzeichnis, in dem sich tigerjython2.jar befindet, erstellst du dann das Unterverzeichnis Lib (falls es noch nicht vorhanden ist) und kopierst die ausgepackten Dateien hinein.

PROGRAMMIERKONZEPTE: *Sprachsynthese, Künstliche Sprache, Text-To-Speech-System*

EINEN DEUTSCHEN ODER ENGLISCHEN TEXT SPRECHEN

MaryTTS stellt dir in der vorliegenden Version drei verschiedene Stimmen zur Verfügung: Eine deutschsprechende Frauenstimme, eine deutschsprechende Männerstimme und eine englischsprechende Männerstimme. Du wählst die Stimme mit der Funktion **selectVoice()** aus. Den zu sprechenden Text übergibst du danach der Funktion **generateVoice()**, welche eine Liste mit den erzeugten Soundsamples zurückgibt, die du mit einem SoundPlayer abspielen kannst.

```
from soundsystem import *

initTTS()
selectVoice("german-man")
#selectVoice("german-woman")
#selectVoice("english-man")

text = "Danke dass du mir eine Sprache gibst. Viel Spass beim Programmieren"
#text = "Thank you to give me a voice. Enjoy programming"
voice = generateVoice(text)
openSoundPlayer(voice)
play()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die auskommentieren Zeilen kannst du verändern, um den Text von den verschiedenen Stimmen sprechen zu lassen. Du musst immer zuerst **initTTS()** aufrufen, um die Sprachsynthese-Software bereit zu stellen.

Du könntest der Funktion *initTTS()* als Parameter noch einen Pfad auf das Verzeichnis mit den *MaryTTS*-Dateien übergeben, standardmässig handelt es sich um das Unterverzeichnis *Lib*.

DAS HEUTIGE DATUM UND DIE AKTUELLE ZEIT ANKÜNDIGEN

Die Anwendungen der Sprachsynthese sind vielfältig. Menschen mit Sehbehinderungen können sich Texte vorlesen lassen, Navigationssysteme und Bahnhof- oder Zugsdurchsagen verwenden oft ebenfalls synthetisch erzeugte Stimmen. Viele interaktive Computergames setzen ebenfalls künstlich erzeugte Stimmen ein.

Dein Programm ermittelt aus dem Computersystem die aktuelle Zeit und liest sie mit einer deutsch- oder englischsprechenden Stimme vor.

```
from soundsystem import *
import datetime

language = "german"
#language = "english"

initTTS()
if language == "german":
    selectVoice("german-woman")
    month = ["Januar", "Februar", "März", "April", "Mai",
        "Juni", "Juli", "August", "September", "Oktober",
        "November", "Dezember"]
if language == "english":
    selectVoice("english-man")
    month = ["January", "February", "March", "April", "May",
        "June", "July", "August", "September", "October",
        "November", "December"]

now = datetime.datetime.now()

if language == "german":
    text = "Heute ist der " + str(now.day) + ". " \
        + month[now.month - 1] + " " + str(now.year) + ".\n" \
        + "Die genaue Zeit ist " + str(now.hour) + " Uhr " + ".\n" \
        + str(now.minute)
if language == "english":
    text = "Today we have  " + month[now.month - 1] + " "  \
        + str(now.day) + ", "+ str(now.year) + ".\n" \
        + "The time is " + str(now.hour) + " hours " + ".\n" \
        + str(now.minute) + " minutes."

print text
voice = generateVoice(text)
openSoundPlayer(voice)
play()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

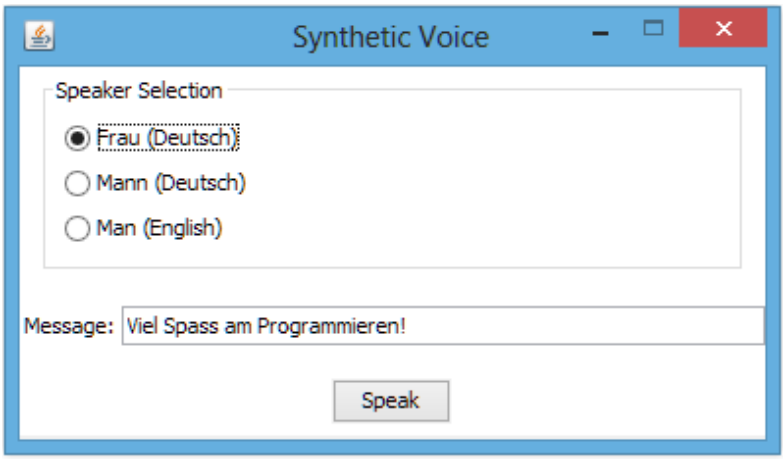
Durch Auswahl der auskommentierten Zeile kannst du zwischen dem deutschen oder englischen Sprecher unterscheiden. Die Klasse **datetime.datetime.now()** liefert dir mit ihren Attributen *year, month, day, hour, minute, second, microsecond* die Information über das aktuelle Datum und die aktuelle Zeit. Wie du siehst, kann man bei der Definition von langen Strings den Rückwärtsbruchstrich (Backslash) als Zeilenverlängerung verwenden.

EINE GRAFISCHE BENUTZEROBERFLÄCHE VERWENDEN

Wie du im Kapitel 3.13 bereits gelernt hast, stellt dir TigerJython Werkzeuge zur Verfügung, mit denen du einfache Benutzeroberflächen (GUIs) mit wenig Aufwand erstellen kannst. Dabei

werden die klassischen Bedienungselemente wie Textfelder, Schaltflächen (Buttons), Markierungsfelder (Checkboxen), Optionsfelder (Radiobuttons) und Schieberegler (Sliders) als Objekte modelliert, die sich in Bereichen (Panes) eines Bildschirmfensters befinden, das während der Programmausführung ständig sichtbar bleibt. (Du kannst dich in der APLU-Dokumentation genauer orientieren.)

In deinem Programm wird der Sprecher mit Radiobuttons ausgewählt und beim Klick auf den Bestätigungsbutton der Text im Textfeld in eine Sprachausgabe umwandelt.




```
from soundsystem import *
from entrydialog import *

speaker1 = RadioEntry("Frau (Deutsch)")
speaker1.setValue(True)
speaker2 = RadioEntry("Mann (Deutsch)")
speaker3 = RadioEntry("Man (English)")
pane1 = EntryPane("Speaker Selection", speaker1, speaker2, speaker3)
textEntry = StringEntry("Message:", "Viel Spass am Programmieren!")
pane2 = EntryPane(textEntry)
okButton = ButtonEntry("Speak")
pane3 = EntryPane(okButton)
dlg = EntryDialog(pane1, pane2, pane3)
dlg.setTitle("Synthetic Voice")

initTTS()

while not dlg.isDisposed():
    if okButton.isTouched():
        if speaker1.getValue():
            selectVoice("german-woman")
            text = textEntry.getValue()
        elif speaker2.getValue():
            selectVoice("german-man")
            text = textEntry.getValue()
        elif speaker3.getValue():
            selectVoice("english-man")
            text = textEntry.getValue()
        if text != "":
            voice = generateVoice(text)
            openSoundPlayer(voice)
            play()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die while-Schleife wird solange durchlaufen, bis das Dialogfenster mit dem Close-Button der Titelzeile geschlossen wird. Bei jedem Durchlauf prüfst du mit **isTouched()**, ob seit dem letzten Aufruf der Button gedrückt wurde. Ist dies der Fall, so holst du mit **getValue()** die

aktuellen Werte der GUI-Elemente und machst daraus wie in den vorhergehenden Programmen eine Sprachausgabe.

Es ist etwas gefährlich, eine solche "enge" *while*-Schleife zu durchlaufen, da man damit unnötig Rechenzeit verschwendet. Allerdings wird beim Aufruf von *isTouched()* das Programm automatisch kurz (1ms) angehalten, damit der Durchlauf etwas gebremst wird.

Um diese Schwierigkeit zu vermeiden, kannst du das Programm auf Eventsteuerung umstellen, indem du das Hauptprogramm mit **putSleep()** anhältst. Während des Schlafens wird keinerlei Rechenzeit benötigt. Um das Programm wieder aufzuwecken, verwendest du einen Callback **onTouched**, den du im Konstruktor von EntryDialog registrierst. Darin rufst du die Funktion **wakeUp()** auf wodurch das in *putSleep()* blockierte Programm weiter läuft.

Da beim Klicken des Close-Buttons automatisch *wakeUp()* aufgerufen wird, kannst du mit *isDisposed()* und break die Endlosschleife und damit das Programm beenden.

```
from soundsystem import *
from entrydialog import *

def onTouched(item):
    wakeUp()

speaker1 = RadioEntry("Frau (Deutsch)")
speaker1.setValue(True)
speaker2 = RadioEntry("Mann (Deutsch)")
speaker3 = RadioEntry("Man (English)")
pane1 = EntryPane("Speaker Selection", speaker1, speaker2, speaker3)
textEntry = StringEntry("Message:", "Viel Spass am Programmieren!")
textEntry2 = StringEntry("Message:", "Enjoy programming!")
pane2 = EntryPane(textEntry)
okButton = ButtonEntry("Speak")
pane3 = EntryPane(okButton)
dlg = EntryDialog(pane1, pane2, pane3, touched = onTouched)
dlg.setTitle("Synthetic Voice")

initTTS()

while True:
    putSleep()
    if dlg.isDisposed():
        break
    if speaker1.getValue():
        selectVoice("german-woman")
        text = textEntry.getValue()
    elif speaker2.getValue():
        selectVoice("german-man")
        text = textEntry.getValue()
    elif speaker3.getValue():
        selectVoice("english-man")
        text = textEntry2.getValue()
    if text != "":
        voice = generateVoice(text)
        openSoundPlayer(voice)
        play()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

AUFGABEN

1. Beschaffe dir oder schreibe selbst ein kurzes Gedicht als Textdatei, z.B.

   Hat der alte Hexenmeister
   sich doch einmal wegbegeben!

Und nun sollen seine Geister
auch nach meinem Willen leben.
Seine Wort und Werke
merkt ich und den Brauch,
und mit Geistesstärke
tu ich Wunder auch.

Walle! walle
manche Strecke,
dass, zum Zwecke,
Wasser fließe
und mit reichem, vollem Schwalle
zu dem Bade sich ergieße.

Johann Wolfgang Goethe (**Download)**

Mit der Zeile *text = open("zauber.txt", "r").read()* kannst du den Text aus der Textdatei *zauber.txt*, die sich im gleichen Verzeichnis wie dein programm befindet, als String einlesen. Lass den Text durch die Frauenstimme vorlesen.

2. Definiere entweder iterativ oder rekursiv die Funktion *fac(n)*, welche die Fakultät n! = 1 * 2 * ... *n zurückgibt.
   Dein Programm soll mit *readInt()* nach einer Zahl zwischen 0 und 10 fragen und den Fragetext auch sprechen. Es berechnet dann die Fakultät n! der eingegebenen Zahl und gibt das Resultat als gesprochenen Text aus.

4.5 AKUSTIK-EXPERIMENTE

EINFÜHRUNG

Du kannst den Computer auch an Stelle eines Experimentalsystems verwenden, beispielsweise um mit dem Soundsystem das menschliche Hören zu untersuchen. Dies ist nicht nur billiger, sondern gibt dir auch eine ungeheure Flexibilität, insbesondere wenn du die Experimente mit einem selbstgeschriebenen Programm durchführst.

PROGRAMMIERKONZEPTE: *Kammerton, Schwebung, Tonleiter*

STIMMEN EINES MUSIKINSTRUMENTS, SCHWEBUNG

Das Gehör kann zwei Töne mit fast gleichen Frequenzen, die allein gespielt werden, nicht unterscheiden. Werden diese aber gleichzeitig gespielt, so ergibt sich ein An- und Abschwellen der Lautstärke, das sehr gut hörbar ist. Um dies selbst erleben, spielst du mit deinem Programm zuerst während 5 Sekunden den Standard-Kammerton a (440 Hz) und nachher während derselben Zeit einen um nur 1 Hertz höheren Ton. Es ist kein Unterschied feststellbar. Beim Abspielen beider Töne hörst du aber ein deutliches Schwebungsphänomen.

```
import time

playTone(440, 5000)
time.sleep(2)
playTone(441, 5000)
time.sleep(2)
playTone(440, 20000, block = False)
playTone(441, 20000)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die globale Funktion *playTone()* hat verschiedene Parameter-Variationen, die du in der TigerJython *Hilfe* unter *APLU Dokumentation* (Koordinatengrafik) nachlesen kannst. Hier verwendest du den benannten Parameter *block*, mit dem du wählen kannst, ob die Funktion blockiert, bis der Ton zu Ende gespielt ist, oder ob sie nach Beginn des Abspielens sofort zurückkehrt. Um mehrere Töne gleichzeitig zu spielen, musst du die nicht-blockierende Variante verwenden.

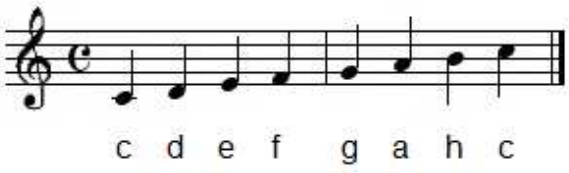
Um Instrumente in einem Orchester, aber auch die Töne eines einzelnen Instruments (Saiteninstrument, Klavier, usw.) zu stimmen, spielt man zwei Töne gleichzeitig und achtet auf Schwebungen.

TONLEITERN

Die wohltemperierte Tonleiter geht vom Standard-Kammerton mit der Frequenz 440 Hz aus und teilt die Oktave (Frequenzverhältnis 2) in 12 Halbtöne mit dem gleichem Frequenzverhältnis r. Demnach gilt

$$r^{12} = 2 \quad \text{oder} \quad r = \sqrt[12]{2} \approx 1.0594630943$$

Du kannst damit leicht die C-Dur-Tonleiter spielen, die gemäss der Notenschrift



aus Ganz- und Halbtönen besteht. Der Kammerton entspricht der Note a.

In der reinen oder natürlichen Tonleiter werden die Tonfrequenzen durch Multiplikation mit einfachen Brüchen aus dem Grundton gebildet. Für die 8 Töne einer Oktave lauten sie:

$$1, \frac{9}{8}, \frac{5}{4}, \frac{4}{3}, \frac{3}{2}, \frac{5}{3}, \frac{15}{8}, 2$$

oder als Zahlenreihe 24, 27, 30, 32, 36, 40, 45, 48. Um sie abzuspielen, kannst du die Frequenzen in einer Liste speichern und diese *playTone()* übergeben. Nachdem du beide Tonleitern einzeln abgespielt hast, hörst du dir zwei verschieden gestimmte Instrumente an, welche die Tonleiter zusammen spielen. Wie du feststellen wirst, tönt es schrecklich.

```
r = 2**(1/12)
a = 440

scale_temp = [a / r**9, a / r**7, a / r**5, a / r**4, a / r**2,
              a, a * r**2, a * r**3]
scale_pure = [3/5 * a, 3/5 * a * 9/8, 3/5 * a * 5/4,  3/5 * a * 4/3,  3/5 * a * 3/2,
              a, 3/5 * a * 15/8, 3/5 * a * 2]

playTone(scale_temp, 1000)
playTone(scale_pure, 1000)

playTone(scale_temp, 1000, block = False)
playTone(scale_pure, 1000)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Bei der wohltemperierten Tonleiter haben benachbarte Halbtöne immer dasselbe Frequenzverhältnis (also nicht etwa gleich Frequenzdifferenzen!). Der Vorteil der wohltemperierten gegenüber der reinen Stimmung ist, dass die Frequenzverhältnisse für alle Tonarten (C-Dur, D-Dur, usw.) immer gleich gross sind. [mehr...]

MELODIEN SPIELEN

Mit *playTone()* kannst du auch zum Spass eine einfache Melodie spielen. Für sich folgende Töne mit gleicher Länge verwendest du ein Tupel mit Tonhöhe und Geschwindigkeitsangabe

und packst diese Tonreihen in eine Liste. Zuletzt ist es noch möglich, ein Musikinstrument zu wählen. Beispielsweise erkennst du in deinem Programm sicher eine Kindermelodie. Erkennst du sie?

```
v = 250
playTone([("cdef", v), ("gg", 2*v), ("aaaa", v//2), ("g", 2*v),
          ("aaaa", v//2), ("g", 2*v), ("ffff", v), ("ee", 2*v),
          ("dddd", v), ("c", 2*v)], instrument="harp")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Es ist zwar erstaunlich, wie einfach du mit *playTone()* eine Melodie spielen kannst. Allerdings tönt diese verglichen mit einem Musikinstrument sehr synthetisch.

AUFGABEN

1. Du kannst ein Lied als Liste der Tonfrequenzen notieren und diese mit einer for-Schleife abspielen:

```
melody = [262, 444, 349, 349, 392, 330, 262, 466, 440, 392, 392, 349]
v = 200
for f in melody:
    playTone(f, v)
```

   a. Kennst du das Lied? Spiel das Lied etwas langsamer ab.

   b. Spiel das Lied eine Oktave höher.

   c. Für deine Singklasse ist die erste Version zu tief und die zweit zu hoch. Transponiere die Melodie, so dass sie anstelle von c' mit g' beginnt.

2. Spiel den Akkord c'', e'', g'' (Terz, Quint) während zwanzig Sekunden mit der wohltemperierten Stimmung ab. (Du kannst dazu *playTone()* mit Ton-Buchstabenangabe verwenden.) Spiel nun denselben Akkord mit der reinen Stimmung. Was stellst du fest?

Dokumentation Sound

Sound

Befehl	Aktion
playTone(freq)	spielt Ton mit gegebener Frequenz (Hz) 1000 ms lange, blockierend
playTone(freq, blocking = False)	das selbe, aber nicht blockierend (um mehrere Töne gleichzeitig abzuspielen)
playTone(freq, duration)	spielt Ton mit gegebener Frequenz und Dauer (ms)
playTone([f1, f2, f3 ...])	spielt hintereinander mehrere Töne mit den geg. Frequenzen und Dauer 1000ms
playTone([(f1, d1),(f2, d2), (f3, d3)...])	spielt hintereinander mehrere Töne mit gegebenen Frequenzen und Dauer
playTone([("c", 700),("e", 1500)], instrument="piano")	spielt Noten mit gegebener Dauer und Instrument (piano, quitar, harp, trumpet, organ, violin... gemäss MIDI Spezifikation).
playTone([("c", 1000),("e", 1500)], instrument="piano", volumen = "10")	wie oben mit gegebener Lautstärke

Module import: from soundsystem import *

Wiedergabe:

getWavMono(filename)	liefert Liste der Abtastwerte der angegeben monauralen Sounddatei. Angabe "wav/xxx.wav" lädt auch vom _wav-Verzeichnis in tigerjython2.jar
getWavStereo(filename)	liefert Liste der Abtastwerte der angegeben binauralen Sounddatei. Angabe "wav/xxx.wav" lädt auch vom _wav-Verzeichnis in tigerjython2.jar
getWavInfo(file)	liefert String mit Information über Abtastrate, usw.
openSoundPlayer(filename)	öffnet einen Soundplayer mit der angegebenen Sounddatei. Nachher kann sie mit den folgenden Player-Funktionen abgespielt werden
openMonoPlayer(filename)	öffnet einen monauralen Soundplayer mit der angegeben Datei. Es kann sich auch um eine binaurale Datei handeln (Mittelwert aus beiden Kanälen)
openStereoPlayer(filename)	öffnet einen binauralen Soundplayer mit der angegeben Datei. Es kann sich auch um eine monaurale Datei handeln (beide Kanäle identisch)
openSoundPlayerMP3(filename)	wie openSoundPlayer(), aber für MP3-Datei
openMonoPlayerMP3(filename)	wie openMonotPlayer(), aber für MP3-Datei
openStereoPlayerMP3(filename)	wie openStereoPlayer(), aber für MP3-Datei
play()	spielt Sound von der aktuellen Position an und kehrt sofort zurück
blockingPlay()	spielt Sound von der aktuellen Position an und wartet, bis er fertig gespielt ist
advanceFrames(n)	schiebt die aktuelle Position um die Anzahl Abtastwerte nach vorne
advanceTime(t)	schiebt die aktuelle Position um die angegebene Zeit nach vorne
getCurrentPos()	gibt die aktuelle Position zurück
getCurrentTime()	gibt die aktuelle Spielzeit zurück
rewindFrames(n)	schiebt die aktuelle Position um die angegebene Anzahl Abtastwerte zurück
rewindTime(t)	schiebt die aktuelle Position um die angegebene Zeit zurück
stop()	hält das Abspielen an und setzt die aktuelle Abspielposition an den Anfang
setVolume(v)	setzt die Lautstärke (v = 0..100)
isPlaying()	gibt True zurück, falls noch nicht zu Ende gespielt ist
mute(bool)	schaltet mit True auf stumm, mit False wieder auf hörbar
playLoop()	wiederholt das Abspielen endlos

replay()	wiederholt das Abspielen einmal
delay(time)	hält Programm um time (Millisekunden) an

Aufnahme und speichern:

openMonoRecorder()	öffnet einen monauralen Soundrecorder
openStereoRecorder()	öffnet einen binauralen Soundrecorder
capture()	beginnt mit der Aufnahme
stopCapture()	beendet die Aufnahme
getCapturedBytes()	liefert die aufgenommenen Samples byteweise zurück
getCapturedSound()	liefert die aufgenommenen Samples als integer Listenwerte zurück (binaural: Kanäle abwechselnd)
writeWavFile(samples, filename)	schreibt die Samples in eine WAV-Datei

Fast Fourier Transform (FFT):

fft(samples, n)	transformiert die ersten n Werte der übergebenen Liste mit Abtastwerten (floats). Rückgabe einer Liste mit n // 2 äquidistanten Spektralwerten (floats). Bei einer Abtastrate von fs reichen diese von 0 bis fs/2 im Abstand fs / n (Auflösung)
sine(A, f, t)	Sinusschwingung mit Amplitude A und Frequenz f (Phase 0) für jeden Float-Wert t
square(A, f, t)	Rechteckschwingung mit Amplitude A und Frequenz f (Phase 0) für jeden Float-Wert t
sawtooth(A, f, t)	Sägezahnschwingung mit Amplitude A und Frequenz f (Phase 0) für jeden Float-Wert t
triangle(A, f, t)	Dreieckschwingung mit Amplitude A und Frequenz f (Phase 0) für jeden Float-Wert t
chirp(A, f, t)	Sinusschwingung mit Amplitude A und zeitlinear ansteigender Frequenz (Startwert f) für jeden Float-Wert t

ROBOTIK

Lernziele

- Du kannst beschreiben, was man unter einem Roboter versteht und kennst einige Einsatzmöglichkeiten von Robotern.
- Du kennst den Unterschied zwischen einem autonomen und fremdgesteuerten Roboter und weisst, warum man Roboter simuliert.
- Du kannst den EV3- oder NXT-Roboter mit einem Python-Programm steuern
- Du kannst an einigen Beispielen erklären, was ein lernender Roboter ist. Du verstehst den Unterschied zwischen dem Teach- und Execute-Mode.
- Du kennst das Prinzip des Regelungskreises und kannst einige Beispiele von Regelungen aufzählen.
- Du kannst in einem Programm Sensorwerte mit Pollen und Events erfassen.

"Will robots inherit the earth? Yes, but they will be our children."

but:

"No computer has ever been designed that is ever aware of what it's doing; but most of the time, we aren't either."

Marvin Minsky, AI Researcher at MIT

5.1 REAL- UND SIMULATIONSMODUS

EINFÜHRUNG

Unter einem Roboter versteht man meist eine computergesteuerte Maschine, die eine Tätigkeit ausführen kann, welche früher durch Menschen verrichtet wurde. Wenn die Maschine sogar mit Hilfe von Kameras und Sensoren die Umgebung erfassen und entsprechend mit Aktoren (Motoren, Ventile, Sprachausgaben, usw.) reagieren kann, spricht man von einem **intelligenten System** oder bei menschenähnlichem Verhalten von einem **Androiden**.

Ein typisches Beispiel ist der Film-Roboter WALL·E, der ein eigenes Bewusstsein besitzt, das so weit geht, dass er Ersatzteile für sich selbst sucht und besondere Gegenstände, die sein Interesse erwecken, in einer Sammlung aufbewahrt. Er ist auch in der Lage, den Rubik-Würfel zusammen zu setzen, was du sicher als ein Zeichen von Intelligenz ansiehst.

Die **Künstliche Intelligenz (KI)** befasst sich mit der interessanten Frage, inwiefern ein Computersystem überhaupt als "*intelligent*" bezeichnet werden kann. Zur Beantwortung dieser Frage muss zuerst definiert werden, was man unter einer intelligenten Maschine versteht. Ein möglicher Lösungsansatz zur Definition von Intelligenz ist der **Turing-Test**.

Hier befasst du dich mit einfacheren Fragen und du lernst mit einem einfachen Roboter umzugehen, der mit Berührungs-, Licht-, Sound-, Infrarot- und Ultraschallsensoren ausgerüstet ist und mit zwei Rädern, die mit Elektromotoren angetrieben sind, vorwärts und rückwärts fahren und sich drehen kann.

Die Steuerung von Motoren und Sensoren übernimmt ein eingebauter Computer, darum nennt man einen Roboter auch ein **Eingebettetes System** (embedded system). Wenn es sich oft um einen einfachen Rechnerchip handelt, spricht man von einem **Mikroprozessor** (oder **Mikrocontroller**). Eingebettete Systeme spielen heutzutage eine ausserordentlich wichtige Rolle, denn du findest sie in vielen Geräten des täglichen Lebens, etwas in allen Smartphones. Aber auch Kaffeemaschinen, Waschmaschinen, TV-Geräte, Photoapparate, usw. sind meist eingebettete Systeme. In einem modernen Auto befinden sich bis zu hundert Mikrocontroller als eingebettete Systeme von der Motorsteuerung bis zum Antiblockierungssystem. Du solltest dir deshalb bewusst sein, dass du im Umgang mit Robotern auch viele allgemeingültige Prinzipien für eingebettete Systeme kennenlernst.

Führt der eingebaute Prozessor ein eigenständiges Programm zur Steuerung des Roboters aus, so spricht man von einem **autonomen** Roboter. Der eingebaute Prozessor kann aber auch lediglich über einen Datenübertragungskanal die von den Sensoren erfassten Daten an einen äusseren Computer senden und von diesem Steuerungsbefehle erhalten. In diesem Fall handelt es sich um einen **fremdgesteuerten** Roboter. Schliesslich kann ein Roboter auch **simuliert** werden. Dabei werden Sensoren, Motoren usw. meist als Softwareobjekte abgebildet. Dem realen Zusammenbau der Komponenten entspricht dann softwaremässig eine Klassenkonstruktion. In der Praxis werden Roboter meist zuerst auf dem Computer simuliert, da damit das Verhalten mit kleinem Aufwand und ohne Gefahren für die Umgebung studiert werden kann.

Mit den weltweit bekannten Robotik-Baukästen von **Lego Mindstorms** kannst du wichtige Aspekte der Robotik spielerisch erlernen. Das System besteht aus einem mikroprozessorgesteuerten Baustein (**Brick**) und einer Vielzahl von Bauteilen, mit denen unterschiedliche Robotermodelle konstruiert werden können. Der Brick hat mehrere Entwicklungsstadien durchlaufen: Früher hiess er RCX, dann NXT und neuerdings EV3.

Der EV3-Brick mit den Motoren und Sensoren ist ein eingebettetes System, das durch einen modernen ARM-Prozessor gesteuert wird. Öffnet man ihn, so werden seine elektronischen Bauteile sichtbar.

PC-Port

4 Motor-Ports

USB-Port

Lautsprecher

SD-Karte

32 bit ARM9 Prozessor
Texas Instrument AM1808
mit 64 MB RAM und
16 MB Flash Speicher

4 Sensor-Ports

Beim Einschalten des Bricks startet auf dem Mikroprozessor eine Firmware oder beim EV3 das Linux-Betriebssystem, und auf dem Display erscheint ein einfaches Menü. Damit kannst du im autonomen Modus bereits auf dem Brick gespeicherte Programme ausführen. Auf dem EV3 musst du für den fremdgesteuerten Modus ein Hilfsprogramm (BrickGate) starten, das Befehle interpretiert, die über eine Bluetooth-Verbindung von aussen empfangen werden, beispielsweise den Befehl, einen Motor in einer bestimmten Drehrichtung einzuschalten oder den Messwert eines Sensors zurück zu melden. Beim NXT ist dieses Programm Teil der Firmware.

Wie bei allen eingebetteten Systemen benötigst du für die Robotik einen externen Computer, auf dem du die Roboterprogramme entwickelst. Im autonomen Modus wird das Programm auf den Brick heruntergeladen, im fremdgesteuerten Modus läuft es auf dem PC.

Autonomer Modus

Fremdgesteuerter Modus

PROGRAMMIERKONZEPTE: *Roboter, Android, Künstliche Intelligenz, Eingebettetes System, Mikroprozessor, Mikrocontroller, Blockierende/Nichtblockierende Methode*

VORARBEITEN

Mit TigerJython kannst du den Roboter simulieren (**Simulationsmodus**) und autonom oder fremdgesteuert verwenden (**Realmodus**). Du verwendest dabei verschiedene Klassenbibliotheken, die aber das gleiche Programmierinterface (Application Programming Interface, API) haben, so dass die Programme für alle Modi praktisch identisch sind. Es müssen lediglich der import und eventuell gewisse im Programm verwendete Timings angepasst werden.

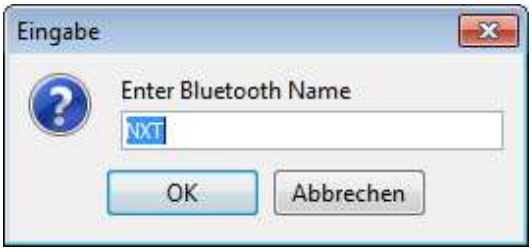
Simulationsmodus:
Steht dir kein NXT oder EV3 zur Verfügung, kannst du das Thema trotzdem im Simulationsmodus durcharbeiten. Die für die Simulation benötigten Bilder sind in der Distribution von TigerJython (im Unterverzeichnis *_sprites*) bereits enthalten.

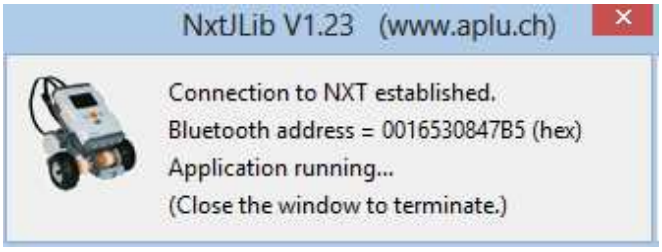
Realmodus:
Die meisten Beispiele verwenden das Basismodell des Lego NXT- bzw. EV3-Roboters, das sich mit zwei Motoren bewegt und mit diversen Sensoren ausgestattet werden kann. Solange du die grundsätzliche Funktionalität nicht veränderst, kannst du auch ein eigenes, davon abweichendes Modell verwenden. Da die Kommunikation mit dem Roboter über Bluetooth erfolgt, muss dein PC Bluetooth-fähig sein und du musst den Brick mit dem PC "paaren".

Vorgehen mit dem Lego-NXT:
Auf dem Lego-NXT kann entweder die originale Lego-Firmware oder die Java-Firmware leJOS installiert sein. Normalerweise ist der Bluetooth-Name des Bricks "NXT".

Für das Paaren gehst du so vor, wie du es mit anderen externen Bluetooth-Geräten wie Smartphones, Bluetooth-Handsets, Druckern, usw. gewohnt bist.

Den NXT kannst du mit Python nur im fremdgesteuerten Modus verwenden. Du schreibst dazu in TigerJython ein ganz normales Python-Programm unter Verwendung des Moduls ch.aplu.nxt und drückst zum Starten den grünen Run-Button. Zuerst wirst du nach dem Bluetooth-Namen gefragt und danach wird die Verbindung zum Brick aufgebaut. Während der Programmausführung bleibt ein Fenster mit Verbindungsinformationen offen. Schliesst du dieses Fenster, so wird die Verbindung zum NXT unterbrochen.





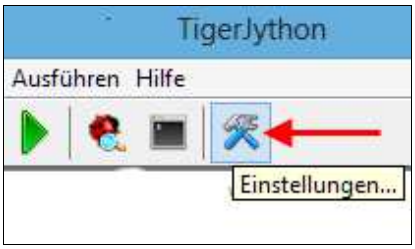
Befinden sich im Raum mehrere Lego-NXT, so müssen ihre Bluetooth-Namen verschieden sein, damit es keine Konflikte gibt. Ein Tool zum Ändern des Namens findest du unter www.legorobotik.ch. Du kannst auch anstelle des Bluetooth-Namens die Bluetooth-Adresse verwenden, die du mit verschiedenen Tools herausfinden kannst (beispielsweise wird sie bei jedem Verbindungaufbau im Connection-Dialog ausgeschrieben).

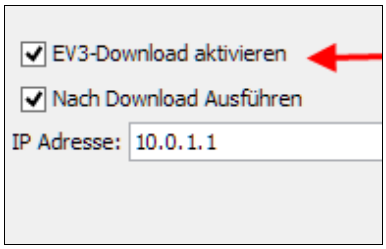
Für die Bluetooth-Kommunikation ist die Bluecove-Bibliothek notwendig. Du lädst die Dateien von **hier** herunter und packst sie im Unterverzeichnis *Lib* des Verzeichnisses, in dem sich *tigerjython2.jar* befindet, aus.

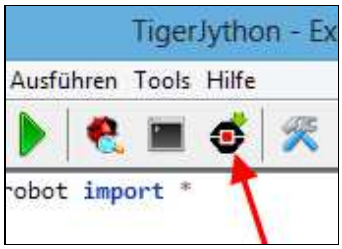
Vorgehen mit dem Lego-EV3:
Auf dem EV3 läuft ein Linux-Betriebssystem, das zusammen mit der leJOS-Software gestartet wird, die sich auf einer SD-Karte befindet. Eine genaue Anleitung, wie du die SD-Karte erstellen musst, findest du **hier**. Entfernst du die SD-Karte, so kannst du den EV3 wieder wie im Originalzustand verwenden. Ist der EV3 unter leJOS gestartet, so kommunizierst du über eine Bluetooth-PAN-Verbindung mit ihm. Dazu musst du den PC mit dem Brick paaren und als Netzwerk-Zugriffspunkt angeben. Eine Anleitung findest du **hier**.

Nach dem Booten des EV3 mit leJOS und der erfolgreichen Anbindung über Bluetooth-PAN musst du auf dem Brick den ***BrickGate*** Server starten, den du im Menu "Programme" findest.

Um den EV3 im autonomen Modus zu verwenden, solltest du in TigerJython in den Einstellungen im Register *Bibliotheken* die Markierungsfelder *EV3-Download aktivieren* und Nach *Download ausführen* ankreuzen. Du erkennst dann in der Werkzeugleiste ein zusätzliches EV3-Symbol.







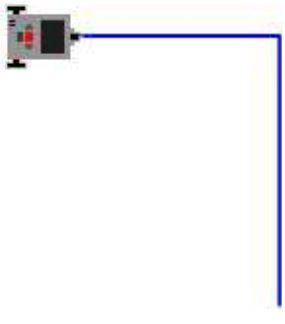
Für den fremdgesteuerten Modus klickst du wie üblich auf den grünen Run-Button. Wie mit dem NXT wirst du zuerst nach dem Bluetooth-Namen gefragt und es öffnet sich dann ein Fenster mit Verbindungsinformationen. Willst du das gleiche Programm autonom ausführen, so klickst du einfach auf den EV3-Button. Das Python-Skript wird dann auf den EV3 heruntergeladen und dort ausgeführt. Sein Name erscheint auch auf dem Display des EV3 und es kann jederzeit ohne Verbindung zum PC mit dem ENTER-Button wieder ausgeführt werden.

In den Programmen wird davon ausgegangen, dass du für den EV3 die neuen Motoren und Sensoren aus dem EV3-Sortiment verwendest. Als Lichtsensor dient der EV3 Colorsensor. Die alten NXT-Motoren und Sensoren werden aber auch für den EV3 immer noch unterstützt. Du musst einfach überall vor den Klassennamen "Nxt" setzen, also NxtMotor, NxtGear, NxtTouchSensor, usw.

VORWÄRTS- UND RÜCKWÄRTS-FAHREN, DREHEN

In deinem ersten Roboterprogramm soll sich der Roboter während bestimmten, im Programm fest codierten Zeiten bewegen. Die Roboterbibliothek ist objektorientiert aufgebaut und bildet die Wirklichkeit modellmässig ab. So wie du bei der Konstruktion des Roboters den Lego-Brick in die Hand nimmst, erstellst du softwaremässig mit **robot = LegoRobot()** eine Instanz der Klasse *LegoRobot()*. Als nächstes nimmst du zwei Motoren und setzt diese zu einem Fahrwerk zusammen, was du softwaremässig mit **gear = Gear()** ausdrücken wirst. Dann schliesst du die Fahrwerkmotoren an die Motor-Ports A und B an, was du softwaremässig mit **addPart()** formulierst.

Mit dem Befehl *gear.forward()* schaltest du beide Motoren gleichzeitig mit gleicher Rotationsgeschwindigkeit ein und der Roboter fährt geradeaus. **Dieser Bewegungszustand bleibt erhalten, bis du etwas anderes unternimmst.** Der Aufruf kehrt aber sofort zurück und dein Programm fährt bei der folgenden Anweisung weiter (man nennt diese eine **nicht-blockierende Methode**). Dein Programm muss also dafür sorgen, dass der Roboter nach einer gewissen Zeit etwas anderes macht. Dazu kannst du das Programm mit **Tools.delay()** warten lassen und dann mit einem anderen Befehl die Bewegung verändern oder stoppen.



Sobald du einen Bewegungsbefehl an den Roboter sendest, wird der bestehende Zustand beendet und der neue Zustand übernommen. Am Schluss des Programms solltest du immer die Methode **exit()** aufrufen. Dabei werden alle Motoren gestoppt und im Realmodus auch die Bluetooth-Verbindung unterbrochen, was nötig ist, damit ein nächster Programmstart erfolgreich ist. (Wird das Programm nicht korrekt endet, so kann es nötig sein, dass du den Brick aus- und wieder einschalten musst.)

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
gear.setSpeed(50)
gear.forward()
Tools.delay(2000)
gear.left()
Tools.delay(580)
gear.forward();
Tools.delay(2000)
robot.exit()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Ein Gear ist ein Fahrwerk mit zwei Motoren. Statt die Motoren einzeln zu steuern, kannst du damit Befehle verwenden, die beide Motoren gleichzeitig beeinflussen.

Die Klassenbibliotheken für den Simulations- und den Realmodus sind so konzipiert, dass die Programme praktisch identisch sind. Du kannst dein Programm zuerst im Simulationsmodus entwickeln und danach mit wenig Anpassungen mit dem realen Roboter ausführen.

Den EV3 kannst du autonom oder fremdgesteuert verwenden. In beiden Fällen muss auf dem EV3 das BrickGate-Programm gestartet sein, das die von Python gesendeten Befehle empfängt und entsprechend interpretiert.

Da bei der Ausführung im autonomen Modus keine Fehler angezeigt werden, solltest du das Programm immer zuerst im fremdgesteuerten Modus ausprobieren (grüner Knopf) und erst dann mit dem EV3-Button auf den Brick herunterladen und dort ausführen.

BEWEGEN MIT BLOCKIERENDEN METHODEN

Statt wie vorhin mit dem Befehl *forward()* den Roboter in Vorwärtsbewegung zu setzen und dann mit *delay(2000)* das Programm 2000 ms warten lassen, kannst du auch die blockierende Methode **forward(2000)** verwenden, die den Roboter ebenfalls in Vorwärtsbewegung versetzt, aber erst nach 2000 ms zurückkehrt. Auch für **left()** und *right()* gibt es blockierende Varianten.

Du kannst damit das vorhergehende Programm mit blockierenden Methoden etwas vereinfachen.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *
```

```
robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
gear.setSpeed(50)
gear.forward(2000)
gear.left(580)
gear.forward(2000)
robot.exit()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du musst zwischen blockierenden und nicht blockierenden Methoden unterscheiden. Nicht blockierende Befehle bewirken eine Zustandsänderung des Roboters und kehren sofort zurück. Blockierenden Bewegungsbefehlen übergibst du ein bestimmtes Zeitintervall, während dem das Programm blockert d.h. die nächste Anweisung wird erst ausgeführt, wenn das Zeitintervall abgelaufen ist.

Auf den ersten Blick scheint es einfacher zu sein, immer blockierende Methoden zu verwenden. Du handelst dir aber damit einen wesentlichen Nachteil ein. Da dein Programm nun blockiert ist, kannst du während dieser Zeit keine anderen Aktionen ausführen, also beispielsweise keine Sensorwerte lesen!

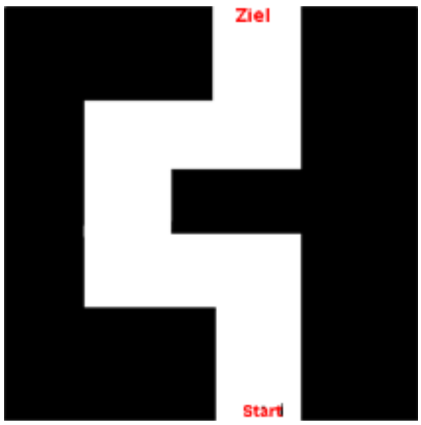
Hängt sich das Programm während der Programmausführung auf, so kannst du es im fremdgesteuerten Modus durch Schliessen des Verbindungsinfo-Fensters abbrechen. Im autonomen Modus drückst du im Notfall die beiden Tasten DOWN+ENTER.

AUFGABEN

1. Schreibe ein Programm mit blockierenden Methoden, um ein Quadrat abzufahren.

2. Schreibe ein Programm mit nicht-blockierenden Methoden, damit sich der Roboter auf der Halbkreiskurve bewegt.

3. Erstelle mit einigen Gegenständen einen Parcours und schreibe ein dazugehörendes Programm so, dass der Roboter vom Start zum Ziel fährt.



   Für die Simulation wird mit R*obotContext.useBackground()* das Hintergrundbild *bg.gif* angezeigt, das sich im Unterverzeichnis *sprites* befindet.
   Mit *RobotContext.setStartPosition()* kannst du den Roboter bei Programmstart an eine bestimmte Stelle setzen. (Bildschirmkoordinaten 0..500, Nullpunkt oben links).

```
RobotContext.setStartPosition(200, 455)
RobotContext.useBackground("sprites/bg.gif")
```

   Du kannst aber auch ein eigenes Bild (Bildgrösse 501x501) erstellen.

ZUSATZSTOFF

INFRAROT-FERNBEDIENUNG

Für den EV3-Roboter gibt es einen EV3 Infrarot-Sensor, der sich sehr vielseitig einsetzen lässt. Er ist im Lego Heim-Bausatz (inklusive der Fernsteuerbox) bereits enthalten, muss aber für den Lego Education-Bausatz zusätzlich beschafft werden. Den IRSensor kannst du auf drei Arten verwenden.

Klasse	**Messgrössen**
IRSeekSensor	Distanz und Richtung zur IR-Quelle der Fernsteuerung
IRRemoteSensor	Gedrückte Buttons der Fernsteuerung
IRDistanceSensor	Distanz zu einem reflektierenden Ziel (Target)

Für erste "Gehversuche" mit dem Roboter ist die Verwendung der Fernbedienung lustig und motivierend. Gegenüber einem vorgegebenen Fernsteuerungsprogramm kannst du mit einfacher Python-Programmierung die Aktionen, die beim Drücken der Fernsteuerung ausgelöst werden, selbst festlegen.

In deinem Programm entscheidest du dich für folgende Funktionen:

Fernsteuerungsbutton	**Aktion**
Links-Oben	auf Linksbogen vorwärts fahren
Rechts-Oben	auf Rechtsbogen vorwärts fahren
Links-Unten+Rechts-Oben	Geradeaus vorwärts fahren
Links-Unten	Stoppen
Rechts-Unten	Programm beenden

```
from ev3robot import *

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
irs = IRRemoteSensor(SensorPort.S1)
robot.addPart(irs)
isRunning = True

while not robot.isEscapeHit() and isRunning:
    command = irs.getCommand()
```

```
    if command == 1:
        gear.leftArc(0.2)
    if command == 3:
        gear.rightArc(0.2)
    if command == 5:
        gear.forward()
    if command == 2:
        gear.stop()
    if command == 4:
        isRunning = False
robot.exit()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Methoden *isEscapeHit(), isEnterHit(), isDownHit(), isUpHit(), isLeftHit(), isRightHit()* liefern *True*, falls du im autonomen Modus die entsprechenden Buttons auf dem EV3 klickst. Im fremdgesteuerten Modus betreffen sie aber die Tastaturtasten ESCAPE, ENTER, CURSOR-DOWN, CURSOR-UP, CURSOR-LEFT, CURSOR-RIGHT. Dabei muss aber das Verbindungsinfo-Fenster aktiv sein (mit Maus hineinklicken).

5.2 INTELLIGENTE ROBOTER

EINFÜHRUNG

Roboter, die den Weg in einer veränderlichen Umgebung finden, haben ein grosses Einsatzgebiet, beispielsweise als fliegende Objekte, in der Unterwassererforschung und bei der Untersuchung von Kanalrohrsystemen. Hier lernst du schrittweise, wie man grundsätzlich einen fahrenden Roboter bauen kann, der sich in einer veränderlichen Umgebung zurecht findet.

PROGRAMMIERKONZEPTE: *Fremdgesteuerter, autonomer, selbstlernender Roboter, Teach-Mode, Execution-Mode, Ereignisschleife*

DER ROBOTER KENNT DEN WEG

Im einfachsten Fall soll ein Roboter den Durchgang in einem sehr speziellen Kanal finden, der aus gleich langen, rechtwinklig angeordneten Kanalelementen besteht.

Die Information über die konstante Länge der Kanalelemente und ob es sich um Rechts- oder Linkskurven handelt, wird im Programm fest einprogrammiert ("verdrahtet").

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)

moveTime = 5000
turnTime = 580
robot = LegoRobot()
gear = Gear()
gear.setSpeed(50)
robot.addPart(gear)
gear.forward(moveTime)
gear.left(turnTime)
gear.forward(moveTime)
gear.right(turnTime)
gear.forward(moveTime)
gear.right(turnTime)
gear.forward(moveTime)
gear.left(turnTime)
gear.forward(moveTime)
robot.exit()
```

MEMO

Die Zeiten **moveTime** und **turnTime** musst du durch eigene Versuche herausfinden und entsprechend anpassen. Sie hängen natürlich mit der Geschwindigkeit des Roboters zusammen. In Wirklichkeit würde man wohl eher an Stelle von Zeiten die zu durchlaufene Strecke und den Drehwinkel angeben.

VOM MENSCH GESTEUERTER ROBOTER

Die konstante Länge der Kanalelemente kennt der Roboter selbst, aber für die Kurvenbewegung wird er durch einen Menschen gesteuert. Der Roboter ist aber nicht lernfähig, er kann sich den Pfad nicht merken, er bleibt "dumm".

Für die Steuerung des Roboters verwendest du im Simulations- und fremdgesteuerten Modus die Cursor-Links- und Cursor-Rechts-Tasten der Tastatur und im autonomen Modus die entsprechenden LEFT- und RIGHT-Buttons. Mit den Methoden *isLeftHit()* und *isRightHit()* kannst du abfragen, ob die Tasten bzw. die Buttons gedrückt und wieder losgelassen wurden. Um das Programm zu beenden, verwendest du die Escape-Taste bzw. den ESCAPE-Button.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)

moveTime = 5000
turnTime = 580

robot = LegoRobot()
gear = Gear()
gear.setSpeed(50)
robot.addPart(gear)
gear.forward(moveTime)

while not robot.isEscapeHit():
    if robot.isLeftHit():
        gear.left(turnTime)
        gear.forward(moveTime)
    if robot.isRightHit():
        gear.right(turnTime)
        gear.forward(moveTime)
robot.exit()
```

MEMO

Der autonome Modus macht hier weniger Sinn, da man ja den Roboter eigentlich fernsteueren möchte. Du kannst für den EV3 dazu statt der Tastatur auch die Infrarot-Fernsteuerung verwenden (siehe Anhang).

DER ROBOTER LERNT IM TEACH-MODE

Computergestützte Systeme, deren Verhalten nicht fest einprogrammiert ist, die also später ihr Verhalten an eine Umgebung anpassen können, nennt man **adaptive Systeme.** Diese sind daher in einem gewissen Sinn **lernfähig**. Industrierobotern werden in einem "**Teach Mode**" durch einen Spezialisten "angelernt", beispielsweise welche Armbewegungen durchzuführen sind. Der Operator verwendet dabei meist ein Eingabesystem ähnlich einer Fernsteuerung. Dabei wird der

Roboter nacheinander in die gewünschten Positionen gefahren und der jeweilige Zustand abgespeichert.

Im "**Execution Mode**" fährt dann der Roboter die gespeicherten Zustände selbständig (und mit hoher Geschwindigkeit) ab.

Wie vorher kennt dein Kanal-Roboter die konstante Länge der Kanalelemente, wird aber für die Kurvenbewegung durch einen Menschen gesteuert. Der Roboter ist aber nun lernfähig, er kann sich den Pfad einprägen und ihn nachher beliebig oft selbständig durchlaufen.

Es ist oft zweckmässig sich vorzustellen, dass sich ein Roboter in jedem Moment in einem bestimmten **Zustand** befindet. Die Zustände werden mit aussagekräftigen Wörtern bezeichnet und als String gespeichert. Du gehst von folgenden Zuständen aus: Der Roboter ist gestoppt, vorwärtsfahrend, link- oder rechtsdrehend und nennst sie: STOPPED, FORWARD, LEFT, RIGHT.

Statt die Tasten- bzw. Button ständig abzufragen, verwendest du hier eleganter das Event-Programmiermodell mit registrierten Callback-Funktionen, die unabhängig vom gerade laufenden Programm immer dann automatisch aufgerufen werden, wenn ein Ereignis eintritt.

Das Hauptprogramm ist in einer Endlos-Schleife damit beschäftigt, in jedem Zustand die entsprechenden Aktionen durchzuführen. Der Zustandswechsel erfolgt im Callback *onButtonHit()*.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)
RobotContext.showStatusBar(30)

def onButtonHit(buttonID):
    global state
    if buttonID == BrickButton.ID_LEFT:
        state = "LEFT"
    elif buttonID == BrickButton.ID_RIGHT:
        state = "RIGHT"
    elif buttonID == BrickButton.ID_ENTER:
        state = "RUN"

moveTime = 5000
turnTime = 580
memory = []
robot = LegoRobot(buttonHit = onButtonHit)
gear = Gear()
gear.setSpeed(50)
robot.addPart(gear)
state = "FORWARD"
```

```
while not robot.isEscapeHit():
    if state == "FORWARD":
        robot.drawString("Moving forward", 0, 3)
        gear.forward(moveTime)
        state = "STOPPED"
        robot.drawString("Teach me!", 0, 3)
    elif state == "LEFT":
        memory.append(0)
        robot.drawString("Saved: LEFT-TURN", 0, 3)
        gear.left(turnTime)
        state = "FORWARD"
    elif state == "RIGHT":
        memory.append(1)
        robot.drawString("Saved: RIGHT-TURN", 0, 3)
        gear.right(turnTime)
        state = "FORWARD"
    elif state == "RUN":
        robot.drawString("Executing memory", 0, 1)
        robot.drawString(str(memory), 0, 2)
        robot.reset()
        robot.drawString("Moving forward", 0, 3)
        gear.forward(moveTime)
        for k in memory:
            if k == 0:
                robot.drawString("Turning left", 0, 3)
                gear.left(turnTime)
            else:
                robot.drawString("Turning right", 0, 3)
                gear.right(turnTime)
            robot.drawString("Moving forward", 0, 3)
            gear.forward(moveTime)
        gear.stop()
        robot.drawString("All done", 0, 3)
        state = "STOPPED"
robot.exit()
```

MEMO

Die Bearbeitung der Zustände erfolgt in einer while-Schleife im Hauptteil des Programms, also nicht in den Callback-Funktionen. Eine solche Schleife nennt man in der Informatik üblicherweise eine **Ereignisschleife** (event loop). Im Callback wird lediglich der entsprechende Zustand ausgewählt (state switch). Mit dieser Programmiertechnik erhält man eine übersichtliche zeitliche Synchronisation zwischen den länger dauernden Aktionen und dem ereignisgesteuerten Callback-Aufruf, der zu jeder beliebigen Zeit auftreten kann. Das "Gedächtnis" besteht aus einer Liste, in der du die Zahlen 0 oder 1 abspeicherst, je nachdem ob der Weg nach links oder rechts abzweigt.

DER SELBSTLERNENDE ROBOTER

Bei bestimmten Anwendungen ist es nicht möglich, dass der Roboter durch einen Operator angelernt wird. Der Roboter kann sich beispielsweise in einem Gebiet aufhalten, dass sich ausserhalb unmittelbarer Kommunikation befindet (z.B. auf dem Mars).

Um den Weg zu finden, muss der Roboter nun mittels eingebauten Sensoren die Umgebung erfassen und entsprechend "handeln" können. Der Mensch erkennt die Umgebung meist mit seinen Augen. Für Roboter ist zwar die Bilderfassung mit einer Kamera einfach, die Analyse des Bildes aber äusserst kompliziert.

Um sich im Kanal zu orientieren, verwendet dein Roboter lediglich einen Berührungssensor (Touchsensor), der ein Ereignis auslöst, wenn er gedrückt wird. Der Kanal soll immer noch aus gleich langen Kanalelementen bestehen. Wenn der Roboter nach dem Durchfahren eines

Kanalelements einen Touchevent erhält, so weiss er, dass er sich an einer Weggabelung befindet. Er fährt dann etwas zurück und versucht ein Vorankommen mit einer Linkskurve.

Stösst er innerhalb kurzer Zeit wieder auf eine Wand, so weiss er, dass der eingeschlagene Weg falsch war. Er kehrt zurück und fährt nach rechts. Dabei merkt er sich, ob er für den richtigen Weg nach links oder rechts abbiegen muss und kann später den Kanal beliebig oft selbständig und ohne anzustossen durchlaufen.

Du lässt den Roboter im Teach-Mode durch den Kanal laufen und drückst nachher die Enter-Taste bzw. den ENTER-Button, um ihn vom Teach-Mode in den Execute-Mode zu versetzen.

```
# Robo2d.py

from simrobot import *
#from nxtrobot import *
#from ev3robot import *

import time

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)
RobotContext.showStatusBar(30)

def onPressed(port):
    global startTime
    global backTime
    robot.drawString("Press event!", 0, 1)
    dt = time.clock() - startTime # time since last hit in s
    gear.backward(backTime)
    if dt > 2:
        memory.append(0)
        gear.left(turnTime) # turning left
    else:
        memory.pop()
        memory.append(1)
        gear.right(2 * turnTime) # turning right
    robot.drawString("Mem: " + str(memory), 0, 1)
    gear.forward()
    startTime = time.clock()

def run():
    for k in memory:
        robot.drawString("Moving forward", 0, 1)
        gear.forward(moveTime)
        if k == 0:
            robot.drawString("Turning left", 0, 1)
            gear.left(turnTime)
        elif k == 1:
            robot.drawString("Turning right", 0, 1)
            gear.right(turnTime)
    gear.forward(moveTime)
    robot.drawString("All done", 0, 1)
    isExecuting = False

moveTime = 5000
turnTime = 580
backTime = 850
memory = []
robot = LegoRobot()
gear = Gear()
gear.setSpeed(50)
robot.addPart(gear)
ts = TouchSensor(SensorPort.S3, pressed = onPressed)
robot.addPart(ts)
startTime = time.clock()
gear.forward()
robot.drawString("Moving forward", 0, 1)
```

```
while not robot.isEscapeHit():
    if robot.isEnterHit():
        robot.reset()
        run()
robot.exit()
```

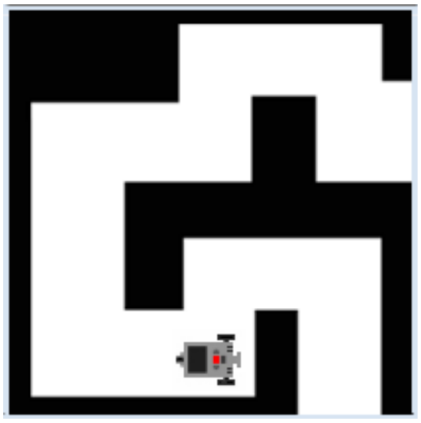
MEMO

Der Touchsensor wird am Port S3 angeschlossen (im Simulationsmodus entspricht dies einer Montageposition vorne in der Mitte). Der Sensor meldet die Touchevents über den Callback *onPress()*, der mit dem benannten Parameter *press* registriert wird. Der Touchsensor ist ein Roboter-Bauteil, der wie üblich mit *addPart()* zum Roboter hinzugefügt wird. Um herauszufinden, ob sich der Roboter in einem Kanalelement oder in einer Sackgasse bewegt hat, bestimmst du mit der in Python eingebauten Clock die Zeit seit dem letzten Touchevent. Ist sie grösser als 2 Sekunden, bewegt er sich im Kanalelement, sonst war es eine Sackgasse. Da der Roboter immer zuerst nach links dreht und dabei eine 0 in sein Gehirn schreibt, muss er im Fall der Sackgasse die fehlerhafte 0 durch 1 ersetzen.

DIE UMGEBUNG WIRD KOMPLEXER

Du hast sicher gemerkt, dass der Roboter durchaus in der Lage ist, selbst zu merken, wie weit er vorwärts fahren muss, bis er an die nächste Abzweigung kommt, da er ja die Zeit messen kann, die er benötigt, bis er am Ende des Kanalelements anstösst. Dadurch ist er in der Lage, sich auch in einem Kanal mit verschieden langen Kanalelementen richtig zu bewegen. Allerdings muss er jetzt nicht nur die Links-Rechts-Information, sondern auch die Laufzeit im Kopf behalten. Du packst am besten beide zusammen gehörenden Informationen in eine Liste *node = [moveTime, k]*, wo *moveTime* die Laufzeit (in ms) und *k = 0* für eine Links-, *k = 1* für eine Rechtskurve stehen.

Die Laufzeit *moveTime* kriegst du nach dem Durchlaufen des Kanalelements, musst sie aber noch um die Zeit korrigieren, um die der Roboter zu weit gefahren ist. Du speicherst sie in einer globalen Variable, da du sie nochmals verwenden musst, falls der Roboter in die Sackgasse gefahren bist.



```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *
import time

RobotContext.useObstacle("sprites/bg2.gif", 250, 250)
RobotContext.setStartPosition(410, 460)
RobotContext.showStatusBar(30)

def pressCallback(port):
    global startTime
```

```
    global backTime
    global turnTime
    global moveTime
    dt = time.clock() - startTime # time since last hit in s
    gear.backward(backTime)
    if dt > 2:
        moveTime = int(dt * 1000) - backTime  # save long-track time
        node = [moveTime, 0]
        memory.append(node) # save long-track time
        gear.left(turnTime) # turning left
    else:
        memory.pop() # discard node
        node = [moveTime, 1]
        memory.append(node)
        gear.right(2 * turnTime) # turning right
    robot.drawString("Memory: " + str(memory), 0, 1)
    gear.forward()
    startTime = time.clock()

def run():
    for node in memory:
        moveTime = node[0]
        k = node[1]
        robot.drawString("Moving forward",0, 1)
        gear.forward(moveTime)
        if k == 0:
            robot.drawString("Turning left",0, 1)
            gear.left(turnTime)
        elif k == 1:
            robot.drawString("Turning right",0, 1)
            gear.right(turnTime)
    gear.forward() # must stop manually
    robot.drawString("All done, press DOWN to stop", 0, 1)
    isExecuting = False

turnTime = 580
backTime = 850

robot = LegoRobot()
gear = Gear()
gear.setSpeed(50)
robot.addPart(gear)
ts = TouchSensor(SensorPort.S3, pressed = pressCallback)
robot.addPart(ts)
startTime = time.clock()
moveTime = 0
memory = []
gear.forward()

while not robot.isEscapeHit():
    if robot.isDownHit():
        gear.stop()
    elif robot.isEnterHit():
        robot.reset()
        run()
robot.exit()
```

MEMO

Im ersten Anlauf erscheint dir die Datenstruktur von *memory* als eine Liste von *node*-Listen kompliziert. Zusammengehörende Daten (hier die Laufzeit für die nächste Strecke und die darauf folgende Links-Rechts-Info) sollten aber immer in einer gemeinsamen Datenstruktur abgelegt werden. Erfreue dich daran, dass der Roboter sich auch mit dem gleichen Programm in einem ganz anderen Kanal (z.B. *bg3.gif*) zurecht findet.

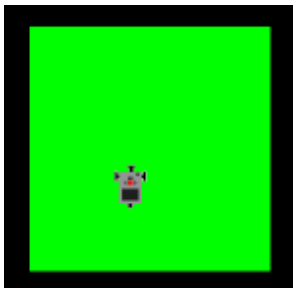
AUFGABEN

1. Der Roboter mit einem Touchsensor soll einen Kanal mit rechtwinkligen Kurven und ungleichlangen Strecken selbständig durchlaufen, aber nicht lernfähig sein. Verwende den Hintergrund *bg2.gif.*

2. Schreib ein Programm für einen Rasenmäher-Roboter mit einem Touchsensor, der streifenweise einen Rasen mäht. Oben und unten stösst er an eine Rasenbegrenzung.

   Verwende für den Simulationsmode folgende RobotContext-Optionen:

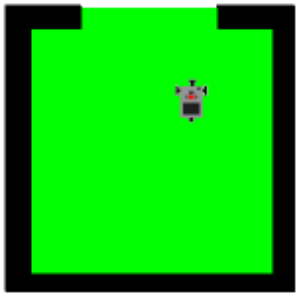
   ```
   RobotContext.useBackground("sprites/fieldbg.gif")
   RobotContext.useObstacle("sprites/field1.gif", 250, 250)
   RobotContext.setStartPosition(350, 300)
   ```

3. Die Rasenbegrenzung hat leider ein Loch, sodass der Roboter den Rasen verlässt. Erstell ein Programm mit einem lernenden Roboter, der nach dem Abfahren der ersten Streifen weiss, wie lange die Streifen sind und den Touchsensor nicht mehr benötigt.

   Verwende für den Simulationsmode folgende RobotContext-Optionen:

   ```
   RobotContext.useBackground("sprites/fieldbg.gif")
   RobotContext.useObstacle("sprites/field2.gif", 250, 250)
   RobotContext.setStartPosition(350, 300)
   ```

ZUSATZSTOFF

TEACH-MODE MIT DER INFRAROT-FERSTEUERUNG

(nur EV3 autonomer Modus)

Du hast bereits im letzen Kapitel gelernt, wie man mit der EV3 Infrarot-Fernsteuerung umgeht. Hier setzt du sie dazu ein, den Roboter im Teach-Mode durch den Kanal zu führen. Sowohl im Teach- wie im Execute-Mode läuft das Programm autonom auf dem EV3, du benötigst also nach dem Download keinen externen Computer mehr.

Dieses Vorgehen kommt der Realität sehr nahe, da es bei Industrierobotern üblich ist, sie mit einer Fernsteuerung "anzulernen" und dann das "gelernte" Programm abzuspielen.

Du verwendest im Teach-Mode die beiden oberen Buttons der Fernsteuerung, um den Roboter nach links oder rechts fahren zu lassen. Drückst du den linken unteren Button, so startet der Execute-Mode. Auch hier ist es elegant, mit Zuständen zu arbeiten.

```
from ev3robot import *

def onActionPerformed(port, command):
    global state
    if command == 1:
        state = "LEFT"
```

```
    elif command == 3:
        state = "RIGHT"
    elif command == 2:
        state = "RUN"

moveTime = 5000
turnTime = 580
memory = []
robot = LegoRobot()
gear = Gear()
gear.setSpeed(50)
robot.addPart(gear)
irs = IRRemoteSensor(SensorPort.S1, actionPerformed = onActionPerformed)
robot.addPart(irs)
state = "FORWARD"
robot.drawString("Learning...", 0, 3)

while not robot.isEscapeHit():
    if state == "FORWARD":
        gear.forward(moveTime)
        state = "STOPPED"
    elif state == "LEFT":
        memory.append(0)
        gear.left(turnTime)
        gear.forward(moveTime)
        state = "STOPPED"
    elif state == "RIGHT":
        memory.append(1)
        gear.right(turnTime)
        gear.forward(moveTime)
        state = "STOPPED"
    elif state == "RUN":
        robot.drawString("Executing...", 0, 3)
        robot.reset()
        gear.forward(moveTime)
        for k in memory:
            if k == 0:
                gear.left(turnTime)
            else:
                gear.right(turnTime)
            gear.forward(moveTime)
        gear.stop()
        robot.drawString("All done", 0, 3)
        state = "STOPPED"
robot.exit()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

5.3 STEUERN UND REGELN

EINFÜHRUNG

Im Zusammenhang mit Maschinen, zu denen auch die Roboter gehören, stellt sich oft das Problem, sie so zu steuern, dass eine bestimmte Messgrösse einen vorgegebenen Wert, den **Sollwert**, möglichst gut einhält. Beispielsweise soll im Auto der Tempomat eine vorgegebene Geschwindigkeit auch dann einhalten, wenn das Auto eine Steigung oder ein Gefälle durchfährt. Dazu muss ein **Regelungssystem** mit einem Sensor die aktuelle Geschwindigkeit, den **Istwert**, bestimmen und mit einem Stellglied die Leistung des Motors entsprechend anpassen, also sozusagen das Gaspedal bedienen.

Weitere Beispiele von technischen Regelungssystemen:

- Temperatur eines Kühlschranks einhalten (Thermostatregelung)
- Ein Flugzeug auf einem bestimmten Kurs halten (Autopilot)
- Füllstand eines Flüssigkeitsreservoirs einhalten (z.B. WC-Spülung)

Auch viele menschliche Tätigkeiten können als Regelungsprozess aufgefasst werden. Beispiele:

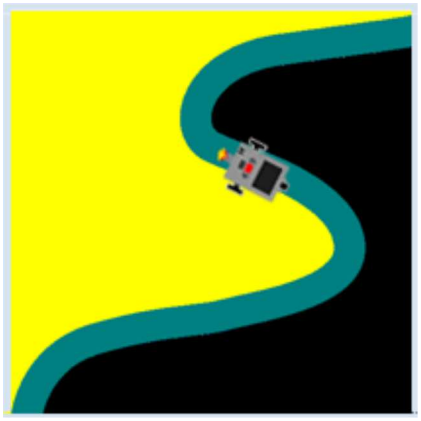
- Auto lenken, dass es auf der Strasse bleibt
- Soviel arbeiten, dass man gerade knapp durch die Promotion kommt
- Beim Stehen auf einem Fuss das Gleichgewicht behalten

PROGRAMMIERKONZEPTE: *Regelungssystem, Istwert, Sollwert, Messfehler*

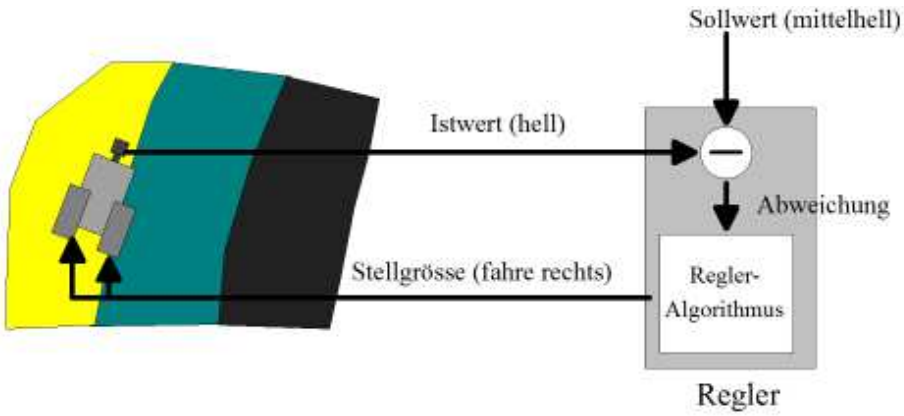
SELF-DRIVING CAR

Das Autofahren ist ein komplexer Regelungsprozess mit vielen Eingangssignalen, die vom Fahrer optisch, aber auch taktil (Kräfte am Körper) erfasst werden. Die geistige Verarbeitung dieser Signale führt zu seinem Verhalten (Drehung des Steuerrades, Drücken der Pedale, usw.).
In der Zukunft werden Fahrzeuge den Weg sogar in einer komplexen verkehrstechnischen Situation ohne einen menschlichen Fahrer finden. Mehrere Forschungsgruppen auf der ganzen Welt arbeiten an diesem Problem und vielleicht wirst du dich einmal an dieser interessanten Forschung beteiligen. In einer stark vereinfachten Situation kannst du hier deine Fähigkeiten bereits auf eine Probe stellen.

Deine Aufgabe ist es, den Roboter, der mit einem Fahrwerk und einem Lichtsensor ausgerüstet ist und damit die Helligkeit der Unterlage misst, auf einer grünen Strasse entlang zu führen, die von einem gelben und schwarzen Gebiet umgeben ist. Der Sensor meldet auf der grünen Unterlage einen mittleren, auf der gelben eine grossen und auf der schwarzen einen kleinen Lichtwert. Es ist die Aufgabe des Regelungssystems, aus den gemessenen Lichtwerten ein Steuersignal für die Motoren zu erstellen, damit der Roboter möglichst gut der Strasse entlang fährt.



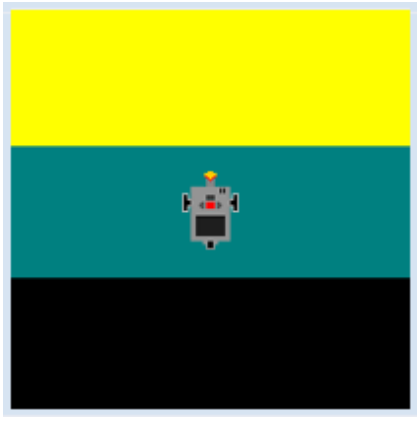
Schematisch kann man diesen Prozess in einem **Regelungskreis** aufzeichnen. Der Lichtsensor misst den aktuellen Lichtwert (Istwert) und liefert ihn an den Regler. Dieser vergleicht ihn mit dem gewünschten Wert (Sollwert) auf der grünen Strasse. Aus der Differenz errechnet der Regler gemäss einem von dir zu erfindenden Regleralgorithmus die Steuergrösse für das Fahrwerk, d.h. die zwei Motoren werden entsprechend geschaltet.



Regelungskreis

Wie du siehst, handelt es sich tatsächlich schematisch um einen "Kreis" vom Fahrzeugsensor, zum Regelungssystem und wieder zurück zu den Fahrzeugmotoren

Bevor du das Programm schreiben kannst, musst du noch die Lichtwerte kennen, die für die gelbe, grüne und schwarze Unterlage vom Sensor geliefert werden. Dazu schreibst du am besten ein kleines Testprogramm, das dir die gemessen Werte auf der Konsole bzw. auf dem Display ausschreibt. Im Realmodus brauchst du den Roboter nicht zu bewegen, sondern du kannst ihn einfach auf die entsprechende Unterlage stellen. Im Simulationsmodus fährst du über die entsprechend gefärbten Gebiete. (Dieser Prozess heisst **Eichung**.) Für den NXT verwendest du den NXT-LightSensor, für den EV3 kannst du den EV3-ColorSensor einsetzen.



```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.setStartPosition(250, 490)
RobotContext.useBackground("sprites/roadtest.gif")

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
ls = LightSensor(SensorPort.S3)
robot.addPart(ls)
```

```
ls.activate(True)
gear.forward()

while not robot.isEscapeHit():
    v = ls.getValue()
    print v
    Tools.delay(100)
robot.exit()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

In deinem Programm verwendest du einen naheliegenden Regelalgorithmus: Ist der Istwert grösser als der Sollwert, so befindet sich das Fahrzeug im gelben Gebiet und du fährst eine Rechtskurve. Ist der Istwert hingegen kleiner als der Sollwert, so befindest es sich im schwarzen Gebiet und du fährst eine Linkskurve. Sonst fährst du geradeaus.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.setStartPosition(50, 490)
RobotContext.useBackground("sprites/road.gif")

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
ls = LightSensor(SensorPort.S3)
robot.addPart(ls)
ls.activate(True)
gear.forward()
nominal = 501

while not robot.isEscapeHit():
    actual = ls.getValue()
    if actual == nominal:
        gear.forward()
    elif actual < nominal:
        gear.leftArc(0.1)
    elif actual > nominal:
        gear.rightArc(0.1)

robot.exit()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

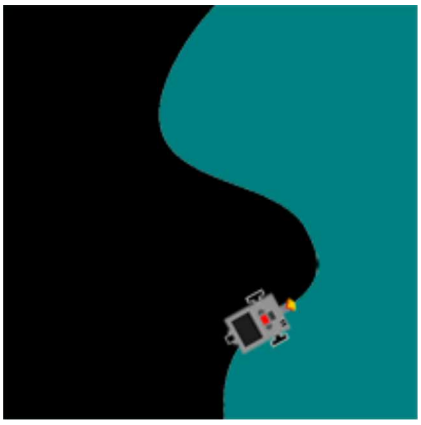
Im Simulationsmodus funktioniert die Regelung gut, im Realmodus aber gar nicht. Du merkst sicher warum: Die Messwerte des Sensors schwanken, auch wenn sich der Sensor über einem einheitlich gefärbten Gebiet befindet. Das ist ja auch zu erwarten, da die Helligkeit selbst auf der gleichen Unterlage auf Grund der Beleuchtungsunterschiede, aber auch wegen **Messfehlern** des Sensors, niemals exakt gleich bleibt. Finde selbst eine Lösung für dieses Problem!

Ein sensibler Parameter ist der Kurvenradius bei *leftArc()* bzw. *rightArc()*. In kleiner Wert führt zu kleineren "Ausreissern" von der Strasse weg, aber zu einem unruhigen, hin-und-her schwingenden Verhalten [**mehr...**], ein grosser Wert zur Beruhigung der Bewegung, aber zu einer weniger präzisen Führung auf der Strasse. Bestätige dies durch einige Versuche mit verändertem Kurvenradius.

AUFGABEN

1. Fahre mit einem Lichtsensor entlang einer schwarz-grünen Kante. Im Simulationsmodus verwendest du den folgenden RobotContext:

   ```
   RobotContext.useBackground("sprites/edge.gif")
   RobotContext.setStartPosition(250, 490)
   ```



2. Fahre mit zwei Lichtsensoren auf einem runden Pfad.

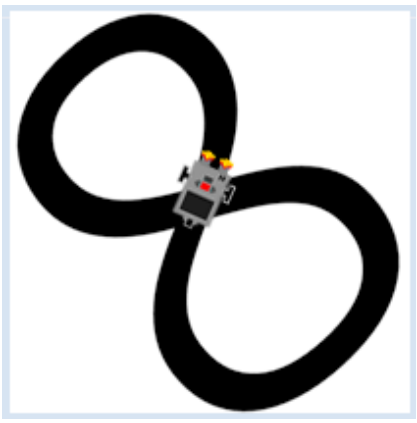
   Im Simulationsmodus verwendest du den RobotContext:

   ```
   RobotContext.useBackground("sprites/roundpath.gif")
   RobotContext.setStartPosition(250, 250)
   RobotContext.setStartDirection(-90)
   ```

   Ändere die Startposition und die Startrichtung so, dass der Roboter bereits auf der Bahn startet.

3. Fahre mit zwei Lichtsensoren auf einer Achterbahn. Im Simulationsmodus verwendest du das Hintergrundbild *track.gif*.

   ```
   RobotContext.useBackground("sprites/track.gif")
   ```



5.4 SENSORIK

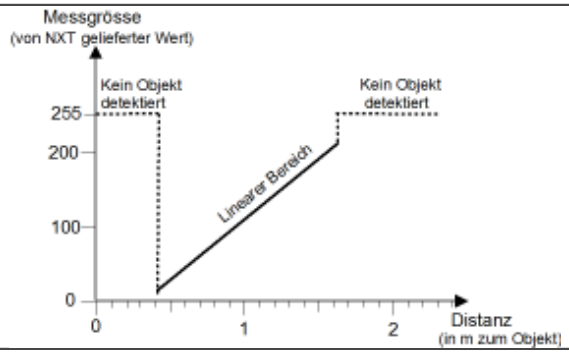
EINFÜHRUNG

Ein Sensor ist ein Messgerät für eine physikalische Grösse, beispielsweise der Temperatur, der Lichtintensität, des Drucks, des Abstands, usw. Meist kann der vom Sensor gelieferte Wert eine beliebige Zahl innerhalb des Messbereichs sein. Es gibt allerdings auch Sensoren, die nur zwei Zustände ähnlich einem Schalter kennen, beispielsweise Füllstanddetektoren, Berührungssensoren, usw.

Die physikalische Grösse wird im Sensor üblicherweise in eine elektrische Spannung umgewandelt und von einer Auswertelektronik weiter verarbeitet [**mehr...**]. Sensoren können intern kompliziert aufgebaut sein, wie beispielsweise Ultraschallsensoren, Gyrosensoren oder Laserdistanzmesser. Unter der **Sensorkennlinie** versteht man den Zusammenhang zwischen dem physikalischen Messwert und dem vom Sensor abgegebenen Wert. Für viele Sensoren verläuft die Kennlinie einigermassen linear, es muss aber der Umrechnungsfaktor und die Nullpunktverschiebung bestimmt werden. Dazu wird der Sensor in einer Messserie mit bekannten Grössen **geeicht**.

Der Ultraschallsensor bestimmt die Distanz zu einem Objekt durch die Laufzeit, die ein kurzer Ultraschall-Puls benötigt, um vom Sensor zum Objekt und wieder zurück zu laufen. Der Sensor liefert für Distanzen im Bereich von ca. 30 cm und 2 m Werte zwischen 0 und 255, wobei 255 (in der Simulation -1) dann abgegeben wird, wenn kein Objekt im Messbereich ist.

Ultraschallsensor



Kennlinie

In den meisten Anwendungen wird ein Sensor so in das Programm eingebunden, dass dieses periodisch den Sensorwert abfragt. Man nennt dies **Pollen des Sensors**. Der in einer Wiederholschleife abgefragte Werte dann im Programm weiter verarbeitet. Die Anzahl Messwerte pro Sekunde (zeitliche Auflösung) hängt vom Sensortyp, der Geschwindigkeit des Rechners und der Datenübertragung vom Brick zum Programm ab. Der Ultraschallsensor liefert nur ungefähr 2 Messwerte pro Sekunde.

Der Zustand von Sensoren, die nur zwei Zustände haben, kann ebenfalls durch Pollen abgefragt werden. Oft ist es aber einfacher, den Wechsel des Zustands, also die Zustandsänderung, als einen **Event** aufzufassen und ihn programmtechnisch mit einem **Callback** zu verarbeiten.

PROGRAMMIERKONZEPTE: *Sensor, Sensorkennlinie, Sensoreichung, Pollen&Event, Triggerpegel*

POLLEN ODER EVENTS VERWENDEN?

In vielen Fällen kannst du dich entscheiden, ob du einen Sensor lieber mit Pollen oder mit Events einbinden willst. Dies hängt etwas von der Anwendung ab. Du kannst die beiden Verfahren miteinander vergleichen, wenn du an einem Brick einen Motor und einen Berührungssensor

anschliesst. Beim Klicken des Berührungssensors soll der Motor eingeschaltet, beim nächsten Klicken ausgeschaltet werden.

Für diese Anwendung **sind Events viel eleganter**, da sie dir das Drücken des Berührungssensors als Aufruf einer Funktion mitteilen. Du musst beim Erzeugen des TouchSensors diese Funktion als benannten Parameter angeben. Beim Pollen ist es nötig, mit einem Flag dafür sorgen, dass du nur **den Übergang** vom nicht gedrückten zum gedrückten Zustand verarbeitest.

Mit Pollen:

```
from nxtrobot import *
#from ev3robot import *

def switchMotorState():
    if motor.isMoving():
        motor.stop()
    else:
        motor.forward()

robot = LegoRobot()
motor = Motor(MotorPort.A)
robot.addPart(motor)
ts = TouchSensor(SensorPort.S3)
robot.addPart(ts)

isOff = True
while not robot.isEscapeHit():
    if ts.isPressed() and isOff:
        isOff = False
        switchMotorState()
    if not ts.isPressed() and not isOff:
        isOff = True
```

Mit Events:

```
#from nxtrobot import *
from ev3robot import *

def onPressed(port):
    if motor.isMoving():
        motor.stop()
    else:
        motor.forward()

robot = LegoRobot()

motor = Motor(MotorPort.A)
robot.addPart(motor)
ts = TouchSensor(SensorPort.S1,
         pressed = onPressed)
robot.addPart(ts)
while not robot.isEscapeHit():
    pass
robot.exit()
```

MEMO

Sensoren können mit Pollen oder Events programmiert werden. Du musst beide Verfahren kennen und in der Lage sein zu entscheiden, welches in einer bestimmten Situation das zweckmässigere ist. Im Eventmodell definierst du Funktionen, deren Name üblicherweise mit "on" beginnt. Diese werden **Callbacks** genannt, weil sie vom System beim Auftreten des

Events automatisch aufgerufen ("zurückgerufen") werden. Du musst Callbacks bei der Erzeugung des Sensorobjekts mit benannten Parametern **registrieren**.

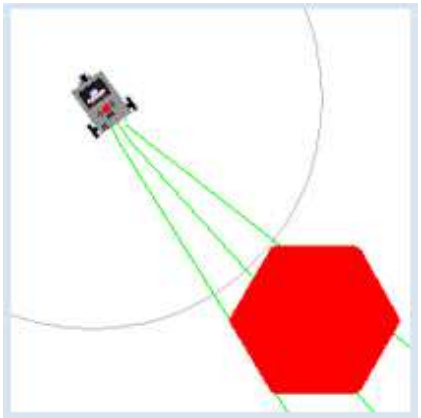
POLLEN EINES ULTRASCHALL-SENSORS

Vorbemerkung: Falls du in deinem EV3-Bausatz keinen Ultraschall-Sensor hast, so kannst du auch den EV3 Infrarotsensor einsetzen.

Einen Sensor musst du immer dann pollen, wenn du laufend seine Messwerte benötigst. Du stellst dir die Aufgabe, dass der Roboter, den du irgendwo auf den Boden stellst, einen Gegenstand (ein Ziel, Target) finden soll und zu diesem hinfährt.

Zum Erfassen eines Targets verwendest du einen Ultraschall-Sensor, der ähnlich wie ein Radar-Zielerfassungssystem eingesetzt wird. Um die Eigenschaften eines Sensors kennen zu lernen und ihn auszuprobieren, solltest du den Aufwand nicht scheuen, ein kurzes Testprogramm zu schreiben, das du später nicht mehr brauchen wirst. Dabei ist es zweckmässig, die Sensorwerte auszuschreiben und eventuell auch hörbar zu machen, da du dann Hände und Augen frei hast, um den Roboter und den Sensor zu bewegen. In einer Schleife fragst du die Sensorwerte ab, wobei die Schleifenperiode angepasst wird, je nachdem ob du im autonomen oder fremdgesteuerten Modus bist.

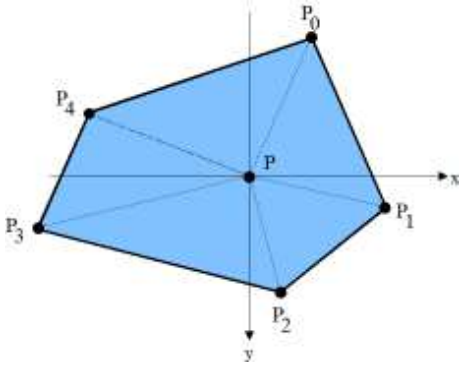
```
# from nxtrobot import *
from ev3robot import *

robot = LegoRobot()
us = UltrasonicSensor(SensorPort.S1)
robot.addPart(us)
isAutonomous = robot.isAutonomous()
while not robot.isEscapeHit():
    dist = us.getDistance()
    print "d = ", dist
    robot.drawString("d=" + str(dist), 0, 3)
    robot.playTone(10 * dist + 100, 50)
    if dist == 255:
        robot.playTone(10 * dist + 100, 50)
    if isAutonomous:
       Tools.delay(1000)
    else:
       Tools.delay(200)
robot.exit()
```



Um ein Target zu finden und zu ihm hinzufahren rotierst du den Roboter wie eine Radarantenne und suchst dabei ständig mit dem Ultraschallsensor nach dem Target. Hast du das Ziel erfasst, so merkst du dir die Richtung und drehst noch weiter, bis kein Echo mehr auftritt. Damit kannst du die scheinbare Grösse des Targets (den Winkelbereich, in dem das Target "sichtbar" ist) bestimmen. Du fährst dann mit dem Roboter in der Mitte des Winkelbereichs und hältst in einer bestimmten Distanz an.

In der Simulation kannst du mit **setBeamAreaColor()** und **setProximityCircleColor()** die Distanzmessung anschaulich machen. Das angezeigte Target entspricht der Bilddatei, welche in *RobotContext.useTarget()* angegeben wird.

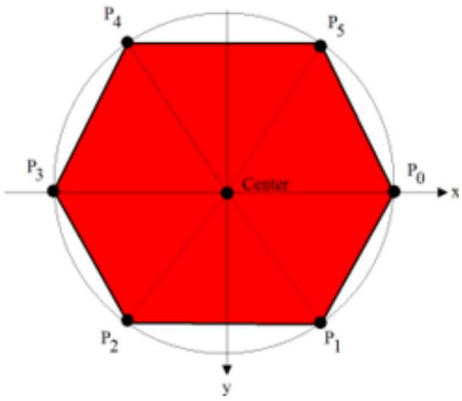
Für die Erfassung des Targets durch den simulierten Sensor wird aber nicht dieses Bild, sondern ein Netz aus Dreiecksmaschen (mesh) verwendet. Diese bestehen aus dem gemeinsamen Zentralpunkt und zwei Eckpunkten. Das angezeigte Target besitzt die Maschen
PP_0P_1, PP_1P_2, PP_2P_3, PP_3P_4, PP_4P_0.



Im Programm gibt du die die Eckpunkte der Maschen als Parameter der Methode *useTarget()* an. Die Koordinaten beziehen sich auf ein Pixelkoordinatensystem mit Nullpunkt im Zentrum und positiver x- Achse nach rechts und positiver y-Achse nach unten.

Für ein Sechseck mit dem Durchmesser 100 lauten die Maschenkoordinaten

[50, 0] , [25, 43], [-25, 43], [-50, 0], [-25, -43], [25, -43].




```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

mesh = [[50, 0], [25, 43], [-25, 43], [-50, 0],
          [-25, -43], [25, -43]]
RobotContext.useTarget("sprites/redtarget.gif", mesh, 400, 400)

def searchTarget():
    global left, right
    found = False
    step = 0
    while not robot.isEscapeHit():
        gear.right(50)
        step = step + 1
        dist = us.getDistance()
        print "d = ", dist
        if dist != -1:  # simulation
        #if dist < 80:   # real
            if not found:
                found = True
                left = step
                print "Left at", left
                robot.playTone(880, 500)
        else:
            if found:
                right = step
                print "Right at ", right
                robot.playTone(440, 5000)
                break

left = 0
right = 0
robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
us = UltrasonicSensor(SensorPort.S1)
robot.addPart(us)
us.setBeamAreaColor(makeColor("green"))
us.setProximityCircleColor(makeColor("lightgray"))
gear.setSpeed(5)
```

```
print "Searching..."
searchTarget()

gear.left((right - left) * 25)   # simulation
#gear.left((right - left) * 100)  # real

print "Moving forward..."
gear.forward()

while not robot.isEscapeHit() and gear.isMoving():
    dist = us.getDistance()
    print "d =", dist
    robot.playTone(10 * dist + 100, 100)
    if dist < 40:
        gear.stop()
print "All done"
robot.exit()
```

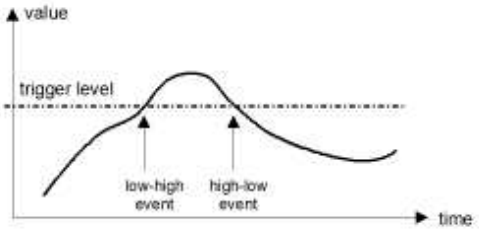
MEMO

Meist bestimmt man die Sensorwert durch wiederholtes Abfragen (Pollen) mit einer getter-Methode (*getValue()*, **getDistance()**, usw.) Du musst gewisse Werte beim Wechsel zwischen Simulationsmodus und Realmodus anpassen, insbesondere Zeitintervalle. Zudem musst du berücksichtigen, dass der Sensor im Simulationsmodus -1 und im Realmodus 255 zurückgibt, wenn er kein Target findet. In der Simulation bestimmt der verwendete Sensorport die Blickrichtung des Ultraschallsensors:

Sensorport	**Detektionsrichtung**
S1	Vorwärts
S2	Links
S3	Rückwärts

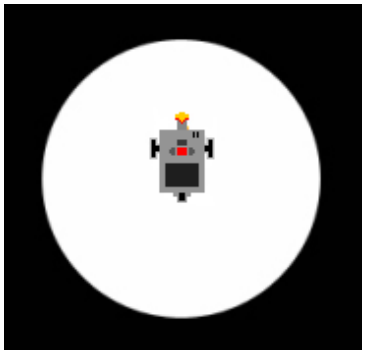
EVENTS MIT EINEM TRIGGERPEGEL

Auch Sensoren, die kontinuierliche Werte liefern, können mit dem Eventmodell programmiert werden. Dabei definiert man einen bestimmten Messwert als **Schwellenwert**, meist **Triggerpegel** genannt. Ein Event wird dann ausgelöst, wenn dieser Pegel überquert wird, entweder von kleineren zu grösseren Werten oder umgekehrt.



Die Sensoren besitzen einen Standardwert für den Triggerpegel. Diesen kannst du aber mit *setTriggerLevel()* verändern.

Dein Programm wacht darüber, dass der fahrende Roboter innerhalb eines kreisförmigen Gebiets bleibt (und damit beispielsweise nicht von einem Tisch fällt). Dabei verwendest du den Lichtsensor, der hier nur auf hell und dunkel reagieren muss. Ist die Unterlage dunkel, so wird der Callback **onDark** ausgelöst.



Für den Realmodus mit dem NXT ist es wichtig, dass du mit **activate(True)** die LED-Beleuchtung des Sensors einschaltest.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.setStartPosition(250, 200)
RobotContext.setStartDirection(-90)
RobotContext.useBackground("sprites/circle.gif")

def onDark(port, level):
    gear.backward(2700)
    gear.left(580)
    gear.forward()

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
ls = LightSensor(SensorPort.S3,
      dark = onDark)
robot.addPart(ls)
ls.setTriggerLevel(100)  # adapt value
gear.forward()
while not robot.isEscapeHit():
    pass
robot.exit()
```

MEMO

Des Überqueren eines bestimmten Messwerts kann als ein Event aufgefasst werden. Man spricht dabei von **Triggerung**. Standardwerte des Triggerpegels:

Sensor	Triggerpegel (Standard)
Soundsensor	50
Lichtsensor	500
Ultraschallsensor	10

Die Vor- und Nachteile des Event-Modells gegenüber Pollens:

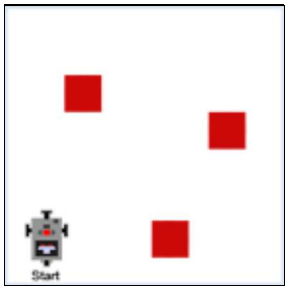
Vorteile des Event-Modells	Nachteile des Event-Modells
Vereinfachte übersichtlichere Programmierung, da der Code im Callback vom übrigen Programm losgelöst ist	Das laufende Programm wird zu unvorhersehbarer Zeit (asynchron) unterbrochen. Dies kann den übrigen Programmablauf stören
Der Event wird immer erfasst, auch wenn der PC langsam ist	Callbacks können unerwünschte Nebenwirkungen haben, z.B. wenn sie globale Variablen oder den Zustand des Roboters verändern
Das Hauptprogramm kann normal weiterlaufen und braucht sich nicht um den Sensor zu "kümmern"	Callbacks laufen in einem eigenen Prozess, daher kann es zu Konflikten zwischen Prozessen (Threads) kommen
Triggerung ist ein zentraler Begriff der Messtechnik	Es kann nur ein bestimmter Wert (Triggerpegel) erfasst werden
Das Event-Modell ist dem Denken in Zuständen angepasst (der Event setzt das System in einen neuen Zustand)	Callbacks sollten in der Regel nur kurz dauernden Code enthalten, da sonst weitere Events verloren gehen können

AUFGABEN

1. Der Roboter soll sich beim ersten Klatschen in Bewegung setzen und bei weiterem Klatschen seine Richtung ändern. Löse das Problem im Realmodus und im Simulationsmodus. Im Realmodus verwendest du den Soundsensor. Im Simulationsmodus brauchst du an deinem PC ein Mikrofon und du musst in der Systemsteuerung den Mikrofonpegel richtig einstellen.

2. Schliesse am Brick einen Motor und einen Berührungssensor an und schreibe ein Programm so, dass beim Drücken des Sensorknopfs der Motor eingeschaltet und beim Loslassen der Motor wieder ausgeschaltet wird.

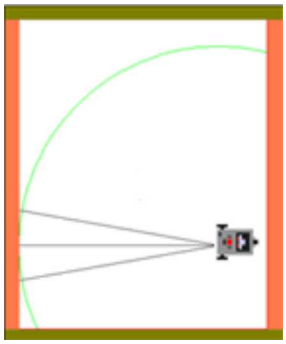
3. Ein Roboter mit einem Ultraschallsensor und einem Berührungssensor soll 3 höhere Gegenstände (Kerzen, Büchsen...) finden, auf sie losfahren und sie umstossen. Im Simulationsmodus kannst du das Umstossen als Touchevent auffassen und *squaretarget.gif* verwenden, um die Gegenstände darzustellen. Das Bild ist 60x60 Pixel gross. Für den RobotContext kannst du folgende Vorlage verwenden. Versuche die Angaben unter *mesh* zu verstehen.



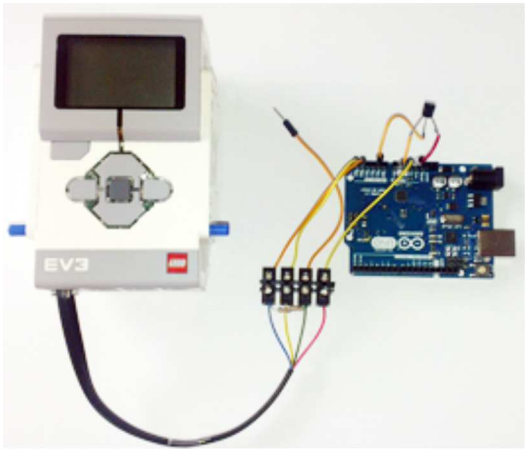

```
mesh = [[-30, -30], Point[-30, 30], Point[30, -30], Point[30, 30]]
RobotContext.useTarget("sprites/squaretarget.gif", mesh, 350, 250)
RobotContext.useObstacle("sprites/squaretarget.gif", 350, 250)
RobotContext.useTarget("sprites/squaretarget.gif", mesh, 100, 150)
RobotContext.useObstacle("sprites/squaretarget.gif", 100, 150)
RobotContext.useTarget("sprites/squaretarget.gif" ,mesh, 200, 450)
RobotContext.useObstacle("sprites/squaretarget.gif", 200, 450)
RobotContext.setStartPosition(40, 450)
```

4*. Ein Roboter mit einem Ultraschallsensor wird zu Beginn an eine zufällige Position in ein rechteckiges Feld gesetzt. Seine Aufgabe ist es, möglichst rasch und genau die Mitte des Feldes zu finden. Die Aufgabe kann im Simulations- und/oder Realmodus gelöst werden.

   Als Target können die Bilddatei *bar0.gif* und *bar1.gif* verwendet werden.



ZUSATZSTOFF: ARDUINO-SENSOREN VERWENDEN



Im Unterschied zum EV3 besitzt das bekannte **Arduino**-Microcontroller-System ein klassisches I/O-System mit digitalen Ein- und Ausgngsports und analogen Eingängen (Analog-Digital-Wandler). Damit lassen sich billigen Sensoren und selbstgebaute elektronische Schaltungen anschliessen. Diese Vorteile kann man nutzen, indem man beide Geräte zusammenschliesst und die Sensoren und Aktoren des Arduino mit dem EV3 anspricht. Die Verbindung der beiden Geräte erfolgt am einfachsten über einen I2C-Link, da beide Geräte über eine I2C-Schnittstelle verfügen. Dabei ist der EV3 ein I2C-Master und der Arduino ein I2C-Slave. Die zusätzlich benötigten Python-Module sind bereits in der Distribution von TigerJython enthalten. Der EV3 kann dabei im autonomen oder direkten Modus betrieben werden. Du kannst dich unter **http://www.aplu.ch/ev3** genauer informieren.

Dokumentation Robotik

Module import: **from simrobot import ***
from ev3robot import *
from nxtrobot import *

LegoRobot:

LegoRobot()	erzeugt Roboter (ohne Motoren) und stellt die Verbindung zum Roboter her
addPart(part)	fügt eine Komponente zum Roboter
clearDisplay()	Löscht die Anzeige [*Simulationsmodus:* Status bar]
drawString(text, x, y)	Schreibt text an Position x, y [*Simulationsmodus:* in Status bar, (x, y) irrelevant]
isButtonHit()	True, falls DOWN-Taste gedrückt [*NXT und Simulationsmodus:* Tastaturtaste *Cursor-Down*]
isEnterHit()	True, falls ESCAPE-Taste gedrückt [*NXT und Simulationsmodus:* Tastaturtaste *Escape*]
isLeftHit()	True, falls LEFT-Taste gedrückt [*NXT und Simulationsmodus:* Tastaturtaste *Cursor-Left*]
isRightHit()	True, falls RIGHT-Taste gedrückt [*NXT und Simulationsmodus:* Tastaturtaste *Cursor-Right*]
isUpHit()	True, falls UP-Taste gedrückt [*NXT und Simulationsmodus:* Tastaturtaste *Cursor-Up*]
playTone(frequency, duration)	spielt einen Ton mit geg, Frequenz (in Hz) in Länge (ms) [*Simulationsmodus:* nicht vorhanden]
setVolume(volume)	setzt die Lautstärke für alle Tonausgaben (0..100)
setLED(pattern)	setzt EV3-LEDS: 0: aus, 1: grün, 2: rot, 3: rot hell, 4: grün blinkend, 5: rot blinkend, 6: rot blinkend hell, 7: grün doppelblinkend, 8: rot doppelblinkend, 9: rot doppelblinkend hell
exit()	stoppt den Roboter und beendet die Verbindung
isConnected()	gibt True zurück, falls die Verbindung unterbrochen ist, oder das Simulationsfenster geschlossen wurde
reset()	*Simulationsmodus:* setzt den Roboter an die Startposition/Startrichtung

Gear:

Gear()	erzeugt ein Fahrwerk mit 2 synchronisierten Motoren an Port A, B
backward()	fährt rückwärts (nicht-blockierende Methode)
backward(ms)	fährt während gegebener Zeit (in ms) rückwärts (blockierende Methode)
isMoving()	gibt True zurück, wenn das Fahrwerk in Bewegung ist
forward()	fährt vorwärts (nicht blockierende Methode)
forward(ms)	fährt während gegebener Zeit (in ms) vorwärts (blockierende Methode)
left()	dreht links (nicht blockierende Methode)
left(ms)	dreht während gegebener Zeit (in ms) links (blockierende Methode)
leftArc(radius)	fährt auf einem Linksbogen mit geg. Radius (nicht- blockierende Methode)
leftArc(radius, ms)	fährt während gegebener Zeit (in ms) auf einem Linksbogen (blockierende Methode)

right()	dreht rechts (nicht-blockierende Methode)
right(ms)	dreht während gegebener Zeit (in ms) rechts (blockierende Methode)
rightArc(radius)	fährt auf einem Rechtsbogen mit geg. Radius (nicht blockierende Methode)
rightArc(radius, ms)	fährt während gegebener Zeit (in ms) auf einem Rechtsbogen (blockierende Methode)
setSpeed(speed)	setzt die Geschwindigkeit
stop()	stoppt das Fahrwerk
getLeftMotorCount()	gibt momentanen Zählerstand für den linken Motor zurück [nicht für Sim]
getRightMotorCount()	gibt momentanen Zählerstand für den rechten Motor zurück [nicht für Sim]
resetLeftMotorCount()	setzt den Zähler des linken Motors auf 0 [nicht für Sim]
resetRightMotorCount()	setzt den Zähler des rechten Motors auf 0 [nicht für Sim]

TurtleRobot:

TurtleRobot()	erzeugt Roboter mit Fahrwerk mit Motoren an Port A, B
backward()	fährt rückwärts (nicht-blockierende Methode)
backward(step)	fährt die gegebene Anzahl Schritte rückwärts (blockierende Methode)
forward()	fährt vorwärts (nicht-blockierende Methode)
forward(step)	fährt die gegebene Anzahl Schritte vorwärts (blockierende Methode)
left()	dreht links (nicht-blockierende Methode)
left(angle)	dreht um den gegebenen Winkel links (blockierende Methode)
right()	dreht rechts (nicht-blockierende Methode)
right(angle)	dreht um den gegebenen Winkel rechts (blockierende Methode)
setTurtleSpeed(speed)	setzt die Geschwindigkeit

Motor:

Motor(MotorPort.port)	erzeugt einen Motor am Motorport A, B, C oder D
backward()	dreht den Motor rückwärts
forward()	dreht den Motor vorwärts
setSpeed(speed)	setzt die Geschwindigkeit
isMoving()	gibt True zurück, wenn der Motor in Bewegung ist
stop()	stoppt den Motor
getMotorCount()	gibt den momentanen Stand des Zählers zurück [nicht für Sim]
resetMotorCount()	setzt den Zähler auf 0 [nicht für Sim]
rotateTo(count)	setzt Zähler auf 0, bewegt Motor bis Zählerstand count und stoppt (blockierend) [nicht für Sim]
rotateTo(count, blocking)	wie rotateTo(count), aber mit blocking False nicht blockierend [nicht für Sim]
continueTo(count)	wie rotateTo(count), aber Zähler nicht auf 0 gesetzt [nicht für Sim]
continueTo(count, blocking)	wie rotateTo(count, blocking), aber Zähler nicht auf 0 gesetzt [nicht für Sim]
continueRelativeTo(count)	wie continueTo(count), aber count ist Inkrement [nicht für Sim]

continueRelativeTo(count, blocking)	wie continueTo(count, blocking), aber count ist Inkrement [nicht für Sim]

LightSensor:

LichtSensor(SensorPort.port)	erzeugt einen Lichtsensor am Port S1, S2, S3 oder S4
LightSensor(SensorPort.port, dark = onDark)	registriert die Callbackfunktion onDark
LightSensor(SensorPort.port, bright = onBright)	registriert die Callbackfunktion onBright
activate(True)	schaltet den LED des Lichtsensors ein (nur beim NXT notwendig)
activate(False)	schaltet den LED des Lichtsensors aus
getValue()	gibt den Wert des Lichtsensors zurück (Zahl ungefähr zwischen 0 und 1000)
setTriggerLevel(level)	setzt einen Triggerlevel
bright(port, level), dark(port, level)	Callback-Funktionen, die als benannte Parameter registriert werden können

ColorSensor:

ColorSensor(SensorPort.port)	erzeugt einen Colorsensor am Port S1, S2, S3, S4
getColor()	gibt die gemessene Farbe zurück als Color-Typ mit den Methoden getRed(), getGreen() und getBlue(), welche den RGB-Wert 0 - 255 liefern
getColorID()	gibt eine Farbidentifikationszahl zurück: 1: schwarz, 2: blau, 3:grün, 4: gelb, 5: rot, 6: weiss
getColorStr()	liefert die Farbe als String (BLACK, BLUE, GREEN, YELLOW, RED, WHITE und UNDEFINED)
getLightValue()	gibt die Helligkeit (im HSG Modell) der gemessenen Farbe zurück

TouchSensor:

TouchSensor(SensorPort.port)	erzeugt einen Berührungssensor am Port S1, S2, S3 oder S4
TouchSensor(SensorPort.port, pressed = onPressed)	registriert die Callbackfunktion onPressed
TouchSensor(SensorPort.port, release = onRelease)	registriert die Callbackfunktion onRelease
isPressed()	gibt True zurück, falls der Touchsensor gedrückt ist
pressed(port), released(port)	Callback-Funktionen, die als benannte Parameter registriert werden können

SoundSensor:

SoundSensor(SensorPort.port)	erzeugt einen Soundsensor am Port S1, S2, S3 oder S4
SoundSensor(SensorPort.port, loud = onLoud)	registriert die Callbackfunktion loudCallback
SoundSensor(SensorPort.port, quiet = onQuiet)	registriert die Callbackfunktion quietCallback
getValue()	gibt die Lautstärke zurück (ungefähr 0 - 100)
setTriggerLevel(level)	setzt einen Triggerlevel
loud(port, level), quiet(port, level)	Callback-Funktionen, die als benannte Parameter registriert werden können

UltrasonicSensor:

UltrasonicSensor(SensorPort.port)	erzeugt einen Ultraschallsensor am Port S1, S2, S3 oder S4
getDistance()	gibt die Entfernung zurück
setTriggerLevel(level)	setzt einen Triggerlevel (default: 10)
far(port, level), near(port, level)	Callback-Funktionen, die als benannte Parameter registriert werden können
setProximityCircleColor(color)	Simulationsmodus: setzt die Farbe des Suchkreises
setMeshTriangleColor(color)	Simulationsmodus: setzt die Füllfarbe der Maschen
eraseBeamArea()	Simulationsmodus: löscht den Strahlbereich

InfraredSensor (nur EV3):

IRRemoteSensor(SensorPort.port)	erzeugt einen Infrarot-Fernsteuerungssensor am Port S1, S2, S3, S4
getCommand()	gibt die aktuelle Kommando-ID zurück: 0: nichts, 1:oben-links, 2:unten-links, 3:oben-rechts, 4:unten-rechts,5:oben-links+oben-rechts,6:oben-links+unten-rechts,7:unten-links+oben-rechts, 8::unten-links+unten-.rechts,9:Zentrum,10:unten-links+oben-links,11:oben-rechts+unten-rechts. Der Kanal wird mit dem roten Schiebeschalter gewählt.1: oben, 4: unten. Er entspricht der Portnummer, wo der Sensor angeschlossen ist.
actionPerformed(port, command)	Callback-Funktion, die als benannter Parameter registriert werden kann
IRSeekSensor(SensorPort.port)	erzeugt einen Infrarot-Suchensor am Port S1, S2, S3, S4 (aktive IR-Quelle der Fernsteuerung, wenn der Zentrums-Button gedrückt wird)
v = getValue()	v.bearing gibt den Richtungswert (-12..12) und v.distance die Distanz (in cm) zur Quelle zurück. Der Kanal wird mit dem roten Schiebeschalter gewählt.1: oben, 4: unten. Er entspricht der Portnummer, wo der Sensor angeschlossen ist.
IRDistanceSensor(SensorPort.port)	erzeugt einen Infrarot-Distanzsensor am Port S1, S2, S3, S4 (reflektierendes Target)
getDistance()	gibt die Entfernung zurück (in cm; 255, falls die die Distanzmessung misslingt)

RobotContext (nur Simulation)

setStartDirection(angle)	setzt die Startrichtung des Roboters (0 gegen Osten, positiv im Uhrzeigersinn)
setStartPosition(x, y)	setzt die Startposition des Roboters (Pixelkoordinaten, Nullpunkt oben links
showStatusBar(height)	fügt eine Statusbar mit gegebener Höhe unten am Fenster an
setStatusText(text)	setzt den Statustext (alter Text wird gelöscht)
useBackground(filename)	fügt ein Hintergrundbild für den Lichtsensor ein
useObstacle(filename, x, y)	fügt ein Hindernis für den Touchsensor an der Position (x, y) ein
useTarget(filename, mesh, x, y)	fügt ein Target für den Ultraschallsensor an der Position (x, y) ein

kapitel sechs

INTERNET

Lernziele

- Du kennst String als Datentyp und kannst mit wichtigen Stringmethoden umgehen.
- Du weisst, was ein HTML-formatiertes Dokument ist und kennst einige HTML-Tags.
- Du kannst ein HTML-Dokument als Datei öffnen oder von einem Webserver herunterladen und in einem Browserfenster darstellen.
- Du kennst das Client-Server-Modell und kannst mit dem HTTP-GET-Befehl eine Datei vom Webserver anfordern.
- Du kennst ein verfahren, wie man ein HTML-Dokument nach bestimmten Informationen durchsuchen kann.
- Du kannst den Datentyp Dictionary beschreiben und weisst, in welchen Fällen er sich besonders gut eignet.
- Du kann programmgesteuert eine Suchabfrage bei Google durchführen.

6.1 HTML, STRINGS

EINFÜHRUNG

HTML (Hyper Text Markup Language) ist eine Dokumentbeschreibungssprache für Webseiten. Eine im Browser dargestellte Webseite, die dir noch so kompliziert erscheint, wird aus einer gewöhnlichen Textdatei erzeugt, die für das Layout neben dem sichtbaren Text zusätzlich Auszeichnungen (**Markups**) enthält. Diese bestehen aus einem **Tag**-Paar mit einem Start- und Endtag. Das Starttag beginnt mit der Spitzklammer < und wird mit der Spitzklammer > geschlossen; das Endtag beginnt mit </ und wird ebenfalls mit > geschlossen.

Das Grundgerüst einer HTML-Textdatei besteht aus den Tags <*html*> und <*body*> sowie den entsprechenden Endtags.

```
<html>
   <body>
      TigerJython Web-Site
   </body>
</html>
```

Die Gross-Kleinschreibung von Tags, sowie Zeilenumbrüche und Einrückungen spielen für das Layout des Dokuments keine Rolle.

PROGRAMMIERKONZEPTE: *HTML, Hyperlink, String, Unveränderlicher Datentyp*

WAS SIND STRINGS?

In vielen Programmen, so auch im Zusammenhang mit dem Web, benötigst du einen Datentyp, der Text abspeichern kann. Darunter versteht man eine Aneinanderreihung von Buchstaben (eine **Zeichenkette**), so wie du sie mit der Tastatur eingeben kannst. Zusätzlich brauchst du einige Steuerzeichen, beispielsweise um einen Zeilenumbruch zu kennzeichnen. In Python verwendest du für Zeichenketten den Datentyp *String*.

Der Text eines Strings wird zwischen doppelten oder einfachen Anführungszeichen gesetzt. Du kannst Strings wie eine Liste auffassen, deren Elemente einzelne Zeichen sind. Die meisten dir bekannten Operationen für Listen sind daher auch für Strings anwendbar, mit einem wichtigen Unterschied: Ein einzelnes Zeichen kannst du mit einem Index aus dem String herausholen (eckiges Klammerpaar), du kannst das Zeichen aber nicht mit einer Zuweisung verändern, da der String ein **unveränderlicher Datentyp** ist. Willst du einen String verändern, so musst du einen neuen String erstellen [**mehr...**].

Dein Programm definiert *HTML*-formatierten Text als String *html* und schreibt ihn auf der Konsole aus.

```
html = "<html><body>TigerJython Web Site</body></html>"
print html
```

Um einen String zeichenweise zu durchlaufen, verwendest du eine for-Schleife mit einem **Index**:

```
html = "<html><body>TigerJython Web Site</body></html>"

for i in range(len(html)):
   print html[i]
```

Eleganter ist es aber, eine for-Schleife mit dem Schlüsselwort **in** zu verwenden:

```
html = "<html><body>TigerJython Web Site</body></html>"
for c in html:
   print c
```

Ein String kann auch spezielle Steuerzeichen enthalten. Diese **Escape-Character** werden mit einem **Rückwärtsbruchstrich** (Backslash) eingeleitet, beispielsweise lautet das Zeichen für eine neue Zeile \n (newline , auch Linefeed <lf> genannt). Du erstellst beispielsweise die ganz am Anfang des Kapitels gezeigte Formatierung mit:

```
html = "<html>\n   <body>\n      TigerJython Web Site\n   </body>\n</html>"
print html
```

Texte kannst du auch aus einer Textdatei einlesen. Erstelle dazu mit irgend einem Texteditor im gleichen Verzeichnis, in dem sich *dein Programm* befindet, die Datei *welcome.html* mit folgendem Inhalt:

```
<html>
   <body>
      <h1>TigerJython Web-Site</h1>
      Guten Tag
   </body>
</html>
```

Mit dem Tag <h1> zeichnest du eine Überschrift aus. Dein Programm liest diese Textdatei in den String html und schreibt ihn wieder in die Konsole aus.

```
html = open("welcome.html").read()
print html
```

MEMO

Ein String ist ein unveränderliches Objekt bestehend aus einzelnen Zeichen. Du kannst einzelne Zeichen mit einem Index lesen. Wenn du aber mit einer Zuweisung versuchst, ein Zeichen zu ersetzen, ergibt sich eine Fehlermeldung.

Textdateien werden mit *open()* geöffnet. Dabei übergibst du den Pfad zur Datei. Dieser kann relativ zum Verzeichnis sein, in dem sich *dein Programm* befindet, aber auch absolut, wenn du einen Bruchstrich (unter Windows eventuell auch noch einen Laufwerkbuchstaben) vorstellst, also beispielsweise

open("test/welcome.html")	welcome.txt im Unterverzeichnis test des Verzeichnises in dem sich dein Programm befindet
open("/myweb/test/welcome.html")	welcome.txt im Verzeichnis /myweb/test des Laufwerks auf dem sich dein Programm befindet
open("d:/myweb/test/welcome.html")	(nur für Windows) welcome.txt im Verzeichnis \myweb\test des Laufwerks d:

Man kann zwei Strings mit dem Additionsoperator + aneinander fügen (konkatenieren). Wichtig ist aber, dass beide Operanden wirklich Strings sind. Beispielsweise führt "pi = " + 3.1459 zu einer Fehlermeldung. Du musst dafür "pi = " + str(3.14159) schreiben, also die Zahl zuerst mit *str()* in einen String konvertieren.

Die wichtigsten Operationen mit Strings

s = "Python"	String definieren (oder auch s = 'Python')
s[i]	Auf Stringzeichen mit Index i zugreifen
s[start:end]	Neuer Teilstring mit Zeichen start bis end, aber ohne end
s[start:]	Neuer Teilstring mit Zeichen von start an
s[:end]	Neuer Teilstring von ersten Zeichen bis end, aber ohne end
s.index(x)	Index des ersten Vorkommens von x (-1: nicht gefunden)
s.find(x)	Index des ersten Vorkommens von x (-1: nicht gefunden)
s.count(x)	Gibt die Anzahl der Auftreten von x zuück
if x in s	Gibt True zurück, falls x in s enthalten ist
if x not in s	Gibt True zurück, falls x nicht in s enthalten ist
s1 + s2	Konkatenation von s1 und s2 als neuer String
s1 += s2	Ersetzt s1 durch die Konkatenation von s1 und s2
s * 4	Neuer String mit Zeichen von s viermal wiederholt
len(s)	Gibt die Anzahl Zeichen zurück

WEB-BROWSER

Die wichtigste Aufgabe eines **Web-Browsers** ist es, die HTML-Tags zu interpretieren und die Seite gemäss den Layoutangaben in einem Bildschirmfenster darzustellen. Du kannst die Datei *welcome.html* mit einem auf deinem PC installierten Web-Browser (Firefox, Explorer, Chrome, Safari, Opera, usw.) anzeigen.

TigerJython stellt dir ein einfaches Browser-Fenster als Instanz der Klasse *HtmlPane* zur Verfügung. Die Methode **insertText()** bewirkt, dass der übergebene String als Webseite im Fenster erscheint.


```
from ch.aplu.util import HtmlPane

html = open("welcome.html").read()
pane = HtmlPane()
pane.insertText(html)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Ein Web-Browser interpretiert die HTML-Markups und stellt das Dokument gemäss den Layoutangaben dar.

HtmlPane kennt nur die grundlegenden HTML-Tags. Die Anzeige von komplexen HTML-Seiten wird nicht unterstützt. Du kannst ein *HtmlPane* auch dazu gebrauchen, deine Programmausgaben in einem eigenen Fenster mit einem gefälligen Layout anzuzeigen, statt in der Konsole auszuschreiben.

HYPERLINKS

Die explosionsartige Verbreitung des Web ist im Wesentlichen darauf zurück zu führen, dass eine Webseite Elemente enthalten kann, die bei einem Mausklick zu einer Webseite führt, die auf irgendeinem anderen, sogar weit entfernten Webserver liegen kann. Elemente dieser Art werden *Hyperlinks* genannt. Mit Hyperlinks lässt sich eine vernetzte Informationsstruktur, ähnlich einem Spinnennetz, aufbauen.

Du erstellst wieder mit einem Texteditor die Datei *welcomex.html*, die ein Link-Tag <a> enthält. Neu ist auch noch das *Paragraph-Tag* <p> enthalten, das neue Abschnitte mit einem Zeilenumbruch definiert.

```
<html>
   <body>
      <h1>TigerJython Web-Site</h1>
      <p>Guten Tag!</p>
      <a href="http://www.tigerjython.ch/">TigerJython Home</a>
   </body></html>
```

Du musst in deinem Programm die Hyperlinks aktivieren, indem du eine Funktion **linkCallback()** definierst und sie mit dem benannten Parameter *linkListener* registrierst. Klickst du auf den Link, so führt dieser Event zum Aufruf des Callbacks, wobei die im Link-Tag enthaltene URL übergeben wird.

```
from ch.aplu.util import HtmlPane

def linkCallback(url):
    pane.insertUrl(url)

html = open("welcomex.html").read()
pane = HtmlPane(linkListener = linkCallback)
pane.insertText(html)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Hyperlinks sind Querverweise in einem Dokument, mit dem zu anderen Dokumenten gesprungen werden kann. Verlinkte Dokumente sind ein charakteristisches Merkmal des World Wide Web.

Die Darstellung von Webseiten mit *HtmlPane* ist leider unvollständig. Du kannst aber mit *HtmlPane.browse()* den Standard-Browser verwenden [**mehr...**].

```
from ch.aplu.util import HtmlPane
HtmlPane.browse("www.tigerjython.com")
```

AUFGABEN

1. Mit dem Tag

```
<img src="gifs/tigerlogo.png" width="120" height="116"></img>
```

kannst du ein Bild einbinden, das sich im Unterverzeichnis *gifs* des Verzeichnisses mit der HTML-Datei befindet. Die Werte von *width* und *height* sollten der Bildgrösse in Pixel entsprechen.

Erstelle eine Datei *showlogo.html* und ein Programm, das in einer *HtmlPane* Folgendes zeigt:

Das Bild *tigerlogo.png* kannst du von **hier** herunterladen.

2. Definiere die Strings *name, vorname, strasse* und *ort*, sowie die Zahlen *hausnummer* und *postleitzahl* z. B. mit deinen persönlichen Angaben. Konkateniere mit dem + Zeichen diese Strings in einen einzigen String *anschrift* derart, dass *print(anschrift)* die Angaben formatiert ausschreibt:

Vorname Name
Strasse Hausnummer
Postleitzahl Ort

3. Die gleichen Angaben wie in der Aufgabe 2 sollen in einer *HtmlPane* erscheinen, mit Postleitzahl und Ort in Fettschrift.

Schreibe einen entsprechenden html-formatierten String und übergebe ihn mit *insertText()* der *HtmlPane*.

Anmerkung: Das Tag <*br*> bewirkt einen Zeilenumbruch, das Tag <*b*> bewirkt Fettschrift.

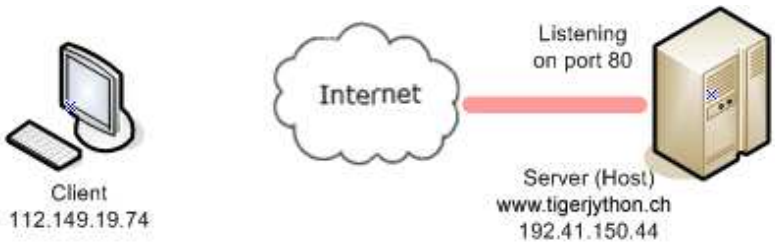
6.2 CLIENT-SERVER-MODELL, HTTP

EINFÜHRUNG

Du weisst bereits, dass eine Webseite, die in deinem Web-Browser angezeigt wird, durch eine gewöhnliche HTML-Textdatei beschrieben ist, die sich in der Regel auf einem Internet-Server (auch **Host** genannt) befindet. Um die Datei zu lokalisieren, verwendet der Browser die URL in der Form *http://servername/dateipfad*.

http steht hier für Hypertext Transfer Protocol und bezeichnet das Verfahren, wie die Datei vom **Server** zu deinem Browser-PC, auch **Client** genannt, transferiert wird. Der Servername, auch IP-Adresse (IP: Internet Protocol) genannt, ist entweder in der "gepunkteten" Form, z.B. 192.41.150.141, oder als Alias, z.B. www.tigerjython.com. Der Dateipfad der HTML-Datei beginnt mit einem Bruchstrich, ist aber auf dem Server relativ zu einem bestimmten Dokumentenpfad.

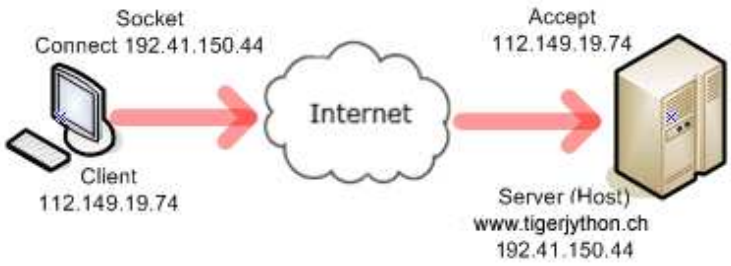
Bei der Kommunikation zwischen dem Client und dem Server wir das **Request-Respond**-Verfahren eingesetzt, eines der wichtigsten Prinzipien der Rechnerkommunikation. Dabei wird davon ausgegangen, dass auf dem Server ein Serverprogramm gestartet ist, das auf einem bestimmten TCP-Port (für das Web Port 80) auf einen Client-Request wartet.



Der Datenaustausch besteht aus folgenden 4 Phasen:

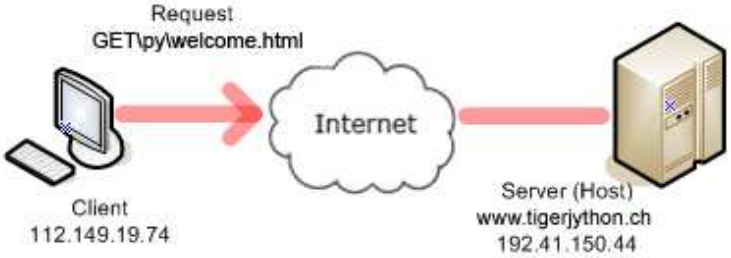
Phase 1:

Der Client erstellt ein Socket-Objekt und verbindet sich mit dem Server. Der Server akzeptiert die Verbindung und merkt sich die Client-Adresse



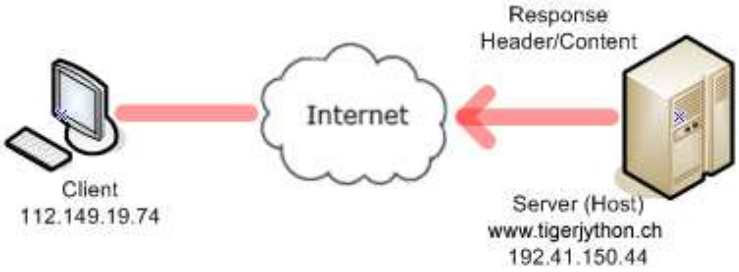
Phase 2:

Der Client sendet einen Request an den Server und übergibt ihm dabei den Pfad auf die gewünschte Datei



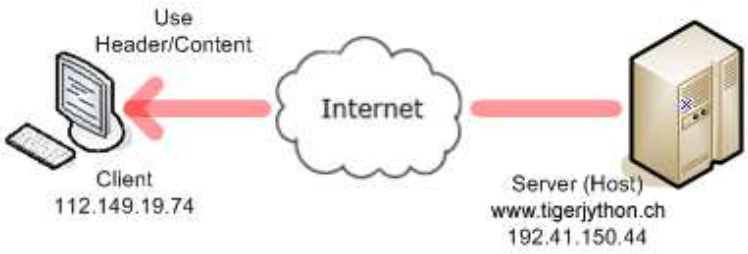
Phase 3:

Der Server verarbeitet den Request und sendet dem Client im Response die Datei



Phase 4:

Der Client empfängt den Response und verarbeitet ihn (Anzeige im Browser-Fenster)



PROGRAMMIERKONZEPTE: *Host, Client, IP-Adresse, Request-Respond-Modell, HTTP, Parsen*

MIT HTTP EINE WEBSEITE ANFORDERN

Dein Client-Programm führt die Phasen 1, 2, und 4 durch und holt dabei die Datei *welcomex.html*, die sich im Unterverzeichnis *py* des Server-Dokumentenpfads befindet.

Die Methode **socket()** der socket-Klasse, liefert in der Variablen *s* ein Socket-Objekt. Dabei müssen zwei Konstanten übergeben werden, die den richtigen Socket-Typ festlegen.

```
import socket
import sys

host = "www.tigerjython.ch"
port = 80
remote_ip = socket.gethostbyname(host)

# Phase 1
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((remote_ip , port))
print "Socket Connected to " + host + " on ip " + remote_ip

# Phase 2
request = "GET /py/welcome.html HTTP/1.1\r\nHost: " + host + "\r\n\r\n"
s.sendall(request)

# Phase 4
reply = s.recv(4096)
print "\nReply:\n"
print reply
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Der Request, um eine Webseite von einem Webserver anzufordern, verwendet das HTTP-Protokoll. Dies ist eine Vereinbarung zwischen Client und Server und legt in allen Einzelheiten den Ablauf des Datentransfers fest. Der GET-Befehl ist im HTTP wie folgt dokumentiert [**mehr...**]:

1. Zeile	GET /py/welcomeex.html HTTP/1.1\r\n	Befehl zum Holen der Datei im Verzeichnispfad des Servers, Version des Protokolls <carriage return><line feed> (Zeilenumbruch)
2. Zeile	Host: hostname\r\n	Name des Hosts <carriage return><line feed>
3. Zeile	\r\n	Leerzeile als Markierung für das Ende des Befehls

HTTP-HEADER UND CONNECT

Der Response besteht aus einem Kopf (Header) mit Statusangaben und dem Inhalt (Content) mit der angeforderten Datei. Um die Webseite darzustellen, schneidest du den Kopf weg und übergibst den Inhalt einer *HtmlPane*.

```
import socket
import sys
from ch.aplu.util import HtmlPane

host = "www.tigerjython.ch"
port = 80
remote_ip = socket.gethostbyname(host)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((remote_ip , port))
request = "GET /py/welcome.html HTTP/1.1\r\nHost: " + host + "\r\n\r\n"
s.sendall(request)
reply = s.recv(4096)

index = reply.find("<html")
html = reply[index:]

pane = HtmlPane()
pane.insertText(html)
```

MEMO

Um den Kopf wegzuschneiden, verwendest du die String-Methode **find(str)**, welche im String nach dem übergebenen Teilstring *str* sucht und den Index des ersten Vorkommens zurückgibt. (Falls der Teilstring nicht vorkommt, wird -1 zurückgegeben.) Nachher kannst du mit einer Slice-Operation elegant den Teilstring, der beim Index beginnt und bis zum Ende geht, herausfiltern.

LESEN DER WETTERPROGNOSEN

Du wirst dich fragen, warum man ein so kompliziertes Verfahren anwenden soll, um eine Webseite anzuzeigen, wenn man dasselbe doch mit einer einzigen Zeile *insertUrl()* von *HtmlPane* erledigen kann. Was du eben gelernt hast, macht aber durchaus Sinn, wenn du beispielsweise den Inhalt der Webseite gar nicht in einem Browserfenster darstellen möchtest, sondern du dich lediglich für gewisse darin eingebettete Informationen interessierst. Als sinnvolle Anwendung holt dein Programm aus der Webseite von Meteo-Schweiz die aktuelle Wetterprognose und schreibt sie als Text aus.

Du kannst dir das Programmierer-Leben doch etwas einfacher machen, wenn du statt der socket-Klasse die Bibliothek *urllib2* heranziehst, um die Datei vom Webserver zu holen. [**mehr...**].

Um herauszufinden, wo sich die gewünschte Information befindet, stellst du in einem Analyseprogramm die Seite als Text im Konsole-Fenster und gleichzeitig als Webseite in deinem Standard-Browser dar.

```
import urllib2
from ch.aplu.util import HtmlPane

url = "http://www.meteoschweiz.admin.ch/web/de/wetter/detailprognose.html"
HtmlPane.browse(url)
html = urllib2.urlopen(url).read()
print html
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Verwendung von Software-Bibliotheken wie *urllib2* vereinfacht den Programmcode, verdeckt aber grundlegende Mechanismen.

PARSEN EINES TEXTS

Du hast jetzt die interessante und anspruchsvolle Aufgabe, die relevante Information aus einem langen Textstring herauszuholen, man spricht auch vom **Parsen** eines Texts.

Als erstens stellst du dir die Teilaufgabe, mit der Funktion **remove_html_tags()** alle HTML-Tags aus dem String zu entfernen. Das Vorgehen ist typisch und der dabei angewendete Algorithmus kann wie folgt beschrieben werden:

Du durchläufst den Text Zeichen-um-Zeichen. Dabei stellst du dir vor, dass du dir zwei **Zustände** merkst: Du bist ausserhalb oder innerhalb eines HTML-Tags. Nur wenn du ausserhalb eines HTML-Tags bist, kopierst du das Zeichen ans Ende eines Ausgabestrings. Der Zustandswechsel erfolgt beim Lesen der Tag-Spitzklammern < bzw. > [**mehr...**].

Für das Entfernen oder Umwandeln von festen Textteilen, verwendest du am besten die Methode **replace(oldstr, newstr)**, welche alle Vorkommen von *oldstr* durch *newstr* ersetzt.

```
import urllib2

def remove_html_tags(s):
    inTag = False
    out = ""

    for c in s:
        if c == '<':
            inTag = True
        elif c == '>':
          inTag = False
        elif not inTag:
            out = out + c
    return out

def remove_umlaute(s):
    s = s.replace("&auml;", "ä")
    s = s.replace("&ouml;", "ö")
    s = s.replace("&uuml;", "ü")
    s = s.replace("&Auml;", "Ä")
    s = s.replace("&Ouml;", "Ö")
    s = s.replace("&Uuml;", "Ü")
    return s

def remove_nbsp(s):
```

```
    s = s.replace(" ", " ")
    return s

def remove_blank_lines(s):
    s = s.replace("\n     \n", "\n")
    return s

url = "http://www.meteoschweiz.admin.ch/web/de/wetter/detailprognose.html"
html = urllib2.urlopen(url).read()
html = remove_html_tags(html)
html = remove_umlaute(html)
html = remove_nbsp(html)

start = html.find("Wetterprognose für die")
end = html.find("Trend mit")
html = html[start:end]
html = remove_blank_lines(html)

print html
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Beim Parsen von Texten wird dieser gewöhnlich zeichenweise durchlaufen. In vielen Fällen helfen aber auch Methoden der String-Klasse [**mehr...**].

In HTML sind Umlaute und gewisse Spezialzeichen durch eine Sequenz, die mit & eingeleitet wird, gemäss folgender Tabelle codiert.

Übersicht über die wichtigsten Sonderzeichen

Befehl	Zeichen	Erklärung
<	<	Kleiner-Zeichen (engl. **lower tag**); leitet Tags ein
>	>	Größer-Zeichen (engl. **greater tag**); beendet Tags
ä	ä	Steht für **a-Umlaut**
Ä	Ä	Steht für **A-Umlaut**
ö	ö	Steht für **o-Umlaut**
Ö	Ö	Steht für **O-Umlaut**
ü	ü	Steht für **u-Umlaut**
Ü	Ü	Steht für **U-Umlaut**
ß	ß	Steht für **sz-Ligatur**
&	&	Steht für **Amp**ersand; wird benötigt, um **&** darzustellen
	(Leerzeichen)	Geschütztes Leerzeichen; damit lassen sich Leerräume erzwingen!

SPRACHSYNTHESE DER WETTERPROGNOSEN

Mit deinen Kenntnissen aus dem Kapitel *Sound*, kannst du den Text der Wetterprognose mit ein paar zusätzlichen Zeilen durch eine synthetische Stimme vorlesen lassen. Du fügst einfach im Programm die folgenden Zeilen an:

```
from soundsystem import *

initTTS()
selectVoice("german-man")
sound = generateVoice(html)
openSoundPlayer(sound)
play()
```

AUFGABEN

1. Unter der URL *http://www.timeanddate.com* werden verschiedene interessante Informationen angeboten, die man mit einem selbstgeschriebenen Programm extrahieren und weiterverwenden kann. Du kannst beispielsweise Wetterinformationen von einer Stadt irgendwo auf der Welt erhalten. Schau dir mit einem Webbrowser beispielsweise die Website *http://www.timeanddate.com/weather/canada/halifax* an.

   Dein Programm soll die aktuelle Temperatur in einer frei wählbaren Stadt ausschreiben.

   Vorgehen: Du schreibst zuerst den ganzen Text aus, den du mit der oben angegebenen URL nach dem Entfernen der HTML-Tags kriegst. Du findest leicht, wo sich die Temperaturangabe befindet. Extrahiere diese mit geeigneten Stringmethoden und schreibe in der Konsole aus: "Die Temperatur in canada/halifax ist .. Grad". Füge jetzt noch einen Eingabedialog hinzu, um den Ort interaktiv wählen zu können.

2*. Die Methode *urllib2.urlopen(url)* wirft eine Exception, falls die url nicht gefunden wird. Setzt man den Aufruf in einen try-except-Block

```
try:
   urllib2.urlopen(url)
except:
   print "Fehler"
```

   so verzweigt das Programm beim Fehler in den except-Block.

   Ergänze das Programm in Aufgabe 1 mit einer sinnvollen Fehlerausgabe, wenn die Stadt nicht gefunden wird.

6.3 GOOGLESEARCH, DICTIONARY

EINFÜHRUNG

Es ist möglich, bekannte Suchmachinen wie Google oder Yahoo zu verwenden, um programmgesteuert eine Websuche durchzuführen. Dazu musst du eine bestimmte URL des Anbieters mit zusätzlichen Parametern ergänzen, die den Suchstring enthalten, und damit einen HTTP-GET-Request durchführen. Diese Angaben werden von einer **Webapplikation**, d.h. einem Programm, dass auf dem Webserver läuft, ausgewertet und als HTTP-Response an dich zurückgegeben [**mehr...**].

Über die URL *http://ajax.googleapis.com/ajax/services/search/web* liefert Google 4 Links mit dem besten Google-Ranking zurück. Dazu müssen nach einem Fragezeichen ? zwei Parameter v und q wie folgt angefügt werden (hier für den Suchstring *tigerjython*) v=1.0&q=tigerjython. Mehrere Suchbegriffe können mit dem + Zeichen verbunden werden. Der Google-Server liefert das Resultat in einer speziellen Formatierung, nämlich der JavaScript Object Notation (**JSON**), zurück. Mit der Bibliotheksklasse *json* kann daraus ein Dictionary erstellt werden. Um daraus bestimmte Informationen zu extrahieren, musst du daher zuerst verstehen, was man in Python unter einem Dictionary versteht.

PROGRAMMIERKONZEPTE: *Webapplikation, Python-Dictionary*

EIN DICTIONARY VERSTEHEN

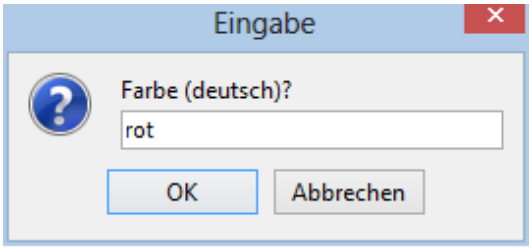
Wie der Name vermuten lässt, handelt es sich bei einem Dictionary um eine Datenstruktur ähnlich einem Wörterbuch. Dabei steht links das Wort in der dir bekannten Sprache und daneben das Wort in der Fremdsprache (wir sehen von Vieldeutigkeiten ab), beispielsweise für einige Farbnamen von deutsch nach englisch:

Deutsch	**Englisch**
blau	blue
rot	red
grün	green
gelb	yellow

(In einem Wörterbuch sind die Wörter alphabetisch geordnet, damit das Auffinden vereinfacht wird.)

Man nennt das linke Wort den **Schlüssel (key)** und das rechte Wort den **Wert (value)**. Ein Dictionary besteht also aus **Schlüssel-Wert-Paaren (key-value-pairs)** [**mehr...**]. Sowohl Schlüssel wie Werte können beliebige Datentypen haben [**mehr...**].

Dein Programm übersetzt die oben angegebenen Farben von deutsch auf englisch. Wenn die Eingabe nicht im Wörterbuch ist, wird der Fehler abgefangen und eine Fehlermeldung ausgeschrieben.



```
dict = {"blau":"blue", "rot":"red", "grün":"green", "gelb":"yellow"}

print "All entries:"
for key in dict:
    print key + " -> " + dict[key]

while True:
    farbe = input("Farbe (deutsch)?")
    if farbe in dict:
        print farbe + " -> " + dict[farbe]
    else:
        print farbe + " -> " + "(nicht übersetzbar)"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Ein Dictionary besteht aus Schlüssel-Werte-Paaren (key-value-pairs). Bei der Definition verwendet man ein geschweiftes Klammerpaar und trennt die Paare mit einem Komma, sowie Schlüssel und Wert mit einem Doppelpunkt (Leerschläge spielen für die Formatierung keine Rolle).

Wichtige Operationen:

dict[key]	Liefert den Wert zum Schlüssel key
dict[key] = value	Fügt ein neues Schlüssel-Wert-Paar hinzu oder modifiziert Wert
len(dict)	Liefert die Anzahl Schlüssel-Wert-Paare
del dict(key)	Löscht das Paar (Schlüssel und Wert) mit dem Schlüssel key
key in dict	Liefert True, falls der Schlüssel key vorhanden ist
dist.clear()	Löscht alle Einträge, es bleibt eine leeres Dictionary

Ein Dictionary kann man mit einer for-Schleife

```
for key in dict:
```

durchlaufen (iterieren).

DICTIONARIES SIND EFFIZIENTE DATENSTRUKTUREN

Du hast richtig gedacht, wenn du einwendest, dass man gepaarte Informationen auch in einer Liste abspeichern könnte. Naheliegend wäre es, die Paare als Teillisten in einer umfassenden Liste zu speichern. Wozu gibt es Dictionary als eigene Datenstruktur?

Der grosse Vorteil von Dictionaries besteht darin, dass du bei Vorgabe des Schlüssels mit der Klammernotation einfach und schnell auf seinen Wert zugreifen oder anders gesagt, dass du Dictionaries effizient durchsuchen kannst. Das effiziente Auffinden von Information spielt natürlich nur bei grossen Datenmengen eine Rolle, wenn es sich also um hunderte, ja tausende von Schlüssel-Werte-Paaren handelt.

Als interessante und nützliche Anwendung soll dein Programm die Postleitzahl irgend einer Stadt in der Schweiz finden. Dazu verwendest du eine Textdatei ***chplz.txt***, die du von hier herunterladen kannst. Du kopierst sie in das Verzeichnis, in dem sich dein Programm befindet. Die Datei ist zeilenweise wie folgt strukturiert (und besitzt keine Leerzeile, auch nicht am Ende):

Aarau:5000
Aarburg:4663
Aarwangen:4912
Aathal Seegraeben:8607
...

Deine erste Aufgabe besteht darin, diese Textdatei in ein Dictionary zu verwandeln. Dazu liest du sie zuerst mit **read()** als String ein und spaltest sie mit split("\n") in einzelne Zeilen auf [**mehr...**].

Für die Erstellung des Dictionary trennst du die Zeilen nochmals mit dem Doppeltpunkt als Trennzeichen in Schlüssel und Wert und fügst das neue Paar mit der Klammernotation in das (zuerst leere) Dictionary ein. Wie vorhin mit den Farben kannst du nun mit der Klammernotation auf eine Postleitzahl zugreifen.

```
file = open("chplz.txt")
plzStr = file.read()
file.close()

pairs = plzStr.split("\n")
print str(len(pairs)) + " pairs loaded"
plz = {}

for pair in pairs:
    element = pair.split(":")
    plz[element[0]] = element[1]

while True:
    town = input("Stadt?")
    if town in plz:
        print "Die Postleitzahl von " + town + " ist " + str(plz[town])
    else:
        print "Habe die Stadt " + town + " nicht gefunden."
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Mit einem Dictionary kann man sehr einfach und schnell auf einen Schlüsselwert zugreifen [**mehr...**].

GOOGLE FÜR EIGENE ZWECKE BRAUCHEN

Dein Programm verwendet die Suchmaschine Google, um mit einem vom Benützer eingegebenen Suchstring nach Websites zu suchen und die von Google gelieferten Informationen auszuschreiben. Dazu sendest du eine GET-Request mit der oben angegebenen URL, die mit dem Suchstring ergänzt ist. Der Response von Google ist ein String, in dem die Informationen mit geschweiften Klammern strukturiert sind. Die Formatierung entspricht der JavaScript Object Notation (JSON). Mit der Methode **json.load()** kann er in ein ineinander geschachteltes Python-Dictionary umgewandelt werden, das sich effizienter parsen lässt. Du kannst die Verschachtelung analysieren, indem du in einer Testphase geeignete Informationen in der Konsole ausschreibst. Später kannst du diese Zeilen wieder auskommentieren oder entfernen. Was findet Google mit dem Suchstring "tigerjython"?

```
import urllib2, json

search = input("Gib Suchstring ein (UND-Verbund mit +):")
url = "http://ajax.googleapis.com/ajax/services/search/web?v=1.0&q="
         + search
responseStr = urllib2.urlopen(url).read()
response = json.loads(responseStr)

#print "response:\n" + str(response) + "\n"

responseData = response["responseData"]
#print "reponseData:\n" + str(responseData) + "\n"

results = responseData["results"]
#print "results:\n" + str(results) + "\n"

for result in results:
    title = result["title"]
    url = result["url"]
    print title + " ---- " + url
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Wie du siehst, kann ein Dictionary als Werte wiederum Dictionaries enthalten. Damit lassen sich hierarchische Informationsstrukturen, ähnlich wie XML, erstellen.

AUFGABEN

1. Verbessere das Postleitzahl-Programm schrittweise so, dass eine Stadt auch dann gefunden wird, wenn man
   a. Leerschläge vor oder nach dem Städtenamen eingibt
   b. sich nicht konsequent an die Gross-Kleinschreibung hält
   c. Umlaute als Ae, Oe, Ue, ae, oe, ue schreibt
   d. Accents weglässt (Achtung: Es gibt einen Konflikt bei ö)
   e. Einige Orte sind mehrdeutig, haben aber eine Zusastzangabe. Wie willst du damit umgeben?
2. Verwende den Google-Search, um vom Suchresultat mit dem höchsten Ranking den Titel und den Inhalt in einer HtmlPane auszuschreiben. Beispielsweise sollte mit dem Suchstring "tigerjython" ungefähr Folgendes erscheinen:

> **Programmieren: TigerJython(Python)**
>
> TigerJython ist eine Programmierumgebung für Python (aktuell 2.7) Sie beruht..... Wesentlich an TigerJython ist, das es ohne Installation auf verschiedenen....

kapitel sieben

GAMES & OOP

Lernziele

- Du weisst, wie man in Python eine Klasse bestehend aus Konstruktor, Instanzvariablen und Methoden definiert und verwendet.
- Du weisst, was man unter einer Klassenhierarchie versteht und kannst abgeleitete Klassen definieren und verwenden.
- Du kannst in einfachen Worten erklären, was man unter Polymorphismus versteht.
- Du kennt den grundlegenden Design der Gamelibrary JGameGrid kannst damit ein einfaches Computergame selbst programmieren.

"Computerspiele sind heute aus der digitalen Medienwelt nicht mehr wegzudenken. Ihr hoher Stellenwert bei den Jugendlichen motiviert viele Pädagogen dazu, die didaktische Verwertbarkeit dieses Mediums zu untersuchen. Game-Based Learning (GBL) ist Gegenstand vieler wissenschaftlicher Studien und gehört heute zum Schulalltag. Im Informatikunterricht können Computerspiele aus dem Blickwinkel des Herstellers angegangen werden. Als hoch dynamische Programme fördern sie bereits beim Anfänger das Denken in Abläufen und eignen sich durch die bewegten Grafikobjekte (Sprites) hervorragend für die Einführung in das objektorientierte Programmieren."

Jarka Arnold, Aegidius Plüss
in "Spiele als Einstieg in das objektorientierte Programmieren"

7. 1 OBJEKTE ÜBERALL

EINFÜHRUNG

Im täglichen Leben bist du von einer Vielzahl von verschiedenen Objekten umgeben. Da Software oft die Wirklichkeit modellmässig abbildet, ist es naheliegend, auch in der Informatik Objekte einzuführen. Man spricht dann von **Objektorientierter Programmierung (OOP)**. Das Konzept der OOP hat sich seit zwei Jahrzehnten in der Informatik als derart wegweisend erwiesen, dass es praktisch in allen heute entwickelten Softwaresystemen angewendet wird [**mehr...**]. Im Folgenden lernst du die wichtigsten Konzepte der OOP kennen, damit du an diesem Hype teilhaben kannst.

Du hast bereits die Turtle als Objekt kennen gelernt. Eine Turtle besitzt **Eigenschaften** (sie hat eine bestimmte Farbe, befindet sich an einer gewissen Position und hat eine bestimmte Blickrichtung) und **Fähigkeiten** (sie kann sich vorwärts bewegen, sich drehen, usw.). In der OOP werden Objekte mit ähnlichen Eigenschaften und Fähigkeiten in **Klassen** eingeteilt. Die Turtleobjekte gehören zur *Klasse Turtle*, man sagt auch, sind **Instanzen** der *Klasse Turtle*. Um ein Objekt zu erzeugen, muss man zuerst eine **Klasse definieren** oder wie bei Turtles eine bereits vordefinierte Klasse verwenden.

Beim Programmieren nennt man die Eigenschaften auch **Attribute** oder **Instanzvariablen**, die Fähigkeiten auch **Operationen oder Methoden**. Es handelt sich um Variablen und Funktionen, ausser dass sie einer bestimmten Klasse angehören und damit in der Klasse "gekapselt" sind. Um sie ausserhalb der Klasse zu verwenden, muss man eine Klasseninstanz und den Punktoperator voranstellen.

PROGRAMMIERKONZEPTE: *Grundbegriffe: Klasse, Objekt (Instanz), Eigenschaft, Fähigkeit, Attribut/Instanzvariable, Methode, Ableitung, Basis-/Superklasse, Konstruktor*

EIN ANPASSUNGSFÄHIGES SPIELFENSTER

Ohne OOP ist es nicht möglich, mit vernünftigem Aufwand ein Computergame zu erstellen, denn die Spielfiguren und andere Gegenstände des Gameboards sind offensichtlich Objekte. Das Gameboard ist ein rechteckiges Bildschirmfenster und wird durch die **Klasse GameGrid** aus der Gamelibrary **JGameGrid** modelliert. TigerJython stellt dir eine Instanz beim Aufruf von **makeGameGrid()** zur Verfügung und mit **show()** wird das Fenster angezeigt. Dabei kannst du das Aussehen des Spielfensters über Parameterwerte deinen Wünschen anpassen. Mit

```
makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
```

wird ein quadratisches Spielfenster angezeigt, dass 10 horizontale und 10 vertikale Zellen der Grösse 60 Pixel aufweist. Du erkennst rote Gitterlinien und ein Hintergrundbild **town.jpg**. (Der letzte Parameter bewirkt, dass der Navigationsbalken unterdrückt wird, der hier nicht benötigt wird.)

```
from gamegrid import *

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
show()
```

MEMO

Die Methoden der Klasse *GameGrid* stehen dir mit *makeGameGrid()* als Funktionen zur Verfügung. Du kannst aber auch selbst eine Instanz erzeugen und die Methoden mit dem Punktoperator aufrufen.

```
from gamegrid import *

gg = GameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
gg.show()
```

Das Spielfenster ist aus quadratischen Zellen aufgebaut, wobei die Zellengrösse (60 pixel) und die Anzahl 10 horizontaler und vertikaler Zellen angegebenen werden. Damit auch die rechte und untere Gitterlinie angezeigt werden, besitzt das Fenster eine Grösse von 601 x 601 pixel. Dies entspricht der (minimalen) Grösse des Hintergrundbildes. Der letzte boolesche Parameter bestimmt, ob eine Navigationsleiste erscheint.

KLASSENDEFINITION MIT ABLEITUNG

Bei der Definition einer Klasse kannst du entscheiden, ob deine neue Klasse ganz eigenständig ist oder aus einer bereits vorhandenen Klasse **abgleitet** wird. In der abgeleiteten Klasse stehen dir alle Eigenschaften und Fähigkeiten der **Oberklasse** (auch **Basisklasse** oder **Superklasse** genannt) zur Verfügung. Anschaulich sagt man, dass die abgeleitete Klasse Eigenschaften und Fähigkeiten **erbt**.

In der Gamelibrary *JGameGrid* werden Spielfiguren **Actors** genannt und sind Instanzen der vordefinierten *Klasse Actor*. Willst du also eine eigene Spielfigur verwenden, so **definierst** du eine Klasse, die **aus Actor abgeleitet** ist.

Deine Klassendefinition beginnt mit dem Schlüsselwort **class**. Es folgt der beliebig wählbare Klassennamen und ein rundes Klammerpaar. Dort schreibst du den Namen der Klasse hin, von der du deine Klasse ableitest. Da du die Spielfigur von *Actor* ableiten willst, gibst du diesen Klassennamen an.

Die Klassendefinition enthält die Definition der **Methoden**, die wie normale Funktionen definiert werden, mit dem Unterschied, dass sie **obligatorisch** den Parameter **self** als ersten Parameter aufweisen müssen. Mit diesem Parameter kannst du auf andere Methoden und Instanzvariablen der eigenen Klasse und ihrer Basisklasse zugreifen.

Die Liste der Methodendefinitionen beginnt üblicherweise mit der Definition einer **speziellen Methode** mit dem Namen __init__ (zwei vor- und nachgestellte Underlines). Diese nennt man **Konstruktor** und sie wird automatisch dann aufgerufen, wenn ein Objekt der Klasse erzeugt wird. In unserem Fall rufst du im Konstruktor von *Alien* den Konstruktor der Basisklasse *Actor* auf, welchem du den Pfad zum Spritebild übergibst.

Als nächstes definierst du die Methode **act()**. Diese spielt für die Gameanimation eine zentrale Rolle, denn sie wird vom Gamemanager in jedem Simulationszyklus automatisch aufgerufen. Dies ist ein besonders intelligenter Trick, damit du dich nicht mit einer Wiederholstruktur selbst um die Animation kümmern musst.

In **act()** legst du fest, was die Spielfigur in jedem Simulationszyklus machen soll. Als Demonstration bewegst du sie hier lediglich mit **move()** in die nächste Zelle. Da *move()* eine Methode der Basisklasse *Actor* ist, musst du sie mit vorgestelltem *self* aufrufen.

Hast du einmal deine Klasse *Alien* definiert, so **erzeugst** du ein Alienobjekt unter Aufruf des Klassennamens und weist es einer Variablen zu. Typisch für die OOP ist es, dass du selbstverständlich beliebig viele Aliens erzeugen kannst. Wie im täglichen Leben haben diese eine eigene **Individualität**, "wissen" also durch ihre *act()*-Methode, wie sie sich bewegen müssen. Um die erzeugten Aliens ins Gameboard einzufügen, verwendest du **addActor()**, wobei du mit *Location()* die Zellenkoordinaten angeben musst (die Zelle mit den Koordinaten (0,0) ist oben links, x nimmt nach rechts, y nach unten zu). Um den Simulationszyklus zu starten, rufst du schliesslich **doRun()** auf.

```
from gamegrid import *

# ---------------- class Alien ----------------
class Alien(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/alien.png")
    def act(self):
        self.move()
makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
spin = Alien() # object creation, many instances can be created
urix = Alien()
addActor(spin, Location(2, 0), 90)
addActor(urix, Location(5, 0), 90)
show()
doRun()
```

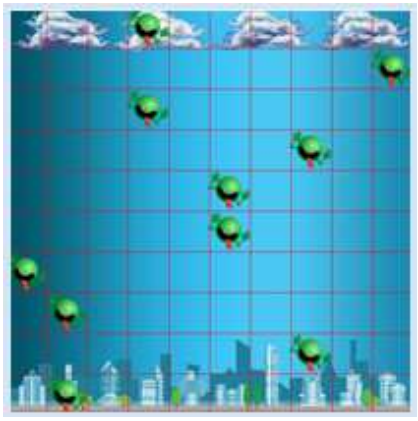
MEMO

Eine Klassendefinition beginnt mit dem Schlüsselwort **class** und **kapselt** die Methoden und Instanzvariablen der Klasse. Der Konstruktor mit dem Namen **__init__** wird bei der Erzeugung eines Objekts automatisch aufgerufen. Um ein Objekt (eine Instanz) zu erzeugen, verwendest du den Klassennamen und übergibst ihm die Parameterwerte, welche __init__ erhalten soll. Alle Spielfiguren werden aus der Klasse *Actor* abgeleitet. In der Methode **act()** definierst du, was die Spielfigur in jedem Simulationszyklus machen soll.

Mit **addActor()** fügst du eine Spielfigur in das Gameboard, wobei du seine Startposition (Location) und seine Startrichtung (in Grad) angibst (0 gegen Osten, positiv in Uhrzeigersinn).

ANGRIFF DER ALIENS

Du erkennst die Stärke und Eleganz des objektorientieren Programmierparadigmas daran, dass du das Gameboard mit wenigen Programmzeilen mit vielen vom Himmel fallenden Aliens bevölkern kannst. Dazu modellierst du im Hauptteil mit einer Wiederholschleife ein Alienraumschiff, das alle 0.2 s einen neuen Alien an zufälliger Stelle der obersten Gitterzeile aussetzt.



```
from gamegrid import *
import random

# ---------------- class Alien ----------------
class Alien(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/alien.png")

    def act(self):
        self.move()

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
show()
doRun()

while not isDisposed():
    alien = Alien()
    addActor(alien, Location(random.randint(0, 9), 0), 90)
    delay(200)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Eine Endlosschleife im Hauptteil des Programms sollte mit **isDisposed()** darauf testen, ob das Gamefenster geschlossen wurde, damit das Programm korrekt beendet wird.

Achtung: Es ist manchmal nötig, TigerJython zu schliessen und wieder zu öffnen, damit gleichnamige, aber veränderte Sprite- oder Hintergrundbilder geladen werden.

SPACEINVADER LIGHT

In deinem ersten selbst geschriebenen Computergame soll der Spieler versuchen, eine Alien-Invasion zu bekämpfen, indem er die angreifenden Aliens mit Mausklicks entfernt. Jeder in der Stadt gelandete Alien führt zu einem Minuspunkt.

Für die Mausunterstützung fügst du einen Maus-Callback mit dem Namen *pressCallback* ein und registrierst ihn als benannten Parameter *mousePressed*. Im Callback holst du dir zuerst aus dem Event-Parameter e die Zellenposition des Mausklicks. Befindet sich in dieser Zelle ein Actor, so kriegst du ihn mit **getOneActorAt()**, ist die Zelle leer, so liefert der Aufruf None. **removeActor()** entfernt den Actor aus dem Gameboard.

```
from gamegrid import *
import random

# ---------------- class Alien ----------------
class Alien(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/alien.png")

    def act(self):
        self.move()

def pressCallback(e):
    location = toLocationInGrid(e.getX(), e.getY())
    actor = getOneActorAt(location)
    if actor != None:
        removeActor(actor)
    refresh()
```

```
makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False,
             mousePressed = pressCallback)
setSimulationPeriod(800)
show()
doRun()

while not isDisposed():
    alien = Alien()
    addActor(alien, Location(random.randint(0, 9), 0), 90)
    delay(1000)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Da *act()* in jeder Simulationsperiode einmal aufgerufen wird, ist die Periodendauer für die Ablaufgeschwindigkeit des Games verantwortlich. Der Standardwert der Simulationsperiode ist 200 ms. Sie kann mit **setSimulationPeriod()** auf einen anderen Wert eingestellt werden.

Das Gameboard wird in jedem Simulationszyklus einmal neu aufgebaut (gerendert), eine Änderung der Spielsituation wird daher erst zu diesem Zeitpunkt sichtbar. Willst du bei einem Mausklick die neue Situation sofort anzeigen, so kannst du das Rendern mit **refresh()** manuell ausführen.

AUFGABEN

1. Erstelle mit einem Bildeditor ein eigenes Hintergrundbild. Füge es in das Verzeichnis sprites im Verzeichnis, in dem sich dein Programm befindet (oder in *<userhome>/gamegrid /sprites*) oder gebe den voll qualifizierten Dateipfad an.

2. Füge mit *addStatusBar(30)* eine 30 pixel hohe Statusbar an und schreibe dort mit *setStatusText()* die Anzahl Aliens aus, die trotz der Abwehr in der Stadt landen konnten.

3. Die gelandeten Aliens sollen nicht einfach verschwinden, sondern sich an der Landestelle in eine andere Form ("sprites/alien_1.gif" oder eigenes Bild) umwandeln und in der Landestelle verharren. (Anleitung: Mit *removeSelf()* kannst du einen alten Alien entfernen und mit *addActor()* einen neuen Actor an derselben Stelle erzeugen.)

4*. Die gelandeten Aliens melden melden an das Alienraumschiff, wo sie gelandet sind, so dass neue Aliens nur noch in "freien" Spalten abspringen. Sobald alle Spalten besetzt sind, ist das Spiel mit einer Anzeige von "GameOver" beendet ("sprites/gameover.gif"). (Anleitung: Der Gamemanager kann mit *doPause()* angehalten werden.)

5*. Erweitere das Game nach deinen eigenen Ideen.

7.2 KLASSEN UND OBJEKTE

EINFÜHRUNG

.

Du hast bereits Bekanntschaft mit wichtigen Konzepten der Objektorientierten Programmierung gemacht und gemerkt, dass du ohne OOP in Python kaum Computergames schreiben kannst. Es ist daher wichtig, dass du etwas systematischer die Begriffe der OOP und ihre Implementierung in Python kennen lernst [**mehr...**]

PROGRAMMIERKONZEPTE: *Vererbung, Klassenhierarchie, Überschreiben, Is-a-Relation, Mehrfachvererbung*

INSTANZVARIABLEN

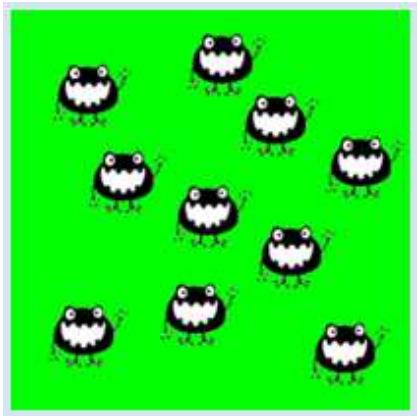
Tiere eignen sich hervorragend, also Objekte modelliert zu werden. Du definierst zuerst eine Klasse *Animal*, die das entsprechende Tierbild im Hintergrund des Gameboard darstellt. Bei der Erzeugung eines Objekts dieser Klasse übergibst du daher dem Konstruktor den Dateipfad auf das Tierbild, damit die Methode *showMe()* soll das Bild anzeigen kann. Sie verwendet dabei Zeichnungsmethoden der Klasse *GGBackground*.

Der Konstruktor, der den Dateipfad erhält, muss ihn als Initialisierungswert in einer Variablen speichern, damit alle Methoden darauf zugreifen können. Eine solche Variable ist ein Attribut oder eine Instanzvariable der Klasse. In Python erhalten Instanzvariablen den Prefix *self* und werden bei der ersten Zuweisung eines Werts erzeugt.

Wie du bereits weist, hat der Konstruktor den speziellen Namen **__init__** (mit zwei vor- und nachgestellten Underlines). Der Konstruktor sowie alle Methoden müssen **self** als ersten Parameter aufweisen, was man gerne vergisst.

Du definierst also den Konstruktor
`def __init__(self, imgPath):`
sowie eine Methode
`def showMe(self, x, y):`

Hast du einmal ein Tierobjekt myAnimal mit
`myAnimal = Animal(bildpfad)`
erzeugt, so rufst du diese Methode mit
`myAnimal.showMe(x, y)`



auf, also ohne den Parameter self. Die OOP macht vor allem dann einen Sinn, wenn du mehrere Objekte derselben Klassen verwendest. Um dies hautnahe zu erleben, soll in deinem Programm bei jedem Mausklick ein neues Tier entstehen.

```
from gamegrid import *

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath  # Instance variable
    def showMe(self, x, y):  # Method definition
         bg.drawImage(self.imagePath, x, y)
```

```
def pressCallback(e):
    myAnimal = Animal("sprites/animal.gif") # Object creation
    myAnimal.showMe(e.getX(), e.getY())  # Method call

makeGameGrid(600, 600, 1, False, mousePressed = pressCallback)
setBgColor(Color.green)
show()
doRun()
bg = getBg()
```

MEMO

Die Eigenschaften oder Attribute eines Objekts werden als Instanzvariablen definiert. Sie haben für jedes Objekt der Klasse individuelle Werte. Der Zugriff auf Instanzvariablen innerhalb der Klasse erfolgt durch Vorstellen von self.

Einer Klasse stehen aber auch die Variablen und Funktionen des Programm-Hauptteils zur Verfügung, beispielsweise alle Methoden der Klasse GameGrid und mit *bg* der Background des Spielfensters. Die Methoden können sogar eine Variable des Hauptteils verändern, falls sie in der Methode als global deklariert wird.

Wenn das Objekt keine Initialisierungen benötigt, so kann die Definition des Konstruktors auch weggelassen werden. Statt das Spritebild dem Konstruktor zu übergeben, verwendest du im folgenden Programm die Variable *imagePath* und kannst damit auf den Konstruktor verzichten.

```
from gamegrid import *
import random

# ---------------- class Animal ----------------
class Animal():
    def showMe(self, x, y):
         bg.drawImage(imagePath, x, y)

def pressCallback(e):
    myAnimal = Animal()
    myAnimal.showMe(e.getX(), e.getY())

imagePath = "sprites/animal.gif"
makeGameGrid(600, 600, 1, False, mousePressed = pressCallback)
setBgColor(Color.green)
show()
doRun()
bg = getBg()
```

VERERBUNG, METHODEN HINZUFÜGEN

Durch Klassenableitung oder Vererbung erstellst du eine Klassenhierarchie und kannst damit einer bestehenden Klasse zusätzliche Eigenschaften und Verhalten hinzufügen. Objekte der abgeleiteten Klasse sind automatisch auch Objekte der übergeordneten Klasse (auch **Ober-, Basis**- oder **Superklasse** genannt) und können daher alle Eigenschaften und Methoden der übergeordneten Klasse verwenden, als ob sie in der abgeleiteten Klasse selbst definiert wären. Beispielsweise soll ein Haustier ein Tier sein, dass zusätzlich noch einen Namen hat, den es mit *tell()* ausschreiben soll. Du definierst daher eine Klasse *Pet*, die von *Animal* abgeleitet ist.

Da du den Tiernamen für jedes Haustier individuell bei seiner Erzeugung festlegen willst,

übergibst du ihn als Initialisierungswert dem Konstruktor von *Pet*, der ihn in einer Instanzvariablen abspeichert.

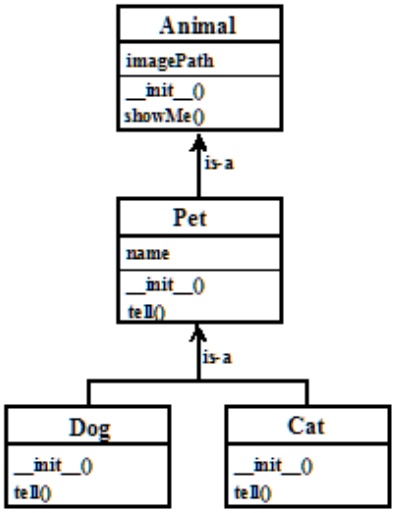
```
from gamegrid import *
from java.awt import Point

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath
    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class Pet ----------------
class Pet(Animal):   # Derived from Animal
    def __init__(self, imgPath, name):
        Animal.__init__(self, imgPath)
        self.name = name
    def tell(self, x, y): # Additional method
        bg.drawText(self.name, Point(x, y))

makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
doRun()
bg = getBg()
bg.setPaintColor(Color.black)

for i in range(5):
    myPet = Pet("sprites/pet.gif", "Trixi")
    myPet.showMe(50 + 100 * i, 100)
    myPet.tell(72 + 100 * i, 145)
```

MEMO

Wie du siehst, kannst du *myPet.showMe()* aufrufen, obschon *showMe()* in der Klasse *Pet* gar nicht definiert ist, denn ein Haustier **ist-auch-ein** Tier. Man nennt die Beziehung von *Pet* und *Animal* daher eine **is-a-Relation**. Die Basisklassen werden bei den abgeleiteten Klassen in eine Klammer hinter den Klassennamen gesetzt. In Python kann man eine Klasse auch aus mehreren Basisklassen ableiteten (**Multiple Inheritance**).

KLASSENHIERARCHIE, METHODEN ÜBERSCHREIBEN

In einer abgeleiteten Klasse können Methoden der Basisklasse auch verändert werden, indem man sie mit dem gleichen Namen und der gleichen Parameterliste neu definiert. Willst du Hunde modellieren, die bei *tell()* auch noch bellen, so leitest du die Klasse *Dog* von *Pet* ab und überschreibst die Methode *tell()*. Analog kannst du eine Katze miauen lassen, indem du eine Klasse *Cat* von *Pet* ableitest und dort ebenfalls *tell()* überschreibst.



Die vier Klassen können anschaulich in einem **Klassendiagramm** aufgezeichnet werden. Darin wird die is-a-Relation besonders deutlich [**mehr...**].

Im Klassendiagramm werden die Klassen als Rechteckbox dargestellt, in die du zuerst den Klassennamen schreibst. Mit einer horizontalen Trennungslinie getrennt folgen als nächstes die Instanzvariablen und dann, angeführt durch den Konstruktor, die Methoden der Klasse. Die Klassenhierarchie wird durch eine geschickte Anordnung und mit Verbindungspfeilen anschaulich gemacht.

```
from gamegrid import *

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath
    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class Pet ----------------
class Pet(Animal):
    def __init__(self, imgPath, name):
        Animal.__init__(self, imgPath)
        self.name = name
    def tell(self, x, y):
        bg.drawText(self.name, Point(x, y))

# ---------------- class Dog ----------------
class Dog(Pet):
    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, name)
    def tell(self, x, y): # Overriding
        bg.setPaintColor(Color.blue)
        bg.drawText(self.name + " tells 'Waoh'", Point(x, y))

# ---------------- class Cat ----------------
class Cat(Pet):
    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, name)
    def tell(self, x, y): # Overriding
        bg.setPaintColor(Color.gray)
        bg.drawText(self.name + "  tells 'Meow'", Point(x, y))

makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
doRun()
bg = getBg()

alex = Dog("sprites/dog.gif", "Alex")
alex.showMe(100, 100)
alex.tell(200, 130)  # Overriden method is called

rex = Dog("sprites/dog.gif", "Rex")
rex.showMe(100, 300)
rex.tell(200, 330)  # Overriden method is called
```

```
xara = Cat("sprites/cat.gif", "Xara")
xara.showMe(100, 500)
xara.tell(200, 530)  # Overriden method is called
```

MEMO

Durch das Überschreiben von Methoden kann man in der abgeleiteten Klassen das Verhalten der Basisklasse verändern. Beim Aufruf von Methoden derselben Klasse oder der Basisklasse muss *self* vorgestellt werden. In der Parameterliste wird *self* aber nicht übergeben.Manchmal möchte man in einer überschriebenen Methode die gleichlautende Methode der Basisklasse verwenden. Um diese aufzurufen, muss man den Klassennamen der Basisklasse voranstellen und in der Parameterliste *self* ebenfalls übergeben.Diese Regel gilt auch für den Konstruktor: Wird im Konstruktor der abgeleiteten Klasse der Konstruktor der Basisklasse verwendet, so muss dieser durch Vorstellen des Klassennamens der Basisklasse und mit dem Parameter *self* aufgerufen werden.

TYPENBEZOGENER METHODENAUFRUF: POLYMORPHISMUS

Eine etwas schwieriger zu verstehende, aber besonders wichtige Eigenschaft von objektorientierten Programmiersprachen ist der Polymorphismus Darunter versteht man den Aufruf von überschriebenen Methoden, wobei der Aufruf automatisch der Klassenzugehörigkeit angepasst wird. An einem einfachen Beispiel erlebst du, was damit gemeint ist.

Du verwendest mit den vorher definierten Klassen eine Liste *animals*

```
animals = [Dog(), Dog(), Cat()]
```

in der sich zwei Hunde und eine Katze befinden. Beim Durchlaufen der Liste und Aufruf von *tell()* mit

```
for animal in animals:
    animal.tell()
```

tritt eine Schwierigkeit auf, denn es gibt ja drei verschiedene Methoden von *tell()*, nämlich je eine in den Klassen *Pet*, *Dog* und *Cat*. Der Computer kann diese Vieldeutigkeit auf drei Arten auflösen. Er kann eine Fehlermeldung abgeben, er kann das *tell()* der Basisklasse *Pet* aufrufen oder er kann herausfinden, um welche Sorte von Pets es sich handelt und das entsprechende *tell()* aufrufen. In einer polymorphen Programmiersprache wie Python gilt das letzte und beste Verhalten.

```
from gamegrid import *
from soundsystem import *

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath
    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class Pet ----------------
class Pet(Animal):
    def __init__(self, imgPath, name):
        Animal.__init__(self, imgPath)
        self.name = name
    def tell(self, x, y):
        bg.drawText(self.name, Point(x, y))

# ---------------- class Dog ----------------
class Dog(Pet):
```

```
    def __init__(self, imgPath, name):
         Pet.__init__(self, imgPath, name)
    def tell(self, x, y): # Overridden
         Pet.tell(self, x, y)
         openSoundPlayer("wav/dog.wav")
         play()

# ---------------- class Cat ----------------
class Cat(Pet):
    def __init__(self, imgPath, name):
        Pet.__init__(self, imgPath, name)
    def tell(self, x, y): # Overridden
        Pet.tell(self, x, y)
        openSoundPlayer("wav/cat.wav")
        play()

makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
doRun()
bg = getBg()

animals = [Dog("sprites/dog.gif", "Alex"),
     Dog("sprites/dog.gif", "Rex"),
     Cat("sprites/cat.gif", "Xara")]

y = 100
for animal in animals:
    animal.showMe(100, y)
    animal.tell(200, y + 30)    # Which tell()????
    show()
    y = y + 200
    delay(1000)
```

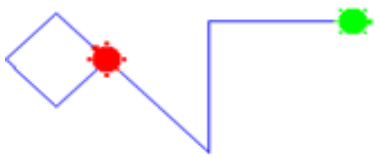
MEMO

Der Polymorphismus sorgt dafür, dass bei überschriebenen Methoden die Klassenzughörigkeit entscheidet, welche Methode aufgerufen wird. Da in Python die Zugehörigkeit zu Klassen sowieso erst zu Laufzeit festgelegt wird, ist der Polymorphismus eine Selbstverständlichkeit.Diese dynamische Datenbindung von Python nennt sich auch ***Ententest*** oder ***Duck-Typing***, gemäss dem Zitat, das James Whitcomb Riley (1849 - 1916) zugeschrieben wird:
"Wenn ich einen Vogel sehe, der wie eine Ente läuft, wie eine Ente schwimmt und wie eine Ente schnattert, dann nenne ich diesen Vogel eine Ente."
Es gibt Fälle, wo eine überschriebene Methode in der Basisklasse zwar definiert ist, aber nichts bewirken soll. Dies erreicht man entweder mit einem sofortigen ***return*** oder mit der leeren Anweisung ***pass***.

AUFGABEN

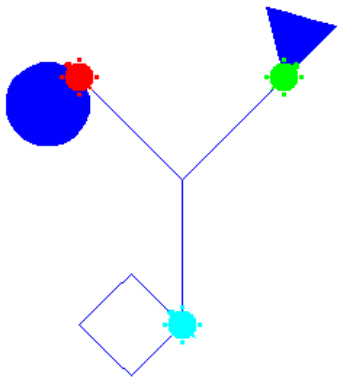
1. Definiere eine aus der Klasse *Turtle* abgeleitete Klasse *TurtleKid*, welche mit *shape()* ein Quadrat zeichnet. Der folgende Hauptteil soll funktionieren:

```
tf = TurtleFrame()
# john ist eine Turtle
john = Turtle(tf)
# john kennt alle Befehle der Turtle
john.setColor("green")
john.forward(100)
john.right(90)
john.forward(100)

# laura ist ein TurtleKid, aber auch eine Turtle
# laura kennt alle Befehle der Turtle
laura = TurtleKid(tf)
laura.setColor("red")
laura.left(45)
laura.forward(100)
# laura kennt aber auch den neuen Befehl
laura.shape()
```



2. Definiere zwei aus *TurtleKid* abgeleitete Klassen *TurtleBoy* und *TurtleGirl*, welche *shape()* so überschreiben, dass ein *TurtleBoy* ein gefülltes Dreieck und *ein TurtleGirl* einen gefüllten Kreis zeichnet Der folgende Hauptteil soll funktionieren:

```
tf = TurtleFrame()

aGirl = TurtleGirl(tf)
aGirl.setColor("red")
aGirl.left(45)
aGirl.forward(100)
aGirl.shape()

aBoy = TurtleBoy(tf)
aBoy.setColor("green")
aBoy.right(45)
aBoy.forward(100)
aBoy.shape()

aKid = TurtleKid(tf)
aKid.back(100)
aKid.left(45)
aKid.shape()
```



3. Zeichne das Klassendiagramm zu Aufgabe 2

ZUSATZSTOFF

STATISCHE VARIABLEN UND STATISCHE METHODEN

Klassen können auch dazu verwendet werden, zusammengehörende Variablen oder Funktionen zu gruppieren und damit den Code übersichtlicher zu machen. Beispielsweise kannst du die wichtigsten physikalischen Konstanten in der Klasse *Physics* zusammenfassen. Man nennt Variablen, die im Klassenkopf definiert werden, **statische Variablen** und ruft sie durch Vorstellen des Klassennamens auf. Es ist also im Gegensatz zu Instanzvariablen nicht nötig, eine Instanz der Klasse zu erzeugen [**mehr...**].

```
import math

# ---------------- class Physics ----------------
class Physics():
    # Avogadro constant [mol-1]
    N_AVOGADRO = 6.0221419947e23
    # Boltzmann constant [J K-1]
    K_BOLTZMANN = 1.380650324e-23
    # Planck constant [J s]
    H_PLANCK = 6.6260687652e-34;
```

```
    # Speed of light in vacuo [m s-1]
    C_LIGHT = 2.99792458e8
    # Molar gas constant [K-1 mol-1]
    R_GAS = 8.31447215
    # Faraday constant [C mol-1]
    F_FARADAY = 9.6485341539e4;
    # Absolute zero [Celsius]
    T_ABS = -273.15
    # Charge on the electron [C]
    Q_ELECTRON = -1.60217646263e-19
    # Electrical permittivity of free space [F m-1]
    EPSILON_0 = 8.854187817e-12
    # Magnetic permeability of free space [ 4p10-7 H m-1 (N A-2)]
    MU_0 = math.pi*4.0e-7

c = 1 / math.sqrt(Physics.EPSILON_0 * Physics.MU_0)
print("Speed of light (calulated): %s m/s" %c)
print("Speed of light (table): %s  m/s" %Physics.C_LIGHT)
```

Eine Sammlung von zusammengehörenden Funktionen kannst du ebenfalls gruppieren, indem du sie als statische Methoden in einer aussagekräftig bezeichneten Klasse definierst. Diese Methoden kannst du dann durch Vorstellen des Klassennamens direkt verwenden, ohne dass du eine Instanz der Klasse erstellen musst. Um eine Methode statisch zu machen, schreibst du vor die Definition @staticmethod.

```
# ---------------- class OhmsLaw ----------------
class OhmsLaw():
    @staticmethod
    def U(R, I):
        return R * I

    @staticmethod
    def I(U, R):
        return U / R

    @staticmethod
    def R(U, I):
        return U / I

r = 10
i = 1.5

u = OhmsLaw.U(r, i)
print("Voltage = %s V" %u)
```

MEMO

Statische Variable (im Unterschied zu Instanzvariablen auch **Klassenvariablen** genannt) gehören zu der Klasse als Ganzes und haben im Gegensatz zu Instanzvariablen für alle Objekte der Klasse den gleichen Wert. Sie können mit vorgestelltem Klassennamen gelesen und verändert werden.

Eine typische Anwendung von statischen Variablen ist ein *Instanzenzähler,* also eine Variable, welche die Anzahl erzeugter Objekte der betreffenden Klasse zählt. Zusammengehörende Funktionen können als statische Methoden einer sinnvoll bezeichneten Klasse gruppiert werden. Bei der Definition muss die Zeile @staticmethod (**function decorator** genannt) vorgestellt werden

7.3 ARCADE GAMES, FROGGER

EINFÜHRUNG

Viele Computergames, die du auf Spielconsolen und im Internet findest, bestehen aus Bildern, die sich über einen Hintergrund bewegen. Manchmal ist sogar der Hintergrund bewegt, insbesondere wenn die Spielfiguren zum Rand des Bildschirmfenster laufen, damit dem Spieler der Eindruck eines Scenarios vermittelt wird, das wesentlich grösser als das Bildschirmfenster ist. Die Spielanimation erfordert zwar eine grosse Rechenleistung, ist aber grundsätzlich einfach zu verstehen: Zu sich in kurzer Zeit folgenden Zeitpunkten wird in der sogenannten Game-Loop der Bildschirminhalt neu berechnet, der Hintergrund und die Bilder der Spielfiguren in einen nicht sichtbaren Bildpuffer kopiert und dann dieser als Ganzes im Fenster dargestellt (gerendert). Werden mehr als ungefähr 25 Bilder pro Sekunden dargestellt, ergibt sich für das Auge ein fliessende Bewegung, bei weniger Bildern ist die Bewegung ruckartig.

In vielen Spielen interagieren die Spielfiguren durch Zusammenstösse. Die Behandlung von Kollisionen ist also für viele Spiele fundamental. Gut ausgebaute Gamelibraries wie JGameGrid unterstützen den Programmierer dabei durch eingebaute Kollisionsdetektion unter Verwendung des Eventmodells. Man definiert dabei, welches die potentiellen Kollisionspartner sind und das System ruft bei einem Kollisionsereignis automatisch einen Callback auf.

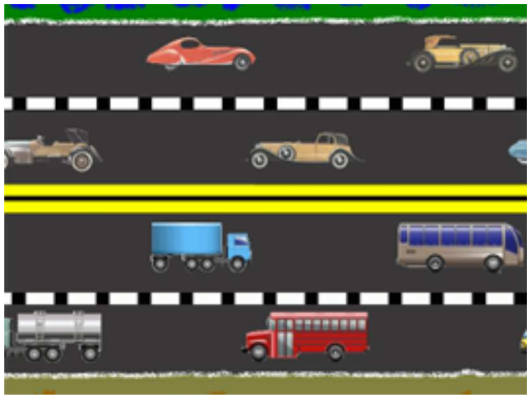
PROGRAMMIERKONZEPTE: *Gamedesign, Pixelgame, Actor, Kollision, Supervisor*

GAME-SZENARIO

Bei der Entwicklung von Computergames ist es wichtig, dass du dir zuerst ein möglichst detailliertes Spielszenario ausdenkst und stichwortartig als Pflichtenheft aufschreibst. Meist sind deine Ansprüche im ersten Anlauf zu hoch und du musst versuchen, das Game soweit zu vereinfachen, dass du lauffähige Teilversionen entwickeln kannst, die du schrittweise erweiterst. Die Kunst besteht darin, dass du das Programm so allgemein schreibst, dass du bei den nachfolgenden Erweiterungen den bestehenden Code nicht stark abändern musst, sondern nur zu ergänzen brauchst. Auf Anhieb gelingt dies aber selbst bei professionellen Programmieren selten, so dass bei der Entwicklung von Games Euphorie und Frustration eng beieinander liegen. Um so grösser ist deine Freude und Genugtuung, wenn du dein eigenes, ganz persönliches Computerspiel vorführen und spielen lassen kannst.

Der Weg zum kompetenten Gameprogrammierer führt über die Entwicklung von bekannten Spielen, die du in einer persönlichen Variante und mit deinen eigenen Spritebildern implementierst. Dabei ist es in der Ausbildungsphase nicht so wichtig, ob diese Games bereits fix-fertig erhältlich sind, denn es geht ja nicht in erster Linie darum, dass du viel damit spielst, sondern dass du lernst, wie man sie entwickelt.

Ein bekanntes Spiel ist das Froschspiel (Frogger). Es hat folgendes lustiges Scenario:
Ein Frosch versucht, sich über eine stark befahrene Strasse zu bewegen und zu einem Teich zu gelangen. Kollidiert er mit einem Fahrzeug, so verliert er sein Leben. Das Ziel ist es, ihn mit den Cursortasten sicher über die Strasse zu bringen. In deiner Implementierung gibt es vier Fahrbahnen, zwei mit gegeneinander laufenden Lastwagen und Bussen, und zwei mit gegeneinander laufenden Oldtimer-Autos.



Zwei mögliche Entwicklungswege stehen dir offen: Du realisierst zuerst die Bewegung des Froschs oder die Bewegung der Fahrzeuge. Nachher fügst du den Kollisionsmechanismus und die Verrechnung der Spielpunkte sowie die Behandlung des Spielendes (Game-Over) hinzu.

In GameGrid werden die Fahrzeuge als Instanzen der Klasse *Car* modelliert, die aus Actor abgeleitet ist. In der Methode **act()** wird die Bewegung der Fahrzeuge programmiert. Als Sprites verwendest du die Bilder *car0.gif,..car19.gif,* die sich in der Distribution von TigerJython befinden. Du kannst natürlich auch eigene Bilder verwenden (sie sollten maximal 70 Pixel hoch und maximal 200 Pixel breit sein und einen transparenter Hintergrund aufweisen).

Für Arcade-Games üblich ist die Verwendung eines Gameboard mit einer Zellengrösse von 1 Pixel. d.h. das Gitter entspricht dem Pixelraster. Als Fenstergrösse wählst du 800 x 600 Pixel und lädst ein Hintergrundbild *lane.gif* der Grösse 801 x 601 Pixel, das das Strassenszenario darstellt. In der Funktion **initCars()** erzeugst du die 20 Car-Objekte und überlegst dir, wo und in welcher Blickrichtung du sie in das Gameboard einfügen willst.

Das Bewegen der Autos mit der Methode **act()** ist einfach: Du schiebst sie mit *move()* weiter und lässt die nach rechts laufenden Autos von rechts nach links bzw. die nach links laufenden Autos von links nach rechts springen, wenn sie aus dem Bildschirmfenster hinausgefahren sind. Beachte dabei, dass die Location-Koordinaten eines Actors auch ausserhalb des Bildschirmfensters verwendet werden können.

```
from gamegrid import *

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)


makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False)
setSimulationPeriod(50)
initCars()
show()
doRun()
```

MEMO

Für Arcade-Games wird meist ein GameGrid mit einer Zellengrösse von 1 Pixel verwendet (Pixelgame). Bei einer Simulationsperiode von 50 ms wird die Spielszene 20 Mal pro Sekunde gerendert, was zu einer relativ gut fliessenden Bewegung führt. Das sporadisch auftretende Rucken ist darauf zurückzuführen, dass der Computer zu wenig Rechenleistung aufweist. Wegen der begrenzten Rechenleistung kann die Simulationsperiode auch nicht wesentlich verkleinert werden.

FROSCH MIT CURSORTASTEN BEWEGEN

Es nächstes stellst du dir die Aufgabe, den Frosch in das Spiel einzubauen. Dieser soll sich bei der Entstehung am unteren Bildrand befinden und mit den Cursor up-, down-, left und right-Tasten bewegt werden.

Da auch der Frosch ein Actor ist, schreibst du zuerst die Klasse *Frog*, die du von Actor ableitest. Ausser dem Konstruktor benötigst du keine Methoden, da der Frosch mit Tastatur-Events bewegt wird. Dazu definierst du den Callback **onKeyRepeated**, den du beim Aufruf von *makeGameGrid()* mit dem benannten Parameter *keyRepeated* registrierst. Dieser Callback wird nicht nur beim Drücken der Taste einmal, sondern auch bei gedrückt gehaltener Taste periodisch aufgerufen. Im Callback prüfst du den Tastencode und bewegst den Frosch entsprechend um 5 Schritte (Pixel) weiter.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
  def __init__(self):
      Actor.__init__(self, "sprites/frog.gif")

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)


makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
             keyRepeated = onKeyRepeated)
setSimulationPeriod(50);
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()
```

MEMO

Um Tastaturevents zu erfassen, können auch die Callbacks *keyPressed(e)* und *keyReleased(e)* registriert werden. Im Unterschied zu *keyRepeated(code)* muss der Keycode aber mit *e.getKeyCode()* aus dem Parameter *e* geholt werden. Zudem ist in diesem Spiel *keyPressed(e)* weniger geeignet, da es nach dem Drücken und Halten der Taste eine Verzögerung gibt, bist die nachfolgenden Press-Events ausgelöst werden. Wenn du die Keycodes nicht kennst, so schreibst du am besten ein kleines Testprogramm, das diese ausschreibt:

```
from gamegrid import *

def onKeyPressed(e):
    print "Pressed: ", e.getKeyCode()

def onKeyReleased(e):
    print "Released: ", e.getKeyCode()

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
  keyPressed = onKeyPressed, keyReleased = onKeyReleased)
show()
```

KOLLISIONSEVENTS

Das Vorgehen zur Detektion von Kollisionen zwischen Aktoren ist einfach: Du sagst bei der Erzeugung eines Fahrzeug *car* mit mit der Methode

```
frog.addCollisionActor(car)
```

dass der Frosch bei einer Kollision mit einem Fahrzeug einen Event auslösen soll, der die Methode **collide()** aufruft, die sich in der Klasse *Frog* befindet. Dort behandelst du den Event nach deinen Wünschen, beispielsweise lässt du den Frosch wieder an die Startposition zurück springen.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/frog.gif")
        self.setCollisionCircle(Point(0, -10), 5)

    def collide(self, actor1, actor2):
        self.setLocation(Location(400, 560))
        return 0

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        frog.addCollisionActor(car)
```

```
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
     keyRepeated = onKeyRepeated)
setSimulationPeriod(50)
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()
```

MEMO

Die Methode **collide()** ist kein eigentlicher Callback, sondern eine Methode der Klasse *Actor*, die in *Frog* überschrieben wird. Darum brauchst du *collide()* auch nicht mit einem benannten Parameter zu registrieren. Standardmässig wird der Kollisionsevent dann ausgelöst, wenn sich die umgebenden Rechtecke der Spritebilder überschneiden. Man kann aber die Kollisionsbereiche sowohl in Form, Grösse und Lage bezüglich das Sprite-Bildes verändern. Dazu stehen folgende Methoden der Klasse *Actor* zur Verfügung:

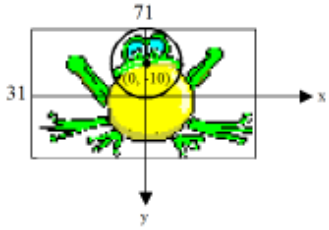
Methode	**Kollisionbereich**
setCollisionCircle(centerPoint, radius)	Kreis mit gegebenem Zentrum und Radius (in Pixel)
setCollisionImage()	Nicht-transparente Bildpixels (nur mit einem Partner der Kreis, Linie oder Punkt als Kollisionsbereich hat)
setCollisionLine(startPoint, endPoint)	Linie zwischen Start- und Endpunkt
setCollisionRectangle(center, width, height)	Rechteck mit gegebenem Zentrum und gegebener Länge und Breite
setCollisionSpot(spotPoint)	Ein Bildpixel

Alle Methoden verwenden ein relatives Pixel-Koordinatensystem mit Nullpunkt in der Mitte des Spritebildes und positiver x-Achse nach rechts und positiver y-Achse nach unten.

Das Froschbild hat eine Grösse von 71 x 41 Pixel. Setzt man daher beispielsweise im Konstruktor von Frog zusätzlich

```
self.setCollisionCircle(Point(0, -10), 5)
```

so muss ein Fahrzeug über den Kreis mit Radius 5 Pixel am Kopf des Frosches fahren, um einen Kollisionsevent auszulösen.



(Da der Kollisionsbereich aus Effizienzgründen zwischengespeichert wird, kann es nötig sein, TigerJython neu zu starten, damit sich Änderungen auswirken.)

SPIEL-SUPERVISOR UND SOUND

Bei vielen Spielen ist es nötig, dass ein "unabhängiger Spiel-Supervisor" für die Einhaltung der Spielregeln, die Verteilung der Punkte und das Ausrufen des Siegers bei Game-Over verantwortlich gemacht wird. Wie im täglichen Leben, ist es auch hier besser, diese Aufgabe nicht einer Spielfigur, sondern einem davon unabhängigen Programmteil zuzuweisen. Besonders gut geeignet ist der Hauptteil des Programms, der ja nach der Spielinitialisierung zu Ende läuft. Du fügst dazu am Ende des bestehenden Programms eine Schleife ein, die mit einer kurzen Periodendauer das Spiel überprüft und entsprechend handelt. Du solltest aber nicht eine ganz enge Schleife ohne **delay()** einbauen, da diese das Programm unnötig belastet, was zu Verzögerungen im übrigen Programmablauf führen kann.

Die Schleife sollte dann abbrechen, wenn das Game-Fenster geschlossen wird, was du mit **isDisposed = True** erreichst. Der Supervisor kann beispielsweise die Anzahl Versuche begrenzen und die Anzahl der erfolgreichen und misslungenen Strassenüberquerungen zählen und anzeigen. Die Behandlung der Situation bei *Game-Over* ist oft speziell trickreich, da an verschiedene Varianten gedacht werden muss. Oft ist es auch so, dass man das Spiel mehrmals spielen will, ohne das Programm neu zu starten. Für den Einbau von Soundeffekten kannst du deine Kenntnisse vom Kapitel Sound beiziehen. Am einfachsten verwendest du die Funktion *playTone()*.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/frog.gif")

    def collide(self, actor1, actor2):
        global nbHit
        nbHit += 1
        playTone([("c''h'a'f'", 100)])
        self.setLocation(Location(400, 560))
        return 0

    def act(self):
        global nbSuccess
        if self.getY() < 15:
            nbSuccess += 1
            playTone([("c'e'g'c''", 200)])
            self.setLocation(Location(400, 560))

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        frog.addCollisionActor(car)
        if i < 5:
            addActor(car, Location(350 * i, 90), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
```

```
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
    keyRepeated = onKeyRepeated)
setSimulationPeriod(50)
setTitle("Frogger")
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()

# Game supervision
maxNbLifes = 3
nbHit = 0
nbSuccess = 0
while not isDisposed():
    if nbHit + nbSuccess == maxNbLifes:  # game over
        addActor(Actor("sprites/gameover.gif"), Location(400, 285))
        removeActor(frog)
        doPause()
    setTitle("#Success: " + str(nbSuccess) + " #Hits " + str(nbHit))
    delay(100)
```

MEMO

Das Zählen der Erfolge mit **nbSuccess** und der Misserfolge mit **nbHit** erfolgt in der Klasse *Frog*. Darum müssen die Variablen dort als global deklariert werden. Man könnte auch statische oder Instanzvariablen der Klasse *Frog* verwenden. Bei Game-Over wird ein Actorbild mit einem Text eingefügt, der Frosch entfernt, der Simulationszyklus mit **doPause()** angehalten und dann die Supervisor-Schleife mit break verlassen. Man könnte auch einen *TextActor* verwenden. Damit ist es möglich, den Text zu Laufzeit anzupassen.

```
rate = nbSuccess / (nbSuccess + nbHit)
ta = TextActor("  Game Over: Success Rate = " + str(rate) + " % ",
     DARKGRAY, YELLOW, Font("Arial", Font.BOLD, 24))
addActor(ta, Location(200, 287))
```

AUFGABEN

1. Ersetze das Hintergrundbild und die Oldtimer-Bilder durch Tierbilder, die in einem Fluss schwimmen (Krokodile, usw.).

2. Führe einen Punktezählung (Score) und eine Zeitlimite für die Überquerung ein: Jede erfolgreiche Überquerung gibt 5 Punkte, jeder Hit -5 Punkte. Das Überschreiten der Zeitlimite ergibt 10 Minuspunkte und lässt den Frosch wieder an den Anfangsort zurückspringen.

3. Ergänze das Spiel nach eigenen Ideen.

7.4 GRIDGAMES, SOLITAIRE BRETTSPIEL

EINFÜHRUNG

Bei einer bestimmten Klasse von Computergames können sich die Spielfiguren nur in Zellen einer Gitterstruktur aufhalten, wobei die Zellen meist gleiche Grösse haben und matrixartig angeordnet sind. Die Berücksichtigung der Ortsbeschränkung auf eine Gitterstruktur vereinfacht die Implementierung des Spiels ganz wesentlich. Wie schon der Name sagt, ist die Gamelibrary *JGameGrid* für gitterartige Games besonders optimiert.

In diesem Kapitel entwickelst du schrittweise das Brett-Solitaire mit dem englischen Brettlayout. Dabei lernst du wichtige Lösungsverfahren kennen, die du auf alle Gitterspiele anwenden kannst.

PROGRAMMIERKONZEPTE: *Spielbrett, Spielregeln, Pflichtenheft, Game-Over*

BRETTINITIALISIERUNG, MAUSSTEUERUNG

Auf einem Spielbrett befinden sich in einer regelmässigen Anordnung Löcher oder Vertiefungen, in die du Stifte stecken bzw. Murmeln legen kannst. Das bekannteste Board-Solitaire verwendet ein Brett mit einer kreuzartigen Anordnung von 33 Vertiefungen und wird englisches Brett genannt. Am Anfang sind alle Löcher ausser dem Zentrumsloch mit Murmeln belegt. Wie der Name *Solitaire* sagt, wird es meistens von einer einzigen Person gespielt.

Ein englisches Board-Solitaire aus Indien, 1830
© 2003 puzzlemuseum.com

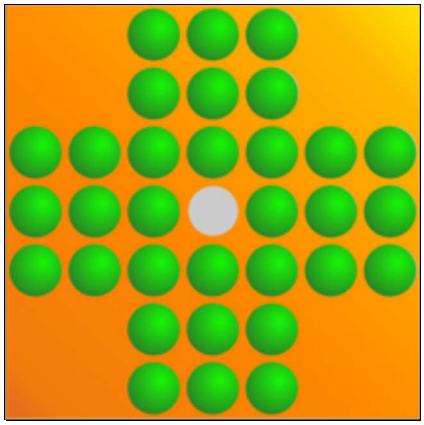
Es gelten folgende **Spielregeln**: Ein Zug besteht darin, eine Murmel auf ein freies Loch zu verschieben, wobei dabei genau eine Murmel in horizontaler oder vertikaler Richtung übersprungen werden muss. Die übersprungene Murmel wird vom Spielbrett entfernt.

Das Ziel besteht darin, alle Murmeln bis auf eine letzte vom Brett "abzuräumen". Das Spiel gilt als besonders gut gelöst, falls sich die letzte Murmel im Zentrum befindet. In der Implementierung als Computergame sollst du eine bestimmte Murmel durch Drücken der Maustaste "packen" und bei gedrückter Maustaste verschieben können. Beim Loslassen der Maustaste wird geprüft, ob der Zug den Spielregeln entspricht. Falls er regelwidrig ist, soll die Murmel wieder an den Anfangsort zurückspringen; ist er legal, so wird die Murmel am neuen Ort angezeigt und die übersprungene Murmel vom Brett entfernt.

Damit ist das **Pflichtenheft** klar und du kannst hinter die Implementierung gehen. Diese erfolgt wie immer schrittweise, wobei bei jedem Schritt ein lauffähiges Programm vorliegen muss. Es liegt auf der Hand, eine GameGrid mit 7x7 Zellen zu verwenden, wobei die Eckzellen nicht verwendet werden. Zuerst zeichnest du in der Funktion *initBoard()* mit dem Hintergrundbild *solitaire_board.png*, die sich in der Distribution von TigerJython befindet, das Brett und realisierst die Maussteuerung mit den Maus-Callbacks *mousePressed*, *mouseDragged* und *mouseReleased*.

Beim **Press-Event** merkst du dir die aktuelle Location und die aktuelle Murmel in dieser Zelle. Diese wird dir mit **getOneActorAt()** zurückgeben, wobei du *None* erhältst, wenn die Zelle leer ist. Der Entwicklungsprozess ist leichter zu beherrschen und Fehler besser aufzufinden, wenn du in einer Statusbar (oder in der Konsole) wichtige Ergebnisse ausschreibst.

Beim **Drag-Event** verschiebst du das sichtbare Bild der Murmel mit **setLocationOffset()** an die aktuelle Cursorposition. Diese kann auch neben der Zellenmitten liegen, sodass sich eine kontinuierlich sichtbare Verschiebung ergibt. Dabei ist es wichtig, dass du den Murmelactor selbst nicht verschiebst, sondern nur sein Spritebild (darum die Bezeichnung *Offset*). Damit umgehst du alle Schwierigkeiten mit übereinander liegenden Actors.



Beim **Release-Event** soll in dieser ersten Version die Kugel wieder an ihre Startlocation zurückspringen. Dies erreichst du mit dem Aufruf von **setLocationOffset(0, 0).**

```
from gamegrid import *

def isMarbleLocation(loc):
    if loc.x < 0 or loc.x > 6 or loc.y < 0 or loc.y > 6:
        return False
    if (loc.x == 0 or loc.x == 1 or loc.x == 5 or loc.x == 6) and \
       (loc.y == 0 or loc.y == 1 or loc.y == 5 or loc.y == 6):
        return False
    return True

def initBoard():
    for x in range(7):
        for y in range(7):
            loc = Location(x, y)
            if isMarbleLocation(loc):
                marble = Actor("sprites/marble.png")
                addActor(marble, loc)
    removeActorsAt(Location(3, 3)) # Remove marble in center

def pressEvent(e):
    global startLoc, movingMarble
    startLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble = getOneActorAt(startLoc)
    if movingMarble == None:
       setStatusText("Pressed at " + str(startLoc) + ". No marble found")
    else:
       setStatusText("Pressed at " + str(startLoc) + ". Marble found")

def dragEvent(e):
    if movingMarble == None:
        return
    startPoint = toPoint(startLoc)
    movingMarble.setLocationOffset(e.getX() - startPoint.x,
                                   e.getY() - startPoint.y)

def releaseEvent(e):
    if movingMarble == None:
        return
    movingMarble.setLocationOffset(0, 0)

makeGameGrid(7, 7, 70, None, "sprites/solitaire_board.png", False,
    mousePressed = pressEvent, mouseDragged = dragEvent,
    mouseReleased = releaseEvent)
setBgColor(Color(255, 166, 0))
setSimulationPeriod(20)
addStatusBar(30)
setStatusText("Press-drag-release to make a move.")
initBoard()
show()
doRun()
```

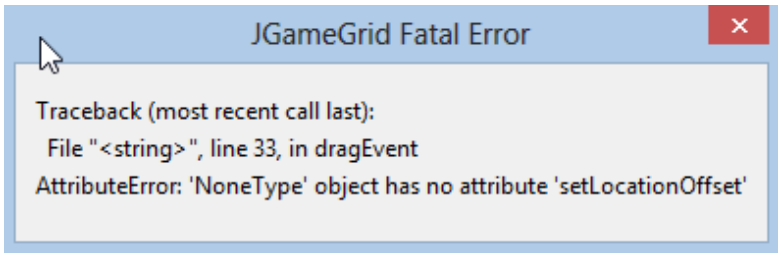
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Statt beim Draggen den Actor selbst zu verschieben, kannst du auch nur sein Spritebild bewegen. Dazu verwendest du **setLocationOffset(x, y)**, wobei x und y relative Koordinaten in Bezug auf den aktuellen Mittelpunkt des Sprites sind.

Im Zusammenhang mit Mausbewegungen musst du zwischen den Koordinaten der Maus und Zellenkoordinaten sorgfältig unterscheiden. Dabei sind die Konversionsfunktionen *toLocationInGrid(pixel_coord)* bzw. *toPoint(location_coord)* wichtig.

Gehst du von einer leeren Zelle aus, so führen der *Drag-* und *Release-Event* zu einem berüchtigten Programm-Absturz, da *movingMarble* den Wert *None* hat und du damit eine Methode aufrufst.



Um den Fehler zu vermeiden, verlässt du die Callbacks gerade zu Beginn mit einem sofortigen return.

IMPLEMENTIERUNG DER SPIELREGELN

Wie würdest du beim wirklichen Spiel die Spielregeln überprüfen? Du müsstest wissen, mit welcher Murmel du gestartet bist, also deren Startlocation *start* kennen. Dann müsstest du wissen, wohin du die Murmel verschieben möchtest, also deren Ziellocation *dest* kennen. Für einen legalen Zug müssen folgenden Bedingungen zutreffen:

1. Bei *start* gibt es eine Murmel
2. Bei *dest* gibt es keine Murmel
3. *dest* ist eine Zelle, die zum Board gehört
4. *start* und *dest* sind entweder horizontal oder vertikal zwei Zellen auseinander
5. An der Zwischenzelle befindet sich eine Murmel

Es ist elegant, diese Bedingungen in einer Funktion **getRemoveMarble(start, dest)** zu implementieren, welche bei einem legalen Zug die zu entfernende Murmel und bei einem illegalen Zug *None* zurückgibt.

Es ist klar, dass du diese Funktion im Release-Event aufrufst und bei einem legalen Zug den zurückgegebenen Actor mit **removeActor()** vom Board nimmst.

```
from gamegrid import *

def getRemoveMarble(start, dest):
    if getOneActorAt(start) == None:
        return None
    if getOneActorAt(dest) != None:
        return None
    if not isMarbleLocation(dest):
        return None
    if dest.x - start.x == 2 and dest.y == start.y:
        return getOneActorAt(Location(start.x + 1, start.y))
    if start.x - dest.x == 2 and dest.y == start.y:
        return getOneActorAt(Location(start.x - 1, start.y))
```

```
    if dest.y - start.y == 2 and dest.x == start.x:
        return getOneActorAt(Location(start.x, start.y + 1))
    if start.y - dest.y == 2 and dest.x == start.x:
        return getOneActorAt(Location(start.x, start.y - 1))

def isMarbleLocation(loc):
    if loc.x < 0 or loc.x > 6 or loc.y < 0 or loc.y > 6:
        return False
    if (loc.x == 0 or loc.x == 1 or loc.x == 5 or loc.x == 6) and \
       (loc.y == 0 or loc.y == 1 or loc.y == 5 or loc.y == 6):
        return False
    return True

def initBoard():
    for x in range(7):
        for y in range(7):
            loc = Location(x, y)
            if isMarbleLocation(loc):
                marble = Actor("sprites/marble.png")
                addActor(marble, loc)
    removeActorsAt(Location(3, 3)) # Remove marble in center

def pressEvent(e):
    global startLoc, movingMarble
    startLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble = getOneActorAt(startLoc)
    if movingMarble == None:
       setStatusText("Pressed at " + str(startLoc) + ". No marble found")
    else:
       setStatusText("Pressed at " + str(startLoc) + ". Marble found")

def dragEvent(e):
    if movingMarble == None:
        return
    startPoint = toPoint(startLoc)
    movingMarble.setLocationOffset(e.getX() - startPoint.x,
                                   e.getY() - startPoint.y)

def releaseEvent(e):
    if movingMarble == None:
        return
    destLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble.setLocationOffset(0, 0)
    removeMarble = getRemoveMarble(startLoc, destLoc)
    if removeMarble == None:
        setStatusText("Released at " + str(destLoc) + ". Not a valid move.")
    else:
        removeActor(removeMarble)
        movingMarble.setLocation(destLoc)
        setStatusText("Released at " + str(destLoc)+ ". Marble removed.")


startLoc = None
movingMarble = None

makeGameGrid(7, 7, 70, None, "sprites/solitaire_board.png", False,
    mousePressed = pressEvent, mouseDragged = dragEvent,
    mouseReleased = releaseEvent)
setBgColor(Color(255, 166, 0))
setSimulationPeriod(20)
addStatusBar(30)
setStatusText("Press-drag-release to make a move.")
initBoard()
show()
doRun()
```

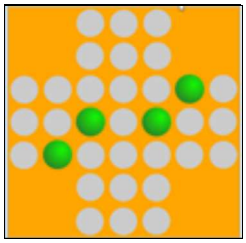
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Statt mit mehreren vorzeitigen *return* die Funktion *getRemoveMarble()* zu verlassen, könnte man die Bedingungen auch mit booleschen Operation verknüpfen. Es ist Ansichtssache, welche Programmiertechnik man als geeigneter betrachtet.

PRÜFUNG AUF GAME-OVER

Es bleibt dir jetzt nur noch die Aufgabe, bei jedem legalen Zug zu prüfen, ob das Spiel beendet ist. Dies ist sicher dann der Fall, falls sich nur noch eine einzige Murmel auf dem Spielfeld befindet und du damit das Spielziel erreicht hast.



Dabei vergisst du aber, dass es noch andere Spielkonstellationen geben könnte, bei denen das Spiel als beendet betrachtet werden muss, nämlich wenn sich noch mehr als eine Murmel auf dem Brett befindet, du aber mit keiner davon einen legalen Zug machen kannst. Es ist zwar nicht ganz sicher, ob man mit legalen Zügen überhaupt einmal in diese Situation kommt, aber du musst ***defensiv* programmieren**, also immer auf der sicheren Seite bleiben, denn du kannst damit rechnen, dass auch beim Programmieren der Murphy-Spruch gilt: "Wenn etwas schief gehen kann, geht es schief".

Um diese Situation in den Griff zu bekommen, kannst du in der Funktion **isMovePossible()** alle noch vorhandenen Murmeln einzeln darauf zu testen, ob man mit ihnen einen legalen Zug machen kann. Dazu prüfst du für jede Murmel, ob es mit irgendeinem Loch eine zu entfernende Zwischenmurmel gibt [**mehr...**].

```
from gamegrid import *

def checkGameOver():
    global isGameOver
    marbles = getActors() # get remaining marbles
    if len(marbles) == 1:
        setStatusText("Game over. You won.")
        isGameOver = True
    else:
        # check if there are any valid moves left
        if not isMovePossible():
           setStatusText("Game over. You lost. (No valid moves available)")
           isGameOver = True

def isMovePossible():
   for a in getActors():  # run over all remaining marbles
        for x in range(7): # run over all holes
            for y in range(7):
                loc = Location(x, y)
                if getOneActorAt(loc) == None and \
                  getRemoveMarble(a.getLocation(), Location(x, y)) != None:
                    return True
   return False

def getRemoveMarble(start, dest):
    if getOneActorAt(start) == None:
        return None
    if getOneActorAt(dest) != None:
        return None
    if not isMarbleLocation(dest):
        return None
    if dest.x - start.x == 2 and dest.y == start.y:
        return getOneActorAt(Location(start.x + 1, start.y))
    if start.x - dest.x == 2 and dest.y == start.y:
        return getOneActorAt(Location(start.x - 1, start.y))
```

```
    if dest.y - start.y == 2 and dest.x == start.x:
        return getOneActorAt(Location(start.x, start.y + 1))
    if start.y - dest.y == 2 and dest.x == start.x:
        return getOneActorAt(Location(start.x, start.y - 1))
    return None

def isMarbleLocation(loc):
    if loc.x < 0 or loc.x > 6 or loc.y < 0 or loc.y > 6:
        return False
    if (loc.x == 0 or loc.x == 1 or loc.x == 5 or loc.x == 6) and \
       (loc.y == 0 or loc.y == 1 or loc.y == 5 or loc.y == 6):
        return False
    return True

def initBoard():
    for x in range(7):
        for y in range(7):
            loc = Location(x, y)
            if isMarbleLocation(loc):
                marble = Actor("sprites/marble.png")
                addActor(marble, loc)
    removeActorsAt(Location(3, 3)) # Remove marble in center

def pressEvent(e):
    global startLoc, movingMarble
    if isGameOver:
        return
    startLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble = getOneActorAt(startLoc)
    if movingMarble == None:
       setStatusText("Pressed at " + str(startLoc) + ".No marble found")
    else:
       setStatusText("Pressed at " + str(startLoc) + ".Marble found")

def dragEvent(e):
    if isGameOver:
        return
    if movingMarble == None:
        return
    startPoint = toPoint(startLoc)
    movingMarble.setLocationOffset(e.getX() - startPoint.x,
                                   e.getY() - startPoint.y)

def releaseEvent(e):
    if isGameOver:
        return
    if movingMarble == None:
        return
    destLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble.setLocationOffset(0, 0)
    removeMarble = getRemoveMarble(startLoc, destLoc)
    if removeMarble == None:
        setStatusText("Released at " + str(destLoc)
                      + ". Not a valid move.")
    else:
        removeActor(removeMarble)
        movingMarble.setLocation(destLoc)
        setStatusText("Released at " + str(destLoc)+
                      ". Valid move - Marble removed.")
        checkGameOver()


startLoc = None
movingMarble = None
isGameOver = False

makeGameGrid(7, 7, 70, None, "sprites/solitaire_board.png", False,
   mousePressed = pressEvent, mouseDragged = dragEvent,
   mouseReleased = releaseEvent)
```

```
setBgColor(Color(255, 166, 0))
setSimulationPeriod(20)
addStatusBar(30)
setStatusText("Press-drag-release to make a move.")
initBoard()
show()
doRun()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

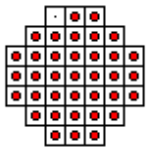
MEMO

Nach jedem Zug testest du mit **checkGameOver()**, ob das Spiel beendet ist. Ist dies der Fall, befindet sich das Spiel in einem ganz **speziellen Zustand,** den du mit der booleschen Variable (ein Flag) **isGameOver = True** kennzeichnest.

Insbesondere musst du bei *Game-Over* auch alle Maus-Aktionen unterbinden. Du erreichst dies mit einem sofortigen **return** aus den Maus-Callbacks.

AUFGABEN

1. Erstelle ein Brett-Solitaire mit einem französischen Brett.



2. Erweitere das Brett-Solitaire mit einem Score, der die Anzahl Züge zählt und ausschreibt. Auch soll das Spiel mit der Space-Taste wieder neu gestartet werden können.

3. Orientiere dich bei einer Lehrperson oder im Internet über Lösungsstrategien des Brett-Solitaire [**mehr...**].

4. Erstelle ein Brett-Solitaire nach deinen eigenen Ideen.

7.5 SPRITE-ANIMATION

EINFÜHRUNG

In der Gamelibrary *JGameGrid* werden alle Spielfiguren aus der Klasse *Actor* abgeleitet, damit sie bereits ohne Programmieraufwand viele wichtige Eigenschaften und Fähigkeiten besitzen. Ihr spezifisches Aussehen erhalten sie aber über ihr Erscheinungsbild, das als Bilddatei, auch Sprite genannt, geladen wird.

Spielfiguren sind in vieler Hinsicht animiert: Sie bewegen sich über das Spielfeld und ändern dabei ihr Erscheinungsbild, z.B. ihre Körperhaltung oder ihre Miene. Aus diesem Grund können einem Actor-Objekt beliebig viele verschiedene Spritebilder zugeordnet werden, die über einen ganzzahligen Index, der Sprite-ID, unterschieden werden. Dies ist einfacher, als Actors mit unterschiedlichen Sprites durch Klassenableitungen zu modellieren.

Spielfiguren werden in der Regel auch ihren Ort, ihre Bewegungsrichtung und ihren Rotationswinkel verändern. Dabei sollte der Rotationswinkel automatisch der Bewegungsrichtung angepasst werden. In *JGameGrid* muss aus Effizenzgründen bereits bei der Definition der Actors angegeben werden, ob diese rotierbar und welche Spritebilder zugeordnet sind. Diese werden bei der Erstellung des Actor-Objekt in einen Bildpuffer geladen, der auch die rotierten Bilder enthält. Zu Laufzeit müssen dann die Bilder weder von der Festplatte geladen noch sonstwie transformiert werden. Standardmässig werden 60 Spritebilder für alle 6 Grad erzeugt.

In *JGameGrid* wird ein Animationskonzept angewendet, das man auch in anderen Game-Libraries, insbesondere in **Greenfoot** [**mehr...**] findet.

Fundamentales Animationsprinzip:

> In der Klasse *Actor()* ist die Methode *act()* definiert, die einen **leeren** Definitionsteil hat, also sofort zurückkehrt. Die von Actor abgeleiteten benutzerdefinierten Spielfiguren überschreiben *act()* und implementieren dabei das spezifische Verhalten der Spielfigur.
>
> Beim Hinzufügen einer Spielfigur zum Spielfenster mit *addActor(),* wird dieser in eine **Act-Order-Liste** (geordnet nach Actor-Klassen) eingefügt. Eine interne **Game-Loop** (hier auch **Simulationszyklus** genannt), durchläuft periodisch diese Liste und ruft wegen der Polymorphie *act()* aller Actors der Reihe nach auf.

Damit dieses geistreiche Prinzip funktioniert, müssen sich allerdings die Actors **kooperativ** verhalten, d.h. **kurz laufenden Code** aufweisen. Insbesondere wirken sich Schleifen und Delays katastrophal aus, da andere Actors dadurch auf den Aufruf ihres eigenen *act()* warten müssen.

Das Zeichnen der Spritebilder erfolgt nach folgendem Prinzip. In der Game-Loop werden die Bilder aller Actoren in der Reihenfolge der **Paint-Order-Liste** in einen Bildschirmpuffer kopiert und dieser am Ende im Spielfenster gerendert. Die Reihenfolge des Durchlaufs legt damit auch die Sichtbarkeit der Spritebilder fest: Spritebilder **später** durchlaufener Actors überdecken die anderen, liegen also sozusagen **weiter oben**. Da die Actors in der Reihenfolge des Aufrufs von *addActor()* in die Paint-Order-Liste eingefügt werden, liegen später hinzugefügte Sprites oberhalb der anderen. [**mehr...**]

Zwar können bei der Initialisierung einem Actor beliebig viele Spritebilder zugeordnet werden, aber diese können zu Laufzeit **nicht verändert** werden. [**mehr...**]

PROGRAMMIERKONZEPTE: *Simulationszyklus, Kooperativer Code, Fabrik-Klasse, Statische Variable, Entkoppelung*

PFEILBOGEN BEWEGEN UND PFEILE ABSCHIESSEN

Mit einer Armbrust, die du mit der Tastatur steuerst, willst du Pfeile abschiessen, die sich auf einer natürlichen Bahn (Wurfparabel) bewegen. Später willst du diese Pfeile verwenden, um herumfliegende Früchte zu halbieren.

Du schreibst eine Klasse *Crossbow*, die du von der Klasse *Actor* ableitest. Beim Aufruf des Konstruktors der Basisklasse *Actor* sagst du mit *True*, dass es sich um einen rotierbaren Actor handelt. Der Wert 2 gibt an, dass er 2 Spritebilder besitzt, nämlich eines mit einer gespannten Armbrust und aufgesetztem Pfeil und eines für die entspannte Armbrust ohne Pfeil. Die Bilddateien werden automatisch unter dem Namen *sprites/crossbow_0.gif* und *sprites/crossbow_1.gif* gesucht und befinden sich in der Distribution von *TigerJython*.

```
Actor.__init__(self, True, "sprites/crossbow.gif", 2)
```

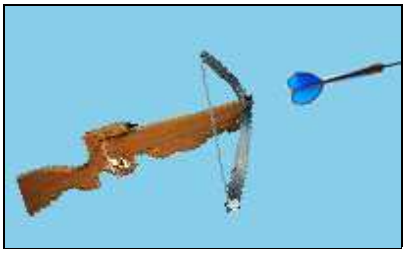
Die Armbrust wird mit Tastaturevents gesteuert: Mit Cursor-up/down veränderst du die Richtung und mit der Space-Taste schiesst du den Pfeil ab. Der Callback *keyCallback()* ist in *makeGameGrid()* als *keyPressed* registriert.

Die Pfeilklasse *Dart* ist bereits etwas komplizierter, müssen sich doch die Pfeile auf einer Wurfparabel in einem x-y-Koordinatensystem mit horizontaler x und nach unten zeigender y-Achse bewegen. Die Flugbahn wird nicht aus einer Kurvengleichung, sondern iterativ als Veränderung in der kurzen Zeit *dt* bestimmt. Aus der Kinematik ist bekannt, dass sich dabei die neuen Geschwindigkeitskoordianten (*vx', vy'*) und die neuen Ortskoordinaten (*px', py'*) nach der Zeit dt wie folgt berechnen (*g = 9.81m/s^2* ist die Gravitationsbeschleunigung):

$$vx' = vx$$
$$vy' = vy + g * dt$$

$$px' = px + vx * dt$$
$$py' = py + vy * dt$$

Die Startwerte (Anfangsbedingungen) ermittelst du in der Methode **reset()**, die beim Hinzufügen der Dartinstanz zum Spielfeld automatisch aufgerufen wird.



In *act()* gibst du dem Pfeil den neuen Ort und die neue Richtung. Um Ressourcen zu sparen, entfernst du ihn vom Board, sobald er ausserhalb des sichtbaren Fenster ist und bringst die Armbrust wieder in Abschussstellung.

```
# GG5a.py

from gamegrid import *
import math

# ------------------- class Crossbow -----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()

    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
```

```
        self.vx = self.speed * math.cos(math.radians(self.getDirection()))
        self.vy = self.speed * math.sin(math.radians(self.getDirection()))

    def act(self):
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy, self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

# ------ End of class definitions --------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
        crossbow.show(1) # crossbow is released
        dart = Dart(100)
        addActorNoRefresh(dart, crossbow.getLocation(),
                                crossbow.getDirection())

screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Beim Aufruf des Konstruktors der Klasse *Actor* wird angegeben, ob der Actor rotierbar ist und ob ihm mehrere Spritebilder zugeordnet sind. [**mehr...**]

Die Richtung des Pfeils drehst du ständig in Richtung der Geschwindigkeit, damit ein natürliches Flugbild entsteht.

FRÜCHTEFABRIK UND BEWEGTE FRÜCHTE

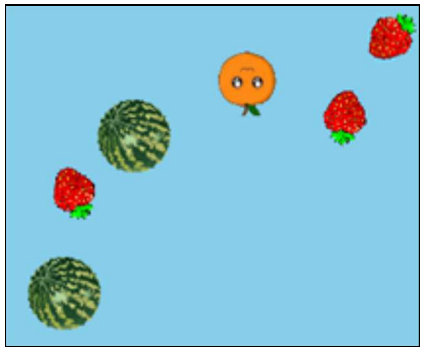
Dein Programm soll drei Sorten von Früchten verwenden: Melonen, Orangen und Erdbeeren. Die Früchte werden laufend in zufälliger Reihenfolge erzeugt und bewegen sich dann vom rechten oberen Bildrand mit zufällig variierter Horizontalgeschwindigkeit auf einer Wurfparabel nach links. Die drei Früchtesorten haben viele Gemeinsamkeiten und ein paar wenige Unterschiede. Es wäre daher **kein guter Entscheid**, die Klassen *Melon*, *Orange* und *Strawberry* aus *Actor* abzuleiten, denn dann müsstest du die Gemeinsamkeiten in jeder Klasse neu implementieren, was zu der verpönten **Codeduplikation** führt. Es ist angebracht, in dieser Situation eine Hilfsklasse *Fruit* zu definieren, in der die Gemeinsamkeiten implementiert sind und die speziellen

Früchte *Melon, Orange und Strawberry* aus *Fruit* abzuleiten.

Die Erzeugung der Früchte delegierst du einer Klasse, die man eine **Fabrik-Klasse** (**factory**) nennt. Obschon sie kein Spritebild besitzt, leitest du sie ebenfalls aus *Actor* ab, damit du *act()* verwenden kannst, um neue Früchte zu erzeugen. Eine *Factory*-Klasse hat eine spezifische Eigenschaft: Obschon sie mehrere Früchte erzeugt, gibt es davon nur eine einzige Instanz [**mehr...**]. Es ist daher nicht üblich, den Konstruktor zu verwenden, der ja zur Erzeugung von mehreren Instanzen vorgesehen ist. Factory-Klassen besitzen darum eine Methode **create()** (oder mit einem ähnlich vielsagenden Namen), die ein einziges Objekt der Klasse erstellt und es als Funktionswert zurückgibt. Jeder weitere Aufruf von *create()* liefert dann nur die bereits erzeugte Factory-Instanz [**mehr...**].

Da ja die Methode **create()** ohne eine Instanz aufgerufen wird, muss sie mit *@staticmethod* **statisch** definiert werden.

Bei der Erzeugung der *FruitFactory* wird mit der Variablen *capacity* auch noch angegeben, welches die maximale Anzahl Früchte ist, welche die Factory erzeugen kann. Zudem kann jeder Actor *setSlowDown()* aufgerufen, um die Aufrufsfrequenz von *act()* zu verlangsamen.



```
from gamegrid import *
import random

# ---------- class Fruit ------------------------
class Fruit(Actor):
    def  __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2) # rotatable, 2 sprites
        self.vx = vx
        self.vy = 0

    def reset(self): # Called when Fruit is added to GameGrid
        self.px = self.getX()
        self.py = self.getY()

    def act(self):
        self.movePhysically()
        self.turn(10)

    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
        self.vy = self.vy + g * self.dt # vx = const
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            self.removeSelf()

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", 2)
        self.vx = vx

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
```

```
class Strawberry(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------- class FruitFactory -------------------
class FruitFactory(Actor):
    myFruitFactory = None
    myCapacity = 0
    nbGenerated = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
                 # slows down act() call for this actor
        return FruitFactory.myFruitFactory

    def act(self):
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            print "Factory expired"
            return

        vx = -(random.random() * 20 + 30)
        r = random.randint(0, 2)
        if r == 0:
            fruit = Melon(vx)
        elif r == 1:
            fruit = Orange(vx)
        else:
            fruit = Strawberry(vx)
        FruitFactory.nbGenerated += 1
        y = int(random.random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)

# ------ End of class definitions --------------------

FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
addActor(factory, Location(0, 0))  # needed to run act()
setSimulationPeriod(30)
doRun()
show()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

In einer statischen Methode steht der Parameter *self* nicht zur Verfügung. Daher müssen alle Variablen, die in *create()* zugewiesen werden, **statische Variablen** sein (voranstellen des Klassennamens) [**mehr...**].

In einer Entwicklungsphase können gewisse Funktionen oder Methoden noch unvollständig codiert sein. Man kann beispielsweise lediglich in die Konsole ausschreiben, dass sie aufgerufen wurden. Du machst davon mit print "*Factory expired*" Gebrauch.

Beim Hinzufügen eines Actors ins GameGrid mit *addActor()* wird der Bildpuffer automatisch auf

den Bildschirm gerendert, damit der Actor sofort sichtbar ist. Falls der Simulationszyklus gestartet ist, wird das Rendern sowieso in jedem Zyklus ausgeführt. Darum sollte in diesem Fall besser **addActorNoRefresh()** verwendet werden, denn zu häufiges Rendern kann zu Bildschirmflackern führen.

ZUSAMMENBAU UND KOLLISIONEN BEHANDELN

Die beiden eben geschriebenen Programmteile könnten auch von zwei Arbeitsgruppen entwickelt worden sein. Die nächste Aufgabe besteht darin, diese Teile zusammenführen, was nicht immer ganz leicht ist. Ist aber wie hier der **Programmstil einheitlich** und der Code weitgehend **entkoppelt**, so wird dadurch das "Mergen" des Codes wesentlich erleichtert

Als neue Funktionalität soll zudem das Halbieren der Früchte beim Zusammentreffen mit einem Pfeil eingebaut werden. Wir haben dies bereits vorbereitet, da ja die Früchte zwei Spritebilder haben, eines für die ganze und eines für die halbierte Frucht.

Wie du weisst, werden Kollisionen zwischen Actors durch einen Kollisionsevent erfasst. Dazu legst du für jeden Actor fest, welche die möglichen Kollisionspartner sind. Überlege dir dazu das Folgende: Erzeugst du beim Abschiessen einen Pfeil, so sind alle gegenwärtig vorhandenen Früchte Kollisionspartner.

Vergiss aber nicht, dass während der Bewegung des Pfeils auch neue Früchte hinzukommen. Darum musst du beim Erzeugen einer Frucht auch alle vorhandenen Pfeile (vielleicht gibt es nur einen) als Kollisionspartner festlegen.

In *JGameGrid* kannst du **addCollisionActors()** eine ganze Liste (allerdings als *ArrayList*) von Actoren als Kollisionspartner übergeben. Mit *getActors(Klasse)* kriegst du eine Liste mit allen Actoren der angegeben Klasse, die du nach Umwandlung in eine *ArrayList* übergeben kannst.

```
from gamegrid import *
import random
import math

# ---------- class Fruit -----------------------
class Fruit(Actor):
    def  __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2)
        self.vx = vx
        self.vy = 0
        self.isSliced = False

    def reset(self): # Called when Fruit is added to GameGrid
        self.px = self.getX()
        self.py = self.getY()
```

```
    def act(self):
        self.movePhysically()
        self.turn(10)

    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            self.removeSelf()

    def sliceFruit(self):
        if not self.isSliced:
            self.isSliced = True
            self.show(1)

    def collide(self, actor1, actor2):
       actor1.sliceFruit()
       return 0

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", 2)
        self.vx = vx

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
class Strawberry(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------- class FruitFactory ------------------
class FruitFactory(Actor):
    myCapacity = 0
    myFruitFactory = None
    nbGenerated = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
        return FruitFactory.myFruitFactory

    def act(self):
        self.createRandomFruit()

    def createRandomFruit(self):
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            print "Factory expired"
            return

        vx = -(random.random() * 20 + 30)
        r = random.randint(0, 2)
        if r == 0:
            fruit = Melon(vx)
        elif r == 1:
            fruit = Orange(vx)
```

```
        else:
            fruit = Strawberry(vx)
        FruitFactory.nbGenerated += 1
        y = int(random.random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)
        # for a new fruit, the collision partners are all existing darts
        fruit.addCollisionActors(toArrayList(getActors(Dart)))

# ------------------- class Crossbow -----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()

    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
        dx = math.cos(math.radians(self.getDirectionStart()))
        self.vx = self.speed * dx
        dy = math.sin(math.radians(self.getDirectionStart()))
        self.vy = self.speed * dy

    def act(self):
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy, self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

    def collide(self, actor1, actor2):
        actor2.sliceFruit()
        return 0

# ------ End of class definitions --------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
        crossbow.show(1) # crossbow is released
        dart = Dart(100)
        addActorNoRefresh(dart, crossbow.getLocation(),
                                crossbow.getDirection())
        # for a new dart, the collision partners are all existing fruits
        dart.addCollisionActors(toArrayList(getActors(Fruit)))


FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
```

```
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
addActor(factory, Location(0, 0))  # needed to run act()
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Hast du mit *addCollisionActor()* oder *addCollisionActors()* die Kollisionspartner deines Actors angegeben, so musst du in der Klasse des Actors die Methode *collide()* einfügen, die bei jeder Kollision automatisch aufgerufen wird. Der Rückgabewert muss ein Integer sein, der sagt, wie manchen Simulationszyklus die Kollisionen inaktiv sind (hier 0 sein). Eine Zahl grösser als 0 ist manchmal nötig, damit sich die beiden Partner wieder voneinander entfernen, bevor Kollisionen aktiv werden.

Beachte auch, dass in **collide(self, actor1, actor2))** *actor1* der Actor der Klasse ist, in der *collide()* definiert ist.

Als Kollisionsgebiete sind standardmässig die umgebenden Rechtecke des Spritebildes aktiv (Sie werden natürlich bei der Rotation der Aktoren mitgedreht). Für den Pfeil könntest du auch mit

```
setCollisionCircle(Point(20, 0), 10)
```

einen Kreis an der Pfeilspitze als Kollisionsgebiet festlegen, damit Früchte, die mit dem hinteren Teil des Pfeils kollidieren, nicht halbiert werden.

SPIELZUSTAND ANZEIGEN UND GAME-OVER BEHANDELN

Zum Dessert verfeinerst du den Code noch, indem du einen Game-Score und Benutzerinformation einbaust. Am einfachsten schreibst du sie in einer Statusbar aus.

Wie du bereits weisst, ist es günstig, im Hauptteil des Programms einen Game-Supervisor zu implementieren. Dieser soll die Anzahl der getroffenen und verpassten Früchte ausschreiben und das Spiel beenden, wenn die Früchtefabrik ihre Kapazität erreicht ist. Er zeigt also den Endstand an, erzeugt einen Game-Over-Actor und verhindert das Weiterspielen.

```
from gamegrid import *
import random
import math

# ---------- class Fruit -----------------------
class Fruit(Actor):
    def  __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2)
        self.vx = vx
        self.vy = 0
        self.isSliced = False

    def reset(self): # Called when Fruit is added to GameGrid
        self.px = self.getX()
        self.py = self.getY()

    def act(self):
        self.movePhysically()
        self.turn(10)
```

```
    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            if not self.isSliced:
                FruitFactory.nbMissed += 1
            self.removeSelf()

    def sliceFruit(self):
        if not self.isSliced:
            self.isSliced = True
            self.show(1)
            FruitFactory.nbHit += 1

    def collide(self, actor1, actor2):
       actor1.sliceFruit()
       return 0

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", 2)
        self.vx = vx

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
class Strawberry(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------- class FruitFactory ------------------
class FruitFactory(Actor):
    myCapacity = 0
    myFruitFactory = None
    nbGenerated = 0
    nbMissed = 0
    nbHit = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
        return FruitFactory.myFruitFactory

    def act(self):
        self.createRandomFruit()

    @staticmethod
    def createRandomFruit():
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            return
        vx = -(random.random() * 20 + 30)
        r = random.randint(0, 2)
        if r == 0:
            fruit = Melon(vx)
        elif r == 1:
```

```
            fruit = Orange(vx)
        else:
            fruit = Strawberry(vx)
        FruitFactory.nbGenerated += 1
        y = int(random.random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)
        fruit.addCollisionActors(toArrayList(getActors(Dart)))

# ------------------- class Crossbow -----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()

    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
        dx = math.cos(math.radians(self.getDirectionStart()))
        self.vx = self.speed * dx
        dy = math.sin(math.radians(self.getDirectionStart()))
        self.vy = self.speed * dy

    def act(self):
        if isGameOver:
            return
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy,self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

    def collide(self, actor1, actor2):
        actor2.sliceFruit()
        return 0

# ------ End of class definitions --------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if isGameOver:
            return
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
        crossbow.show(1) # crossbow is released
        dart = Dart(100)
        addActorNoRefresh(dart,crossbow.getLocation(),crossbow.getDirection())
        # for a new dart, the collision partners are all existing fruits
        dart.addCollisionActors(toArrayList(getActors(Fruit)))


FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
```

```
g = 9.81
isGameOver = False

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
addStatusBar(30)
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
addActor(factory, Location(0, 0))  # needed to run act()
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()

while not isDisposed() and not isGameOver:
   # Don't show message if same
   oldMsg = ""
   msg="#hit:"+str(FruitFactory.nbHit)+" #missed:"+str(FruitFactory.nbMissed)
   if  msg != oldMsg:
        setStatusText(msg)
        oldMsg = msg
   if FruitFactory.nbHit + FruitFactory.nbMissed == FACTORY_CAPACITY:
       isGameOver = True
       removeActors(Dart)
       setStatusText("You smashed " + str(FruitFactory.nbHit) + " out of "
       + str(FACTORY_CAPACITY) + " fruits")
       addActor(Actor("sprites/gameover.gif"), Location(300, 200))

   delay(100)
```

MEMO

Bei Game-Over sollten die meisten Benutzeraktionen verboten werden. Am einfachsten ist es, ein Flag *isGameOver = True* einzuführen, mit dem man die Aktionen durch vorzeitiges *return* in den betreffenden Funktionen und Methoden unterbindet. Es soll möglich bleiben, auch bei Game-Over die Armbrust zu bewegen, aber verboten sein, damit zu schiessen.

AUFGABEN

1. Zähle die Anzahl Pfeile und beschränke sie auf eine sinnvolle Maximalzahl. Ist diese erreicht, soll das Spiel ebenfalls beendet werden. Füge entsprechende Statusangaben hinzu.

2. Füge einen Punktescore für das Halbieren der Früchte hinzu:

   Melone: 5 Punkte
   Orange: 10 Punkte
   Erdbeere: 15 Punkte

3. Wir nach Game-Over die Enter-Taste gedrückt, so soll das Spiel von neuem beginnen.

4. Erweitere oder modifiziere das Spiel nach eigenen Ideen.

Wichtigste Methoden der Klassenbibliothek JGameGrid

Module import: from gamegrrid import *

Klasse GameGrid (globale Funktionen nach Aufruf von makeGameGrid)

Methode	Aktion
GameGrid(nbHorzCells, nbVertCells, cellSize, color)	erzeugt ein Spielfenster mit der gegebenen Anzahl horizontalen und vertikalen Zellen, Zellengrösse, mit sichtbaren Gitterlinien in gegebener Farbe, mit Navigationsbalken
GameGrid(nbHorzCells, nbVertCells, cellSize, color, bgImagePath)	erzeugt ein Spielfenster mit der gegebenen Anzahl horizontalen und vertikalen Zellen, Zellengrösse, mit Gitterlinien, mit Hintergrundbild, mit Navigationsbalken
GameGrid(nbHorzCells, nbVertCells, cellSize, None, bgImagePath, False)	erzeugt ein Spielfenster mit der gegebenen Anzahl horizontalen und vertikalen Zellen, Zellengrösse, ohne Gitterlinien, mit Hintergrundbild ohne Navigationsbalken
act()	wird nach dem Start des Simulationszyklus periodisch aufgerufen
addActor(actor, location)	fügt den Actor an der gegebenen Position zum Spielfenster hinzu
addKeyListener(listener)	registriert den Tastaturlistener
addMouseListener(listener, mouseEventMask)	registriert den Mauslistener
addStatusbar(height)	fügt ein Statusfenster zum Gamegrid
delay(time)	wartet definierte Zeit (in Millisec)
doPause()	unterbricht den Simulationszyklus
doStep()	führt die Simulation Schritt für Schritt durch
doReset()	setzt alle Actor die Startposition und startet die Simulation neu
doRun()	startet den Simulationszyklus
getActors(Actor class)	gibt alle Actors der gegebenen Klasse in einer Liste zurück
getBg()	gibt die Referenz auf GGBackground zurück
getBgColor()	gibt die Hintergrundfarbe zurück
getKeyCode()	gibt den Tastaturcode der letzten gedrückten Taste zurück
getOneActorAt(location)	gibt den Actor in der geg. Zelle zurück (Null, falls keiner)
getOneActor(Actor class)	gibt den Actor der geg. Klasse zurück (Null, falls keiner)
getRandomEmptyLocation()	gibt eine zufällige leere Zellenlocation zurück
getRandomLocation()	gibt eine zufällige Zellenlocation zurück
hide()	versteckt das Spielfenster ohne es zu schliessen
isAtBorder(location)	gibt *True* zurück, wenn sich die Zelle am Rand des Spielfensters befindet
isEmpty(location)	gibt *True* zurück, wenn die Zelle leer ist
isInGrid(location)	gibt *True* zurück, wenn sich die Zelle innerhalb des Spielfensters befindet
kbhit()	ergibt *True*, wenn eine Taste gedrückt wurde
toLocation(x, y)	gibt die Zelle mit den Pixel-Koordinaten x und y zurück

openSoundPlayer("wav/ping.wav")	stellt eine Sounddatei bereit. Folgende Sounds sind im tigerjython.jar vorhanden: bird.wav, boing.wav, cat.wav, click.wav,explore.wav,frog.wav. notify.wav, boing.wav
play()	spielt den bereitgestellten Sound ab
refresh()	aktualisiert das Spielfenster
registerAct(onAct)	registriert den Callback onAct, der in jedem Simulationszyklus aufgerufen wird
registerNavigation(started=onStart, stepped=onStep, paused=onPause, resetted = onReset, periodChanged = onPerionChange)	registriert die Callbacks onStart, onStep, onPause, onReset, onPeriodChange, die bei angezeigtem Navigationsbalken aufgerufen werden (nicht alle nötig)
removeActor (actor)	entfernt den Actor vom Spielfenster
removeActorsAt(location)	entfernt alle Actors, die sich in der angegebenen Zelle befinden
removeAllActors()	entfernt alle Actors vom Spielfenster
reset()	setzt die definierte Simulation in die Ausgangsposition zurück, mit Ausnahme der Actors, welche entfernt worden sind
show()	zeigt das Spielfenster an
setBgColor(color)	setzt die Hintergrundfarbe
setSimulationPeriod (milisec)	setzt die Periode des Simulationsloops
setStatusTest(text)	setzt den Test in die Statusbar
setTitle(text)	setzt den Titel in der Fenster-Titelleiste

Klasse Actor

Actor(spritepath)	erzeugt einen Actor mit dem gegebenen Spritebild
Actor(True, spritepath)	erzeugt einen rotierbaren Actor mit dem gegebenen Spritebild
Actor(spritepath, nbSprites)	erzeugt einen Actor mit mehreren Sprites
Actor(True, spritepath, nbSprites)	erzeugt einen rotierbaren Actor mit mehreren Spritebildern
act()	wird nach dem Start des Simulationszyklus periodisch aufgerufen
addActorCollisionListener(listener)	registriert den Kollisionslistener
addCollisionActor(actor)	registriert den Kollisionspartner
addMouseTouchListener (listener)	registriert den MouseTouchListener
collide(actor1, actor2)	Callback beim Auftreten einer Kollision, gibt die Anzahl der Simulationszyklen zurück, während den weitere Events unterdrückt werden
getCollisionActors()	gibt eine Liste der Kollisionskandidaten zurück
getDirection()	gibt die Bewegungsrichtung zurück
getIdVisible()	gibt Id des sichtbaren Sprites zurück
getNeighbours(distance)	gibt eine Liste aller Actors zurück, die sich in der gegebenen Distanz befinden
getNextMoveLocation (location)	gibt die *location* nach dem nächsten *move()* zurück
getX()	gibt die aktuelle horizontale Zellenkoordinate zurück
getY()	gibt die aktuelle vertikale Zellenkoordinate zurück
hide()	macht den Actor unsichtbar, entfernt ihn aber nicht. Nach *reset()* wird er wieder sichtbar

isInGrid()	gibt *True* zurück, wenn sich der Actor innerhalb des Spielfensters befindet
isHorzMirror()	gibt *True* zurück, wenn die Figur horizontal gespiegelt ist
isVertMirror()	gibt *True* zurück, wenn die Figur vertikal gespiegelt ist
isMoveValid()	gibt *True* zurück, wenn ein *move()* den Actor innerhalb des Spielfensters belässt
isNearBorder()	gibt *True* zurück, wenn sich der Actor am Rand des Spielfensters befindet
isVisible()	gibt *True* zurück, wenn der Actor sichtbar ist
move()	bewegt den Actor mit der aktuellen Richtung in eine benachbarte Zelle
move(distance)	bewegt den Actor um die gegebene Distanz
reset()	aufgerufen, wenn der Actor zum Gamegrid hinzugefügt od. Reset-Button gedrückt wird
setCollisionCircle (spriteId,center, radius)	legt den Kreis innerhalb des Actors fest, der für Kollisionen verwendet wird
setCollisionLine(spriteId, startPoint, endPoint)	legt eine Linie innerhalb des Actors fest, die für Kollisionen verwendet wird
setCollisionRectangle(spriteId, center, width, height)	legt das Rechteck innerhalb des Actors fest, das für Kollisionen verwendet wird
setCollisionSpot(spriteId, spot)	legt den Punkt innerhalb des Actors fest, der für Kollisionen verwendet wird
setHorzMirror(True)	spiegelt das Bild horizontal
setVertMirror(True)	spiegelt das Bild vertikal
setSlowDown(factor)	verlangsamt den Aufruf der Methode *act()* des Actors mit dem gegebenen Faktor
setLocation(location)	setzt den Actor in die gegeben Zelle
setLocationOffset(point)	verschiebt die Mitte des Spritebild relativ zur Zellenmitte (location nicht verändert)
setPixelLocation(location)	setzt den Actor auf die gegebene Pixelkoordinate (location/offset werden angepasst)
setX(x)	setzt die x-Koordinate auf den angegebenen Wert
setY(y)	setzt die y-Koordinate auf den angegebenen Wert
show()	Sprite mit der id 0 wird sichtbar
show(spriteId)	Sprite mit der angegebener id wird sichtbar
showNextSprite ()	zeigt das nächste Sprite-Bild
showPreviousSprite()	zeigt das vorangehende Sprite-Bild
removeSelf()	der Actor wird vernichtet. Nach *reset()* erscheint er nicht mehr
reset()	wird aufgerufen und wenn der Reset-Button geklickt wird
turn(angle)	ändert die Bewegungsrichtung um den gegebenen Winkel (in Grad im Uhrzeigersinn)

Klasse Location

Location(x, y)	erzeugt Location Objekt mit gegebenen horizontalen und vertikalen Zellenkoordinaten

Location(location)	erzeugt Location Objekt mit der gegebenen Location (clone)
clone()	gibt neue Location mit den gleichen Koordinaten zurück
equals(location)	gibt True zurück, falls die aktuelle Location identisch mit der übergegebenen ist
get4CompassDirectionTo(location)	gibt eine Liste mit 4 benachbarten Location (WEST, EAST, NORTH, SOUTH) zurück
getCompassDirectionTo(location)	gibt eine Liste mit 8 benachbarten Locations (auch diagonal)
getDirectionTo(location)	gibt die Richtung von der aktuellen zu der gegebenen Position in Grad zurück
getNeighbourLocation(direction)	gibt eine der 8 benachbarten Zellen zurück. Es wird die Zelle genommen, die am nächsten bei der gegebener Richtung liegt
getNeighbourLocations(distance)	gibt Liste mit allen Zellen mit Zentren innerhalb der gegebenen Distanz zurück
getX()	gibt die aktuelle horizontale Zellenkoordinate zurück
getY()	gibt die aktuelle vertikale Zellenkoordinate zurück

Klasse GGBackground

clear()	löscht den Hintergrund und übermal ihn mit Hintergrundfarbe
clear(color)	löscht den Hintergrund und übermalt ihn mit der geg. Farbe
drawCircle(center, radius)	zeichnet einen Kreis mit dem gegebenen Mittelpunkt und Radius
drawLine(x1,y1, x2, y2)	zeichnet eine Strecke mit gegebenen Endpunkten
drawLine(pt1, pt2)	zeichnet eine Strecke mit den gegebenen Endpunkten
drawPoint(pt)	zeichnet einen Punkt
drawRectangle(pt1, pt2)	zeichnet ein Rechteck mit den gegebenen Diagonaleckpunkten
drawText(text, pt)	schreibt Text an die Position mit dem gegeben Anfangspunkt
fillCell(location, color)	füllt die gegebene Zelle mit der gegebenen Farbe
fillCircle(center,radius)	zeichnet gefüllten Kreis mit dem gegebenen Mittelpunkt und Radius
getBgColor()	gibt die aktuelle Hintergrundfarbe zurück
getColor(location)	gibt die Hintergrundfarbe im Zentrum einer Zelle zurück
save()	speichert den aktuellen Hintergrund
setBgColor(color)	ändert die Hintergrundfarbe
setFont(font)	setzt die Schriftart
setLineWidth(width)	setzt die Liniendicke
setPaintColor(color)	setzt die Zeichnungsfarbe
setPaintMode()	es wird ohne Rücksicht auf den vorhandenen Hintergrund gezeichnet
setXORMode(color)	bei zweimaligen Zeichnen wird er den Hintergrund wieder herstellt
restore()	holt den vorher gespeicherten Hintergrund zurück

kapitel acht

COMPUTEREXPERIMENTE

Lernziele

- Du kannst einfache Probleme aus der Stochastik mit einer Computersimulation unter Verwendung von Zufallszahlen lösen.
- Du verstehst, dass Zufallsexperimente statistischen Schwankungen unterworfen sind und kannst Resultate als Häufigkeitsverteilung darstellen und interpretieren.
- Du kannst mit einer Computersimulation eine Stichprobe nach ihrer Signifikanz untersuchen und kennst den Begriff des Chi-Quadrat-Tests.
- Du weisst, wie man den Computer zur Simulation von Populationen einsetzt.
- Du weisst, was die Mandelbrot-Menge ist und wie man sie grafisch darstellt.
- Du weisst, was Grund- und Obertöne sind und kennst den Begriff des Spektrums.

"Trau' keiner Statistik, die du nicht selbst gefälscht hast."

Winston Churchill zugeschrieben

8.1 SIMULATIONEN

EINFÜHRUNG

In der Forschung und Industrie, aber auch in der Finanzwelt spielen Computersimulationen eine wichtige Rolle. Dabei wird das Verhalten eines realen Systems mit Hilfe eines Computers nachgebildet. Computersimulationen haben gegenüber realen Experimenten und Untersuchungen den Vorteil, dass sie kostengünstig und umweltschonend durchführbar, sowie ungefährlich sind. Allerdings können sie die Wirklichkeit meist nie exakt wiedergeben. Die Gründe dafür sind vielfältig:

- die Wirklichkeit kann wegen Messfehlern nie exakt in Zahlen gefasst werden (ausser bei Abzählungen)
- das Zusammenwirken der Komponenten ist oft nicht exakt bekannt, da die zu Grunde liegenden Gesetze nicht exakt sind [**mehr...**] oder nicht alle Einflüsse berücksichtigt werden [**mehr...**]

Immerhin werden Computersimulationen mit steigender Rechenleistung der Computer immer präziser, man denke etwa an die Wetterprognosen für die nächsten Tage.

Der Zufall spielt in unserem Leben eine ausserordentlich grosse Rolle, denn viele Entscheide fällen wir nicht auf Grund rein logischer Argumente, sondern durch eine intuitive Abschätzung von Wahrscheinlichkeiten. Die Verwendung des Zufalls kann aber auch Probleme mit exakten Lösungen stark vereinfachen. So ist es beispielsweise sehr zeitaufwendig, auf einer Strassenkarte mit vielen Verbindungsmöglichkeiten den kürzesten Weg von A nach B mit einem Algorithmus exakt zu bestimmen; für die Praxis genügend ist es, den mit grosser Wahrscheinlichkeit kürzesten Weg zu finden [**mehr...**]

PROGRAMMIERKONZEPTE:*Computersimulation, Computerexperiment, Statistische Schwankungen*

DER COMPUTER ALS SPIELPARTNER

Deine Kameradin Nora schlägt dir folgendes Spiel vor: "Du darfst drei Würfel werfen. Treten dabei Sechser auf, so hast du gewonnen und ich gebe dir eine Murmel. Wenn keine Sechser vorkommen, habe ich gewonnen und du gibst mir eine Murmel".

Auf den ersten Blick scheint dir das Spiel fair zu sein, denn du überlegst rasch, dass für jeden Würfel die Wahrscheinlichkeit eines Sechser 1/6 beträgt, also die Wahrscheinlichkeit, im ersten ersten, zweiten oder dritten Wurf eine Sechs zu haben 1/6 + 1/6 + 1/6 = 1/2 ist.

Mit dem Computer und deinen Programmierkenntnissen kannst du diese Überlegung nachprüfen. Dabei gehst du davon aus, dass es gleichgültig ist, ob du die 3 Würfe hintereinander oder miteinander machst oder anders gesagt, dass die Wahrscheinlichkeit, mit einem Würfel eine bestimmte Zahl zu erhalten, unabhängig von den anderen Würfeln immer 1/6 ist.

Es gibt zwei Möglichkeiten das Problem anzupacken, entweder **statistisch** oder **kombinatorisch.** Die statistische Lösung entspricht dem realen Spiel. Du simulierst das Werfen der Würfel, indem du im Programm **oftmals** 3 Zufallszahlen zwischen 1 und 6 erzeugst und die

Gewinnfälle zählst.

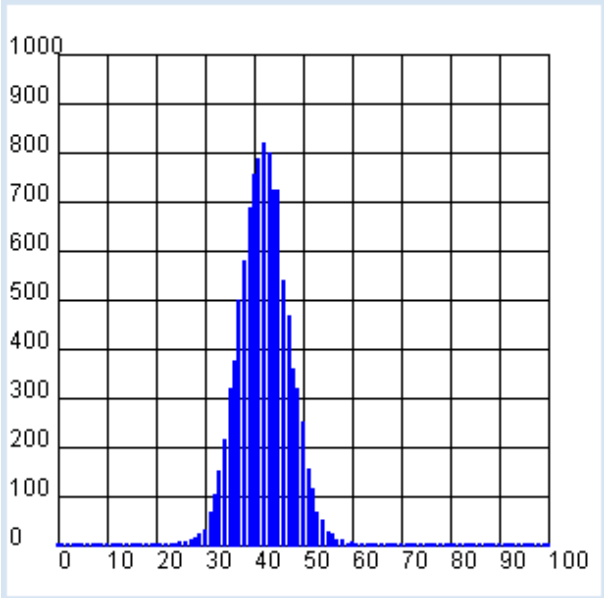
```
from random import randint

n = 1000 # Anzahl Spiele
gewonnen = 0
repeat n:
    a = randint(1, 6)
    b = randint(1, 6)
    c = randint(1, 6)
    if a == 6 or b == 6 or c == 6:
        gewonnen += 1

print "Gewonnen in:", gewonnen, " von", n, " Spielen"
print "Meine Gewinnwahrscheinlichkeit:", gewonnen / n
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Es ergibt sich eine Gewinnwahrscheinlichkeit von ungefähr 0.42 und nicht 0.5, wie du vermutet hast. Der Wert ändert aber von Simulation zu Simulation leicht, denn er ist **statistischen Schwankungen** unterworfen. Wie du intuitiv vermutest, ist das Resultat umso **genauer**, **je mehr** Versuche du machst.

Die statistischen Schwankungen sind bei Computersimulationen von grosser Wichtigkeit.

Um sie zu untersuchen, führst du den Versuch mit gewollt wenig Spielen (sagen wir 100) oftmals durch (sagen wir 10000 mal) und zeichnest ein **Häufigkeitsdiagramm** der gewonnenen Spiele. Es ergibt sich eine interessante, für die Statistik typische, glockenförmige **Verteilung**.



Im Programm verwendest du ein *GPanel* als Grafikfenster. Du kannst mit **drawGrid()** auch ein Koordinatengitter anzeigen. Einen einzelnen Hunderter-Versuch führst du mit der Funktion **sim()** aus, welche die Zahl der gewonnenen Spiele zurückgibt, deren Schwankung du untersuchen willst.

```
from gpanel import *
from random import randint

z = 10000
n = 100

def sim():
    gewonnen = 0
    repeat n:
        a = randint(1, 6)
        b = randint(1, 6)
        c = randint(1, 6)
        if a == 6 or b == 6 or c == 6:
            gewonnen += 1
    return gewonnen

makeGPanel(-10, 110, -100, 1100)
drawGrid(0, 100, 0, 1000)
h = [0] * (n + 1)
title("Simulation gestartet. Biite warten...")
repeat z:
```

```
    x = sim()
    h[x] += 1
title("Simulation beendet")

lineWidth(2)
setColor("blue")
for x in range(n + 1):
    line(x, 0, x, h[x])
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Das Maximum der Verteilung liegt bei etwa 42, da ja die Wahrscheinlichkeit zu gewinnen etwa 0.42 ist und man 100 Spiele macht. Spielst du mit Nora 100 Mal, so kann es trotz deiner Gewinnchance von nur 0.42 durchaus sein, dass du in über 50 Fällen das Spiel gewinnst. Die Wahrscheinlichkeit dafür ist aber gering (ca. 5 %) und darum ist das Spiel nicht fair.

Computerexperimente mit Zufallszahlen unterliegen statistischen Schwankungen, die umso kleiner werden, je grösser die Anzahl der Versuche ist.

Für die kombinatorische Lösung lässt du den Computer alle möglichen Würfe mit 3 Würfeln ausführen. Der erste Wurf kann die Zahlen 1 bis 6 ergeben, ebenfalls der zweite und der dritte. In der geschachtelten for-Schleife bildest du also alle Zahlentripel und zählst mit der Variablen *moegliche* alle Möglichkeiten und mit der Variablen *guenstige* solche, die mindestens eine 6 enthalten, bei denen du also gewinnst.

```
moegliche = 0
guenstige = 0
for i in range(1, 7):
    for j in range(1, 7):
        for k in range(1, 7):
            moegliche += 1
            if i == 6 or j == 6 or k == 6:
                guenstige += 1
print "günstige:", guenstige, "mögliche:", moegliche
print "Meine Gewinnwahrscheinlichkeit:", guenstige / moegliche
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Es ergeben sich 91 günstige von 216 möglichen Fällen und damt eine Gewinnwahrscheinlichkeit *w = guenstige / moegliche* von 91/216 = 0.42. Diesen Wert hast du auch mit der Computersimulation erhalten.

ZUSATZSTOFF

Du kannst natürlich die Aufgabe auch ganz ohne Computer lösen. Dazu überlegst du Folgendes: Es gibt für den Gewinn drei Ereignisse E1, E2, E3:

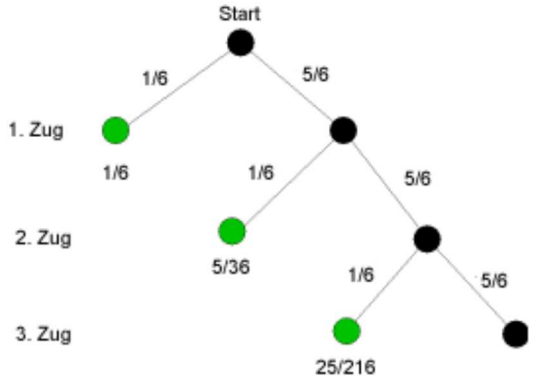
E1: Beim ersten Wurf eine 6 werfen. Wahrscheinlichkeit: 1/6

E2: Beim ersten Wurf keine 6, aber im 2. Wurf eine 6 zu werfen.
Wahrscheinlichkeit: 5 /6 * 1/6

E3: Beim ersten und zweiten Wurf keine 6, aber im 3. Wurf eine 6 zu werfen.
Wahrscheinlichkeit: 5/6 * 5 /6 * 1/6

Da E1, E2 und E3 unabhängig sind, ist die Wahrscheinlichkeit die Summe, also 1 /6 + 5 /36 + 25 /216 = 91/216 = 0.421296.

Du kannst den Prozess baumartig darstellen:



Für die Lösung gibt es auch noch einen *Königsweg*: Die Wahrscheinlichkeit, in den drei Würfen keine 6 zu werfen ist p = 5/6 * 5/6 * 5/6 = 125/216. Die gesuchte Wahrscheinlichkeit also w = 1 - p = 91/216.

AUFGABEN

1. Der Herzog Ferdinando de Medici von Florenz stellte um 1600 fest, dass man beim Werfen von drei Würfeln die Augensumme 9 und 10 mit einer gleich grossen Zahl von Würfelaugen erhalten kann:

Augensumme 9	**Augensumme 10**
1 + 2 + 6	1 + 3 + 6
1 + 3 + 5	1 + 4 + 5
2 + 2 + 5	2 + 2 + 6
2 + 3 + 4	2 + 3 + 5
3 + 3 + 3	2 + 4 + 4

   Der Herzog stellte aber fest, dass beim Dreierwurf die Augensummen 9 und 10 nicht gleichwahrscheinlich sind und fragte den Mathematikprofessor Galileo Galilei um Rat. Berechne diese Wahrscheinlichkeiten mit einer Computersimulation
   a. statistisch
   b. kombinatorisch

2. Bestimme mit einer statistischen Computersimulation die Wahrscheinlichkeit, dass in einer Klasse mit 20 Kindern (mindestens) zwei Kinder am gleichen Tag Geburtstag haben (keine Schaltjahre).
3. Der Chevalier de Méré befragte um 1650 in Paris den Mathematiker Blaise Pascal nach den Gewinnchancen der beiden folgenden Ereignisse:
   a. beim 4-maligen Würfeln mindestens eine Sechs werfen
   b. beim 24-maligen Würfeln mit jeweils 2 Würfeln mindestens eine Doppelsechs werfen.
   Er war der Meinung, dass die Gewinnchancen gleich seien, denn bei b) ist zwar die Gewinnwahrscheinlichkeit bei einem Einzelwurf 6 mal so klein, dafür wird der Versuch aber 6 mal so oft wiederholt. Hat er recht?

4*. Bestimme im Spiel mit Nora, wie gross die Wahrscheinlichkeit ist, dass du bei einem Hunderterspiel in mehr als 50 Fällen gewinnst.

8.2 POPULATIONEN

EINFÜHRUNG

Computersimulationen werden oft dafür verwendet, um ausgehend von einer Zeitaufnahme oder einer gewissen Zeitspanne in der nahen Vergangenheit Voraussagen über das Verhalten eines Systems in der Zukunft zu machen. Solche Prognosen können von grosser strategischer Bedeutung sein und uns beispielsweise in einem Szenario, das zu einer Katastrophe führt, zu rechtzeitigem Umdenken veranlassen. Heute besonders aktuelle Themen sind die Voraussage des Weltklimas und das Bevölkerungswachstum.

Unter einer Population versteht man ein System von Individuen, deren Zahl sich auf Grund von inneren Mechanismen, Wechselwirkungen und äusseren Einflüssen in Laufe der Zeit ändert. Sieht man von äusseren Einflüssen ab, so spricht man von einem abgeschlossenen System. Für viele Populationen ist die Veränderung der Populationsgrösse proportional zur aktuellen Grösse der Population. Aus der Zuwachsrate berechnet sich die Veränderung des aktuellen Werts mit:

Neuer Wert - alter Wert = alter Wert * Zuwachsrate * Zeitintervall

Weil links die Differenz des neuen Werts vom alten Wert steht, nennt man diese Beziehung eine **Differenzengleichung**. Die Zuwachsrate kann man auch als Zunahmewahrscheinlichkeit pro Individuum und Zeiteinheit auffassen. Ist sie negativ, so nimmt die Grösse der Population ab. Die Zuwachsrate kann sich durchaus im Laufe der Zeit ändern.

PROGRAMMIERKONZEPTE: *Differenzengleichung, Zuwachsrate, Exponentielles/Begrenztes Wachstum, Sterbetafeln, Alterspyramide, Räuber-Beute-System*

EXPONENTIELLES WACHSTUM

Bevölkerungsprognosen sind von grossem Interesse und können politische Entscheidungsprozesse massiv beeinflussen. Jüngstes Beispiel ist die Auseinandersetzung um die Regulierung des Ausländeranteils in der Wohnbevölkerung.

Du erhältst vom Bundesamt für Statistik (Quelle: **http://www.bfs.admin.ch**, Stichwort: STAT-TAB) die Einwohnerzahlen der Schweiz je für das Jahresende 2010 und 2011:

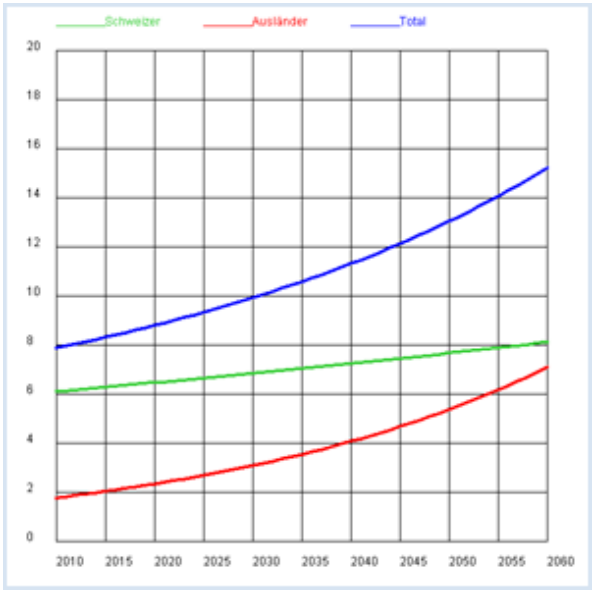
2010: Total z_0 = 7 870 134, davon Schweizer s_0 = 6 103 857

2011: Total z_1 = 7 954 662, davon Schweizer s_1 = 6 138 668

Kannst du daraus eine Prognose des Ausländeranteils für die nächsten 50 Jahre erstellen? Mit den Zahlen errechnest du zuerst die Zahl der Ausländer $a_0 = z_0 - s_0$ bzw. $a_1 = z_1 - s_1$ und damit die jährliche Zuwachsrate zwischen 2010 und 2011 für Schweizer und Ausländer

$$rs = \frac{s_1 - s_0}{s_0} = 0.57\% \qquad \text{bzw.} \qquad ra = \frac{a_1 - a_0}{a_0} = 2.81\%$$

Es ist nun für dich leicht, **unter der Voraussetzung, dass diese Zuwachsraten gleich gross bleiben**, die Bevölkerungszusammensetzung für die nächsten 50 Jahre zu untersuchen. Du kannst dies mit dem Taschenrechner, mit einem Tabellenkalkulationsprogramm oder mit Python machen. In einer grafischen Darstellung visualisierst du die errechneten Werte.




```
from gpanel import *

# Quelle: Bundesamt für Statistik, STAT-TAB
z2010 = 7870134 # Total 2010
z2011 = 7954662 # Total 2011
s2010 = 6103857 # Schweizer 2010
s2011 = 6138668 # Schweizer 2011

def drawGrid():
    # Horizontal
    for i in range(11):
        y = 2000000 * i
        line(0, y, 50, y)
        text(-3, y, str(2 * i))
    # Vertical
    for k in range(11):
        x = 5 * k
        line(x, 0, x, 20000000)
        text(x, -1000000, str(int(x + 2010)))

def drawLegend():
    setColor("lime green")
    y = 21000000
    move(0, y)
    draw(5, y)
    text("Schweizer")
    setColor("red")
    move(15, y)
    draw(20, y)
    text("Ausländer")
    setColor("blue")
    move(30, y)
    draw(35, y)
    text("Total")

makeGPanel(-5, 55, -2000000, 22000000)
title("Bevölkerungswachstum extrapoliert")
drawGrid()
drawLegend()

a2010 = z2010 - s2010 # Ausländer 2010
a2011 = z2011 - s2011 # Ausländer 2011

lineWidth(3)
setColor("blue")
line(0, z2010, 1, z2011)
setColor("lime green")
```

```
line(0, s2010, 1, s2011)
setColor("red")
line(0, a2010, 1, a2011)

rs = (s2011 - s2010) / s2010  # Zuwachsrate Schweizer
ra = (a2011 - a2010) / a2010  # Zuwachsrate Ausländer

# Iteration
s = s2011
a = a2011
z = s + a
sOld = s
aOld = a
zOld = z
for i in range(0, 49):
    s = s + rs * s  # Modellannahme
    a = a + ra * a  # Modellannahme
    z = s + a
    setColor("blue")
    line(i + 1, zOld, i + 2, z)
    setColor("lime green")
    line(i + 1, sOld, i + 2, s)
    setColor("red")
    line(i + 1, aOld, i + 2, a)
    zOld = z
    sOld = s
    aOld = a
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

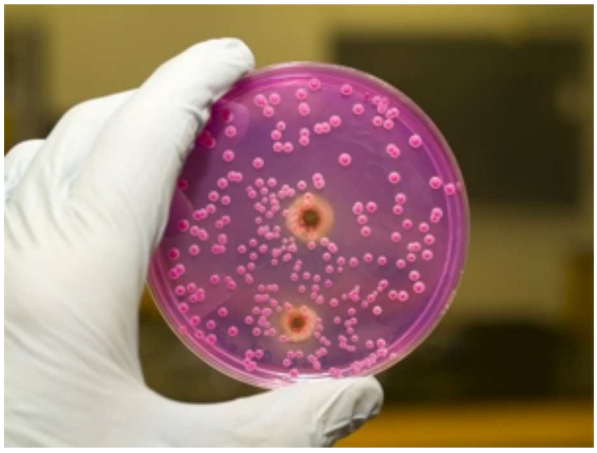
Wie du aus den Zahlen entnehmen kannst, steigt der Ausländeranteil von 2010 bis 2035, also innert 25 Jahren, auf das Doppelte und in weiteren 25 Jahren auf das Vierfache. Die Populationsgrösse nimmt offenbar bei konstanter Zuwachsrate weit überproportional zu. Bezeichnest du mit T die Verdoppelungszeit, so gilt offenbar mit der Anfangsgrösse A für die Populationsgrösse y nach der Zeit t

$$y = A * 2^{\frac{t}{T}}$$

Da die Zeit im Exponent steht, nennt man diese rasante Zunahme ein **exponentielles Wachstum**.

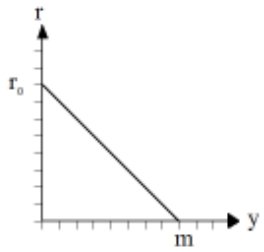
BEGRENZTES WACHSTUM

Viele Populationen befinden sich in einer Umgebung mit beschränkten Ressourcen. Dem rasanten exponentiellen Anstieg mit einer konstanten Zuwachsrate r sind daher Grenzen gesetzt. Bereits vor rund 100 Jahren hat der Biologe Carlson für eine Hefebakterien-Kultur folgende Mengen (mg) nach je einer Stunde experimentell bestimmt:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
9.6	18.3	29.0	47.2	71.1	119.1	174.6	257.3	350.7	441.0	513.3	559.7	594.8	629.4	640.8	651.1	655.9	659.8	661.8

Den experimentellen Verlauf kannst du mit einem Modell verstehen, in dem das vorerst exponentielle Wachstum eine Sättigung erfährt. Dabei lässt du die Zuwachsrate mit zunehmender Populationsgrösse y linear abnehmen, bis sie bei einem bestimmten **Sättigungswert** m Null wird.



Wie du leicht durch Einsetzen von y = 0 und y = m nachprüfen kannst, gilt:

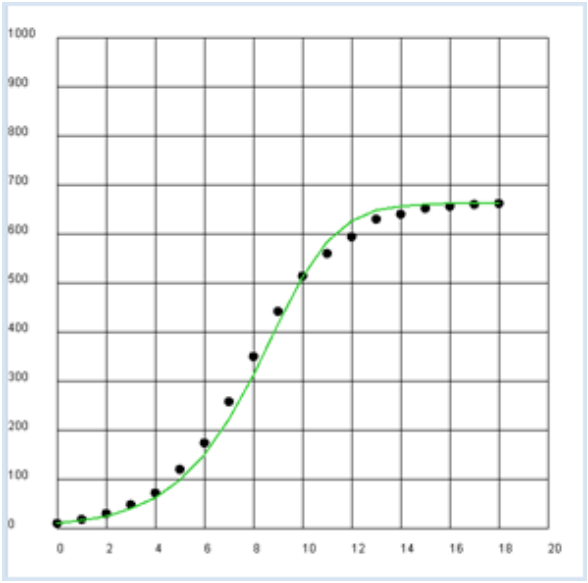
$$r = r_o * (1 - \frac{y}{m}) = \frac{r_o}{m} * (m - y)$$

Unter dieser Voraussetzung kannst du den zeitlichen Verlauf mit einem kurzen Programm grafisch darstellen, wo du auch die experimentellen Werte einzeichnest. Dazu iterierst du die Differenzengleichung, die du mit

dy: neuer Wert - alter Wert
y: alter Wert
dt: Zeitschritt
λ: Zuwachsrate

so schreiben kannst:

$$dy = y * r * dt = y * r_0 * (1 - \frac{r_0}{m}) * dt$$



Mit dem Anfangswert y_0 = 9.6 mg, der Sättigungsmenge m = 662 mg und der anfänglichen Zuwachsrate r_0 = 0.62 /h ergibt sich eine gute Übereinstimmung zwischen Theorie und Experiment.

```
from gpanel import *

z = [9.6, 18.3, 29.0, 47.2, 71.1, 119.1, 174.6, 257.3, 350.7, 441.0, 513.3,
559.7, 594.8, 629.4, 640.8, 651.1, 655.9, 659.6, 661.8]

def r(y):
    return r0 * (1 - y / m)

r0 = 0.62
y = 9.6
m = 662

makeGPanel(-2, 22, -100, 1100)
title("Bakterienwachstum")
drawGrid(0, 20, 0, 1000)
lineWidth(2)
for n in range(0, 19):
    move(n, z[n])
    setColor("black")
    fillCircle(0.2)
```

```
    if n > 0:
        dy = y * r(y)
        yNew = y + dy
        setColor("lime green")
        line(n - 1, y, n, yNew)
        y = yNew
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

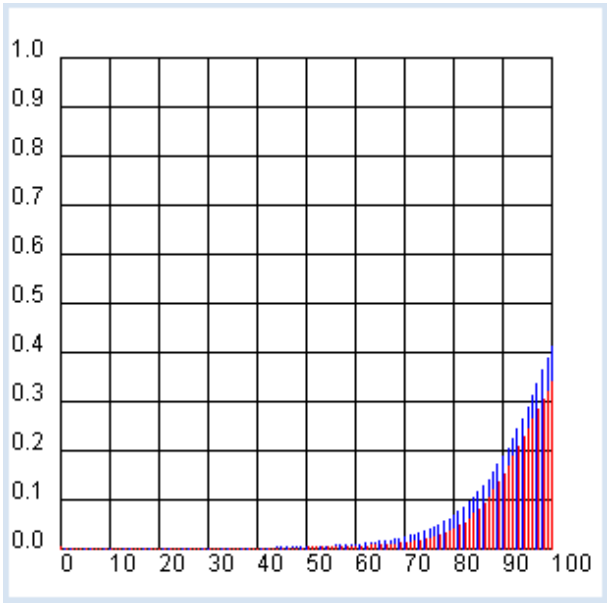
MEMO

Geht man von einer linearen Abnahme der Zuwachsrate aus, so ergibt sich für die Populationsgrösse eine typische S-kurvenartige Sättigungskurve (auch **logistisches Wachstum** oder Sigmoide genannt).

STERBETAFELN

Ein Mass für die Gesundheit einer Population ist die Wahrscheinlichkeit ein bestimmtes Altersjahr zu überleben bzw. in einem hohen Alter zu sterben. Willst du die Altersverteilung für die Schweizer Bevölkerung untersuchen, so verwendest du wieder aktuelle Daten vom Bundesamt für Statistik und zwar die sogenannten *Sterbetafeln* (Quelle: **http://www.bfs.admin.ch,** Stichwort: STAT-TAB) [**mehr...**]. Diese enthalten tabellarisch getrennt nach Männern und Frauen die beobachteten Wahrscheinlichkeiten qx und *qy*, in einem bestimmten Alter zu sterben. Ihre Bestimmung ist grundsätzlich einfach: Man betrachtet am Ende eines Jahres getrennt nach Männern und Frauen alle Todesfälle im vergangenen Jahr und bildet die Häufigkeit der 0-jährigen (gestorben zwischen Geburt und einem Altersjahr), der 1-jährigen, usw. Diese Zahlen teilt man nachher durch die Zahl in dieser Altersgruppe am Anfang des Jahres.

(Du kannst bei STAT-TAB eine Excel-Tabelle erstellen lassen und die Spalten für *qx* und *qy* in eine Textdatei *qx.dat* bzw. *qy.dat* kopieren oder die Dateien von **hier** herunterladen. Kopiere sie in das Unterverzeichnis *Lib* des Verzeichnis, in dem sich *tigerjython2.jar* befindet.) Im Programm liest du die Daten in eine Liste *qx* bzw. *qy* ein. Da die Zahlen zur besseren Lesbarkeit manchmal noch Leerzeichen und Apostroph enthalten, musst du diese entfernen. Zuerst erstellst du lediglich eine grafische Darstellung der eingelesenen Daten.




```
import exceptions
from gpanel import *

def readData(filename):
    table = []
    fData = open(filename)

    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
```

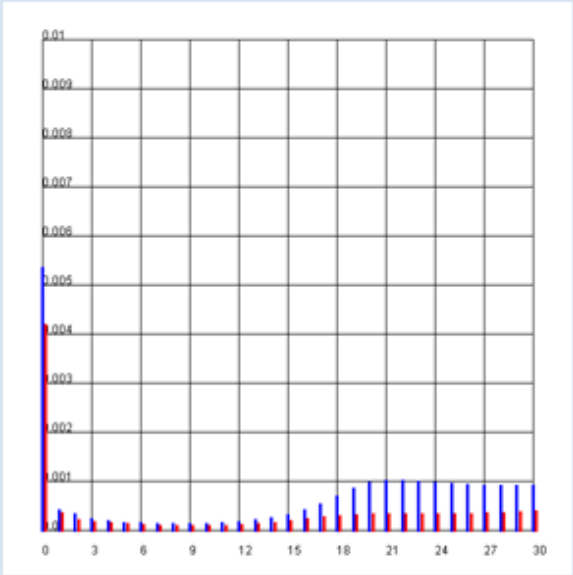
```
            q = float(line)
        except exceptions.ValueError:
            break
        table.append(q)
    fData.close()
    return table

makeGPanel(-10, 110, -0.1, 1.1)
title("Sterbewahrscheinlichkeit (Blau -> Männer, Rot -> Frauen)")
drawGrid(0, 100, 0, 1.0)
qx = readData("Lib/qx.dat")
qy = readData("Lib/qy.dat")
for t in range(101):
    setColor("blue")
    p = qx[t]
    line(t, 0, t, p)
    setColor("red")
    q = qy[t]
    line(t + 0.2, 0, t + 0.2, q)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

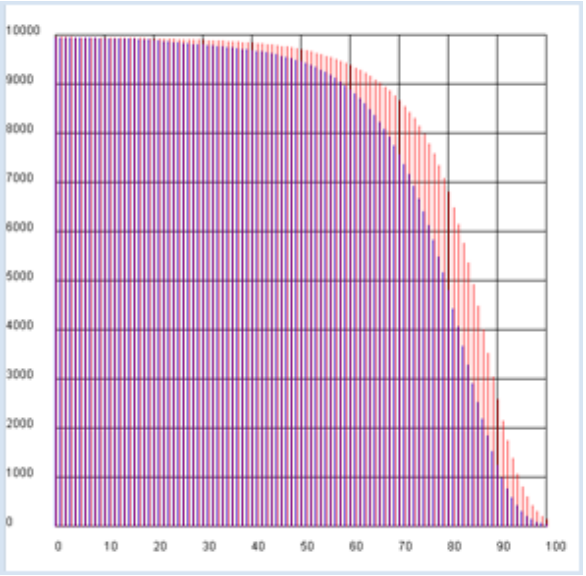
Der Verlauf zeigt deutlich, dass Frauen im Mittel älter als Männer werden. Interessant ist auch der Verlauf in den ersten 30 Lebensjahren.

So sterben im ersten Lebensjahr deutlich mehr Knaben als Mädchen, ebenfalls in der Altersgruppe der 15 bis 30-jährigen. Mache dir darüber deine eigenen Gedanken.



ZEITLICHE ENTWICKLUNG EINER POPULATION

Mit Hilfe der Sterbetafel und einem Computerprogramm kannst du viele interessante demographische Fragestellungen wissenschaftlich korrekt anpacken. Hier gehst du von 10000 Neugeborenen aus und untersuchst, wie sich diese Population in den nächsten 100 Jahren entwickelt. Du verwendest dabei die Werte *qx* und *qy* als negative Zuwachsraten



```
import exceptions
from gpanel import *

n = 10000 # Groesse der Population

def readData(filename):
    table = []
    fData = open(filename)

    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

makeGPanel(-10, 110, -1000, 11000)
title("Populationsverlauf (Blau -> Männer, Rot -> Frauen)")
drawGrid(0, 100, 0, 10000)
qx = readData("Lib/qx.dat")
qy = readData("Lib/qy.dat")
x = n # Männer
y = n # Frauen
for t in range(101):
    setColor("blue")
    rx = qx[t]
    x = x - x * rx
    line(t, 0, t, x)
    setColor("red")
    ry = qy[t]
    y = y - y * ry
    line(t + 0.2, 0, t + 0.2, y)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

LEBENSERWARTUNG VON FRAUEN UND MÄNNERN

In der vorhergehenden Untersuchung wurde nochmals deutlich, dass Frauen länger als Männer leben. Du kannst den Unterschied auch mit einer einzigen Grösse ausdrücken, die man **Lebenserwartung** nennt. Es handelt sich dabei um den Mittelwert des erreichten Alters.
Rufe dir kurz in Erinnerung, wie ein Mittelwert, beispielsweise der Notendurchschnitt für eine Schulklasse, definiert ist: Du bildest die Summe s der Noten aller Schüler und dividierst sie durch die Anzahl n der Schüler. Wenn du der Einfachheit davon ausgehst, dass nur die ganzzahligen Noten von 1 bis 6 vorkommen, so könntest du s auch so berechnen:

s = Schülerzahl mit Note 1 * 1 + Schülerzahl mit Note 2 * 2 + ... Schülerzahl mit Note 6 * 6

oder allgemeiner:

Mittelwert = Summe von (Häufigkeit des Werts * Wert) dividiert durch Totalzahl

Wenn du die Häufigkeiten aus einer Häufigkeitsverteilung h der Werte x (hier der Noten 1 bis 6) entnimmst, so nennt man den Mittelwert auch **Erwartungswert** und du kannst dafür

$$E = \frac{x_1 * h_1 + x_2 * h_2 + ... + x_n * h_n}{h_1 + h_2 + ... + h_n}$$

schreiben. Wie du siehst, werden in der Summe die Häufigkeiten h_i mit ihrem Wert x_i **gewichtet**.

Die Lebenserwartung ist nichts anderes als der Erwartungswert für das Alter, in dem Frauen bzw. Männer sterben. Um sie mit einer Computersimulation zu berechnen, gehst du von einer bestimmten Grösse von je n = 10000 Männern und Frauen aus und bestimmst die Zahl *hx* der Männer bzw. *hy* der Frauen, die im Alter zwischen t und t + 1 sterben. Offenbar lassen sich diese Zahlen aus der Grösse der Populationen x und y zur Zeit t, die du im vorhergehenden Programm berechnet hast, und den Sterberaten *rx* bzw. *ry* so ausdrücken:

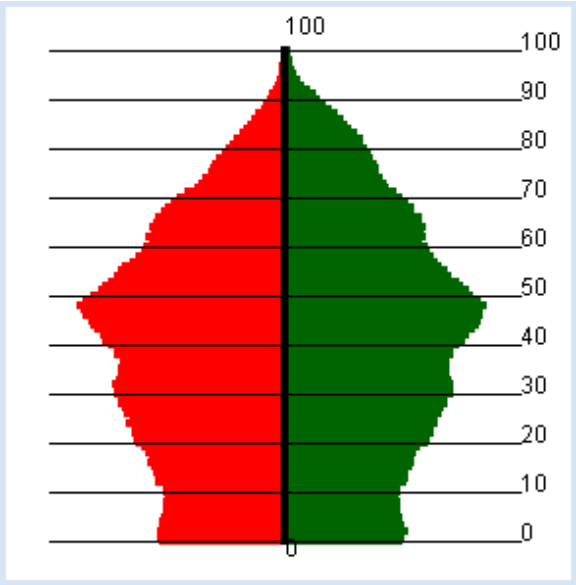
hx = x * rx bzw. hy = y * ry

```
n = 10000 # Groesse der Population

def readData(filename):
    table = []
    fData = open(filename)

    while True:
        line = fData.readline().replace(" ", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

qx = readData("Lib/qx.dat")
qy = readData("Lib/qy.dat")
x = n
y = n
xSum = 0
ySum = 0
for t in range(101):
    rx = qx[t]
    x = x - x * rx
    mx = x * rx # Todesfälle Mann
    xSum = xSum + mx * t #Beitrag Mann
    ry = qy[t]
    y = y - y * ry
    my = y * ry # Todesfälle Frau
    ySum = ySum + my * t #Beitrag Frau

print "Lebenserwartung Mann:", xSum / 10000
print "Lebenserwartung Frau:", ySum / 10000
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Mit den Daten der Schweizer Bevölkerung ergeben sich für die Lebenserwartung des Mannes rund 76 Jahre und für diejenige der Frau rund 81 Jahre.

ALTERSPYRAMIDE



Für demografische Untersuchungen wird die Bevölkerung oft in Jahresgruppen eingeteilt und daraus ein Häufigkeitsdiagramm erstellt. Hat man zwei Gruppen, die man vergleichen möchte, so kannst du die Häufigkeiten der einen Gruppe nach links und der anderen Gruppe nach rechts auftragen. Wendest du dieses Verfahren für Männer und Frauen an, so entsteht eine hübsche pyramidenartige Grafik.

Die aktuellen Daten (31. Dezember 2012) entnimmst du wieder einer Tabelle, die du vom Bundesamt für Statistik beziehst und aus der Excel-Tabelle in die Testdateien *zx.dat* und *zy.dat* kopierst (Quelle: **http://www.bfs.admin.ch**, Stichwort: STAT-TAB). Du kannst sie auch von **hier** herunterladen.

```
import exceptions
from gpanel import *

def readData(filename):
    table = []
    fData = open(filename)
    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

def drawAxis():
    text(0, -3, "0")
    line(0, 0, 0, 100)
    text(0, 103, "100")

makeGPanel(-100000, 100000, -10, 110)
title("Alterspyramide (Grün -> Männer, Rot -> Frauen)")
lineWidth(4)
zx = readData("Lib/zx.dat")
zy = readData("Lib/zy.dat")
for t in range(101):
    setColor("red")
    x = zx[t]
    line(0, t, -x, t)
    setColor("darkgreen")
    y = zy[t]
    line(0, t, y, t)
setColor("black")
drawAxis()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

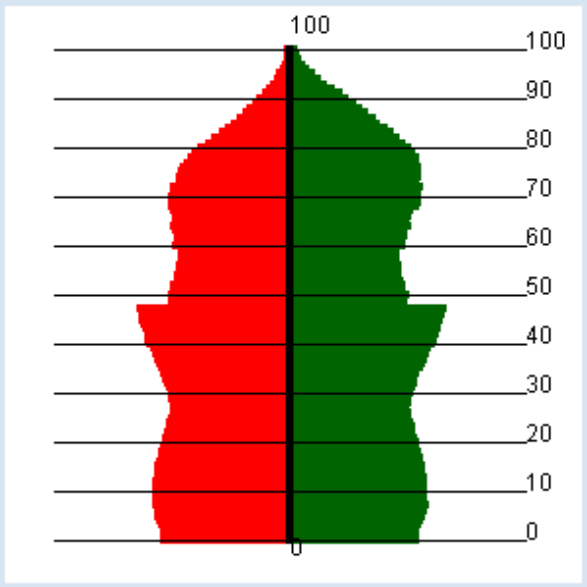
MEMO

Es ist deutlich der Babyboom in den Jahren 1955 - 1965 (47 - 57 Jährige) zu erkennen.

VERÄNDERUNG DER ALTERSSTRUKTUR

Eine Untersuchung der Veränderung der Altersstruktur über Jahrzehnte gibt wichtige Hinweise, wie sich eine Gesellschaft verändert. Du kannst die aktuelle Altersstruktur unter folgenden Voraussetzungen für die nächsten 100 Jahre simulieren:

- Es gibt keine Zu- und Abwanderungen von Aussen (geschlossene Gesellschaft)
- Die Todesfälle werden gemäss der Sterbetafeln berücksichtigt
- Jede Frau im geburtsfähigen Alter von 20 bis und mit 39 bringt eine bestimmte Anzahl k Kindern zur Welt (Knaben und Mädchen gleichwahrscheinlich). Gehe vorerst von k = 2 aus.



Mit einem **Tastendruck** kannst immer um ein Jahr vorschalten.

```
import exceptions
from gpanel import *

k = 2.0

def readData(filename):
    table = []
    fData = open(filename)
    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

def drawAxis():
    text(0, -3, "0")
    line(0, 0, 0, 100)
    text(0, 103, "100")
    lineWidth(1)
    for y in range(11):
        line(-80000, 10* y, 80000, 10 * y)
        text(str(10 * y))

def drawPyramid():
    clear()
```

```
    title("Kinderzahl: " + str(k) + ", Jahr: " + str(year) +
          ", Einwohnerzahl: " + str(getTotal()))
    lineWidth(4)
    for t in range(101):
        setColor("red")
        x = zx[t]
        line(0, t, -x, t)
        setColor("darkgreen")
        y = zy[t]
        line(0, t, y, t)
    setColor("black")
    drawAxis()
    repaint()

def getTotal():
    total = 0
    for t in range(101):
        total += zx[t] + zy[t]
    return int(total)

def updatePop():
    global zx, zy
    zxnew = [0] * 110
    zynew = [0] * 110
    # Getting older and die
    for t in range(101):
        zxnew[t + 1] = zx[t] - zx[t] * qx[t]
        zynew[t + 1] = zy[t] - zy[t] * qy[t]
    # Make baby
    r = k / 20
    nbMother = 0
    for t in range(20, 40):
        nbMother += zy[t]
    zxnew[0] = r / 2 * nbMother
    zynew[0] = zxnew[0]
    zx = zxnew
    zy = zynew

makeGPanel(-100000, 100000, -10, 110)
zx = readData("Lib/zx.dat")
zy = readData("Lib/zy.dat")
qx = readData("Lib/qx.dat")
qy = readData("Lib/qy.dat")
year = 2012
enableRepaint(False)
while True:
    drawPyramid()
    getKeyWait()
    year += 1
    updatePop()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Es zeigt sich, dass die Zukunft der Population sehr sensibel von der Zahl *k* abhängt. Sogar bei *k = 2* nimmt sie langfristig ab.

Um bei Drücken einer Taste das Bildschirmflackern zu vermeiden, solltest du mit **enableRepaint(False)** das automatische Rendern abschalten. Die Grafik wird dann in **drawPyramid()** mit **clear()** nur im Hintergrundspeicher (offscreen buffer) gelöscht und erst nach der Neuberechnung mit **repaint()** neu auf dem Bildschirm gerendert.

AUFGABEN

1. Eine Population besteht zur Zeit 0 aus 2 Individuen. Jedes Jahr nimmt sie mit der Geburtenrate 10% zu (Anzahl Geburten pro Jahr und Individuum). Simuliere sie in den ersten 100 Jahren (grafische Darstellung als Balkendiagramm).

2a. Bei einer nicht alternden Population bleibt die Sterbewahrscheinlichkeit unabhängig vom Alter immer gleich gross. Für Lebewesen gibt es keine solchen Populationen, hingegen verhalten sich radioaktive Atome (Radionuklide) genau so. Statt von Sterbewahrscheinlichkeit spricht man hier von Zerfallswahrscheinlichkeit. Simuliere eine Population mit 10000 Radionukliden, deren Zerfallswahrscheinlichkeit 0.1 beträgt für die ersten 100 Jahre (grafische Darstellung als Balkendiagramm).

2b. Trage im Diagramm als vertikale Linien die Zeiten ein, bei denen die Population ungefähr auf 1/2 , auf 1/4, auf 1/8 und auf 1/16 der Anfangsgrösse geschrumpft ist. Welche Vermutung hast du?

2c* Der radioaktive Zerfall erfolgt nach dem Gesetz

$$N = N_0 * e^{-\lambda t}$$

No: Anzahl Radionuklide zur Zeit t = 0
N: Anzahl Radionuklide zur Zeit t
λ: Zerfallswahrscheinlichkeit pro Zeiteinheit (Zerfallskonstante)

Trage den möglichst gut angepassten Kurvenverlauf mit roter Farbe in der Grafik unter 2a ein.

3. Die Lebenserwartung kann auch durch eine statistische Computersimulation berechnet werden. Dazu simulierst du das Leben eines einzelnen Individuums von Jahr zu Jahr. Du lässt dazu den Computer eine Zufallszahl zwischen 0 und 1 ziehen und lässt das Individuum sterben, falls diese Zahl kleiner als die Sterbewahrscheinlichkeit q ist. Die dabei erreichte Lebensdauer summierst du auf. Nachdem du die Simulation für 10000 Individuen durchgeführt hast, teilst du die erreichte Summe durch 10000. Bestimme mit diesem Verfahren mit den Werten von *qy.dat* die Lebenserwartung einer Frau.

ZUSATZSTOFF

RÄUBER-BEUTE-SYSTEME

Sehr interessant ist das Verhalten von zwei Populationen, die sich in einem bestimmten Ökosystem befinden und sich gegenseitig beeinflussen. Du gehst von folgendem Szenario aus:

In einem geschlossenen Gebiet halten sich Hasen und Füchse auf. Die Hasen vermehren sich mit konstanter Zuwachsrate *rx*. Trifft ein Fuchs auf einen Hasen, so wird er mit der einer bestimmten Wahrscheinlichkeit vom Fuchs gerissen. Die Füchse ihrerseits sterben mit der Sterbewahrscheinlichkeit *ry*. Ihre Zuwachsrate ist durch den Verzerr von Hasen bestimmt.

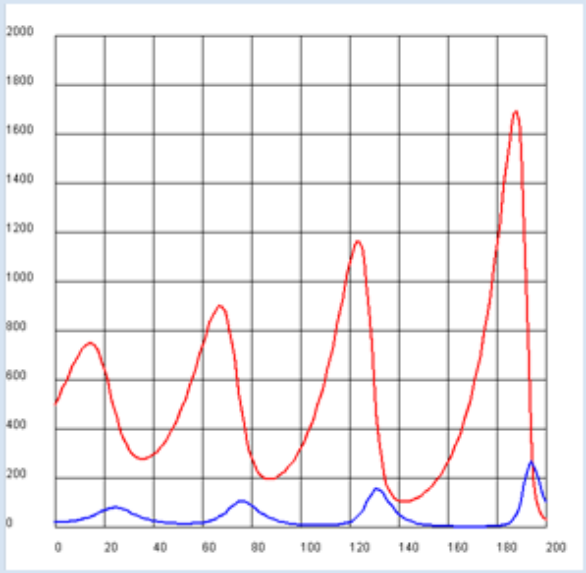
Wenn man annimmt, dass die Wahrscheinlichkeit, dass sich Füchse und Hasen treffen gleich dem Produkt der Zahl der Hasen x und der Füchse y ist, so ergeben sich zwei Differenzengleichungen für x und y [**mehr...**].

xNeu - x = rx * x - gx * x * y
yNeu - y = -ry * y + gy * x * y

Du gehst von folgenden Werten aus:

rx = 0.08
ry = 0.2
gx = 0.002
gy = 0.0004

und verwendest Anfangspopulationen von x = 500 Hasen und y = 20 Füchsen. Die Simulation führst du vorerst für 200 Generationen durch.




```
from gpanel import *

rx = 0.08
ry = 0.2
gx = 0.002
gy = 0.0004

def dx():
    return rx * x - gx * x * y

def dy():
    return -ry * y + gy * x * y

x = 500
y = 20

makeGPanel(-20, 220, -200, 2200)
title("Räuber-Beute-System (rot: Hasen, blau: Füchse)")
drawGrid(0, 200, 0, 2000)
lineWidth(2)
for n in range(200):
    xNew = x + dx()
    yNew = y + dy()
    setColor("red")
    line(n, x, n + 1, xNew)
    setColor("blue")
    line(n, y, n + 1, yNew)
    x = xNew
    y = yNew
```

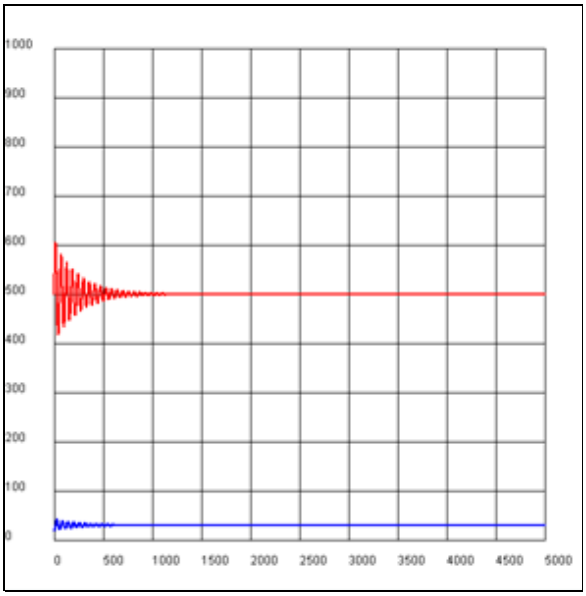
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

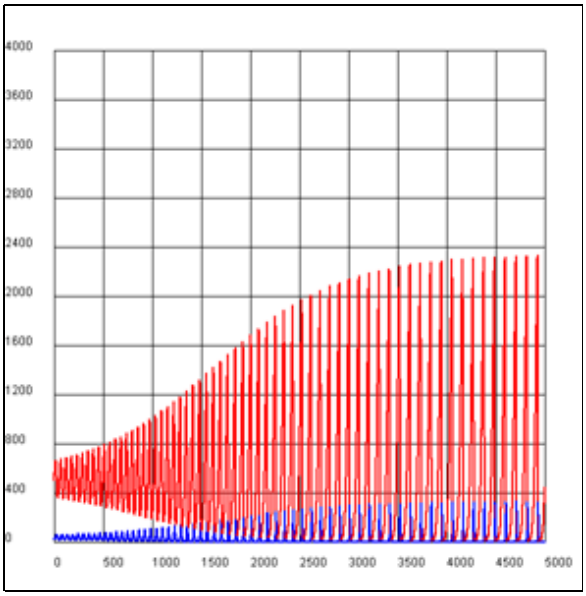
Die Zahl der Hasen und Füchse schwankt auf und ab. Qualitativ ist dies wie folgt zu verstehen: Da sich die Füchse von den Hasen ernähren, vermehren sich die Füchse dann besonders stark, wenn es viele Hasen gibt. Da dadurch die Population der Hasen dezimiert wird, geht auch die Vermehrung der Füchse zurück. Immerhin steigt trotzdem die Zahl der Hasen (sogar über alle Grenzen).

AUFGABEN

1. Führe mit sonst gleichen Werten wie oben für die Hasen eine Begrenzung des Lebensraumes gemäss dem logistischen Wachstum mit einer Zuwachsrate *rx' = rx(1 - x/m)* ein. Zeige, dass für m = 2000 die Schwingung mit der Zeit abklingt, hingegen für m = 3500 regelmässig wird.

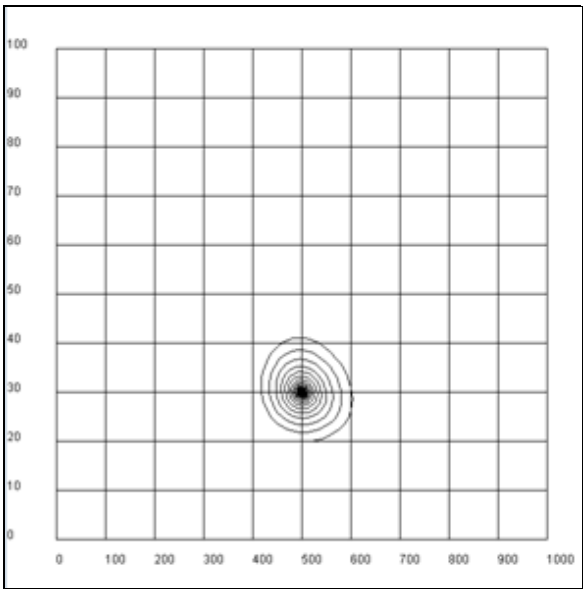


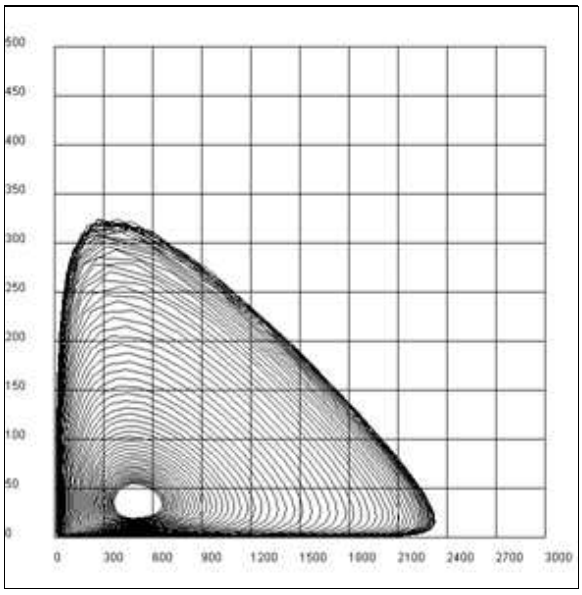
m = 2000, Schwingung klingt ab



m = 3500, Schwingung wird stabil

2. Eine Grafik, in der die Grössen der Population gegeneinander aufgetragen werden, nennt man ein **Phasendiagramm**. Schreibe ein Programm. dass das Phasendiagramm für die zwei Fälle von Aufgabe 1 zeichnet. Verstehst du das Verhalten?





8.3 HYPOTHESEN, STATISTISCHE TESTS

EINFÜHRUNG

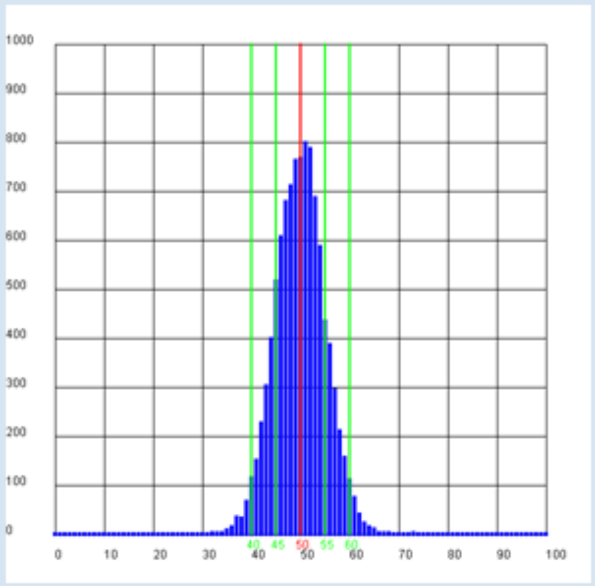
Du machst eine Hypothese (genannt **Nullhypothese**), beispielsweise dass eine vor dir liegende Münze nicht gefälscht ist , d.h. dass die Wahrscheinlichkeit für Kopf und Zahl gleich gross sind (p = ½). Oder du hast einen Würfel vor dir und machst die Hypothese, dass dieser nicht gezinkt ist, d.h. dass alle 6 Zahlen mit der gleichen Wahrscheinlichkeit p = 1/6 auftreten. In diesem Kapitel lernst du ein Verfahren kennen, um deine Hypothese zu prüfen, allerdings auch hier wieder nicht mit absoluter Sicherheit, sondern du lässt zu, dass du dich beim Verwerfen der Hypothese mit der Wahrscheinlichkeit 5 % (Signifikanzniveau) täuschst.

PROGRAMMIERKONZEPTE: *Nullhypothese, Signifikanz, Streuung, Chi-Quadrat-Test*

EINE SIGNIFIKANT GEFÄLSCHTE MÜNZE

Gehst du von der Nullhypothese aus, dass die Münze nicht gefälscht ist und wirfst du ihn n = 100 Mal, so erhältst du eine bestimmte Anzahl k mal Kopf und n - k mal Zahl.

Wiederholst du den Hundertertest mehrmals, sagen wir z = 10000 mal, so ergibt sich eine Verteilung für k, die du durch eine Simulation bestimmen kannst. Wie du erwartest, ist sie glockenförmig um den Mittelwert m = 50 verteilt [**mehr...**].



Du stellst dir nun die interessante Frage, in welchem Bereich +-s um den Mittelwert herum ein vorgegebener Prozentsatz, z.B. 68 % aller Hundertertests liegen. Auch s, genannt **Streuung**, kannst du in der Computersimulation bestimmen, indem du ausgehend vom Mittelwert nach links und nach rechts die Häufigkeitswerte aufsummierst, bis du 6800 erreichst.

Du zeichnest auch noch den Bereich mit 95% aller Fälle ein und erhältst etwa die doppelte Streuung.

```
from gpanel import *
import random

n = 100 # Grösse der Testgruppe
p = 0.5
z = 10000

def showDistribution():
    setColor("blue")
    lineWidth(4)
    for t in range(n + 1):
        line(t, 0, t, h[t])

def showMean():
    global mean
```

```
    sum = 0
    for t in range(n + 1):
        sum += h[t] * t
    mean = int(sum / z + 0.5)
    setColor("red")
    lineWidth(2)
    line(mean, 0, mean, 1000)
    text(mean - 1, -30, str(mean))

def showSpreading(level):
    sum = h[mean]
    for s in range(1, 20):
        sum += h[mean + s] + h[mean - s]
        if sum > z * level:
            break
    setColor("green")
    lineWidth(2)
    line(mean + s, 0, mean + s, 1000)
    text(mean + s - 1, -30, str(mean + s))
    line(mean - s, 0, mean - s, 1000)
    text(mean - s - 1, -30, str(mean - s))

def sim():
    sum = 0
    repeat n:
       w = random.random()
       if w < p:
           sum +=1
    return sum

makeGPanel(-0.1 * n, 1.1 * n, -100, 1100)
title("Münzenwurf, Verteilung von Zahl")
drawGrid(0, n, 0, 1000)
h = [0] * (n + 1)

repeat z:
    k = sim()
    h[k] += 1

showDistribution()
showMean()
showSpreading(0.68)
showSpreading(0.95)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Machst du oft eine Stichprobe mit 100 Münzen, die nicht gefälscht sind, so liegen in 68 % aller Fälle die Zahl der geworfenen Köpfe im Bereich 50 +-5, in 95% aller Fälle im Bereich +-10 [**mehr...**] .

Machst du also mit deiner vor dir liegenden Münze einen Hundertertest und erhältst für die Anzahl Köpfe einen Wert, der grösser als 60 oder kleiner als 40 ist, so verwirfst du die Hypothese, dass die Münze nicht gefälscht ist, d.h. du sagst, die Münze sei gefälscht. Dabei irrst du dich mit einer Wahrscheinlichkeit von 5 % (genannt Signifikanzniveau). Manchmal sagst du auch prägnant, ***die vorliegende Münze ist signifikant gefälscht***.

EIN SIGNIFIKANT GEZINKTER WÜRFEL

Du hast einen Würfel vor dir und möchtest testen, ob es sich um einen fairen Würfel handelt, bei dem alle Augenzahlen mit der Wahrscheinlichkeit 1/6 auftreten. Du machst daher die Hypothese: *Der Würfel ist nicht gezinkt*.

Du lernst hier ein etwas anderes Verfahren als bei der Münze kennen, da bei einem Wurf nicht nur zwei, sondern 6 Möglichkeiten auftreten, nämlich die Augenzahlen 1 bis 6. Um auf sicher zu gehen, wirfst du den Würfel oftmals, sagen wir 600 mal und schreibst dir die Häufigkeiten der Augenzahlen auf.

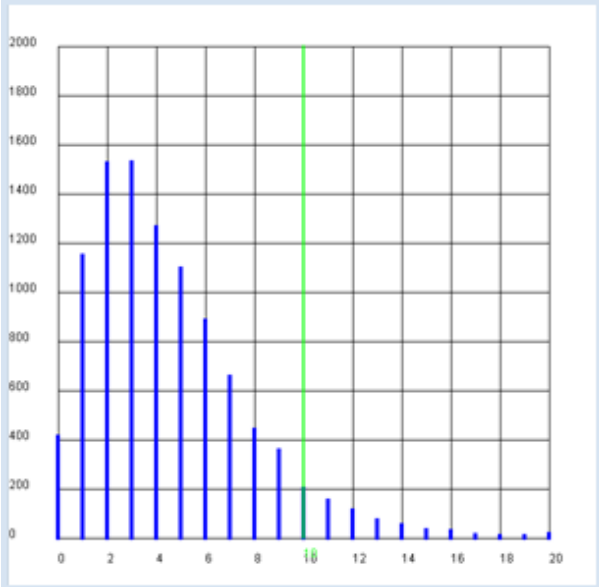
Augenzahl	Beobachtete Häufigkeit (u)	Theoretische Häufigkeit (Erwartungswert e)
1	112	100
2	128	100
3	97	100
4	103	100
5	88	100
6	72	100
Total	600	600

Beobachtete und theoretische Häufigkeit

Um ein Mass für die Abweichung der Beobachtung von der Theorie einzuführen, berechnest du für jede Augenzahl die relative quadratische Abweichung $(u - e)^2 / e$ und summierst diese Werte auf. Das Resultat nennen wir χ^2 (ausgesprochen "Chi-Quadrat").

Es stellt sich die interessante Frage, wie χ^2 verteilt ist, d.h. mit welchen Häufigkeiten verschiedene Werte von χ^2 bei vielen Sechshunderter-Versuchen auftreten. Um dies herauszufinden, führst du wieder eine Computersimulation mit 10000 Versuchen durch und bestimmst die Verteilung. Dabei rundest du der Einfachheit halber die erhaltenen Werte auf ganze Zahlen [**mehr...**].

Gleichzeitig trägst du auch hier wieder einen kritischen Wert für χ^2 ein, unterhalb der 95% aller Fälle liegen. Aus der Simulation ergibt sich s = 11 [**mehr...**].




```
from gpanel import *
import random

n = 600 # Anzahl Würfe
p = 1 / 6
z = 10000

def showDistribution():
    setColor("blue")
    lineWidth(4)
    for i in range(21):
        line(i, 0, i, h[i])

def showLimit(level):
    sum = 0
    for i in range(21):
        sum += h[i]
        if sum > z * level:
            break
    setColor("green")
    lineWidth(2)
    line(i, 0, i, 2000)
    text(i, -80, str(i))
```

```
    return i

def chisquare(u):
    chisquare = 0
    e = n * p
    for i in range(1, 7):
        chisquare += ((u[i] - e) * (u[i] - e)) / e
    return chisquare

def sim():
    u = [0] * 7
    repeat n:
        t = random.randint(1, 6)
        u[t] += 1
    return chisquare(u)

makeGPanel(-2, 22, -200, 2200)
title("Chi-Quadrat Simulation wird durchgeführt. Bitte warten...")
drawGrid(0, 20, 0, 2000)
h = [0] * 21

repeat z:
    c = int(sim())
    if c < 20:
        h[c] += 1
    else:
        h[20] += 1

title("Chi-Quadrat Test beim Würfeln")
showDistribution()
s = showLimit(0.95)

# Observed series
u1 = [0, 112, 128, 97, 103, 88, 72]
u2 = [0, 112, 108, 97, 113, 88, 82]
c1 = chisquare(u1)
c2 = chisquare(u2)
print "Würfel mit", u1, "Xi-Quadrat:", c1, "Gezinkt?", c1 > s
print "Würfel mit", u2, "Xi-Quadrat:", c2, "Gezinkt?", c2 > s
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Computersimulation ergibt folgendes Resultat: In 95% aller Fälle ist χ^2 kleiner oder gleich dem kritischen Wert 11. Damit hast du ein Verfahren gefunden um zu prüfen, ob dein Würfel gezinkt ist: Du berechnest aus der beobachteten Häufigkeit χ^2. Ist der Wert grösser als 11, so kannst du mit 5% Irrtumswahrscheinlichkeit sagen, dass deine Nullhypothese eines fairen Würfels falsch ist, der Würfel also gezinkt ist.

Mit den Häufigkeiten aus der oberen Tabelle ergibt sich $\chi^2 = 18.7$. Der Würfel ist also mit mit grosser Wahrscheinlichkeit gezinkt. Mit einem anderen Würfel erwürfelst du mit 600 Würfen die Häufigkeiten u2 = [112, 108, 97, 113, 88, 82]. Da du daraus $\chi^2 = 8.5$ berechnest, ist dieser Würfel mit grosser Wahrscheinlichkeit nicht gezinkt.

UNTERSCHIEDE IM MENSCHLICHEN VERHALTEN

Den χ^2-Test kannst du auch auf eine Untersuchung über das Verhalten von zwei Menschengruppen anwenden. Interessant ist oft die Frage, ob das Verhalten in einem bestimmten Kontext von weiblichen und männlichen Personen als statistisch unterschiedlich taxiert werden muss, oder ob sie sich beide Gruppen gleich verhalten.

Du gehst davon aus, dass in einer Sekundarschule die Verwendung von Facebook untersucht wird. Dabei werden in Parallelklassen einer Sekundarschule insgesamt 106 Mädchen (Frauen) und 86 Knaben (Männer) gefragt, ob sie ein Facebook-Konto besitzen. Die Umfrage ergibt:

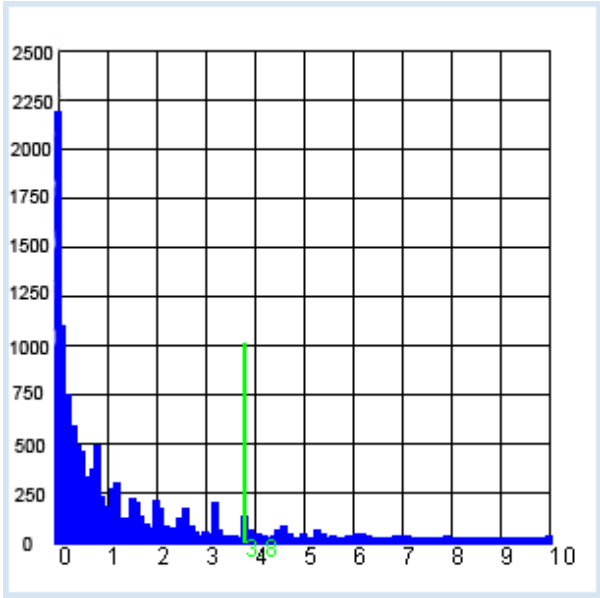
	Facebook Ja	Facebook Nein	Total	% Ja-Anteil
Frauen	87	19	106	82.0%
Männer	62	24	86	72.1%
Total	149	43	192	77.7%

Der prozentuale Anteil ist zwar bei Frauen wesentlich grösser als bei Männern. Es stellt sich aber die Frage, ob dieser Mehranteil statistisch signifikant ist.

Für die Simulation bestimmst du zuerst aus der Totalzahl n von Frauen und Männer die Wahrscheinlichkeit p, ein Konto zu besitzen:

p = (frauen_ja + maenner_ja) / n

Mit diesem Wert simulierst du mit Zufallszahlen und mit der Totalzahl der Frauen die Zahl der weiblichen Kontenbesitzer. Es ergeben sich dabei f0 Frauen mit einem Konto und f1 Frauen ohne Konto. Dasselbe machst du für Männer und es ergeben sich m0 Männer mit einem Konto und m1 Männer ohne Konto. Diese Zahlen entsprechen den Werten u bei der Berechnung von χ^2.



$$\chi^2 = \text{Summe von } (u - e)^2 / e$$

Für alle vier Fälle musst du nun noch der Erwartungswert e bestimmen. Gehst du davon aus, dass p = (f0 + m0) / n die Gesamtwahrscheinlichkeit für ein Ja und entsprechend 1 - p die Gesamtwahrscheinlichkeit für ein Nein ist, so berechnest du

Erwartungswert Frauen-Ja: ef0 = tot Zahl Frauen * p
Erwartungswert Männer-Ja: em0 = tot Zahl Männer * p
Erwartungswert Frauen-Nein ef1 = tot Zahl Frauen * (1 - p)
Erwartungswert Männer-Nein: em1 = tot Zahl Männer * p

Das übrige Programm bleibt gegenüber dem Würfeltest weitgehend unverändert.

```
from gpanel import *
import random

z = 10000
# Umgfragewerte
frauen_ja = 87
frauen_nein = 19
maenner_ja = 62
maenner_nein = 24

def showDistribution():
    setColor("blue")
    lineWidth(4)
    for i in range(101):
        line(i/10, 0, i/10, h[i])

def showLimit(level):
```

```
    sum = 0
    for i in range(101):
        sum += h[i]
        if sum > level * z:
            break
    setColor("green")
    lineWidth(2)
    limit = i / 10
    line(limit, 0, limit, 1000)
    text(limit, -80, str(limit))
    return limit

def chisquare(f0, f1, m0, m1):
    # f: Frauen, m: Männer, 0:ja, 1:nein
    w = (f0 + m0) / n # Wahrscheinlichkeit für ein ja
    # Erwartungswerte
    ef0 = (f0 + f1) * w # Frauen-ja
    em0 = (m0 + m1) * w # Männer-ja
    ef1 = (f0 + f1) * (1 - w) # Frauen-nein
    em1 = (m0 + m1) * (1 - w) # Männer-nein
    # Abweichungen (u - e)*(u - e) / e aufsummieren
    chisquare = (f0 - ef0) * (f0 - ef0) / ef0 \
              + (m0 - em0) * (m0 - em0) / em0 \
              + (f1 - ef1) * (f1 - ef1) / ef1 \
              + (m1 - em1) * (m1 - em1) / em1
    return chisquare

def sim():
    # Frauen simulieren
    f0 = 0 # ja
    f1 = 0 # nein
    for i in range(frauen_alle):
        t = random.random()
        if t < p:
           f0 += 1
        else:
           f1 += 1
    # Männer simulieren
    m0 = 0 # ja
    m1 = 1 # nein
    for i in range(maenner_alle):
        t = random.random()
        if t < p:
           m0 += 1
        else:
           m1 += 1
    return chisquare(f0, f1, m0, m1)

frauen_alle = frauen_ja + frauen_nein
maenner_alle = maenner_ja + maenner_nein
n = frauen_alle + maenner_alle  # alle
p = (frauen_ja + maenner_ja) / n  # Ja-Wahrscheinlichkeit für alle
print "Facebook Ja (alle):", round(100 * p, 1), "%"
pf = frauen_ja / frauen_alle
print "Facebook Ja (Frauen):", round(100 * pf, 1), "%"
pm = maenner_ja / maenner_alle
print "Facebook Ja (Männer:)", round(100 * pm, 1), "%"

makeGPanel(-1, 11, -250, 2750)
title("Chi-Quadrat Test Verwendung von Facebook")
drawGrid(0, 10, 0, 2500)
h = [0] * 101

repeat z:
    c = int(10 * sim())  # Vergrösserungsfaktor 10
    if c < 100:
        h[c] += 1
    else:
        h[100] += 1
```

```
showDistribution()
s = showLimit(0.95)

c = chisquare(frauen_ja, frauen_nein, maenner_ja, maenner_nein)
print "Kritischer Wert:", s
print "Beobachtet:", c,
if c <= s:
   print "- Gleiches Verhalten"
else:
   print "- Ungleiches Verhalten"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Das Resultat ist erstaunlich: Die χ^2-Signifikanzgrenze liegt bei 3.8 [**mehr...**]. Für die Umfragewerte ergibt sich der kleinere Wert 2.7. Trotzdem bei den Mädchen der Kontenanteil wesentlich höher ist, kann also statistisch nicht bewiesen werden, dass sie sich bezüglich Facebook wesentlich von den Knaben unterscheiden.

AUFGABEN

1. Du hast eine andere Idee, um zu überprüfen, ob ein Würfel gezinkt ist. Dazu simulierst du wie beim 100er Münzenwurf oftmals einen 600er Würfelwurf und bestimmst die Verteilung der Augensumme.

   Zeigt der Würfel aus der oben angegebenen Verteilung u1 bei einer 5% Irrtumswahrscheinlichkeit ebenfalls eine Fälschung? Wie steht es mit dem Würfel mit der Verteilung u2?

2. Ein klassischer Roulette-Tisch hat die 37 Zahlen 0..36, die mit gleicher Wahrscheinlichkeit auftreten sollten. Ein cleverer Spieler möchte gewisse Unregel feststellen, um seine Gewinnwahrscheinlichkeit zu erhöhen. Er notiert sich bei 1000 Spielen die Häufigkeit der auftretenden Zahlen und erhält

   u = [20, 26, 20, 22, 20, 27, 18, 28, 21, 36, 20, 28, 25, 19, 22, 25, 33, 25, 28, 25, 32, 29, 22, 32, 28, 31, 26, 25, 32, 32, 25, 20, 25, 44, 40, 24, 45]

   Überprüfe mit einem χ^2-Test die Nullhypothese, dass das Roulette fair ist.

3. Um ein Medikament zu testen, wird in einer Blindstudie einer Gruppe von kranken Personen das Medikament verschrieben und einer anderen Gruppe ein Placebo verabreicht. Nach der Behandlung ergeben sich folgende Werte:

	Nach Behandlung geheilt	Nach Behandlung krank	% Geheilt-Anteil
Mit Medikament behandelt	22	13	62.9 %
Mit Placebo behandelt	11	17	39.3 %

   Der Anteil der Geheilten ist mit medikamentöser Therapie also wesentlich grösser als ohne. Darf angenommen werden, dass das Medikament wirksam ist?

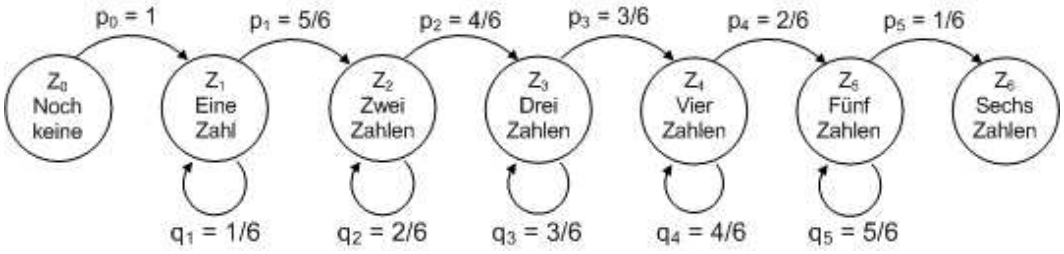
8.4 MITTLERE WARTEZEITEN

EINFÜHRUNG

Es gibt viele Systeme, deren zeitliches Verhalten du als Übergang von einem Systemzustand zu einem nächsten beschreiben kannst. Dabei ist der Übergang vom aktuellen Systemzustand Z_i zum Nachfolgezustand Z_k durch Wahrscheinlichkeiten p_{ik}bestimmt. Im Folgenden würfelst du mit einem Würfel und betrachtest die bereits geworfenen Augenzahlen als Zustandsgrösse:

Z_0 : noch keine Augenzahl geworfen
Z_1 : eine Augenzahl geworfen
Z_2 : zwei (verschiedene) Augenzahlen geworfen
usw.

Du kannst den Übergang der Zustände im folgenden Schema veranschaulichen (sogenannte **Markoff-Kette**):



Die Wahrscheinlichkeiten erklären sich wie folgt: Hast du bereits n Augenzahlen geworfen, so ist die Wahscheinlichkeit, wieder eine dieser Zahlen zu erhalten n/6 und die Wahrscheinlichkeit, eine noch nicht nicht vorhandene zu erhalten (6 - n)/6. Du stellst dir die interessante Frage, wie oft du im Mittel den Würfel werfen musst, um alle sechs Zahlen zu erwürfeln.

PROGRAMMIERKONZEPTE: *Markoff-Kette, Wartezeit, Wartezeitparadoxon*

MITTLERE WARTEZEITEN

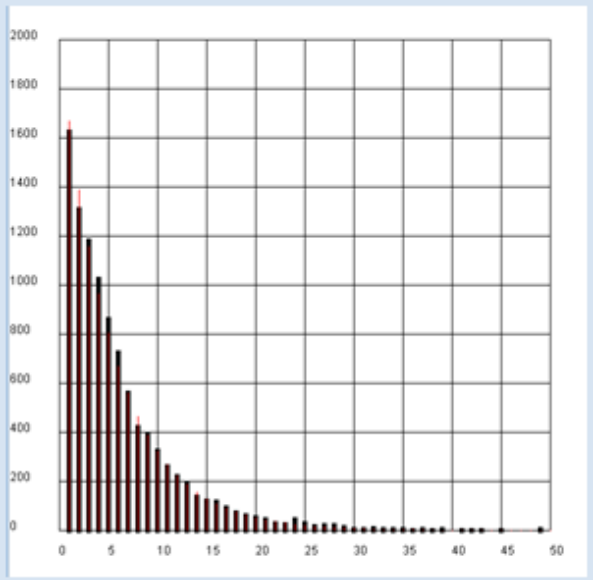
Wenn du in gleichen Zeitschritten würfelst, so kannst du dich auch fragen, wie lange im Mittel der Prozess dauert. Diese Zeit nennst du die **mittlere Wartezeit.** Du erhältst sie aus folgender Überlegung: Die totale Zeit, um von Z_0 zu Z_6 zu gelangen, ist die Summe der Wartezeiten für alle Übergänge. Wie gross sind aber die einzelnen Wartezeiten?

In deinem ersten Programm findest du experimentell die wichtigste Eigenschaft von Wartezeitproblemen heraus.:

Ist p die Wahrscheinlichkeit um von Z_1 nach Z_2 zu gelangen, so beträgt die Wartezeit (in einer geeigneten Einheit) u = 1/p.

In der Simulation untersuchst du die Wartezeit, um eine bestimmte Augenzahl, sagen wir eine 6, zu würfeln. Ein Simulationsversuch besteht hier nicht aus einem einzigen Würfelwurf. Vielmehr wirfst du in der Funktion *sim()* soviel Mal, bis du die 6 erhalten hast und gibst die Anzahl der dazu nötigen Würfe zurück. Diesen Versuch wiederholst du 10000 Mal und bildest das Mittel der benötigten Würfe.

Gleichzeitig stellst du in einem Häufigkeitsdiagramm die Zahl Versuche dar, bei denen du in k = 1, 2, 3,... Würfen die 6 erhieltest (bei k = 50 hörst du auf).




```
from gpanel import *
import random

n = 10000
p = 1/6

def sim():
    k = 1
    r = random.randint(1, 6)
    while r != 6:
        r = random.randint(1, 6)
        k += 1
    return k

makeGPanel(-5, 55, -200, 2200)
drawGrid(0, 50, 0, 2000)
title("Warten auf 6")
h = [0] * 51
lineWidth(5)
sum = 0
repeat n:
    k = sim()
    sum += k
    if k <= 50:
        h[k] += 1
        line(k, 0, k, h[k])
mean_exp = sum / n

lineWidth(1)
setColor("red")
sum = 0
for k in range(1, 1000):
    pk = (1 - p)**(k - 1) * p
    nk = n * pk
    sum += nk * k
    if k <=50:
        line(k, 0, k, nk)
mean_theory = sum / n
title("Experiment: " + str(mean_exp) + " Theorie: " + str(mean_theory))
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Das Resultat ist intuitiv einleuchtend: Da die Wahrscheinlichkeit, eine bestimmte Augenzahl zu würfeln p= 1/6 ist, benötigst du im Mittel u = 1 / p = 6 Würfe um diese Augenzahl zu erhalten.

Es ist auch instruktiv, die theoretischen Werte der Häufigkeiten als rote Linie einzutragen. Dazu überlegst du dir, dass für die Wahrscheinlichkeiten eine 6 zu werfen, gilt:

- im ersten Wurf eine 6: $p_1 = p$
- im ersten Wurf keine 6, aber im zweiten eine 6: $p_2 = (1 - p) * p$
- weder im ersten und zweiten keine 6, aber im dritten eine 6: $p_3 = (1 - p) * (1 - p) * p$
- im k-ten Wurf eine 6: $p_k = (1 - p)^{k-1} * p$

Für die theoretischen Häufigkeiten multiplizierst du diese Wahrscheinlichkeiten mit der Anzahl n der Versuche [**mehr...**].

PROGRAMMIEREN STATT VIEL RECHNEN

Um nun die oben formulierte Aufgabe zu lösen, die mittlere Wartezeit zu berechnen, bis du alle Augenzahlen mindestens einmal geworfen hast, kannst du einerseits den theoretischen Weg einschlagen. Du fasst den Prozess als Markoff-Kette auf und addierst die Wartezeiten für die einzelnen Übergänge:

u = 1 + 6/5 + 6/4 + 6/3 + 6/2 + 6 = 14.7

Andererseits kannst du ein einfaches Programm schreiben, um diese Zahl in einer Simulation zu bestimmen. Dazu gehst du immer gleich vor: Du schreibst eine Funktion *sim()*, in welcher der Computer mit Zufallszahlen eine einzelne Lösung sucht und gibst die benötigte Anzahl Schritte als Returnwert zurückgibt. Dann wiederholst du diese Tätigkeit sehr oft, sagen wir einige tausend Mal, und bestimmst den Mittelwert.

Es ist elegant, in *sim()* eine Liste z zu verwenden, in die du die geworfenen Zahlen einfügst, falls sie sich nicht bereits darin befinden. Wenn die Liste 6 Elemente hat, hast du alle Augenzahlen geworfen.

```
import random

n = 10000

def sim():
    z = []
    i = 0
    while True:
        r = random.randint(1, 6)
        i += 1
        if not r in z:
            z.append(r)
        if len(z) == 6:
            return i

sum = 0
repeat n:
    sum += sim()

print "Mittlere Wartezeit:", sum / n
```

MEMO

Du erhältst in der Computersimulation den leicht schwankenden Wert von 14.68, was der theoretischen Voraussage entspricht. Der Computer kann also auch dazu dienen, schnell zu überprüfen, ob ein theoretisch errechneter Wert überhaupt stimmen kann.

Die theoretische Bestimmung von Wartezeiten kann aber bereits bei einfachen Problemen sehr aufwendig sein. Fragst du etwa nach der mittleren Wartezeit, bis du durch aufeinanderfolgendes Würfeln eine bestimmte Augensumme erreichst, so ist das Problem mit einer Computersimulation extrem einfach zu lösen.

```
import random

n = 10000
s = 7  # zu erreichende Augensumme

def sim():
    i = 0
    total =  0
    while True:
        i += 1
        r = random.randint(1, 6)
        total += r
        if total >= s:
            break
    return i

sum = 0
repeat n:
    sum += sim()

print "Mittlere Wartezeit:", sum / n
```

MEMO

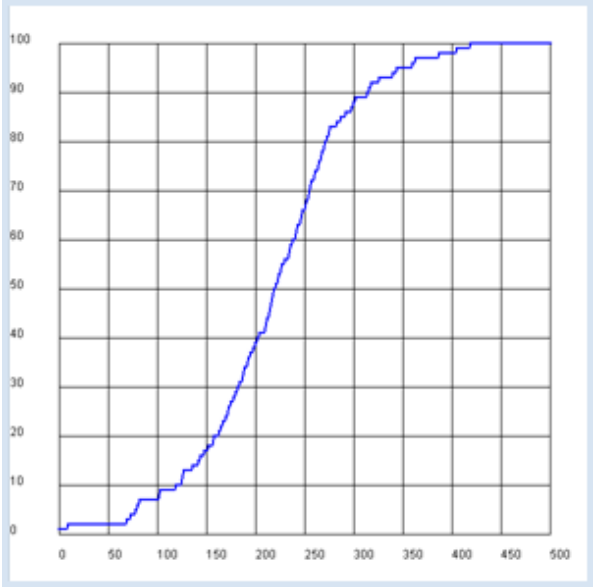
Du erhältst für die Augensumme 7 eine mittlere Wartezeit von etwa 2.52. Dieses Resultat ist insofern erstaunlich, als dass der Erwartungswert für die Augenzahlen ja 3.5 ist und man deswegen annehmen könnte, dass man für die Augensumme 7 im Mittel 2x würfeln muss. Die theoretische Berechnung, für die du mit einem Zeitaufwand von mehreren Stunden rechnen musst, ergibt 117 577 / 46 656 = 2.5008. Sogar Mathematiker verwenden darum den Computer, um theoretische Resultate einem schnellen Test zu unterziehen und Vermutungen zu überprüfen.

AUSBREITUNG EINER KRANKHEIT

Du gehst von der folgenden Geschichte aus, die zwar erfunden ist, aber durchaus gewisse Parallelen zu aktuellen Lebensgemeinschaften hat.

"Auf einer einsamen Karibikinsel leben abgeschnitten von der Aussenwelt 100 Menschen. Ein alter Mann infiziert sich durch den Verzerr eines Zugvogels, den er gedankenlos gegessen hat, mit einer ansteckenden Krankheit. Trifft ein kranker Einwohner mit einem gesunden zusammen, so wird dieser in kurzer Zeit krank. Jede Stunde treffen sich zwei Menschen zufällig."

Mit einer Computersimulation willst du untersuchen, wie sich die Krankheit ausbreitet. Dazu bestimmst du die Zahl der Infizierten in Abhängigkeit von der Zeit.



Es ist elegant, die Population durch eine Liste mit booleschen Werten zu modellieren, wobei gesund mit *False* und krank mit *True* codiert sind. Der Vorteil dieser Datenstruktur besteht darin, dass sich die Wechselwirkung beim Zusammentreffen von zwei Personen in der Funktion *pair()* mit einer logischen OR-Verknüpfung ausdrücken lässt:

1. Person vorher	**2. Person vorher**	**1.& 2. Person nachher**
gesund (False)	gesund (False)	gesund (False)
gesund (False)	krank (True)	krank (True)
krank (True)	gesund (False)	krank (True)
krank (True)	krank (True)	krank (True)

```
from gpanel import *
import random

def pair():
    # Select two distinct inhabitants
    a = random.randint(0, 99)
    b = a
    while b == a:
        b = random.randint(0, 99)
    z[a] = z[a] or z[b]
    z[b] = z[a]

def nbInfected():
    sum = 0
    for i in range(100):
        if z[i]:
            sum += 1
    return sum

makeGPanel(-50, 550, -10, 110)
title("Ausbreitung einer Krankheit")
drawGrid(0, 500, 0, 100)
lineWidth(2)
setColor("blue")

z = [False] * 100
tmax =  500
t = 0
a = random.randint(0, 99)
```

```
z[a] = True # random infected inhabitant
move(t, 1)

while t <= tmax:
    pair()
    infects = nbInfected()
    t += 1
    draw(t, infects)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du findest ein zeitliches Verhalten, bei dem die Zunahme zuerst langsam, dann rasant und dann wieder langsam ist. Qualitativ ist dir dieses Verhalten sicher plausibel, denn zuerst ist die Wahrscheinlichkeit gering, dass sich ein Kranker mit einem Gesunden trifft, da sich vor allem Gesunde untereinander treffen. Am Schluss ist die Wahrscheinlichkeit wiederum gering, dass ein übrig gebliebener Gesunder einem Kranken begegnet, da sich vor allem Kranke untereinander treffen [**mehr...**].

Interessant ist die Frage, wie lange es im Mittel geht, bis alle Einwohner krank sind. Du kannst das Problem direkt mit einer Computersimulation lösen, in der du vielmals dieselbe Population simulierst und dabei die Schritte zählst, bis alle krank sind.

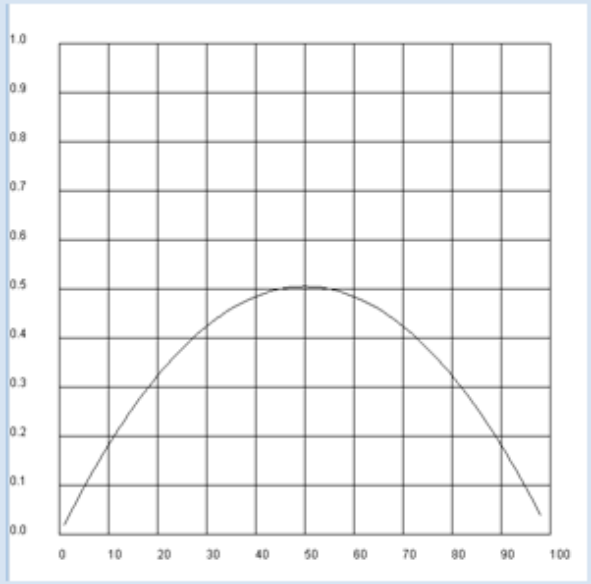
```
import random

n = 1000 # Anzahl Experimente

def pair():
    # Select two distinct inhabitants
    a = random.randint(0, 99)
    b = a
    while b == a:
        b = random.randint(0, 99)
    z[a] = z[a] or z[b]
    z[b] = z[a]

def nbInfected():
    sum = 0
    for i in range(100):
        if z[i]:
            sum += 1
    return sum

def sim():
    global z
    z = [False] * 100
    t = 0
    a = random.randint(0, 99)
    z[a] = True # random infected inhabitant
    while True:
        pair()
        t += 1
        if nbInfected() == 100:
            return t

sum = 0
for i in range(n):
    u = sim()
    print "Experiment #", i + 1, "Wartezeit:", u
    sum += u

print "Mittlere Wartezeit:", sum / n
```

Du kannst die Ausbreitung der Krankheit aber auch als Markoff-Kette auffassen. Ein bestimmter Zustand ist durch die Zahl der Infizierten charakterisiert. Die Wartezeit, bis alle krank sind, ist die Summe der Wartezeiten für die Übergänge von k Kranken zu k+1 Kranken für k von 1 bis 99. Dazu benötigst du die Wahrscheinlichkeit p_k für diesen Übergang. Es gilt:

p_k = Summe der Wahrscheinlichkeiten als ersten einen Kranken und als zweiten einen Gesunden auszuwählen und umgekehrt:

$$p_k = \frac{k}{n} * \frac{n-k}{n-1} + \frac{n-k}{n} * \frac{k}{n-1} = 2 * \frac{k*(n-k)}{n*(n-1)}$$

Im Programm stellst du pk auch noch gerade grafisch dar und ermittelst die Summe der Reziprokwerte.




```
from gpanel import *

n  = 100

def p(k):
    return 2 * k * (n - k) / n / (n - 1)

makeGPanel(-10, 110, -0.1, 1.1)
drawGrid(0, 100, 0, 1.0)

sum = 0
for k in range(1, n - 1):
    if k == 1:
        move(k, p(k))
    else:
        draw(k, p(k))
    sum += 1 / p(k)

title("Wartezeit bis alle krank: " + str(sum))
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Mit der Theorie der Markoff-Ketten ergibt sich eine mittlere Wartezeit von 463 Stunden, also rund 20 Tage.

PROGRAMM FÜR PARTNERSUCHE

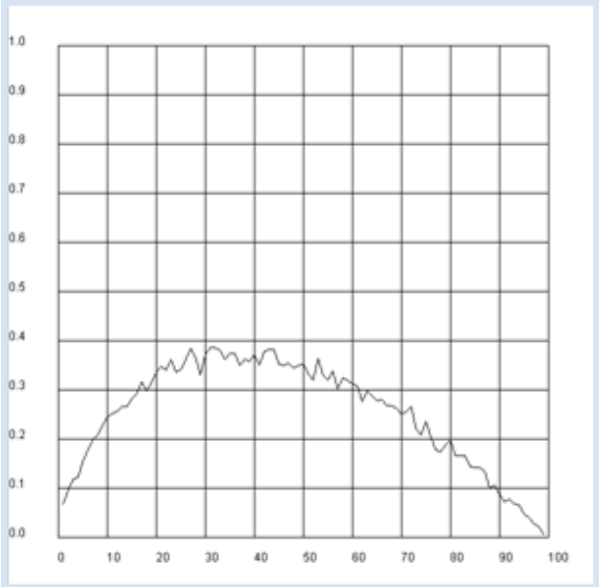
Für die Praxis interessant ist die Frage nach einer optimalen Strategie bei der Wahl eines Lebenspartners. Dabei gehst du von folgender Modellannahme aus: 100 mögliche Partner besitzen aufsteigende Qualifikationswerte. Sie werden dir in zufällig durchmischter Reihenfolge vorgestellt und du kannst sie in dieser "Lernphase" durch deine bisher erreichte Lebenserfahrung in der Reihenfolge ihrer Qualifikationswerte richtig einordnen. Du weist allerdings nicht, welches

der maximale Qualifikationsgrad ist. Bei jeder Vorstellung musst du dich für den Partner entscheiden oder du weist ihn zurück. Wie musst du vorgehen, damit du mit hoher Wahrscheinlichkeit den Partner mit der besten Qualifikation wählst?

Für die Simulation erstellst du in der Funktion *sim(x)* eine Liste t mit den 100 Qualifikationswerten 0..99 in beliebiger Reihenfolge. Es handelt sich um eine Zufallspermutation der Zahlen 0..99, die du in Python elegant mit *shuffle()* erzeugen kannst.

Nachfolgend führst du den Auswahlprozess durch, wobei du von einer fest vorgegebenen Länge x der Lernphase ausgehst, und bestimmst den Index des in diesem Prozess gewählten Partners und seinen Qualifikationswert.

Du simulierst nun mit bestimmtem x den Vorgang 1000 Mal und bestimmst, wie gross die Wahrscheinlichkeit ist, dass du dabei den Partner mit maximalem Qualifikationswert kriegst. Diese Wahrscheinlichkeit trägst du schliesslich in Abhängigkeit der Länge x der Lernphase grafisch auf.

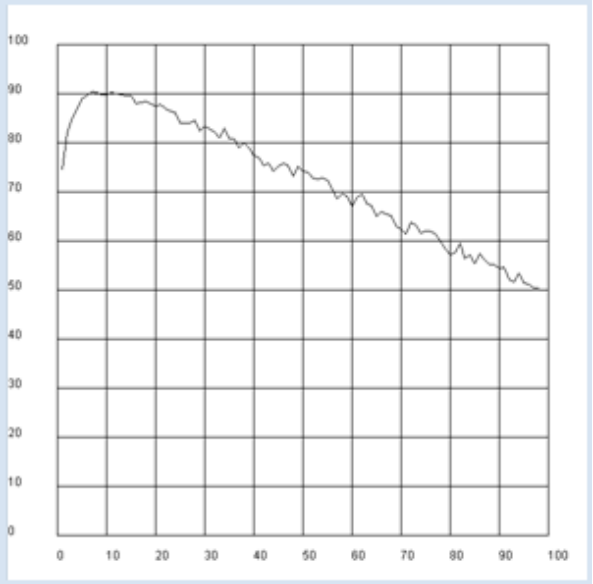



```
import random
from gpanel import *

n = 1000 # Number of simulations
a = 100 # Number of partners

def sim(x):
    # Random permutation [0..99]
    t = [0] * 100
    for i in range(0, 100):
        t[i] = i
    random.shuffle(t)
    best = max(t[0:x])
    for i in range(x, 100):
        if  t[i] > best:
            return [i, t[i]]
    return [99, t[99]]

makeGPanel(-10, 110, -0.1, 1.1)
title("Wahrscheinlichkeit, von 100 den besten Partner zu finden")
drawGrid(0, 100, 0, 1.0)

for x in range(1, 100):
    sum = 0
    repeat n:
        z = sim(x)
        if z[1] == 99:  # best score
            sum += 1
    p = sum / n
    if x == 1:
        move(x, p)
    else:
        draw(x, p)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Es zeigt sich, dass du bei einer Länge der Lernphase von etwas 37 die grösste Wahrscheinlichkeit hast, den Partner mit der besten Qualifikation zu finden. Du kannst die Optimierung des Auswahlverfahrens aber auch nach einem anderen Kriterium durchführen, indem du nicht so sehr nach dem besten Partner, sondern nach einem mit möglichst hohem Qualifikationswert Ausschau hältst. Dazu untersuchst du mit einer ähnlichen Simulation zu gegebener Länge x der Lernphase den Mittelwert der Qualifikation des gewählten Partners.



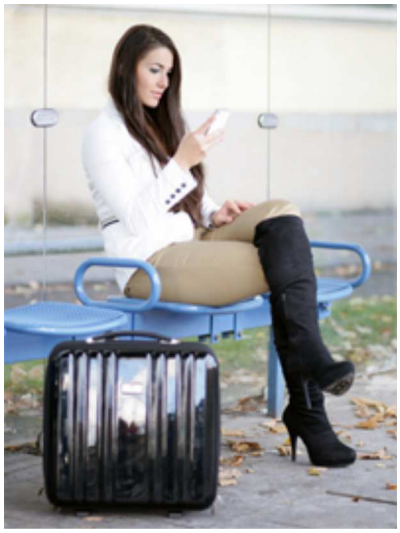

```
import random
from gpanel import *

n = 1000 # Number of simulations

def sim(x):
    # Random permutation [0..99]
    t = [0] * 100
    for i in range(0, 100):
        t[i] = i
    random.shuffle(t)
    best = max(t[0:x])
    for i in range(x, 100):
        if  t[i] > best:
            return [i, t[i]]
    return [99, t[99]]

makeGPanel(-10, 110, -10, 110)
title("Mittlere Qualifikation nach Warten auf Partner")
drawGrid(0, 100, 0, 100)

for x in range(1, 99):
    sum = 0
    repeat n:
        u = sim(x)
        sum += u[1]
    y = sum / n
    if x == 1:
       move(x, y)
    else:
       draw(x, y)
```

MEMO

Es zeigt sich ein völlig anders Bild: Du solltest dich nach dem Kriterium einer möglichst grossen mittleren Qualifikation bereits nach einer Lernphase von ungefähr 10 für den nächst besser bewerteten Partner entscheiden.

Du kannst die Simulation auf für eine realistischere Zahl von Partnern durchführen und bemerkst, dass die optimale Lernphase kurz ist.

WARTEZEITPARADOXON

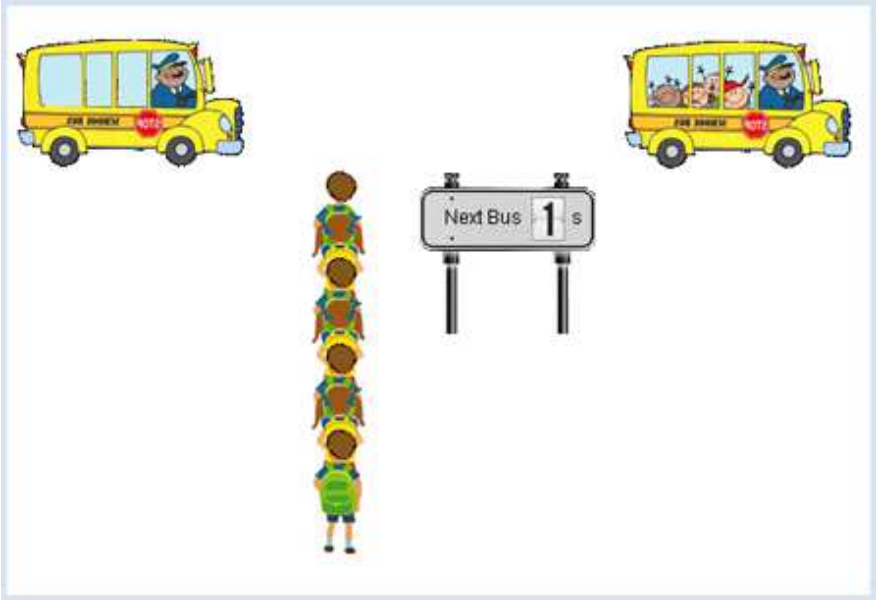


Das Warten an einer Haltestelle von öffentlichen Verkehrsmitteln (Bus, Tram, S-Bahn, U-Bahn) gehört zum Alltag. Wir wollen uns hier überlegen, wie gross die mittlere Wartezeit ist, falls du zu völlig zufälliger Zeit an die Haltestelle eines Busses kommst (also den Fahrplan nicht kennst). Dabei gehst du zuerst davon aus, dass die Busse exakt im Takt von 6 Minuten hintereinander fahren.

Es ist einleuchtend, dass du manchmal nur sehr kurz und manchmal maximal fast 6 Minuten warten musst. Im Mittel beträgt die Wartezeit also 3 Minuten. Wie steht es aber, wenn die Busse nicht mehr exakt im Takt fahren, sondern beispielsweise mit gleich verteilter Wahrscheinlichkeit im Bereich von 2 und 10 Minuten?

Da sie auch dann noch im Mittel alle 6 Minuten vorbei kommen, nimmst du vielleicht an, dass sich an der mittleren Wartezeit von 3 Minuten nichts ändert. Die erstaunliche, und darum paradoxe Antwort ist, dass die mittlere Wartezeit nun grösser als 3 Minuten wird.

In einer animierten Simulation sollst du herausfinden, wie gross die Wartezeit unter der Annahme ist, dass die Busse gleichverteilt im Bereich 2 und 10 Minuten (in der Simulation sind es Sekunden) hintereinander fahren. Dabei verwendest du mit Vorteil die Game-Library *JGameGrid* ein, da sich die beteiligten Objekte, wie Busse und Fahrgäste als Sprite-Objekte modellieren lassen.



Der Programmcode erfordert einige Erklärungen:

Da es sich beim Bus und den Fahrgästen offensichtlich um Objekte handelt, werden diese in den Klassen *Bus* bzw. *Passenger* modelliert. Du Busse werden am Ende des Hauptteil des Programms in einer Endlosschleife gemäss der statistischen Vorgaben erzeugt Schliesst man das Grafikfenster, so bricht die Endlosschleife wegen *isDisposed() = False* ab und das Programm endet.

Die Fahrgäste müssen periodisch erzeugt und in der Wartereihe angezeigt werden. Dazu schreibst du am besten eine Klasse *PassengerFactory* die von *Actor* abgeleitet wird. Diese besitzt zwar keine Spritebild, ihr *act()* kann aber dazu verwendet werden, die Fahrgäste zu erzeugen und ins GameGrid einzufügen. Mit dem Zykluszähler *nbCycles* kannst du die Periode wählen, mit der die Objekte erzeugt werden (der Simulationszyklus ist auf 50 ms eingestellt).

Im *act()* der Klasse *Bus* bewegst du den Bus vorwärts und prüfst mit der x-Koordinate, ob er bei der Haltestelle angekommen ist. Du rufst in diesem Moment die Methode *board()* der Klasse *PassengerFactory* auf, wodurch die wartenden Fahrgäste aus der Wartereihe entfernt werden. Gleichzeitig änderst du mit *show(1)* das Spritebild des Busses und zeigst die neue Wartezeit zum nachfolgenden Bus auf der Anzeigetafel an. Damit diese Aktionen nur einmal aufgerufen werden, verwendest du die boolesche Variable *isBoarded*.

Die Anzeigetafel als Instanz der Klasse *InformationPanel* ist ein zusätzliches Gadget, um den Fahrgästen und Zuschauern des Programms die Distanz zum nächst folgenden Bus anzuzeigen. Die Anzeige wird wieder in der Methode *act()* verändert, indem mit *show()* eines der 10 Spritebilder *digit_0.png* bis *digit_9.png* ausgewählt wird.

```
from gamegrid import *
import random
import time

min = 2
max = 10

def random_t():
    return min + (max - min) * random.random()

# ---------------- class PassengerFactory ----------
class PassengerFactory(Actor):
    def __init__(self):
        self.nbPassenger = 0

    def board(self):
        for passenger in getActors(Passenger):
            passenger.removeSelf()
            passenger.board()
        self.nbPassenger = 0

    def act(self):
        if self.nbCycles % 10 == 0:
            passenger = Passenger(random.randint(0, 1))
            addActor(passenger, Location(400, 120 + 27 * self.nbPassenger))
            self.nbPassenger += 1

# ---------------- class Passenger -----------------
class Passenger(Actor):
    totalTime = 0
    totalNumber = 0

    def __init__(self, i):
        Actor.__init__(self, "sprites/pupil_" + str(i) + ".png")
        self.createTime = time.clock()

    def board(self):
        self.waitTime = time.clock() - self.createTime
        Passenger.totalTime += self.waitTime
        Passenger.totalNumber += 1
        mean = Passenger.totalTime / Passenger.totalNumber
        setStatusText("Mean waiting time: " + str(round(mean, 2)) + " s")

# ---------------- class Car ----------------------
class Bus(Actor):
    def __init__(self, lag):
        Actor.__init__(self, "sprites/car1.gif")
        self.lag = lag
        self.isBoarded = False

    def act(self):
        self.move()
        if self.getX() > 320 and not self.isBoarded:
            passengerFactory.board()
```

```
            self.isBoarded = True
            infoPanel.setWaitingTime(self.lag)
        if self.getX() > 1650:
            self.removeSelf()

# ---------------- class InformationPanel ----------
class InformationPanel(Actor):
    def __init__(self, waitingTime):
        Actor.__init__(self, "sprites/digit.png", 10)
        self.waitingTime = waitingTime

    def setWaitingTime(self, waitingTime):
        self.waitingTime = waitingTime

    def act(self):
        self.show(int(self.waitingTime + 0.5))
        if self.waitingTime > 0:
            self.waitingTime -= 0.1

periodic = askYesNo("Departures every 6 s?")
makeGameGrid(800, 600, 1, None, None, False)
addStatusBar(20)
setStatusText("Acquiring data...")
setBgColor(Color.white)
setSimulationPeriod(50)
show()
doRun()
if periodic:
    setTitle("Warting Time Paradoxon - Departure every 6 s")
else:
    setTitle("Waiting Time Paradoxon - Departure between 2 s and 10 s")

passengerFactory = PassengerFactory()
addActor(passengerFactory, Location(0, 0))

addActor(Actor("sprites/panel.png"), Location(500, 120))
addActor(TextActor("Next Bus"), Location(460, 110))
addActor(TextActor("s"), Location(540, 110))
infoPanel = InformationPanel(4)
infoPanel.setSlowDown(2)
addActor(infoPanel, Location(525, 110))

while not isDisposed():
    if periodic:
        lag = 6
    else:
        lag = random_t()
    bus = Bus(lag)
    addActor(bus, Location(-100, 40))
    a = time.clock()
    while time.clock() - a < lag and not isDisposed():
        delay(10)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

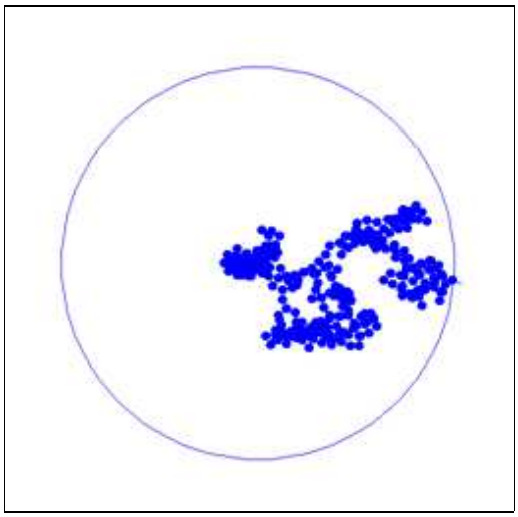
MEMO

Die Simulation zeigt, dass die mittlere Wartezeit mit rund 3.5 Minuten deutlich länger als die vermuteten 3 Minuten ist. Du kannst dir diese Verlängerung wie folgt erklären: Kommen die Busse gleichverteilt einmal mit 2 Minuten und mit 10 Minuten Abstand, so ist es viel wahrscheinlicher, dass deine Ankunft in das Warteintervall von 2 bis 10 Minuten fällt als in das Warteintervall von 0 bis 2 Minuten. Daher wartest du sicher länger als 3 Minuten.

AUFGABEN

1. Du kriegst mit der Wahrscheinlichkeit von je 1/3 jeden Tag eine Zehn-, Zwanzig- oder Fünfzig-Rappen-Münze. Wie viele Tage vergehen im Mittel, bis du dir damit ein Buch, das zehn Franken kostet, kaufen kannst.

2. Ein verirrter Mensch startet in der Mitte des Fensterns und bewegt sich bei jedem Schritt 10 Pixel weit in zufälliger Richtung (random walk). Welches ist die mittlere Wartezeit u, bis er zum ersten mal weiter als die Distanz r vom Start entfernt ist? Simuliere die Bewegung mit einer (versteckten) Turtle für die Werte r = 100, 200, 300. Welchen Zusammenhang vermutest du zwischen r und u?



3. Modifiziere das Programm zum Wartezeitenparadoxon so, dass die Busse mit gleicher Wahrscheinlichkeit 1/2 entweder 2 Sekunden oder 10 Sekunden auseinander liegen und bestimme die mittlere Wartezeit.

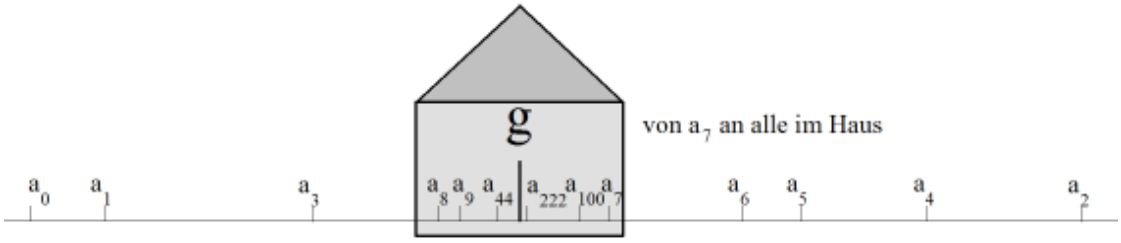
8.5 FOLGEN, KONVERGENZ

EINFÜHRUNG

Aneinandergereihte Zahlen faszinieren die Menschen seit langer Zeit. Bereits in der Antike um 200 v. Chr. hat Achimedes die Zahl Pi durch eine Zahlenfolge angenähert, indem er den Umfang der in den Kreis eingezeichneten regelmässigen Vielecke mit zunehmender Eckenzahl betrachtete. Bereits für ihn war offenbar klar, dass der Kreis als Grenzfall eines regelmässigen Vielecks mit immer grösser werdender Eckenzahl aufgefasst werden kann und damit auch der Umfang der Vielecke gegen den Umfang des Kreises **streben** musste.

Eine Zahlenfolge besteht aus den Zahlen a_0, a_1, a_2, usw., also aus den Gliedern a_n mit dem Index n = 0, 1, 2, usw. (Manchmal beginnt man auch mit n = 1.) Ein Bildungsgesetz bestimmt eindeutig, wie gross jede der Zahlen ist. Ist a_k für alle noch so grossen natürlichen Zahlen definiert, so spricht man von einer unendlichen Zahlenfolge. Das Gesetz kann als expliziter Ausdruck von n angegeben sein. Ein Glied kann sich aber auch aus vorhergehenden Gliedern berechnen lassen (rekursives Gesetz). In diesem Fall müssen auch noch Startwerte bekannt sein.

Eine konvergente Zahlenfolge hat eine sehr anschauliche Eigenschaft: Es gibt für sie eine eindeutige Zahl g, genannt Grenzwert, an die sich die Glieder immer mehr annähern, was so zu verstehen ist, dass du ein beliebig kleines Umgebungsintervall von g wählen kannst und alle Glieder von einem gewissen n nicht mehr aus diesem Intervall herausfallen. Die Umgebung kannst du dir wie ein Haus um den Grenzwert vorstellen: Die Folge kann vorerst eventuell wilde Sprünge machen, aber von einem bestimmten n an befinden sich alle Glieder im noch so kleinen Haus.



Zahlenfolgen können mit dem Computer experimentell untersucht werden. Zur Veranschaulichung verwendest du verschiedene grafische Darstellungen: Du kannst beispielsweise wie im oberen Bild auf einer Achse die Glieder als Striche oder Punkte eintragen und untersuchen, ob sich ein **Häufungspunkt** ergibt. Du kannst aber auch in einem x-y-Diagramm die n auf der x-Achse und a_n auf der y-Achse als Punkte eintragen und untersuchen, wie sich der Graf **für grosse Werte von n** verhält

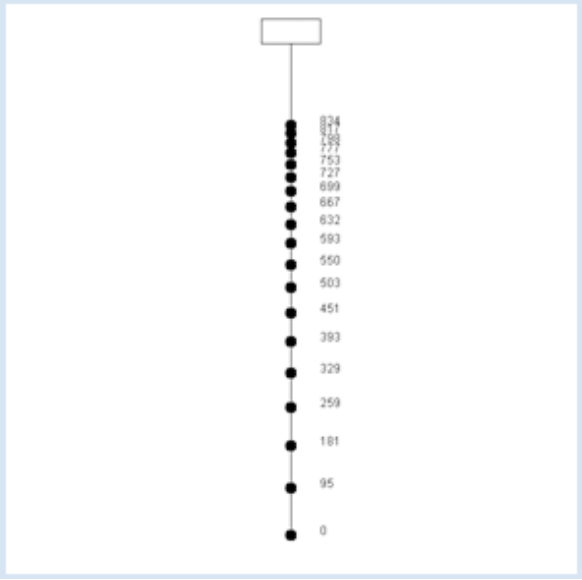
PROGRAMMIERKONZEPTE: *Häufungspunkt, Konvergenz, Feigenbaum-Diagramm, Chaos*

DER JÄGER UND SEIN HUND

Ein Jäger spaziert mit der Geschwindigkeit u = 1 m/s mit seinem Hund zur d = 1000 m entfernten Jagdhütte. Weil der Jäger für den Hund allerdings deutlich zu langsam läuft, verfährt der Hund wie folgt: Er läuft allein mit der Geschwindigkeit u = 20 m/s bis zur Jagdhütte, kehrt dort unverzüglich um und läuft seinem Herrn entgegen. Sobald er diesen erreicht, dreht er wieder um, läuft bis zur Jagdhütte und so weiter.

Du möchtest mit einem Programm diesen Vorgang simulieren. In deinem Programm zeichnest du bei jedem Tastendruck die Lage des Jägers beim Zusammentreffen mit dem Hund als schwarzen Punkt ein und schreibst daneben die Position aus. Die aufeinander folgenden Werte von a bilden eine Zahlenfolge, deren Verhalten du studieren willst. Da für Jäger und Hund zwischen zwei Zusammentreffen die gleiche Zeit verstreicht, gilt für die Zunahme der Position des Jägers:

$$\frac{da}{u} = \frac{2*(d-a)-da}{v}$$



woraus du mit wenig Algebra die folgende Beziehung nachweisen kannst:

$$da = c*(d-x) \qquad \text{mit} \qquad c = \frac{2*u}{u+v}$$

Wie zu erwarten ist, häufen sich die Zahlen a_n gegen die Grenzzahl 1000.

```
from gpanel import *

u = 1 # m/s
v = 20 # m/s
d = 1000  # m
a = 0     # hunter
h = 0     # dog
c = 2 * u / (u + v)
it = 0

makeGPanel(-50, 50, -100, 1100)
title("Hunter-Dog problem")
line(0, 0, 0, 1000)
line(-5, 1000, 5, 1000)
line(-5, 1050, 5, 1050)
line(-5, 1000, -5, 1050)
line(5, 1000, 5, 1050)

while not isDisposed():
    move(0, a)
    fillCircle(1)
    text(5, a, str(int(a)))
    getKeyWait()
    da = c * (d - a)
    dh = 2 * (d  - a) - da
    h += dh
    a += da
    it += 1
    title("it = " + str(it) + "; hunter = " + str(a) +
          " m; dog = " + str(h) + " m")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Man sagt, dass die Folge der Zahlen a_n **konvergiert** und dass ihr Grenzwert 1000 ist. Überlege dir, warum diese Problemstellung theoretischer Natur ist. Sie entspricht aber der antiken Anekdote, wonach Achilles zum Wettlauf mit einer Schildkröte aufgefordert wurde. Da er (immerhin) zehnmal so schnell war wie die Schildkröte, sollte diese allerdings zehn Meter Vorsprung erhalten. Achilles weigerte sich anzutreten mit der Begründung, er habe keine Chance, die Schildkröte einzuholen. Er argumentierte so: In der Zeit, die er für die ersten 10 m benötige, sei die Schildkröte schon wieder 1m voraus. In der Zeit, die er für diesen Meter benötige, sei sie schon wieder 10 cm weiter. In der Zeit, die er für diese 10 cm benötige, wäre sie wieder 1cm voraus und so weiter. Was meinst du dazu?

DAS FEIGENBAUM-DIAGRAMM

Im Zusammenhang mit der Populationsdynamik hast du das logistische Wachstum kennengelernt. Dabei wird die Populationsgrösse *xnew* in der nächsten Generation aus der aktuellen Grösse x aus einer quadratischen Beziehung berechnet. Im Folgenden wird der Zusammenhang vereinfacht:

$$xnew = r * x * (1 - x)$$

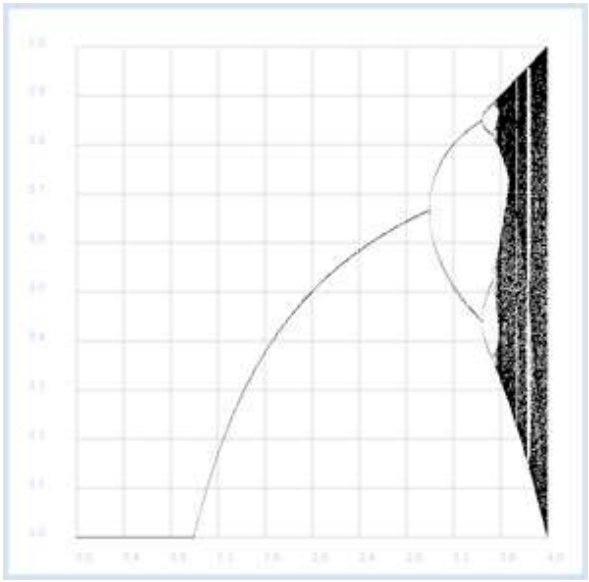
wo r ein frei wählbarer Parameter ist. Du stellst dir die interessante Frage, ob die daraus entstehende rekursiv definierte Folge

$$a_{n+1} = r * a_n * (1 - a_n)$$

mit $a_0 = 0.5$ konvergiert und welches in diesem Fall ihr Grenzwert ist.

Du untersuchst das Verhalten mit einem extrem einfachen Programm, wo du für 1000 äquidistante Werte von r im Bereich 0 bis 4 die ersten 1000 Glieder der Folge als Punkte aufzeichnest.

Du beginnst bei einem festen r immer mit dem gleichen Startwert a0 = 0.5 und zeichnest die Glieder erst von der n = 500 an, da du dich nur dafür interessierst, ob die Folge konvergent oder divergent ist.



```
from gpanel import *

def f(x, r):
     return r * x * (1 - x)

makeGPanel(-0.6, 4.4, -0.1, 1.1)
title("Tree Diagram")
drawGrid(0, 4.0, 0, 1.0, "gray")
for z in range(1001):
    r = 4 * z / 1000
    a = 0.5
    for i in range(1001):
        a = f(a, r)
        if i > 500:
            point(r, a)
```

MEMO

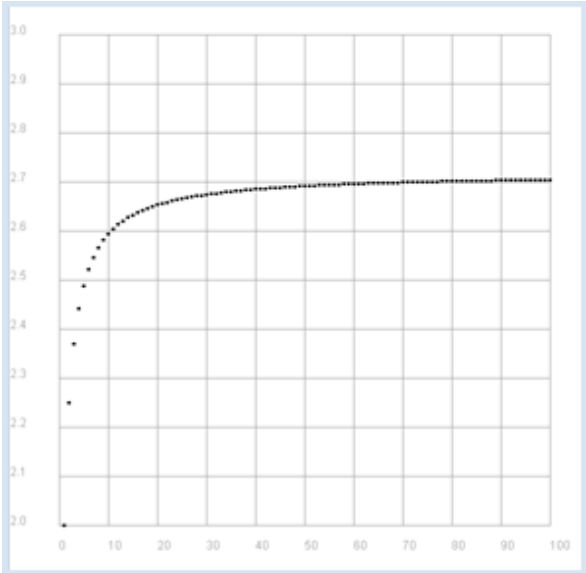
Im Experiment erkennst du für ein bestimmtes r die Häufungspunkte der Folge. Du kannst auf Grund der Computersimulation folgende Vermutungen aufstellen: Für r < 1 gibt es einen Häufungspunkt bei 0, die Folge konvergiert also gehen 0. Im Bereich zwischen 1 und r konvergiert die Folge ebenfalls. Für noch grössere r gibt es vorerst zwei und später mehr Häufungspunkte, die Folge konvergiert also nicht mehr. Für noch grössere Werte von r springt die Folge chaotisch hin und her.

DIE EULERSCHE ZAHL

Ein der berühmtesten Folgen bilden die Zahlen mit dem Bildungsgesetz:

$$a_n = (1+\frac{1}{n})^n$$ mit n = 1, 2, 3, ...

Intuitiv ist es gar nicht klar, was diese Folge mit grösser werdendem n macht, denn einerseits nähert sich 1 + 1/n immer mehr der Zahl 1, dafür wird diese Zahl aber mit einem immer grösseren Exponenten potenziert. Das Rätsel kannst du mit einem einfachen Computerexperiment lüften.




```
from gpanel import *

def a(n):
     return (1 + 1/n)**n

makeGPanel(-10, 110, 1.9, 3.1)
title("Euler Number")
drawGrid(0, 100, 2.0, 3.0, "gray")
for n in range(1, 101):
    move(n, a(n))
    fillCircle(0.5)
```

MEMO

Die Folge $a_n = (1+\frac{1}{n})^n$ konvergiert gegen eine Zahl in der Grössenordnung von 2.7.

Es handelt sich um die **Eulersche Zahl e**, die wohl berühmteste Zahl überhaupt.

SCHNELL KONVERGIERENDE FOLGEN ZUR BERECHNUNG VON PI

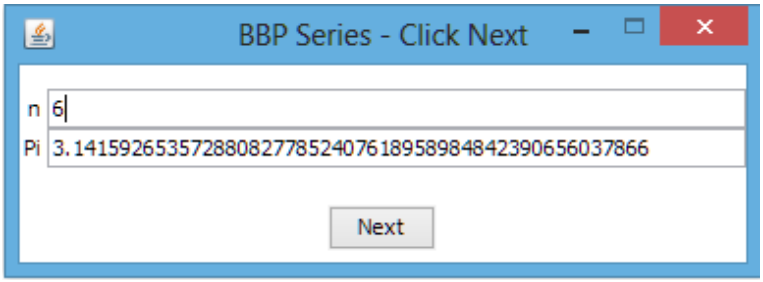
Die Berechnung von π auf möglichst viele Stellen stellt seit dem Altertum eine Herausforderung dar. Erst 1995 entdeckten die Mathematiker Bailey, Borwein und Plouffe eine Summenformel, die BBP-Formel. Sie bewiesen, dass man π exakt als Grenzwert einer Folge erhält, deren n-tes Glied die Summe von

$$\frac{1}{16^k}\left(\frac{4}{8k+1}-\frac{2}{8k+4}-\frac{1}{8k+5}-\frac{1}{8k+6}\right)$$ von k = 0 bis k = n ist

Dein Programm verwendet die Python-Klasse **Decimal**, die Dezimalzahlen mit hoher Genauigkeit zur Verfügung stellt. Aus einem Integer oder Float erzeugt der Konstruktor eine solche Zahl, wobei die üblichen mathematischen Operationszeichen direkt verwendbar sind.

Mit *getcontext().prec* legt man die Genauigkeit fest. Diese entspricht ungefähr der Stellenzahl der verwendeten Dezimalzahlen.

Dein Programm berechnet bei jedem Tastendruck das nächste Folgeglied und stellt den Wert in einem EntryDialog dar.




```
from entrydialog import *
from decimal import *
getcontext().prec = 50

def a(k):
    return 1/16**Decimal(k) * (4 / (8 * Decimal(k) + 1) - 2
    / (8 * Decimal(k) + 4) - 1
    / (8 * Decimal(k) + 5) - 1 / (8 * Decimal(k) + 6))

inp = IntEntry("n", 0)
out = StringEntry("Pi")
pane0 = EntryPane(inp, out)
btn = ButtonEntry("Next")
pane1 = EntryPane(btn)
dlg = EntryDialog(pane0, pane1)
dlg.setTitle("BBP Series - Click Next")
n = 0
s = a(0)
out.setValue(str(s))
while not dlg.isDisposed():
    if btn.isTouched():
       n = inp.getValue()
       if n == None:
           out.setValue("Illegal entry")
       else:
           n += 1
           s += a(n)
           inp.setValue(n)
           out.setValue(str(s))
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Bereits mit 40 Iterationen verändert sich die angezeigte Zahl für π nicht mehr.

Das Programm endet beim Schliessen des Anzeigefensters da *isDisposed() True* ist.

AUFGABEN

1. Die Fibonacci-Folge ist so definiert, dass ein Glied gleich der Summe der beiden Vorgänger ist, wobei das erste und zweite Glied 1 sind. Berechne die ersten 30 Glieder und stelle sie in einer x-y-Grafik dar.

2. Die Fibonacci-Folge divergiert, hingegen konvergiert die Folge der Quotienten zweier aufeinander folgenden Glieder. Stelle wie in Aufgabe 1 diese Quotientenfolge grafisch dar und bestimme den ungefähren Wert des Grenzwerts.

3. Betrachte mit dem Startwert a0 = 1 die Folge

$$a_{n+1} = \frac{1}{2} * (a_n + \frac{2}{a_n})$$

Zeichne die Glieder als Punkte auf einer Zahlengeraden und schreibe ihren Wert in ein Ausgabefenster. Wie du siehst, konvergiert die Folge offenbar. Du kannst den Grenzwert x etwas grosszügig wie folgt berechnen: Für grosse n dürfen sich nachfolgende Glieder kaum mehr unterscheiden, es gilt also im Grenzfall

$$x = \frac{1}{2} * (x + \frac{2}{x})$$ und aufgelöst $x = \sqrt{2}$

Formuliere dieses Resultat als Anleitung, wie man die Quadratwurzel von 2 näherungsweise mit den 4 Grundrechenoperationen bestimmen kann. Wie ändert sich der Algorithmus zur Bestimmung der Quadratwurzel aus einer beliebigen Zahl z?

ZUSATZSTOFF

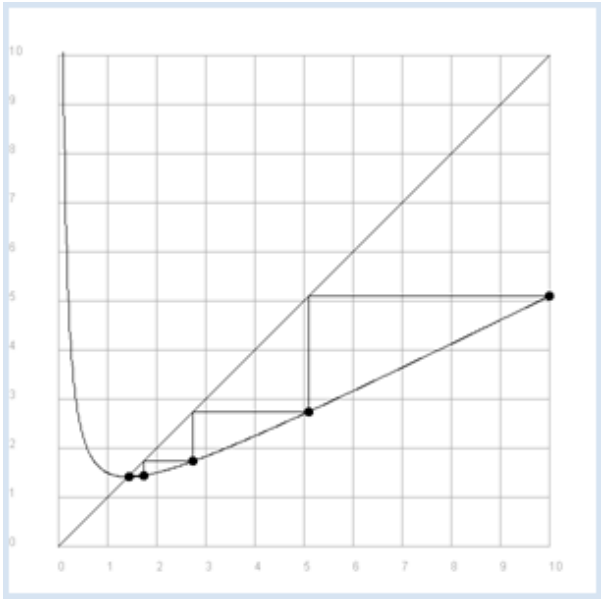
ITERATIVE LÖSUNG EINER GLEICHUNG

Wie du in Aufgabe 3 gesehen hast, ist $x = \sqrt{2}$ die Lösung der Gleichung

$$x = f(x) \quad \text{mit} \quad f(x) = \frac{1}{2} * (x + \frac{2}{x})$$

Es ist illustrativ, das Verfahren zur Lösung dieser Gleichung grafisch darzustellen. Dazu zeichnest du im gleichen Koordinatensystem sowohl den Funktionsgrafen y = f(x) und die Winkelhalbierende y = x ein. Die Lösung ist der Schnittpunkt der beiden Kurven.

Die iterative Lösung entspricht dem sukzessiven Durchlauf eines Punkts auf dem Funktionsgrafen horizontal hin zur Winkelhalbierenden und vertikal nach unten zum nächsten Punkt auf dem Funktionsgrafen.



```
from gpanel import *

def f(x):
    return 1 / 2 * (x + 2 / x)

makeGPanel(-1, 11, -1, 11)
title("Iterative square root begins at x = 10. Press a  key...")
drawGrid(0, 10, 0, 10, "gray")

for i in range(10, 1001):
    x = 10 / 1000 * i
    if i == 10:
        move(x, f(x))
    else:
        draw(x, f(x))

line(0, 0, 10, 10)

x = 10
move(x, f(x))
fillCircle(0.1)
it = 0
while not isDisposed():
    getKeyWait()
    it += 1
    xnew = f(x)
    line(x, f(x), xnew, f(x))
    line(xnew, f(x), xnew, f(xnew))
    x = xnew
    move(x, f(x))
    fillCircle(0.1)
    title("Iteration " + str(it) + ": x = " + str(x))
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du siehst deutlich, dass sich die Punkte bei jedem Tastendruck sehr schnell zum Schnittpunkt hin bewegen. Bereits mit wenigen Iterationen ergibt sich eine Lösung mit 10-stelliger Genauigkeit.

8.6 KORRELATION, REGRESSION

EINFÜHRUNG

In den Naturwissenschaften, aber auch im täglichen Lebens spielt die Erfassung von Daten in Form von Messgrössen eine wichtige Rolle. Dabei braucht aber nicht immer ein Messinstrument zum Einsatz zu kommen. Es kann sich beispielsweise auch um Umfragewerte im Zusammenhang mit einer statistischen Untersuchung handeln. Nach der Datenerfassung geht es meist darum, die Messwerte zu **interpretieren**. Es kann sich um qualitative Aussagen handeln, beispielsweise dass die Messwerte im Laufe der Zeit ansteigen oder absinken, aber auch um die experimentelle Überprüfung eines Naturgesetzes.

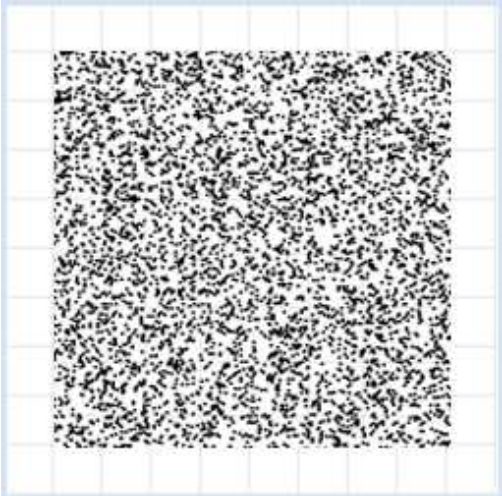
Ein grosses Problem besteht darin, dass Messungen fast immer Schwankungen ausgesetzt und nicht exakt reproduzierbar sind. Trotz dieses "Messfehlern" möchte man wissenschaftlich korrekte Aussagen machen können. Die Messfehler entstehen dabei nicht immer wegen mangelhaften Messgeräten, sondern können in der Natur des Experiments liegen. So liefert die "Messung" der Augenzahl eines geworfenen Würfels grundsätzlich streuende Werte zwischen 1 und 6. Bei jeder Art von Messung spielen darum statistische Überlegungen eine zentrale Rolle.

Oft werden in einer Messserie zwei Grössen x und y miteinander gemessen und man stellt sich die Frage, ob diese in einem Zusammenhang stehen und einer Gesetzmässigkeit unterworfen sind. Trägt man die (x, y)-Werte als Messpunkte in ein Koordinatensystem ein, so spricht man von **Datenvisualisierung**. Fast immer erkennt man bei blosser Betrachtung der **Verteilung der Messwerte**, ob die Daten in einer gegenseitigen Abhängigkeit stehen.

PROGRAMMIERKONZEPTE: *Datenvisualisierung, Messwertverteilung, Wolkendiagramm, Rauschen, Kovarianz, Korrelationskoeffizient, Regression, Bester Fit*

UNABHÄNGIGE UND ABHÄNGIGE DATEN VISUALISIEREN

Du kannst leicht in einem x-y-Diagramm unabhängige Daten simulieren, indem du für x und y gleichverteilte Zufallszahlen verwendest. Obschon es hier nicht nötig ist, kopierst du die Messwerte in die Datenlisten *xval* und *yval* und stellst sie erst danach als Datenpunkte dar.

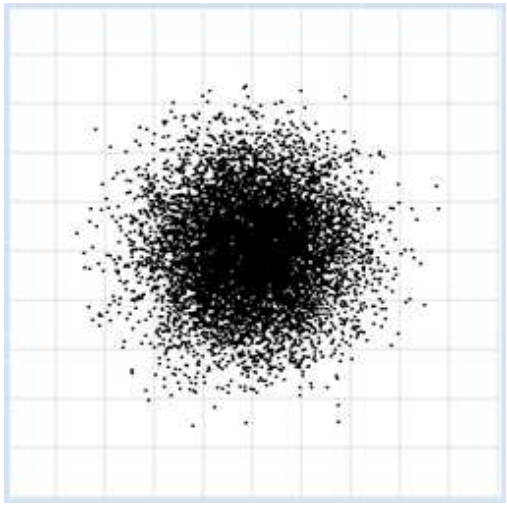


```
import random
from gpanel import *

z = 10000

makeGPanel(-1, 11, -1, 11)
title("Gleichverteilte Wertepaare")
drawGrid(10, 10, "gray")
```

```
xval = [0] * z
yval = [0] * z

for i in range(z):
    xval[i] = 10 * random.random()
    yval[i] = 10 * random.random()
    move(xval[i], yval[i])
    fillCircle(0.03)
```

Galaxienähnliche Grafiken erhältst du, falls die x- und y-Werte zwar immer noch unabhängig voneinander sind, aber ihre Verteilung um einen Mittelwert normalverteilt ist. Hier wird als Mittelwert 5 und als Streuung 1 verwendet. Man spricht auch von einem **Scatter Plot** oder **Wolkendiagramm**.

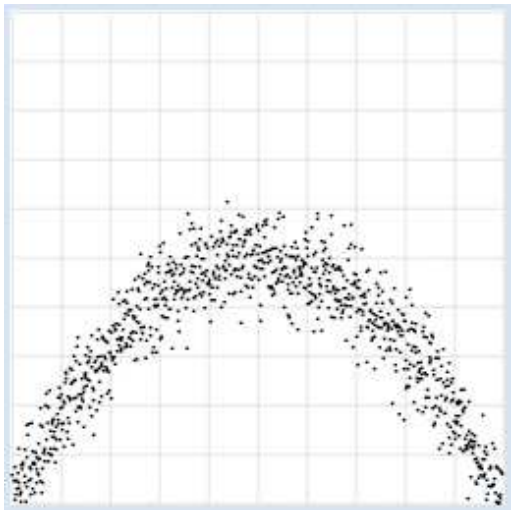
Mit der Funktion *random.gauss(mittelwert, streuung)* kannst du sehr einfach normalverteilte Zufallszahlen erzeugen.



```
import random
from gpanel import *

z = 10000

makeGPanel(-1, 11, -1, 11)
title("Normalverteilte Wertepaare")
drawGrid(10, 10, "gray")

xval = [0] * z
yval = [0] * z

for i in range(z):
    xval[i] = random.gauss(5, 1)
    yval[i] = random.gauss(5, 1)
    move(xval[i], yval[i])
    fillCircle(0.03)
```

Abhängigkeiten zwischen x- und y-Werten kannst du so simulieren, dass x in einem bestimmten Bereich in äquidistanten Schritten ansteigt und y eine Funktion von x ist, die aber statistischen Schwankungen unterworfen ist. Solche Schwankungen nennt man in der Physik auch **Rauschen**.

Du zeichnest das Wolkendiagramm für eine Parabel y = -x * (0.2 * x - 2) und normalverteiltem Rauschen im Bereich x = 0..10 mit einer Schrittweite von 0.01.



```
import random
from gpanel import *
```

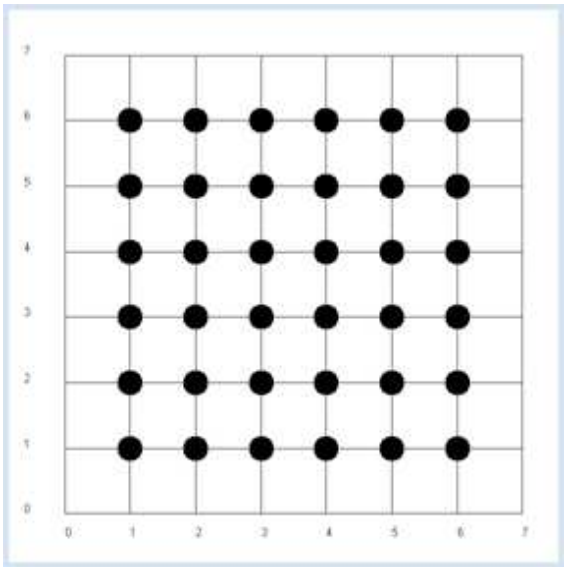
```
import math

z = 1000
a = 0.2
b = 2

def f(x):
    y = -x * (a * x - b)
    return y

makeGPanel(-1, 11, -1, 11)
title("y = -x * (0.2 * x - 2) mit normalverteiltem Rauschen")
drawGrid(0, 10, 0, 10, "gray")

xval = [0] * z
yval = [0] * z

for i in range(z):
    x = i / 100
    xval[i] = x
    yval[i] = f(x) + random.gauss(0, 0.5)
    move(xval[i], yval[i])
    fillCircle(0.03)
```

MEMO

Durch Datenvisualisierung erkennt man gegenseitige Abhängigkeiten von Messgrössen auf den ersten Blick. Bei im Intervall [a, b] gleichverteilten Zufallszahlen kommen die Zahlen in jedem gleichlangen Teilintervall mit gleicher Häufigkeit vor. Normalverteilte Zahlen haben eine glöckenförmige Verteilung, wobei 68% der Zahlen im Intervall (Mittelwert - Streuung) und (Mittelwert + Streuung) liegen.

KOVARIANZ ALS MASSZAHL FÜR ABHÄNGIGKEITEN

Hier geht es darum, gegenseitige Abhängigkeiten nicht nur in einem Diagramm sichtbar zu machen, sondern auch durch eine Zahl auszudrücken. Wie schon oft, gehts du von einem konkreten Beispiel aus und betrachtest den Doppelwurf von Würfeln, wobei vorerst x die Augenzahl des ersten und y die Augenzahl des zweiten Würfels sind. Du führst das Wurfexperiment oftmals aus und zeichnest die Wertepaare als Wolkendiagramm. Da die beiden Messwerte x und y unabhängig sind, ergibt sich eine regelmässige Punktwolke. Berechnest du die Erwartungswerte von x und y, so ergibt sich bekanntlich 3.5.



Da x und y unabhängig, gilt für die Wahrscheinlichkeit p_{xy}, beim Doppelwurf das Paar x, y zu erhalten $p_{xy} = p_x * p_y$.

Die Vermutung liegt nahe, dass allgemein die folgende **Produktregel** gilt.

Sind x und y unabhängig, so ist der Erwartungswert von x*y gleich dem Produkt der Erwartungswerte von x und y [**mehr...**] .

In der Simulation wird diese Vermutung bestätigt.

```
from random import randint
from gpanel import *

z = 10000 # Anzahl Doppelwürfe

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

makeGPanel(-1, 8, -1, 8)
title("Doppelwurf. Unabhängige Zufallsvariablen")
addStatusBar(30)
drawGrid(0, 7, 0, 7, 7, 7, "gray")

xval =  [0] * z
yval =  [0] * z
xyval = [0] * z

for i in range(z):
    a = randint(1, 6)
    b = randint(1, 6)
    xval[i] = a
    yval[i] = b
    xyval[i] = xval[i] * yval[i]
    move(xval[i], yval[i])
    fillCircle(0.2)

xm = mean(xval)
ym = mean(yval)
xym = mean(xyval)
setStatusText("E(x) = " + dec2(xm) + \
          ",  E(y) = " + dec2(ym) + \
          ",  E(x, y) = " + dec2(xym))
```

MEMO

Es ist zweckmässig, eine Statusbar zu verwenden, um die Resultate auszuschreiben. Die Funktion *dec2()* rundet die Werte auf 2 Stellen und gibt sie als String ab.

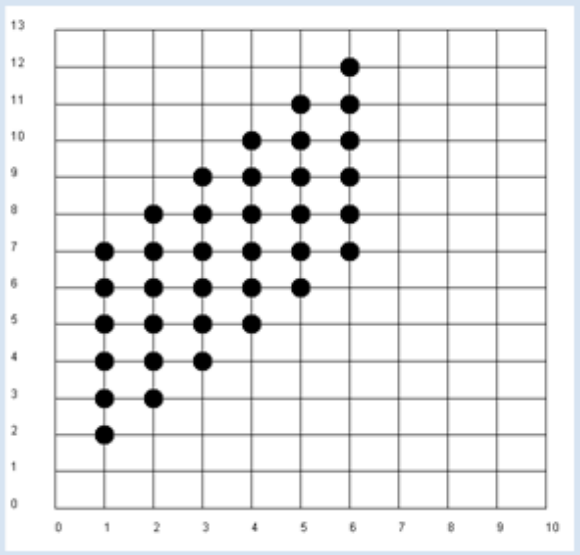
Die Werte unterliegen natürlich einer statistischen Schwankung.

In der nächsten Simulation untersuchst du nicht mehr die geworfenen Augenpaare, sondern die Augenzahl x des ersten Würfels und die Summe y der beiden Augenzahlen. Es ist offensichtlich, dass nun y in einer Abhängigkeit von x ist, denn wenn beispielsweise x = 1 ist, so ist die Wahrscheinlichkeit für y = 4 nicht dieselbe wie wenn x = 2 ist.

Deine Simulation bestätigt, dass die Produktregel tatsächlich nicht mehr gilt, falls x und y voneinander abhängig sind. Es liegt daher nahe, die Abweichung von der Produktregel, also die Differenz

$$c = E(x*y) - E(x)*E(y)$$

Kovarianz genannt, als Mass für die Abhängigkeit von x und y einzuführen. Gleichzeitig siehst du in deinem Programm, dass die Kovarianz auch als Summe der quadratischen Abweichungen vom Mittelwert berechnet werden kann.




```
from random import randint
from gpanel import *

z = 10000 # Anzahl Doppelwürfe

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

makeGPanel(-1, 11, -2, 14)
title("Doppelwurf. Abhängige Zufallsvariablen")
addStatusBar(30)
drawGrid(0, 10, 0, 13, 10, 13, "gray")
xval =  [0] * z
yval =  [0] * z
xyval = [0] * z
for i in range(z):
    a = randint(1, 6)
    b = randint(1, 6)
    xval[i] = a
    yval[i] = a + b
    xyval[i] = xval[i] * yval[i]
    move(xval[i], yval[i])
    fillCircle(0.2)
xm = mean(xval)
ym = mean(yval)
xym = mean(xyval)
c = xym - xm * ym
setStatusText("E(x) = " + dec2(xm) + \
          ",  E(y) = " + dec2(ym) + \
          ",  E(x, y) = " + dec2(xym) + \
          ",  c = " + dec2(c) + \
          ",  covariance = " + dec2(covariance(xval, yval)))
```

MEMO

Du erhältst in der Simulation für die Kovarianz den Wert von ungefähr 2.9. Die Kovarianz eignet sich also sehr wohl als Mass für die Abhängigkeit von Zufallsvariablen.

DER KORRELATIONSKOEFFIZIENT

Die eben eingeführte Kovarianz hat noch einen Nachteil. Je nachdem wie die Messgrössen x und y skaliert sind, ergeben sich andere Werte, auch wenn die Werte offenbar gleichermassen voneinander abhängig sind. Du kannst dies leicht einsehen. Nimmst du beispielsweise zwei Würfel, die statt der Augenzahlen 1, 2,...,6 die Augenzahlen 10, 20,...,60 haben, so ändert sich die Kovarianz stark, obschon die Abhängigkeit dieselbe ist. Aus diesem Grund hat führt man eine **normierte Kovarianz** ein, die man **Korrelationskoeffizient** nennt, indem man die Kovarianz durch die Streuungen der x und y-Werte dividiert:

$$Korrelationskoeffizient(x, y) = \frac{Kovarianz(x, y)}{Streuung(x) * Streuung(y)}$$

Der Korrelationskoeffizient bleibt immer im Bereich -1 bis 1. Ein Wert nahe bei 0 entspricht einer kleinen Abhängigkeit, ein Wert nahe bei 1 einer grossen Abhängigkeit, wobei zunehmende Werte von x zunehmenden Werte von y entsprechen, für einen Wert nahe bei -1 entsprechen zunehmende Werte von x abnehmenden Werten von y.

Im Programm verwendest du wieder den Doppelwurf und untersuchst die Abhängigkeit der Summe der Augenzahlen von der Augenzahl des ersten Würfels.

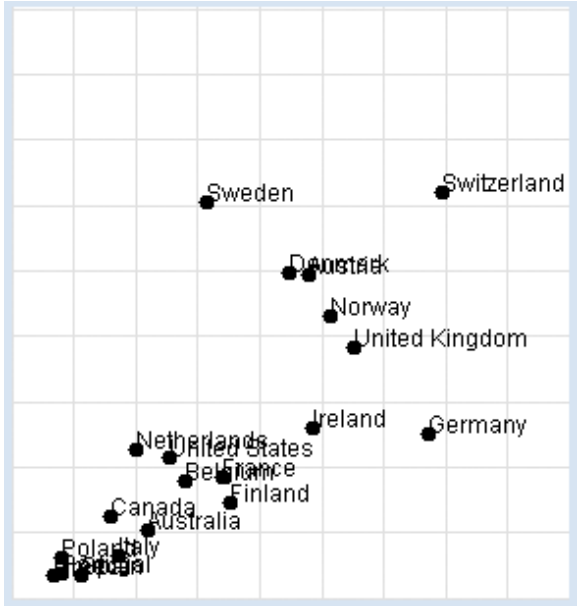
```
from random import randint
from gpanel import *
import math

z = 10000 # Anzahl Doppelwürfe
k = 10 # Scalefactor

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx
```

```
def correlation(xval, yval):
    return covariance(xval, yval) / (deviation(xval) * deviation(yval))

makeGPanel(-1 * k, 11 * k, -2 * k, 14 * k)
title("Doppelwurf. Abhängige Zufallsvariablen")
addStatusBar(30)
drawGrid(0, 10 * k, 0, 13 * k, 10, 13, "gray")

xval =  [0] * z
yval =  [0] * z
xyval = [0] * z

for i in range(z):
    a = k * randint(1, 6)
    b = k * randint(1, 6)
    xval[i] = a
    yval[i] = a + b
    xyval[i] = xval[i] * yval[i]
    move(xval[i], yval[i])
    fillCircle(0.2 * k)

xm = mean(xval)
ym = mean(yval)
xym = mean(xyval)
c = xym - xm * ym
setStatusText("E(x) = " + dec2(xm) + \
          ",  E(y) = " + dec2(ym) + \
          ",  E(x, y) = " + dec2(xym) + \
          ",  covariance = " + dec2(covariance(xval, yval)) + \
          ",  correlation = " + dec2(correlation(xval, yval)))
```

MEMO

Du kannst den Skalenfaktor k verändern und der Korrelationskoeffzient bleibt immer ungefähr 0.71, hingegen ändert sich die Kovarianz stark. Statt Korrelationskoeffizient sagt man auch nur kurz **Korrelation**.

MEDIZINWISSENSCHAFTLICHE PUBLIKATION



Mit deinen bisherigen Kenntnissen bist du bereits in der Lage, eine wissenschaftlichen Publikation, die im Jahr 2012 in der renommierten Zeitschrift "New England Journal of Medicine" erschienen ist, [**mehr...**] zu verstehen und zu beurteilen. Dabei wird der Zusammenhang zwischen dem Schokoladekonsum und der Zahl der Nobelpreisträger in verschiedenen Industriestaaten untersucht, oder anders gesagt, wird der Frage nachgegangen, ob es einen Zusammenhang zwischen Schokoladeessen und Intelligenz gibt. Im Artikel geht der Autor von folgenden Datenquellen aus, die du auch selbst im Internet findet:

Nobelpreise:
http://en.wikipedia.org/wiki/List_of_countries_by_Nobel_laureates_per_capita

Schokoladekonsum:

http://www.chocosuisse.ch/web/chocosuisse/en/documentation/facts_figures.html
http://www.theobroma-cacao.de/wissen/wirtschaft/international/konsum

Du willst die Untersuchung selbst nachvollziehen. In deinem Programm verwendest du für den Ländernamen, den Schokoladekonsum in kg pro Jahr und Einwohner und die Zahl der Nobelpreisträger pro 10 Millionen Einwohner eine Liste *data* mit Teillisten aus drei Elementen. Im Programm stellst du die Daten grafisch dar und bestimmst den Korrelationskoeffizienten.

```
import random
from gpanel import *
import math

data = [["Australia", 4.8, 5.141],
        ["Austria", 8.7, 24.720],
        ["Belgium", 5.7, 9.005],
        ["Canada", 3.9, 6.253],
        ["Denmark", 8.2, 24.915],
        ["Finland", 6.8, 7.371],
        ["France", 6.6, 9.177],
        ["Germany", 11.6, 12.572],
        ["Greece", 2.5, 1.797],
        ["Italy", 4.1, 3.279],
        ["Ireland", 8.8, 12.967],
        ["Netherlands", 4.5, 11.337],
        ["Norway", 9.2, 21.614],
        ["Poland", 2.7, 3.140],
        ["Portugal", 2.7, 1.885],
        ["Spain", 3.2, 1.705],
        ["Sweden", 6.2, 30.300],
        ["Switzerland", 11.9, 30.949],
        ["United Kingdom", 9.8, 19.165],
        ["United States", 5.3, 10.811]]

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

def correlation(xval, yval):
    return covariance(xval, yval) / (deviation(xval) * deviation(yval))

makeGPanel(-2, 17, -5, 55)
drawGrid(0, 15, 0, 50, "lightgray")

xval = []
```

```
yval = []

for country in data:
    d = country[1]
    v = country[2]
    xval.append(d)
    yval.append(v)
    move(d, v)
    fillCircle(0.2)
    text(country[0])

title("Chocolate-Brainpower-Correlation: " + dec2(correlation(xval, yval)))
```

MEMO

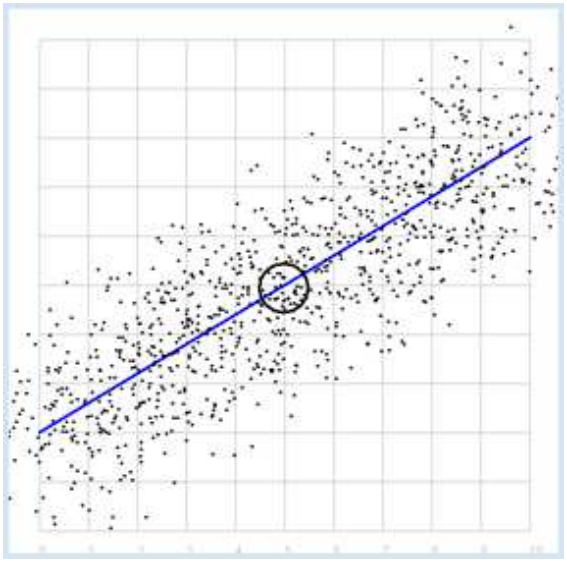
Es ergibt sich eine hohe Korrelation von ungefähr 0.71, die man auf verschiedene Arten interpretieren kann. Richtig ist es zu sagen, dass es einen Zusammenhang zwischen den beiden Datenreihen gibt, allerdings lässt sich über den Grund nur spekulieren. Insbesondere ist damit ein kausaler Zusammenhang zwischen Schololadenkonsum und Intelligenz keineswegs nachgewiesen. Grundsätzlich gilt: Bei einer hohen Korrelation zwischen x und y kann die Grösse x die Ursache für das Verhalten von y sein. Ebenso kann die Grösse y die Ursache für das Verhalten von x sein, oder x und y mit anderen gemeinsamen, vielleicht sogar unbekannten Ursachen im Zusammenhang stehen.

Diskutiere Sinn und Zweck dieser Untersuchung auch mit Personen aus deinem Umfeld und frage sie nach ihrer Meinung.

MESSFEHLER UND RAUSCHEN

Selbst bei einem exakten Zusammenhang zwischen *x* und *y* können Messfehler oder andere Einflüsse zu schwankenden Messwerten führen. In dieser Simulation gehst du von einem linearen Zusammenhang zwischen *x* und *y* aus, wobei aber die Funktionswerte *y* normalverteilten Schwankungen unterworfen werden.

Du bestimmst den Korrelationskoeffizienten und zeichnest den Punkt mit den Koordinaten (*xm*, *ym*) ein, wo *xm* bzw. *ym* die Erwartungswerte von *x* und *y* sind.



```
# Si6h.py
# Gerade und Rauschen

import random
from gpanel import *
import math

z = 1000
a = 0.6
b = 2
sigma = 1

def f(x):
```

```
    y = a * x + b
    return y

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

def correlation(xval, yval):
    return covariance(xval, yval) / (deviation(xval) * deviation(yval))

makeGPanel(-1, 11, -1, 11)
title("y = 0.6 * x + 2 Normalverteilte Messfehler")
addStatusBar(30)
drawGrid(0, 10, 0, 10, "gray")
setColor("blue")
lineWidth(3)
line(0, f(0), 10, f(10))

xval = [0] * z
yval = [0] * z

setColor("black")
for i in range(z):
    x = i / 100
    xval[i] = x
    yval[i] = f(x) + random.gauss(0, sigma)
    move(xval[i], yval[i])
    fillCircle(0.03)

xm = mean(xval)
ym = mean(yval)
move(xm, ym)
circle(0.5)

setStatusText("E(x) = " + dec2(xm) + \
          ",  E(y) = " + dec2(ym) + \
          ",  correlation = " + dec2(correlation(xval, yval)))
```

MEMO

Es ergibt sich erwartungsgemäss eine hohe Korrelation, die umso grösser wird, je kleiner die Streuung *sigma* der Messwerte gewählt wird. Für *sigma = 0* ist die Korrelation exakt 1.
Es zeigt sich auch, dass der Punkt mit den Erwartungswerten *P(0.5, 0.5)* auf der Geraden liegt.

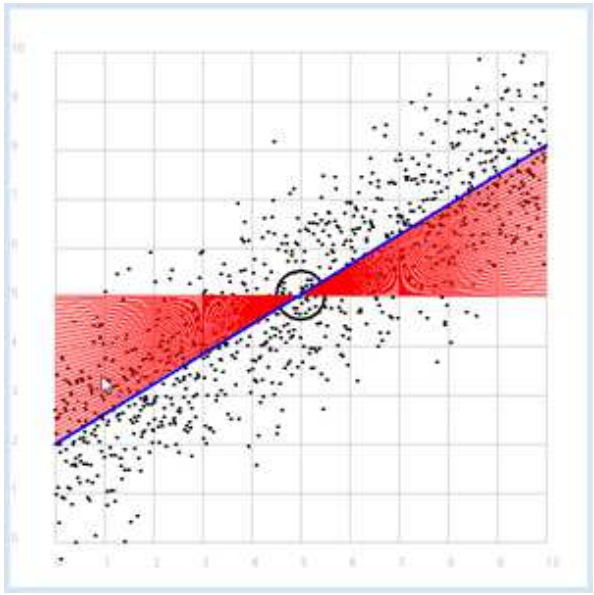
REGRESSIONSGERADE, BESTER FIT

Vorher bist du von einer Geraden ausgegangen, die du "verrauscht" hast. Hier stellst du dir die umgekehrte Frage: Wie findest du die Gerade wieder heraus, wenn du nur die beiden Messreihen hast? Die gesuchte Gerade nennst du **Regressionsgerade**.

Immerhin weisst du schon, dass die Regressionsgerade durch den Punkt P mit den Erwartungswerten geht. Um sie zu finden, gehst du wie folgt vor:

Du legst eine beliebige Gerade durch P und bestimmst für alle x die Quadrate der Abweichungen der Messwerte y von der Geraden (vertikaler Abstand). Für die Regressionsgerade müssen diese Abweichungen möglichst klein sein. Dazu bestimmst du eine **Fehlersumme**, indem du die einzelnen Abweichungen addierst.

Du findest in der Simulation die beste Gerade, indem du deine gelegte Gerade schrittweise um den Punkt P drehst und immer wieder die Fehlersumme bildest. Diese wird absinken. Kurz bevor die Fehlersumme wieder ansteigt, hast du **den besten Fit** gefunden.



```
import random
from gpanel import *
import math

z = 1000
a = 0.6
b = 2

def f(x):
    y = a * x + b
    return y

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
```

```
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

sigma = 1

makeGPanel(-1, 11, -1, 11)
title("Simulierte Datenpunkte. Drücke eine Taste...")
addStatusBar(30)
drawGrid(0, 10, 0, 10, "gray")
setStatusText("Press any key")

xval = [0] * z
yval = [0] * z

for i in range(z):
    x = i / 100
    xval[i] = x
    yval[i] = f(x) + random.gauss(0, sigma)
    move(xval[i], yval[i])
    fillCircle(0.03)

getKeyWait()
xm = mean(xval)
ym = mean(yval)
move(xm, ym)
lineWidth(3)
circle(0.5)

def g(x):
    y = m * (x - xm) + ym
    return y

def errorSum():
    sum = 0
    for i in range(z):
        x = i / 100
        sum += (yval[i] - g(x)) * (yval[i] - g(x))
    return sum

m = 0
setColor("red")
lineWidth(1)
error_min = 0
while m < 5:
    line(0, g(0), 10, g(10))
    if m == 0:
        error_min = errorSum()
    else:
        if errorSum() < error_min:
            error_min = errorSum()
        else:
            break
    m += 0.01

title("Regressionsgerade gefunden")
setColor("blue")
lineWidth(3)
line(0, g(0), 10, g(10))
setStatusText("Gefundene Steigung: " +  dec2(m) + \
              ", Theorie: " + dec2(covariance(xval, yval)
             /(deviation(xval) * deviation(xval))))
```

MEMO

Statt den besten Fit mit einer Computersimulation zu finden, kannst du ihre Steigung auch mit

$$m = \frac{Kovarianz(x, y)}{Streuung(x)^2}$$

berechnen. Da die Regressionsgerade durch den Punkt P mit den Erwartungswerten E(x) und E(y) geht, besitzt sie also die Geradengleichung:

$$y - E(y) = m * (x - E(x))$$

AUFGABEN

1. Das bekannte Engelsche Gesetz aus der Wirtschaftswissenschaft besagt, dass in Haushalten mit steigendem Einkommen im Durchschnitt auch die Ausgaben für Nahrungsmittel zwar absolut wachsen, jedoch relativ abnehmen. Zeige mit dem Datenmaterial dessen Richtigkeit.

   a. Visualisiere den Zusammenhang zwischen Einkommen und absoluten Nahrungsmittelausgaben

   b. Visualisiere den Zusammenhang zwischen Einkommen und relativen Nahrungsmittelausgaben

   c. Bestimme die Korrelation zwischen Einkommen und absoluten Nahrungsmittelausgaben

   d. Bestimmte die Regressionsgerade zwischen Einkommen und absoluten Nahrungsmittelausgaben

   Daten:

Monatseinkommen	4000	4100	4200	4300	4400	4500	4600	4700	4800	4900
Ausgaben %	64	63.25	62.55	61.90	61.30	60.75	60.25	59.79	59.37	58.99

5000	5100	5200	5300	5400	5500	5600	5700	5800	5900	6000
58.65	58.35	58.08	57.84	57.63	57.45	57.30	57.17	57.06	56.97	56.90

2. Die heute allgemein akzeptierte Entwicklungsgeschichte des Universums geht davon aus, dass es vor langer Zeit einen Urknall (Big Bang) gegeben hat, und sich seitdem das Universum ausdehnt. Im Vordergrund steht die Frage, wie lange der Urknall zurückliegt, was man als **Alter des Universums** bezeichnet. Der Astronom Hubble hat 1929 seine weltberühmten Untersuchungen publiziert, in denen er feststellte, dass es einen linearen Zusammenhang zwischen der Distanz d der Galaxien und ihrer Fluchtgeschwindigkeit v gibt. Das Hubble-Gesetz lautet:

   v = H * d

   wo H die Hubble-Konstante genannt wird.

   Du kannst die astrophysikalischen Überlegungen nachvollziehen, indem du von folgenden experimentellen Daten ausgehst, die vom Hubble-Weltraumteleskop stammen:

Galaxie	Distanz [Mpc)	Geschwindig-keit [km/s]
NGC0300	2	133
NGC095	9.16	664
NGC1326A	16.14	1794
NGC1365	17.95	1594
NGC1425	21.88	1473
NGC2403	3.22	278

NGC2541	11.22	714
NGC2090	11.75	882
NGC3031	3.63	80
NGC3198	13.8	772
NGC3351	10	642
NGC3368	10.52	768
NGC3621	6.64	609
NGC4321	15.21	1433
NGC4414	17.7	619
NGC4496A	14.86	1424
NGC4548	16.22	1384
NGC4535	15.78	1444
NGC4536	14.93	1423
NGC4639	21.98	1403
NGC4725	12.36	1103
IC4182	4.49	318
NGC5253	3.15	232
NGC7331	14.72	999

1 Megaparsec (Mpc) = 3.09 * 10^{19}km

a) Zeichne die Daten in einem Scatterplot auf. Kopiere sie dazu die Datenliste in dein Programm.

```
data = [
["NGC0300", 2.00, 133],
["NGC095", 9.16, 664],
["NGC1326A", 16.14, 1794],
["NGC1365", 17.95, 1594],
["NGC1425", 21.88, 1473],
["NGC2403", 3.22, 278],
["NGC2541", 11.22, 714],
["NGC2090", 11.75, 882],
["NGC3031", 3.63, 80],
["NGC3198", 13.80, 772],
["NGC3351", 10.0, 642],
["NGC3368", 10.52, 768],
["NGC3621", 6.64, 609],
["NGC4321", 15.21, 1433],
["NGC4414", 17.70, 619],
["NGC4496A", 14.86, 1424],
["NGC4548", 16.22, 1384],
["NGC4535", 15.78, 1444],
["NGC4536", 14.93, 1423],
["NGC4639", 21.98, 1403],
["NGC4725", 12.36, 1103],
["IC4182", 4.49, 318],
["NGC5253", 3.15, 232],
["NGC7331", 14.72, 999]]

# from Freedman et al, The Astrophysical Journal, 553 (2001)
```

b) Zeige, dass die Werte gut korrelieren und bestimme die Steigung H der Regressionsgeraden

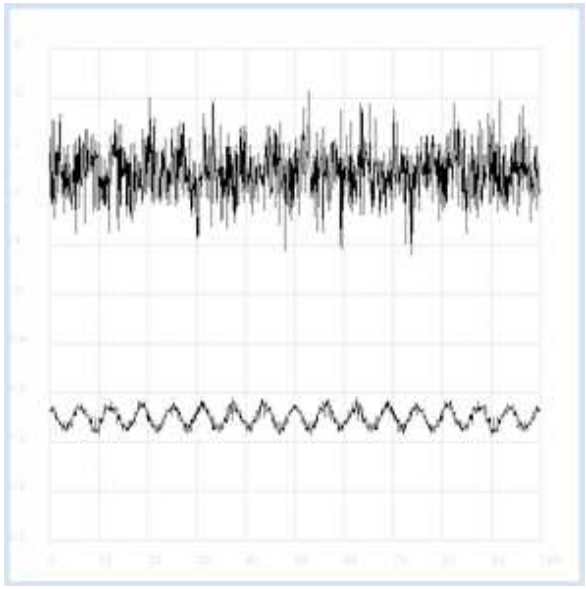
c) Gehst du davon aus, dass die Geschwindigkeit v einer bestimmten Galaxie konstant geblieben ist, so ist ihre Distanz d = v * T, wo T das Alter des Universums ist. Mit dem Hubble-Gesetz v = H * d folgt daraus T = 1 / H. Bestimme T.

ZUSATZSTOFF

MIT AUTOKORRELATION VERSTECKTE INFORMATION FINDEN

Intelligente Wesen auf einem weit entfernten Planeten möchten Kontakt mit anderen Lebewesen aufnehmen. Sie senden dazu Radiosignale aus, die eine bestimmte Regelmässigkeit aufweisen (du kannst dir darunter eine Art Morsesignale vorstellen). Auf dem langen Übertragungsweg wird das Signal immer schwächer und immer mehr von statistisch schwankendem Rauschen überdeckt. Empfangen wir dieses Signal auf der Erde mit einem Radioteleskop, so hören wir vorerst nur das Rauschen. Die Statistik und ein Computerprogramm kann uns aber helfen, das ursprünglich Radiosignal wieder herauszuholen. Wenn du nämlich den Korrelationskoeffizienten des Signals mit seinem eigenen zeitverschobenen Signal berechnest, so verringern sich die statistischen Rauschanteile. (Eine Korrelation mit sich selbst nennt man **Autokorrelation**).

Du kannst diese für die Signalanalyse bedeutsame Eigenschaft mit einem Python-Programm simulieren. Das ursprüngliche Nutzsignal ist ein Sinuston und du überlagerst ihm das Rauschen, indem du jedem Abtastwert eine Zufallszahl (mit einer Normalverteilung) addierst. Das so verrauschte Signal zeigst du im oberen Teil der Grafik und wartest auf einen Tastendruck. Das Nutzsignal ist nicht mehr erkennbar.



Nachfolgend bildest du die Autokorrelation des Signals und zeichnest den Verlauf des Korrelationskoeffizienten im unteren Teil der Grafik auf. Das Nutzsignal ist nun wieder klar erkennbar.

```
import random
from gpanel import *
import math

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
```

```
    return sx

def correlation(xval, yval):
    return covariance(xval, yval)/(deviation(xval)*deviation(yval))
def shift(offset):
    signal1 = [0] * 1000
    for i in range(1000):
       signal1[i] = signal[(i + offset) % 1000]
    return signal1

makeGPanel(-10, 110, -2.4, 2.4)
title("Verrauschtes Signal. Drücke Taste...")
drawGrid(0, 100, -2, 2.0, "lightgray")

t = 0
dt = 0.1
signal = [0] * 1000
while t < 100:
    y = 0.1 * math.sin(t) # Pure signal
#    noise = 0
    noise = random.gauss(0, 0.2)
    z = y + noise
    if t == 0:
        move(t, z + 1)
    else:
        draw(t, z + 1)
    signal[int(10 * t)] = z
    t += dt

getKeyWait()
title("Signal nach Autokorrelation")
for di in range(1, 1000):
    y = correlation(signal, shift(di))
    if di == 1:
        move(di / 10, y - 1)
    else:
        draw(di / 10, y - 1)
```

Noch erlebnisreicher ist es, zuerst das verrauschte Signal und dann das daraus extrahierte Nutzsignal anzuhören. Dazu verwendest du dein Wissen aus dem Kapitel **Sound**.

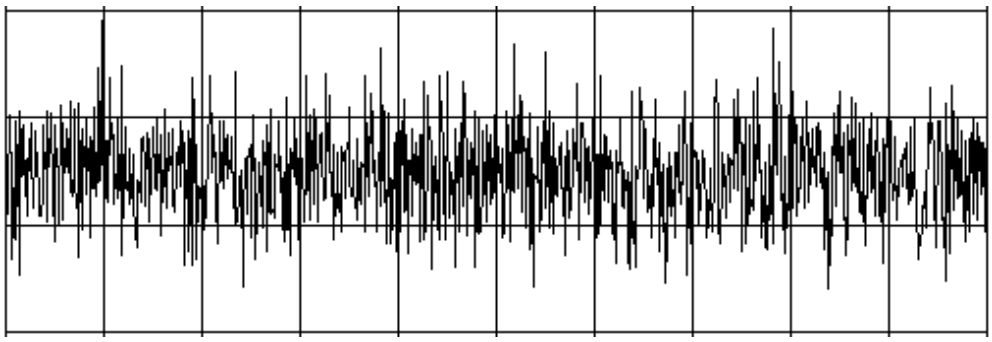
```
from soundsystem import *
import math
import random
from gpanel import *

n = 5000

def mean(xval):
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, k):
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (xval[(i + k) % n] - xm)
    return cxy / n

def deviation(xval):
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx
```

```
makeGPanel(-100, 1100, -11000, 11000)
drawGrid(0, 1000, -10000, 10000)
title("Press <SPACE> to repeat. Any other key to continue.")

signal = []
for i in range(5000):
    value = int(200 *  (math.sin(6.28 / 20 * i) + random.gauss(0, 4)))
    signal.append(value)
    if i == 0:
        move(i, value + 5000)
    elif i <= 1000:
        draw(i, value + 5000)

ch = 32
while ch == 32:
    openMonoPlayer(signal, 5000)
    play()
    ch = getKeyCodeWait()

title("Autocorrelation running. Please wait...")
signal1 = []
xm = mean(signal)
sigma = deviation(signal)
q = 20000 / (sigma * sigma)
for di in range(1, 5000):
    value = int(q *  covariance(signal, di))
    signal1.append(value)
title("Autocorrelation Done. Press any key to repeat.")
for i in range(1, 1000):
    if i == 1:
        move(i, signal1[i] - 5000)
    else:
        draw(i, signal1[i] - 5000)

while True:
    openMonoPlayer(signal1, 5000)
    play()
    getKeyCodeWait()
```

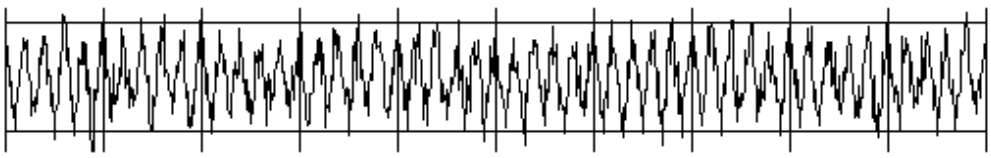
MEMO

Statt das Programm auszuführen, kannst du die beiden Signale auch als WAV-Datei anhören:

Rauschsignal **(hier klicken)**



Nutzsignal **(hier klicken)**



8.7 KOMPLEXE ZAHLEN & FRAKTALE

EINFÜHRUNG

Komplexe Zahlen sind in der Mathematik von grosser Wichtigkeit, da sie die Menge der reellen Zahlen so erweitern, dass viele Sätze einfacher und allgemeiner formuliert werden können. Sie spielen darüber hinaus in Naturwissenschaften und Technik eine wichtige Rolle, insbesondere in der Physik und Elektrotechnik [**mehr...**]. Glücklicherweise sind komplexe Zahlen in Python ein fest eingebauter Datentyp und es stehen die arithmetischen Operatoren für Addition, Subtraktion, Multiplikation und Division zur Verfügung. Zudem gibt es im Modul *cmath* viele bekannte Funktionen mit komplexen Argumenten.

Komplexe Zahlen können in der Gaussschen Zahlenebene als Zeiger oder als Punkte dargestellt werden. Damit sich auch ein Turtlefenster, ein GPanel und eine Pixelgitter von JGameGrid als Gausssche Zahlenebene auffassen lassen, stehen in den entsprechenden Bibliotheken alle Funktionen mit Koordinatenparametern (x, y) auch direkt für komplexe Zahlen zur Verfügung.

PROGRAMMIERKONZEPTE: *Datentyp complex, Konforme Abbildung, Mandelbrot-Fraktal*

GRUNDOPERATIONEN MIT KOMPLEXEN ZAHLEN

In Python verwendest du für die imaginäre Einheit an Stelle von i das in der Elektrotechnik übliche Symbol j. Du kannst die komplexe Zahl mit Realteil 2 und Imaginärteil 3 auf mehrere Arten definieren:

```
z = 2 + 3j        oder    z = 2 + 3 * 1j        oder    z  = complex(2, 3)
```

Verwende für die folgenden Beispiele die praktische TigerJython-Konsole.

Den Real- und Imaginärteil erhältst du mit *z.real* und *z.imag*. Beachte, dass es sich nicht um einen Funktionsaufruf handelt, sondern um einen Variablenwert (den du nur lesen kannst). Real- und Imaginärteil sind immer Floats.

```
>>> z = 2 + 3j
>>> z
    (2+3j)
>>> z.real
    2.0
>>> z.imag
    3.0
```

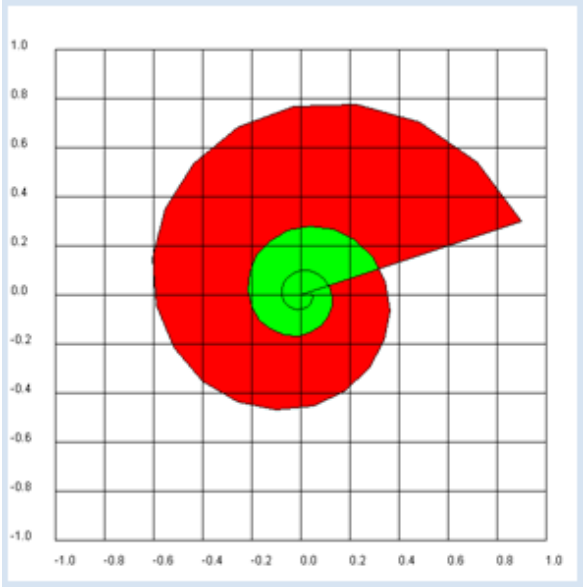
Das Quadrat der imaginären Einheit 1j ist -1. Anders gesagt, ist die imaginäre Einheit 1j gleich der Quadratwurzel aus -1. Um die Quadratwurzel aus komplexen Zahlen zu ziehen, musst du an Stelle von *math* das Modul *cmath* importieren.

```
>>> z = 1j
>>> z * z
    (-1+0j)
>>> import cmath
>>> cmath.sqrt(-1)
    1j
```

Die Standardfunktion *abs()* liefert nicht nur den Betrag für Integer und Floats, sondern auch für komplexe Zahlen Du kannst für komplexe Zahlen die üblichen Operationszeichen +, -, *, / und den Potenzoperator ** verwenden. Es gelten dieselben Rangordnungen wie für Floats.

```
>>> z = 3 + 4j
>>> abs(z)
    5.0
>>> 2 * (z + 1) - z
    (5+4j)
>>> z**2 / z
    (3+4j)
```

In deinem Programm bildest du Potenzen einer komplexen Zahl z = 0.9 + 0.3j. Diese hat einen Betrag von etwas kleiner als 1. Da bei der Multiplikation von zwei komplexen Zahlen die Beträge multipliziert und die Phasen addiert werden, bewegen sich die Potenzen offenbar auf einer Spirale, die du in einem *GPanel* sehr einfach einzeichnen kannst. Beachte, dass du das Füllen vor dem Zeichnen des Gitter machen musst, da mit *fill()* immer geschlossene Gebiete gefüllt werden.




```
from gpanel import *
makeGPanel(-1.2, 1.2, -1.2, 1.2)
title("Gausssche Zahlenebene")

z = 0.9 + 0.3j
for n in range(1, 60):
    y = z**n
    draw(y)
fill(0.2, 0, "white", "red")
fill(0.0, 0.2, "white", "green")
drawGrid(-1.0, 1.0, -1.0, 1.0)
```

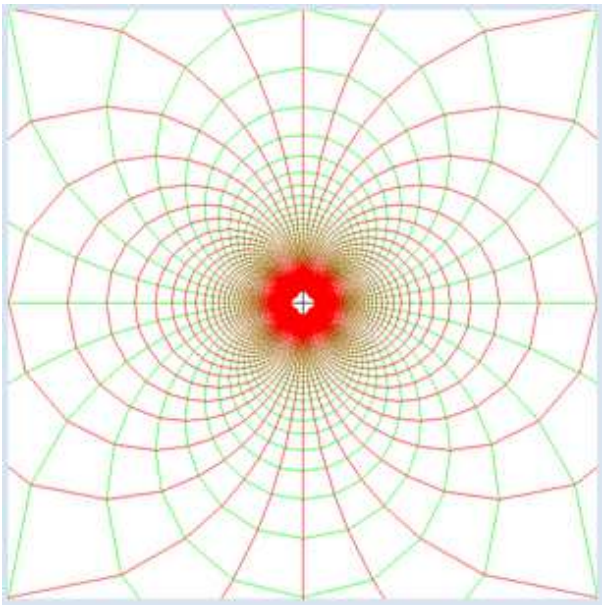
MEMO

draw(z) bewirkt das gleiche wie *draw(z.real, z.imag)*, ist aber einfacher. Du verwendest GPanel-Koordinaten, die auf allen Seiten 10% grösser als der verwendete Koordinatenbereich ist. In *drawGrid()* musst du die Koordinaten als Float angeben, damit das Gitter mit Floats angeschrieben wird.

KONFORME ABBILDUNGEN

Bei zweidimensionalen Abbildungen wird jedem Punkt *P(x, y)* ein Bildpunkt *P'(x', y')* zugeordnet. Du kannst die Punkte auch als komplexe Zahlen auffassen und sagen, dass bei der Abbildung jeder Zahl z ein Funktionswert z' zugeordnet wird und dafür z' = f(z) schreiben.

Du wählst im Folgenden die Funktion *z' = f(z) = 1/z* (Inversion) und bildest ein rechtwinkliges Koordinatengitter ab. Dazu wählst du in der Gaussschen Zahlenebene den Bereich von -5 bis 5 und denkst dir 201 Gitterlinien im Abstand von 1 / 20. Die Bilder der horizontalen Linien zeichnest du grün, die der vertikalen Linien rot ein. Es entsteht ein hübsches Bild.



```
from gpanel import *

# function f(z) = 1/z
def f(z):
    if z == 0:
        return 0
    return 1 / z

min = -5.0
max = 5.0
step = 1 / 20
reStep = complex(step, 0)
imStep = complex(0, step)

makeGPanel(min, max, min, max)
title("Konforme Abbildung für f(z) = 1 / z")
line(min, 0, max, 0)  # Real axis
line(0, min, 0, max) # Imaginary axis

# Transform horizontal line per line
setColor("green")
z = complex(min, min)
while z.imag < max:
    z = complex(min, z.imag) # left
    move(f(z))
    while z.real < max: # move along horz. line
        draw(f(z))
        z = z + reStep
    z = z + imStep

# Transform vertical line per line
setColor("red")
z = complex(min, min)
while z.real < max:
    z = complex(z.real, min) # bottom
    move(f(z))
    while z.imag < max:  # move along vert. line
        draw(f(z))
        z = z + imStep
    z = z + reStep
```

MEMO

Wie du aus der Grafik entnimmst, schneiden sich die Bilder der Gitterlinien wieder rechtwinklig. Eine Abbildung, bei der der Winkel zwischen sich schneidenden Linien erhalten bleibt, nennt man **winkeltreu** und man spricht von einer **konformen Abbildung**.

MANDELBROT-FRAKTALE

Fraktale Bilder sind sehr typisch und vielen Menschen bekannt. Wahrscheinlich ist dir das *Apfelmännchen* auch schon begegnet, das du hier selbst programmieren wirst. Der Algorithmus für viele Fraktale beruht auf komplexen Zahlen und es ist schon deswegen lohnend, sich mit ihnen zu befassen. Als Vater der **Fraktalen Geometrie** gilt der Mathematiker Benoit Mandelbrot (1924-2010).

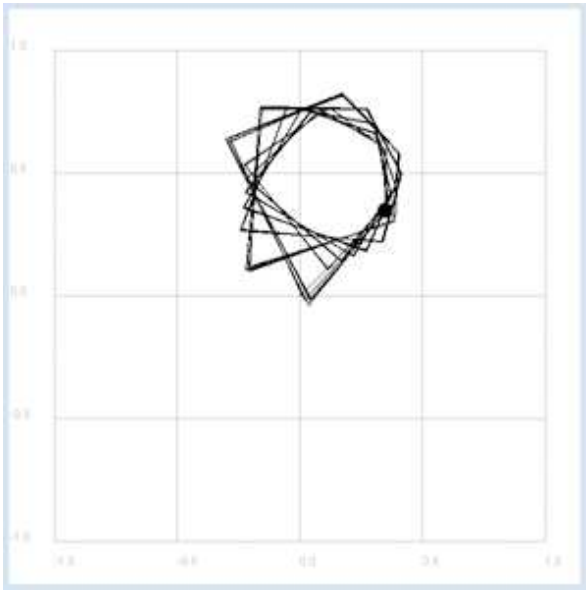
Mandelbrot an der Einführungsvorlesung der Légion d'honneur (2006) (© Wiki)

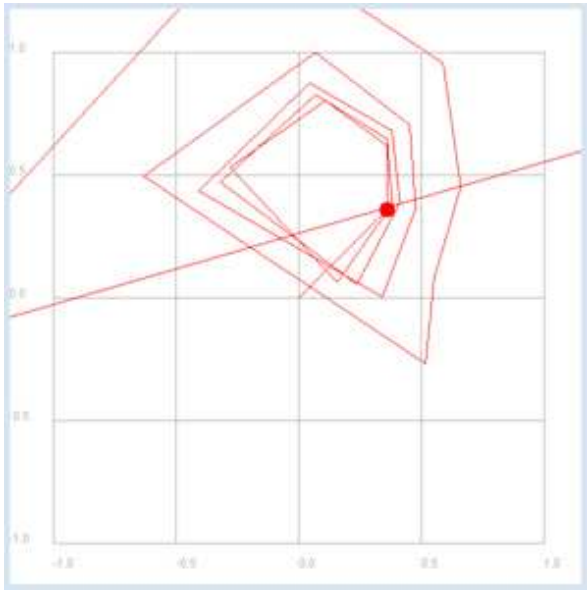
Um Mandelbrot-Fraktale zu erstellen, betrachtest du zu der vorgegebenen komplexen Zahl c eine (rekursiv definierte) Folge von komplexen Zahlen nach dem Bildungsgesetz:

$z' = z^2 + c$ mit dem Anfangswert $z_0 = 0$

Falls der Betrag der Folgenglieder in einem beschränkten Bereich bleibt, also nicht über alle Grenzen wächst, gehört c zur **Mandelbrot-Menge**.

Um dich mit dem Algorithmus vertraut zu machen, zeichnest du die Folgenglieder für zwei komplexe Zahlen $c_1 = 0.35 + 0.35j$ und $c_2 = 0.36 + 0.36j$. Dabei siehst du sofort, dass c_1 zur Mandelbrot-Menge gehört, c_2 hingegen nicht.

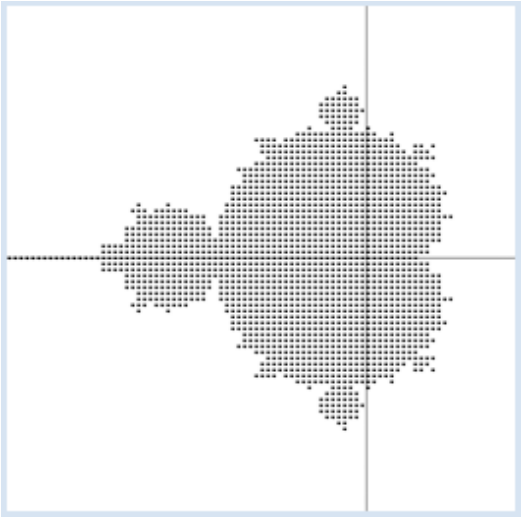




```
from gpanel import *

def f(z):
    return z * z + c

makeGPanel(-1.2, 1.2, -1.2, 1.2)
title("Mandelbrot Iteration")

drawGrid(-1, 1.0, -1, 1.0, 4, 4, "gray")

isMandelbrot = askYesNo("c in Mandelbrot-Menge?")
if isMandelbrot:
    c = 0.35 + 0.35j
    setColor("black")
else:
    c = 0.36+ 0.36j
    setColor("red")

title("Mandelbrot Iteration mit c = " + str(c))
move(c)
fillCircle(0.03)

z = 0j
while True:
    if z == 0:
        move(z)
    else:
        draw(z)
    z = f(z)
    delay(100)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Um herauszufinden, welche Zahlen eines bestimmten Bereichs der Gaussschen Zahlenebene zur Mandelbrot-Menge gehören, führst du für komplexe Zahlen c in einem bestimmten Gitterbereich die Iteration durch. Dabei nimmst du stark vereinfachend an, dass c nicht zur Mandelbrot-Menge gehört, wenn der Betrag eines Folgenglieds in den ersten 50 Iterationen grösser als R = 2 wird. Bleibt der Betrag von z bis am Ende der 50 Iterationen kleiner als 2, so so nimmst du an, dass c zur Mandelbrot-Menge gehört und zeichnest dort einen schwarzen Punkt.

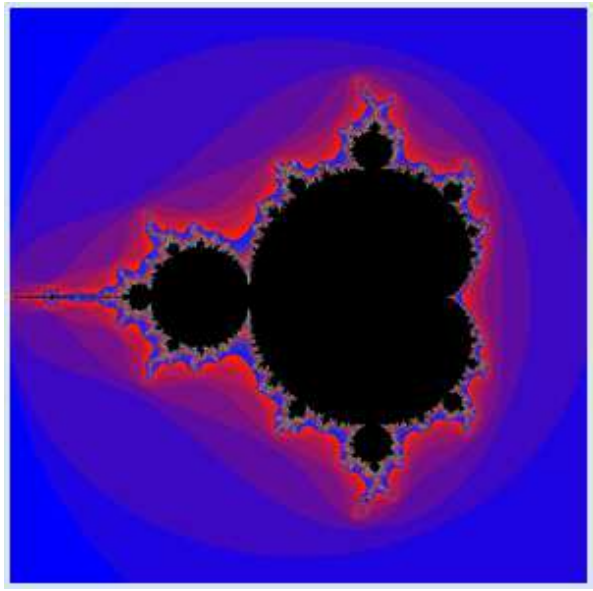


```
from gpanel import *

def isInSet(c):
    z = 0
    for n in range(maxIterations):
        z = z*z + c
        if abs(z) > R: # diverging
            return False
    return True

maxIterations = 50
R = 2
xmin = -2
xmax = 1
xstep = 0.03
ymin = -1.5
ymax = 1.5
ystep = 0.03

makeGPanel(xmin, xmax, ymin, ymax)
line(xmin, 0, xmax, 0)  # Real axis
line(0, ymin, 0, ymax) # Imaginary axis
title("Mandelbrot-Menge")
y = ymin
while y <= ymax:
    x = xmin
    while x <= xmax:
        c = x + y*1j
        inSet = isInSet(c)
        if inSet:
            move(c)
            fillCircle(0.01)
        x += xstep
    y += ystep
```

Das Apfelmännchen wird in der Grafik bereits sichtbar. Noch schönere Figuren kannst du zeichnen, wenn du für Zahlen c, die **nicht** zur Mandelbrot-Menge gehörten, einen Farbpunkt zeichnest, dessen Farbe etwas darüber aussagt, wie schnell die Folge divergiert.

Als Mass für die Divergenz wird dabei die Anzahl *itCount* der Iterationen genommen, bei der der Betrag von z das erste Mal grösser als *R = 2* wird.

Für die Zuordnung von *itCount* zu einer Farbe kannst du in *getIterationColor()* irgend eine Auswahl treffen, die nach deinem ästhetischen Geschmack ein besonders schönes Fraktal erzeugt.



```
from gpanel import *

def getIterationColor(it):
    color = makeColor((30 * it) % 256,
                      (4 * it) % 256,
                      (255 - (30 * it)) % 256)
    return color

def mandelbrot(c):
    z = 0
    for it in range(maxIterations):
        z = z*z + c
        if abs(z) > R: # diverging
            return it
    return maxIterations

maxIterations = 50
R = 2
xmin = -2
xmax = 1
xstep = 0.003
ymin = -1.5
ymax = 1.5
ystep = 0.003

makeGPanel(xmin, xmax, ymin, ymax)
title("Apfelmännchen")
enableRepaint(False)
y = ymin
while y <= ymax:
    x = xmin
    while x <= xmax:
        c = x + y*1j
        itCount = mandelbrot(c)
        if itCount == maxIterations: # inside Mandelbrot set
            setColor("black")
        else: # outside Mandelbrot set
           setColor(getIterationColor(itCount))
        point(c)
        x += xstep
    y += ystep
    repaint()
```

MEMO

Um das Zeichnen etwas zu beschleunigen, wird *enableRepaint(False)* gesetzt und nur am Ende jeder Zeile mit *repaint()* neu gerendert. Das Mandelbrot-Fraktal besitzt die bemerkenswerte Eigenschaft, dass bei einer Vergrösserung eines Ausschnitts wieder eine ähnliche Struktur erscheint.

AUFGABEN

1. Ein hübsches Fraktal kannst du erhalten, wenn du für ein Gitter in der Gaussschen Zahlenebene im Bereich -20 bis 20 nur diejenigen Gitterpunkte z einzeichnest, für welche das gerundete Betragsquadrat eine gerade Zahl ist, also *int(abs(z) * abs(z)) % 2 == 0* gilt. Wähle eine Schrittweite von 0.1.

2. Untersuche die Abbildungen der Gaussschen Zahlenebene

   a) $z' = f(z) = z^2$

   b) $z' = f(z) = a * z$ mit komplexem $a = 2 + 1j$

   c) $z' = f(z) = e^z$

   d) $z' = f(z) = \frac{1-z}{1+z}$ (Möbius-Transformation)

   Bilde das rechtwinklige Koordinatengitter im Bereich von -5 bis 5 mit einer Schrittweite von 1/10 ab. Beschreibe die Abbildung in Worten und mutmasse, ob sie konform ist.

3. Zeichne einige Mandelbrot-Fraktal mit anderer Farbzuordnung, beispielsweise:

Anzahl Iterationen	**Farbe**
< 3	dunkelgrau
< 5	grün
< 8	rot
< 10	blau
< 100	gelb
sonst	schwarz

ZUSATZSTOFF

WECHSELSTROM UND IMPEDANZ

Elektrische Schaltungen für sinusförmige Wechselspannungen und Wechselströmen, die aus passiven Bauelementen (Widerständen, Kondensatoren, Spulen) aufgebaut sind, können wie Gleichstromschaltungen behandelt werden, wenn du für Spannungen, Ströme und Widerstände komplexe Grössen verwendest. Ein allgemeiner komplexer Widerstand heisst auch **Impedanz** und wird oft mit Z und für rein imaginäre Widerstände mit X bezeichnet. Die Impedanz eines ohmschen Widerstands ist R, einer Spule X_L = jωL (L: Induktivität) und eines Kondensators X_C = 1 / jωC (C: Kapazität), wobei ω = 2πf (f: Frequenz) ist.

Eine komplexe Wechselspannung u = u(t) läuft in der Gaussschen Zahlenebene gleichförmig auf einem Kreis. Wird sie an eine Impedanz Z angelegt, so fliesst der Strom i(t) und nach dem Ohmschen Gesetz gilt u = Z * i. Da bei der Multiplikation von komplexen Zahlen die Phasen

addiert und die Beträgt multipliziert werden, läuft u um die Phase von Z phasenverschoben vor i

phase(u) = phase(Z) + phase(i)

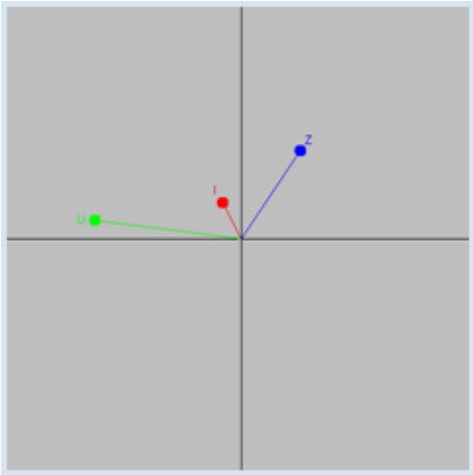
Also läuft i auch auf einem Kreis. Für die Beträge (Amplituden) gilt:

*| u | = | Z | * | i |*

In deinem Programm stellst du diese Beziehungen in der Gaussschen Zahlenebene anschaulich dar, wobei du von den Werten

| u | = 5V und *Z = 2 +3j*

und einer Frequenz von f = 10 Hz ausgehst. Du verwendest für die Animation ein GPanel mit *enableRepaint(False)*, da die Grafik in jedem Zeitschritt vollständig gelöscht, neu aufgebaut und mit *repaint()* gerendert wird.

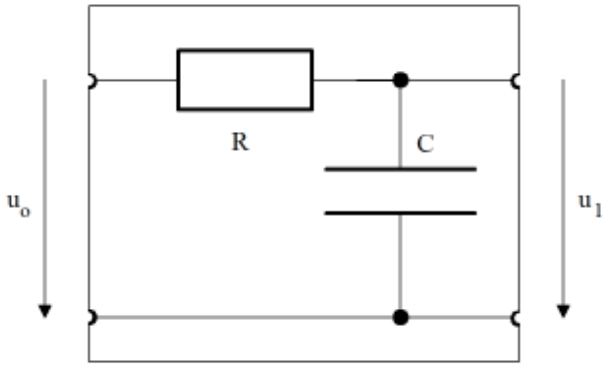



```
from gpanel import *
import math

def drawAxis():
   line(min, 0, max, 0)  # Real axis
   line(0, min, 0, max) # Imaginary axis


def cdraw(z, color, label):
    oldColor = setColor(color)
    line(0j, z)
    fillCircle(0.2)
    z1 = z + 0.5 * z / abs(z) - (0.1 + 0.2j)
    text(z1, label)
    setColor(oldColor)

min = -10
max = 10
dt = 0.001

makeGPanel(min, max, min, max)
enableRepaint(False)
bgColor("gray")
title("Komplexe Spannungen und Ströme")

f = 10 # Frequency
omega = 2 * math.pi * f

t = 0
uA = 5
Z = 2 + 3j

while True:
    u = uA * (math.cos(omega * t) + 1j * math.sin(omega * t))
    i = u / Z
    clear()
    drawAxis()
    cdraw(u, "green", "U")
    cdraw(i, "red", "I")
    cdraw(Z, "blue", "Z")
    repaint()
    t += dt
    delay(100)
```

MEMO

Elektrische Schaltungen mit passiven Bauelementen lassen wie Gleichstromschaltungen behandeln, wenn man Spannung, Strom und Widerstand als komplexe Zahlen auffasst.

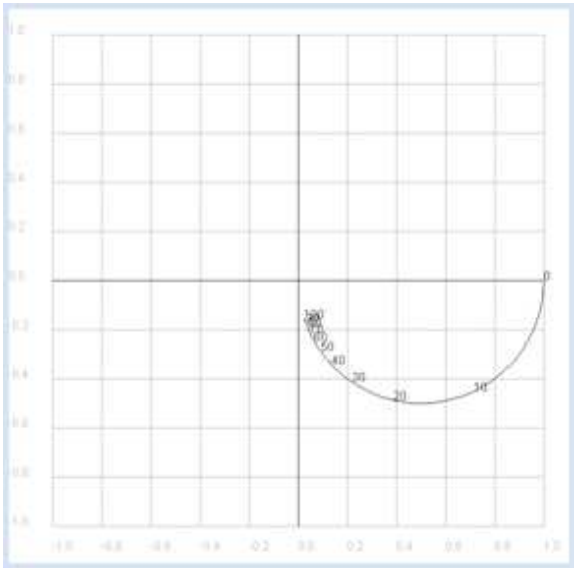
Du wendest diese Erkenntnis auf eine einfache Schaltung an, die lediglich aus einem Widerstand und einem Kondensator besteht. Dabei interessierst du dich für die Ausgangsspannung u_1 in Abhängigkeit von der Frequenz f, wobei du die Eingangsspannung u_o, sowie R und C als gegeben betrachtest.



Die Berechnung ist einfach: Für die Serieschaltung von R und C ergibt sich die Impedanz Z = R + X_C und damit der Strom i = u_o / Z, also wiederum mit dem Ohmschen Gesetz die Ausgangspannung

$$u_1 = X_C * i = \frac{X_C}{R + X_C} * u_o \quad \text{oder} \quad u_1 = v * u_o \quad \text{mit} \quad v = \frac{X_C}{R + X_C}$$

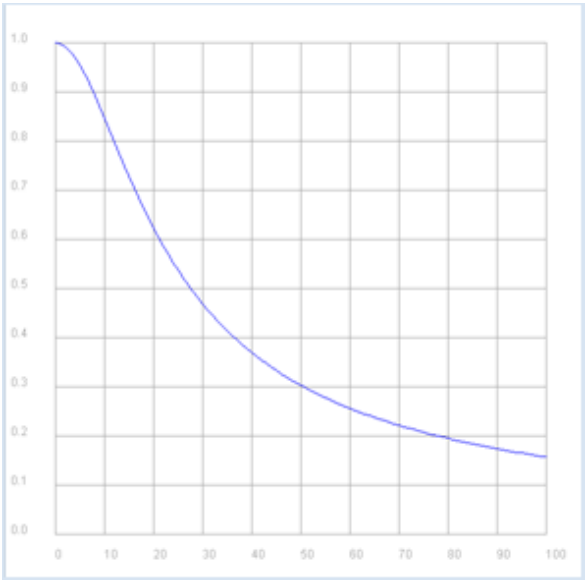
v nennt man den komplexen Verstärkungsfaktor. Du kannst ihn zur Veranschaulichung in der komplexen Ebene für verschiedene Werte von f auftragen und siehst, dass der Betrag vom Wert 1 bei der Frequenz 0 mit zunehmender Frequenz abnimmt. Tiefe Frequenzen werden also gut übertragen, hohe hingegen schlecht. Man nennt diese Schaltung daher einen **Tiefpass**.

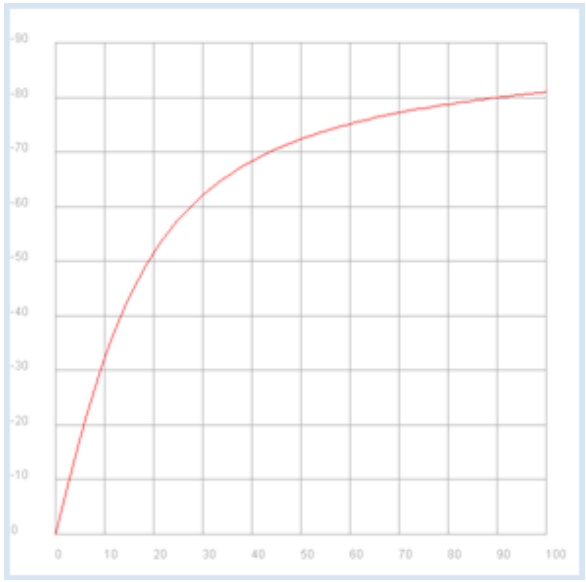


```
from gpanel import *
from math import pi

def drawAxis():
    line(-1, 0, 1, 0)  # Real axis
    line(0, -1, 0, 1) # Imaginary axis

makeGPanel(-1.2, 1.2, -1.2, 1.2)
drawGrid(-1.0, 1.0, -1.0, 1.0, "gray")
setColor("black")
drawAxis()
title("Komplexer Verstärkungsfaktor - Tiefpass")

R = 10
C = 0.001
def v(f):
    if f == 0:
        return 1 + 0j
```

```
    omega = 2 * pi * f
    XC = 1 / (1j * omega * C)
    return XC / (R + XC)

f = 0 # Frequency
while f <= 100:
    if f == 0:
        move(v(f))
    else:
        draw(v(f))
    if f % 10 == 0:
        text(str(f))
    f += 1
    delay(10)
```

Im **Bode-Plot** wird der Verstärkungsfaktor nach Betrag und Phase aufgeteilt und in Funktion der Frequenz aufgetragen (üblich sind allerdings logarithmische Skalen).






```
from gpanel import *
import math
import cmath

R = 10
C = 0.001

def v(f):
    if f == 0:
        return 1 + 0j
    omega = 2 * math.pi * f
    XC = 1 / (1j * omega * C)
    return XC / (R + XC)

makeGPanel(-10, 110, -0.1, 1.1)
drawGrid(0, 100, 0, 1.0, "gray")
title("Bode-Plot - Tiefpass, Betrag (Drücke Taste)")
setColor("blue")
f = 0
while f <= 100:
    if f == 0:
        move(f, abs(v(f)))
    else:
        draw(f, abs(v(f)))
    f += 1

getKeyCodeWait(True)
clear()
```

```
window(-10, 110, 9, -99)
setColor("darkgray")
drawGrid(0, 100, 0, -90, 10, 9)
title("Bode-Plot - Tiefpass, Phase")
setColor("red")
f = 0
while f <= 100:
    if f == 0:
        move(f, math.degrees(cmath.phase(v(f))))
    else:
        draw(f, math.degrees(cmath.phase(v(f))))
    f += 1
```

MEMO

Im Bode-Plot wird nochmals besonders deutlich, dass die vorliegende Schaltung tiefe Frequenzen gut und hohe Frequenzen schlecht überträgt. Zwischen Eingang- und Ausgangssignal besteht zudem eine Phasenverschiebung im Bereich 0 bis -90 Grad. Man sagt, dass die Ausgangsspannung der Eingangsspannung "nachhinkt" bzw. dass die Eingangsspannung der Ausgangsspannung "vorausläuft".

AUFGABEN

1. Die Frequenz

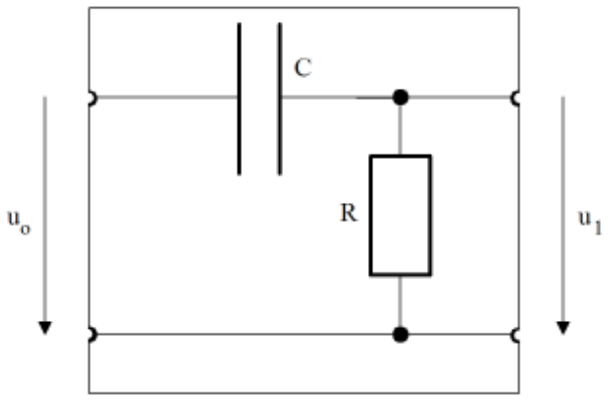
   $$f_c = \frac{1}{2\pi RC}$$

   heisst Grenzfrequenz (cutoff frequency). Zeige für den RC-Tiefpass mit R = 10 Ohm und C = 0.001 F dass bei dieser Frequenz der Betrag des Verstärkungsfaktors
   $1/\sqrt{2}$ ist.

2. Oft gibt man den Betrag des Verstärkungsfaktors in Dezibel (dB) an. Man definiert dB = 20 log |v| (Zehnerlogarithmus). Zeichne für den RC-Tiefpass mit R = 10 Ohm und C = 0.001 F das Bode-Diagramm mit einer dB-Skala bis -100 dB und einer logarithmischen Frequenzskala im Bereich 1Hz..100 kHz.

   Bestätige mit der grafischen Darstellung, dass für höhere Frequenzen der Abfall 20 dB/Frequenzdekade beträgt.

3. Die folgende Schaltung ist ein **Hochpassfilter** (R = 10 Ohm, C = 0.001 F).



   Zeichne wie in Aufgabe 2 den Bode-Plot für den Verstärkungsfaktor und diskutiere das Frequenzverhalten.

```
window(-10, 110, 9, -99)
setColor("darkgray")
drawGrid(0, 100, 0, -90, 10, 9)
title("Bode-Plot - Tiefpass, Phase")
setColor("red")
f = 0
while f <= 100:
    if f == 0:
        move(f, math.degrees(cmath.phase(v(f))))
    else:
        draw(f, math.degrees(cmath.phase(v(f))))
    f += 1
```

MEMO

Im Bode-Plot wird nochmals besonders deutlich, dass die vorliegende Schaltung tiefe Frequenzen gut und hohe Frequenzen schlecht überträgt. Zwischen Eingang- und Ausgangssignal besteht zudem eine Phasenverschiebung im Bereich 0 bis -90 Grad. Man sagt, dass die Ausgangsspannung der Eingangsspannung "nachhinkt" bzw. dass die Eingangsspannung der Ausgangsspannung "vorausläuft".

AUFGABEN

1. Die Frequenz

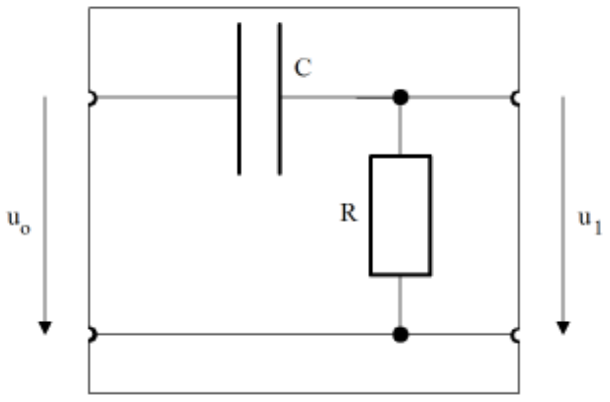
   $$f_c = \frac{1}{2\pi RC}$$

   heisst Grenzfrequenz (cutoff frequency). Zeige für den RC-Tiefpass mit R = 10 Ohm und C = 0.001 F dass bei dieser Frequenz der Betrag des Verstärkungsfaktors
   $1/\sqrt{2}$ ist.

2. Oft gibt man den Betrag des Verstärkungsfaktors in Dezibel (dB) an. Man definiert dB = 20 log |v| (Zehnerlogarithmus). Zeichne für den RC-Tiefpass mit R = 10 Ohm und C = 0.001 F das Bode-Diagramm mit einer dB-Skala bis -100 dB und einer logarithmischen Frequenzskala im Bereich 1Hz..100 kHz.

   Bestätige mit der grafischen Darstellung, dass für höhere Frequenzen der Abfall 20 dB/Frequenzdekade beträgt.

3. Die folgende Schaltung ist ein **Hochpassfilter** (R = 10 Ohm, C = 0.001 F).

   Zeichne wie in Aufgabe 2 den Bode-Plot für den Verstärkungsfaktor und diskutiere das Frequenzverhalten.

8.8 SPEKTRALANALYSE

EINFÜHRUNG

Fällt Licht auf dein Auge oder hörst du einen Ton, so entsteht ein Signal, das man als eine Funktion der Zeit *y(t)* auffassen kann. Für eine einzige Spektralfarbe oder einen reinen Ton handelt es sich um eine sinusförmige Schwingung mit der Amplitude *A* und Frequenz *f*, mathematisch ausgedrückt [**mehr...**]:

$$y(t) = A\sin(\omega * t) \text{ mit } \omega = 2 * \pi * f$$

Ein komplexeres Signal, beispielsweise von einem konstant ausgehaltenen Ton eines Musikinstruments, ist noch immer periodisch, aber nicht mehr sinusförmig. Der berühmte Mathematiker Joseph Fourier (1768-1830) hat aber bewiesen, dass man jede periodische Funktion auch als eine Summe von Sinusschwingungen, als sogenannte **Fourierreihe**, auffassen kann. Er hat damit einen Grundstein von unschätzbarem Wert für den Fortschritt in der modernen Mathematik, Physik und Technik gelegt. Die Aufspaltung eines Signals in seine sinusförmigen Frequenzanteile nennt man **Spektralanalyse**.

PROGRAMMIERKONZEPTE: *Sinusschwingung, Fourierreihe, Fast Fourier Transform (FFT), Spektrum, Sonogramm*

SPEKTRUM EINES KLANGS, OBERTÖNE

Für den typischen Klangcharakter einer Stimme oder eines Musikinstruments sind die sinusförmigen Frequenzanteile wichtig, die darin enthalten sind. Für einen exakt periodischen Klang bestehen diese aus dem Grundton und den Obertönen, deren Frequenz ein ganzzahliges Vielfaches der Frequenz des Grundtons ist. Trägst du die Amplituden der Frequenzanteile in einer Grafik auf, so erhältst du das **Spektrum des Klangs**. Ein Gerät, mit dem du das Spektrum bestimmen kannst, nennt man einen **Spektralanalysator**.

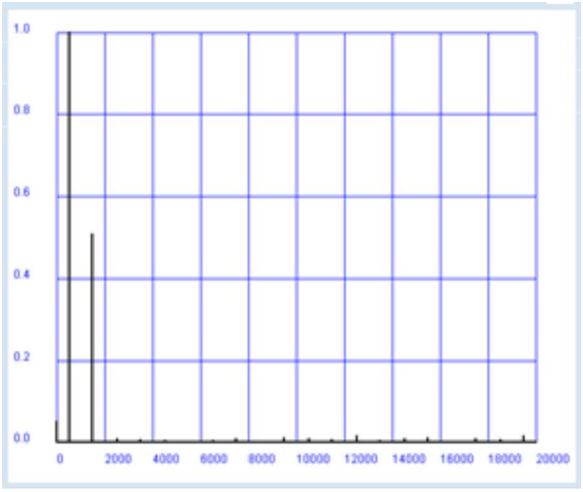
TigerJython kann das Spektrum auf Grund eines Algorithmus bestimmen, der sehr berühmt ist und **Fast Fourier Transform** (FFT) heisst. Um die FFT zu verwenden, musst du allerdings einige zusätzliche JAR-Bibliotheken, die *Piotr Wendykier* geschrieben hat, von **hier** herunterladen und in das Unterverzeichnis Lib des Verzeichnisses, in dem sich *tigerjython2.jar* befindet, kopieren.

Um die FFT durchzuführen, übergibst du der Funktion *fft(samples, n)* eine Liste *samples* mit den zeitlich äquidistanten Abtastwerten und der Angabe n, wieviele davon vom Anfang der Liste an für die FFT zu verwenden sind.

Als Rückgabewerte erhältst du eine Liste mit den Amplituden der *n/2* (normalisierten) Frequenzanteile zurück. Diese liegen im Abstand *r = fs / n,* wobei *fs* die Abtastrate (sampling frequency) ist. *r* nennt man die **Auflösung** des Spektrums.

Diese *n/2* Rückgabewerte im Abstand *r* "bevölkern" das Frequenzgebiet von 0 bis *n/2*r = fs/2*, oder kurz gesagt: **Bei einer Abtastrate von fs liefert die FFT das Spektrum von 0 bis *fs/2***. Für eine Musik-CD mit der typischen Abtastrate *fs* = 44100 Hz entspricht dies einem Spektrum bis 22050 Hz, das den ganzen vom Menschen hörbaren Bereich umfasst.

Um den Spektrumanalysator zu testen, verwendest du vorerst einen Klang *"wav/doublesine.wav"* aus der Distribution von TigerJython, der zwei Sinustöne überlagert. Der Klang wurde mit einer Abtastrate von *fs* = 40'000 Hz aufgenommen. Nimmst du *n* = 10'000 Abtastwerte, so gibt dir die Funktion *fft(samples, n)* 5'000 Frequenzanteile mit der Auflösung *r* = 40'000 / 10'000 = 4 Hz im Bereich 0..20'000 Hz zurück, die du in einem GPanel als Spektrum mit vertikalen Linien grafisch darstellst.



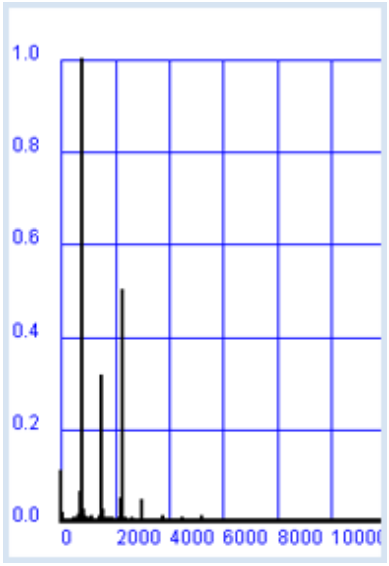

```
from soundsystem import *
from gpanel import *

def showSpectrum(text):
    makeGPanel(-2000, 22000, -0.2, 1.2)
    drawGrid(0, 20000, 0, 1.0, 10, 5, "blue")
    title(text)
    lineWidth(2)
    r = fs / n # Resolution
    f = 0
    for i in range(n // 2):
        line(f, 0, f, a[i])
        f += r

fs = 40000 # Sampling frequency
n = 10000 # Number of samples
samples = getWavMono("wav/doublesine.wav")
openMonoPlayer(samples, fs)
play()
a = fft(samples, n)
showSpectrum("Audio Spectrum")
```

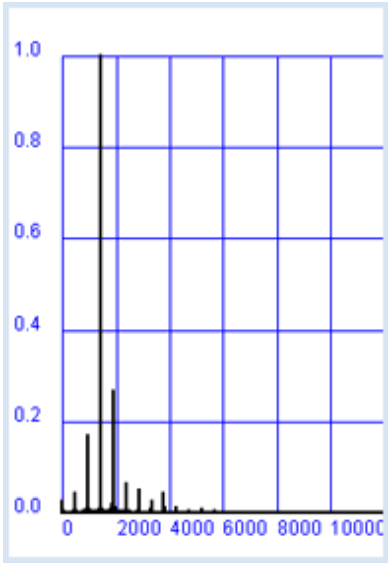
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Wie du vermutest (und hörst), handelt es sich um die Frequenzen 500 Hz und 1.5 kHz mit einem Amplitudenverhältnis von 1 : 1/2. Zusätzlich gibt es noch einige Störungsanteile. Die Frequenz 0 entspricht einem konstanten Signalanteil (offset).

Du hast jetzt mit TigerJython einen feudalen Spektrumanalysator vor dir, mit dem du die Grund- und Obertöne von Musikinstrumenten und menschlichen oder tierischen Lauten untersuchen kannst. In der Distribution findest du bereits den Ton einer Flöte ("wav/flute.wav") und einer Oboe ("wav/oboe.wav"), deren Klangcharakteristik sehr unterschiedlich sind.

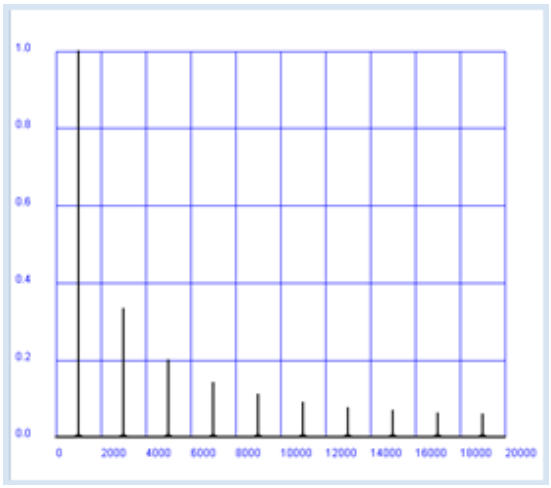




SPEKTRUM FÜR SELBSTDEFINIERTE FUNKTIONEN

Gemäss dem Satz von Fourier ist jede periodische Funktion mit der Frequenz f als Überlagerung von Sinusfunktionen mit den Frequenzen f, 2*f, 3*f, usw. darstellbar (Fourierreihe).

Mit deinem Fourieranalysator kannst du die Amplituden dieser Frequenzkomponenten experimentell bestimmen. Hier betrachtest du eine Rechteckschwingung mit der Frequenz f = 1 kHz. Die eingebaute Funktion *square(A, f, t)* liefert dir während der ersten Hälfte der Periode den Wert A und in der zweiten-A.



Du wählst eine Abtastrate von fs = 40 kHz und bestimmst die Soundsamples für eine Dauer von 3 s (120'000 Werte). Dann spielst du den Soundclip ab. Für das Spektrum verwendest du aber lediglich 10'000 Werte und stellst das Spektrum dar.

```
from soundsystem import *
from gpanel import *

def showSpectrum(text):
    makeGPanel(-2000, 22000, -0.2, 1.2)
    drawGrid(0, 20000, 0, 1.0, 10, 5, "blue")
    title(text)
    lineWidth(2)
    r = fs / n # Resolution
    f = 0
    for i in range(n // 2):
        line(f, 0, f, a[i])
        f += r

n = 10000
fs = 40000 # Sampling frequency
f = 1000 # Signal frequency

samples = [0] * 120000  # sampled data for 3 s
t = 0
dt = 1 / fs # sampling period
for i in range(120000):
   samples[i] = square(1000, f, t)
```

```
    t += dt

openMonoPlayer(samples, 40000)
play()
a = fft(samples, n)
showSpectrum("Spectrum Square Wave")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Das Experiment betätigt die Theorie, wonach das Spektrum einer Rechteckfunktion aus den ungeraden Vielfachen der Grundfrequenz besteht und sich die Amplituden der Spektralanteile wie 1, 1/3, 1/5, 1/7, usw. verhalten. Du kannst aber nie experimentell herausfinden, dass die spektralen Anteile theoretisch bis ins Unendliche reichen.

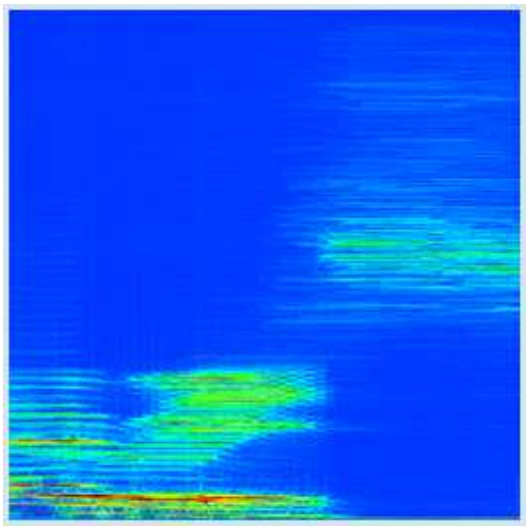
SONOGRAMME

Die FFT ist ein perfektes Tool, um den spektralen Verlauf eines sich zeitlich ändernden Lauts aufzuzeichnen, beispielsweise eines gesprochenen Worts. Das Signal ist in diesem Fall natürlich nicht mehr periodisch, man kann aber davon ausgehen, dass es wenigstens stückweise einigermassen periodisch ist. Darum wendet man die FFT für kurze Signalblöcke an, beispielsweise für eine Blocklänge von 100 ms und wiederholt dies alle 2.5 ms. Damit ergibt sich alle 2.5 ms ein neues Spektrum, das man in einem Sonogramm als kolorierte vertikale Linie darstellen kann.

In deinem Programm beginnst du am Anfang der Abtastwerte und analysierst eine Blocklänge von 2000 Werten. Den nächsten Block beginnst du 50 Samples später, usw. In Python kannst du mit einer Slice-Operation

`samples[k * 50:]` mit `k = 0, 1, 2,...`

Es entsteht dadurch ein **Sonogramm**, beispielsweise für das gesprochene Wort "harris", das sich als "*wav/harris.wav*" in der Distribution von TigerJython befindet.



```
from soundsystem import *
from gpanel import *

def toColor(z):
    w = int(450 +  300 * z)
    c = X11Color.wavelengthToColor(w)
    return c

def drawSonogram():
    makeGPanel(0, 190, 0, 1000)
    title("Sonogramm von 'Harris'")
    lineWidth(4)
    # Analyse blocks every 50 samples
    for k in range(191):
        a = fft(samples[k * 50:], n)
        for i in range(n / 2):
            setColor(toColor(a[i]))
```

```
            point(k, i)

fs = 20000 # Sampling freq->spectrum 0..10 kHz
n = 2000 #  Size of block for analyser

samples = getWavMono("wav/harris.wav")
openMonoPlayer(samples, fs)
play()
drawSonogram()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

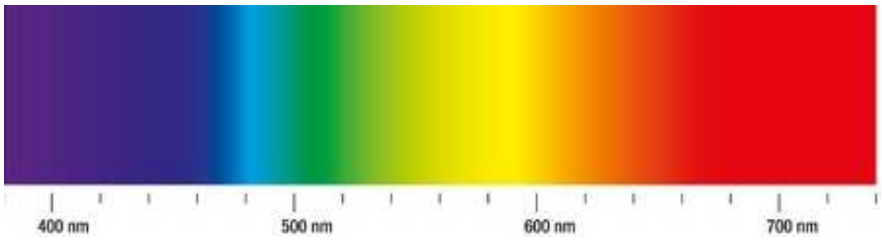
Das Sonogramm zeigt vertikal Frequenzen im Bereich 0..10 kHz, horizontal den zeitlichen Verlauf von 0 bis 190 * 50 / 20000 = 0.475 s.

Für die Umsetzung von Zahlen in Farben ist es zweckmässig, die Funktion *X11Color.wavelengthToColor*() zu verwenden, die Wellenlängen des Farbspektrums im Bereich 380...780 nm in Farben umsetzt.

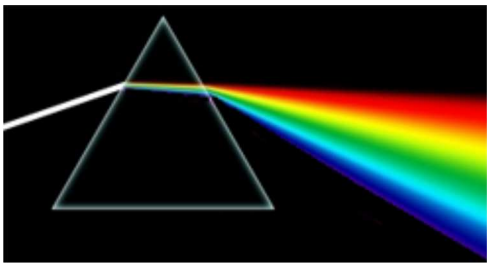
Deutlich sichtbar sind die hohen Spektralanteile beim Zischlaut "s", bei dem die Grundtöne gänzlich fehlen.

LICHTSPEKTREN

Auch Licht kann spektral zerlegt werden, um die darin enthaltenen Wellenlängenanteile zu bestimmen. Das sichtbare Spektrum liegt im Wellenlängenbereich von ungefähr 380 nm bis 780 nm



Als Spektrumanalysator für Licht kennst du sicher das Prisma, bei dem das Licht für verschiedenen Wellenlängen gemäss dem Brechungsgesetz verschieden stark abgelenkt (gebrochen) wird.



In deinem Programm simulierst du den Übergang eines weissen Lichtstrahls in Glas und zeigst in einer Vergrösserung den Lichtweg für verschiedene Farben.

```
from gpanel import *

# K5 glass
B = 1.5220
C = 4590  # nanometer^2
# Cauchy equation for refracting index
```

```
def n(wavelength):
    return B + C / (wavelength * wavelength)

makeGPanel(-1, 1, -1, 1)
title("Brechung am K5-Glas")
bgColor("black")
setColor("white")
line(-1, 0, 1, 0)


lineWidth(4)
line(-1, 1, 0, 0)
lineWidth(1)

sineAlpha = 0.707

for i in range(51):
     wavelength = 380 + 8 * i
     setColor(X11Color.wavelengthToColor(wavelength))
     sineBeta = sineAlpha / n(wavelength)
     x = (sineBeta - 0.45) * 100 - 0.5  # magnification
     line(0, 0, x, -1)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Um eine schöne Grafik zu erhalten, spaltest du die Farben mehr auf, als dies in Wirklichkeit der Fall ist.

AUFGABEN

1. Untersuche andere Musikinstrumente oder Stimmen bezüglich ihres Obertongehalts und versuche, deine Feststellungen auf Grund des typischen Klangcharakters zu verstehen.

2. Neben der globalen Funktion *square(A, f, t)* stehen dir noch die Funktionen *sine(a, f, t), triangle(A, f, t), sawtooth(A, f, t)* zur Verfügung. Stelle eine Vermutung über das Spektrum einer Dreieckschwingung und einer Sägezahnschwingung auf. Mit *sine()* kannst du auch Überlagerungen von Sinusschwingungen untersuchen.

3. Untersuche in Sonogrammen die Unterschiede zwischen verschiedenen Sprecherinnen und Sprechern, die das gleiche Wort sprechen.

8.9 GRUPPENDYNAMIK

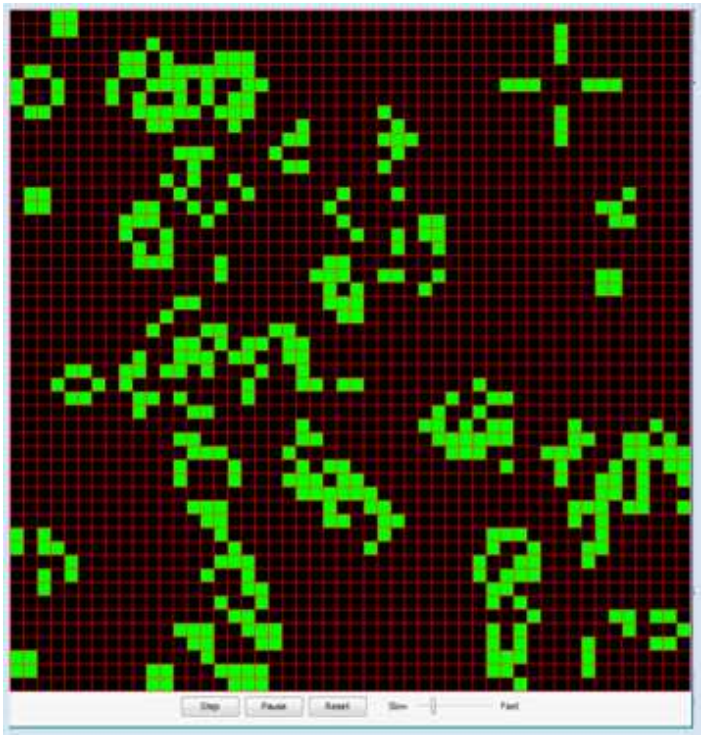
EINFÜHRUNG

Systeme mit vielen Partnern, die aufeinander einwirken, sind weit verbreitet. Computerprogramme können solche Systeme oft mit erstaunlich wenig Aufwand simulieren, da der Computer den Zustand von Tausenden, ja Millionen von Einzelindividuen speichern und zeitlich verfolgen kann. Handelt es sich aber um atomare Vielteilchensysteme mit einer Teilchenzahl in der Grössenordnung von 10^{23}, so stösst auch der Computer an seine Grenzen. Zur Simulation solcher Systeme müssen vereinfachende Verfahren eingesetzt werden, etwa indem man das System in einzelne grössere Zellen einteilt. Beispiele dafür sind die Simulation der Erdatmosphäre für die Wetterprognose und die Voraussage der langfristigen Klimaentwicklung.

PROGRAMMIERKONZEPTE: *Computersimulation, Populationsdynamik, Schwarmverhalten*

CONWAY'S GAME OF LIFE

Es wird eine zweidimensionale gitterartige Anordnung von Individuen untersucht, wo jedes Individuum mit seinen 8 nächsten Nachbarn in Wechselwirkung steht. Sie wurde um 1970 vom britischen Mathematiker John Conway vorgeschlagen und machte ihn damit auch ausserhalb der Mathematik weltberühmt. Fast alle gebildeten Menschen haben wenigstens eine Idee, um was es sich beim Game of Life handelt. Hier wirst du es mit Python selbst programmieren.



Die Population ist in Zellen angeordnet und entwickelt sich in diskreten Zeitschritten (Generationen). Jede Zelle kann *lebend* oder *tot* sein.

Beim Übergang zur nächsten Generation wird der aktuelle Zustand festgehalten und der Nachfolgezustand jeder Zelle auf Grund ihrer 8 nächsten Nachbarn mit folgenden vier Übergangsregeln bestimmt:

1. Lebt die Zelle, so stirbt sie, wenn sie weniger als zwei lebende Nachbarn hat (Vereinsamung)
2. Lebt die Zelle, so lebt sie weiter, wenn sie zwei oder drei lebende Nachbarn hat (Gruppenzusammengehörigkeit)
3. Lebt die Zelle, so stirbt sie, wenn sie mehr als drei lebende Nachbarn hat (Überbevölkerung)

4. Ist eine Zelle tot, so wird sie lebendig, wenn sie genau drei lebende Nachbarn hat (Reproduktion). Sonst bleibt sie tot.

Die Zellenstruktur von *GameGrid* ist ideal, um das Spiel zu implementieren. Für die Population verwendest du eine zweidimensionale Liste *a[x][y]* mit den Werten 0 für eine tote und 1 für eine lebende Zelle. Der Simulationszyklus wird als Generationszyklus aufgefasst und im Callback *onAct()* die aktuelle Population von der Liste *a* in die neue Population *b* umkopiert, die am Schluss als die aktuelle Liste angesehen wird. Im Callback *onReset()*, der beim Klick auf den Reset-Button aufgerufen wird, wählst du 1000 zufällige lebende Zellen.

Um die Callbacks zu aktivieren, musst du sie mit *registerAct()* bzw. *registerNavigation()* registrieren.

```
from gamegrid import *

def onReset():
    for x in range(s):
        for y in range(s):
            a[x][y] = 0  # All cells dead
    for n in range(z):
        loc = getRandomEmptyLocation()
        a[loc.x][loc.y] = 1
    showPopulation()

def showPopulation():
    for x in range(s):
        for y in range(s):
            loc = Location(x, y)
            if a[x][y] == 1:
                getBg().fillCell(loc, Color.green, False)
            else:
                getBg().fillCell(loc, Color.black, False)
    refresh()

def getNumberOfNeighbours(x, y):
    nb = 0
    for i in range(max(0, x - 1), min(s, x + 2)):
        for k in range(max(0, y - 1), min(s, y + 2)):
            if not (i == x and k == y):
                if a[i][k] == 1:
                    nb = nb + 1
    return nb

def onAct():
    global a
    # Don't use the current, but a new population
    b  = [[0 for x in range(s)] for y in range(s)]
    for x in range(s):
        for y in range(s):
            nb = getNumberOfNeighbours(x, y)
            if a[x][y] == 1:  # living cell
                if nb < 2:
                    b[x][y] = 0
                elif nb > 3:
                    b[x][y] = 0
                else:
                    b[x][y] = 1
            else:             # dead cell
                if nb == 3:
                    b[x][y] = 1
                else:
                    b[x][y] = 0
    a = b # Use new population as current
    showPopulation()

# ================== global section ==================
s = 50   # Number of cells in each direction
```

```
z = 1000 # Size of population at start
a  = [[0 for x in range(s)] for y in range(s)]
makeGameGrid(s, s, 800 // s, Color.red)
registerAct(onAct)
registerNavigation(resetted = onReset)
setTitle("Conway's Game Of Life")
onReset()
show()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

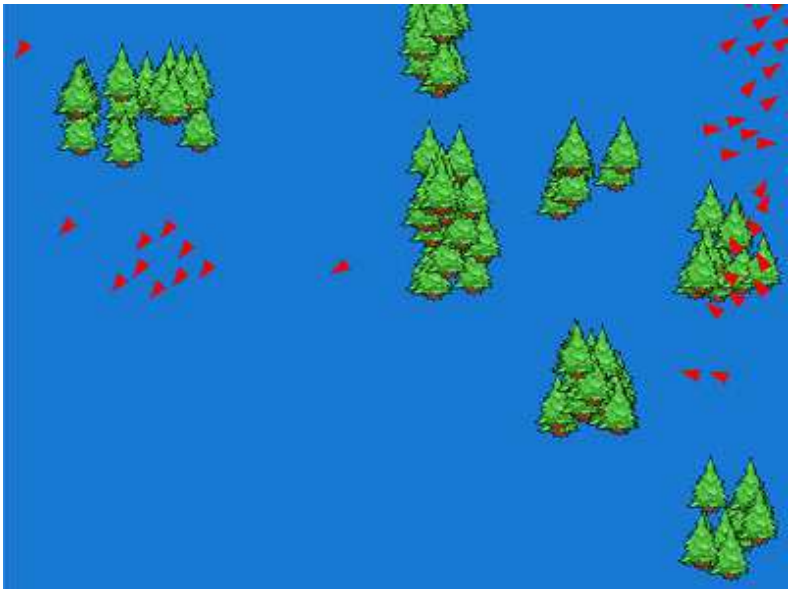
Das Game of Live ist ein Beispiel eines **zellulären Automaten**, der aus Gitterzellen besteht, die miteinander wechselwirken. Sie sind hervorragend geeignet, um das Verhalten komplexer natürlicher Systeme zu untersuchen. Beispiele:

- biologisches Wachstum, Entstehung des Lebens
- soziales, geologisches, ökologisches Verhalten
- Verkehrsaufkommen und Verkehrsregelung
- Entstehung und Entwicklung des Kosmos, von Galaxien und Sternen

Der Wissenschaftler und Chefentwickler von Mathematica Stefan Wolfram hat 2002 in seinem bekannten Buch "A New Kind of Science" darauf hingewiesen, dass solche Systeme mit einfachen Programmen untersucht werden können und wir mit der Computercomputersimulation am Anfang einer neuen Ära stehen, wissenschaftliche Erkenntnisse zu gewinnen.

Bei der Initialisierung der zweidimensionalen Liste wird eine spezielle Python-Syntax mit der Bezeichnung **List Comprehension** verwendet (siehe **Zusatzstoff**).

SCHWARMVERHALTEN



Wie du aus dem täglichen Leben weisst, haben grosse Gruppen von Lebewesen oft die Tendenz, sich in Gruppen zusammenzuschliessen. Dies ist besonders gut bei Vögeln, Fischen und Insekten zu beobachten. Aber auch demonstrierende Menschen zeigen dieses "Schwarmverhalten". Für die Bildung eines Schwarms spielen einerseits äussere (globale) Einflüsse, aber auch die Wechselwirkung zwischen Partnern in der näheren Umgebung, also lokale Einflüsse, eine Rolle.

Craig Reynolds hat 1986 gezeigt, dass folgende drei Regeln zu einer Schwarmbildung zwischen Individuen (er nannte sie *Boids*) führen:

1. **Kohäsionsregel**: Bewege dich in Richtung des Mittelpunkts (Schwerpunkts) der Individuen in deiner Nachbarschaft
2. **Separationsregel**: Bewege dich weg, wenn dir ein Individuum zu nahe kommt
3. **Alignmentregel**: Bewege dich in etwa gleicher Richtung wie deine Nachbarn

Für die Implementierung verwendest du wieder *GameGrid*, um den Aufwand für die Animation klein zu halten. Es ist zweckmässig, ein Gitter mit Pixelgrösse zu verwenden und den Ort, die Geschwindigkeit und die Beschleunigung der Actors mit Float-Vektoren aus der Klasse *GGVector* anzugeben. In jeder Simulationsperiode bestimmst du zuerst mit *setAcceleration()* den neuen Beschleunigungsvektor gemäss der drei Regeln. Daraus ergeben sich der neue Geschwindigkeits- und Ortsvektor aus

$$\vec{v}' = \vec{v} + \vec{a} * dt \qquad \text{und} \qquad \vec{r}' = \vec{r} + \vec{v} * dt$$

Da die absolute Zeitskala unwesentlich ist, kannst du für den Zeitschritt *dt = 1* setzen.

Bei der Anwendung der Separationsregel führen nicht nur nahe fliegende Vögel zu einer Abstossung, sondern auch Hindernisse (hier die Bäume).

Der Rand des Flugbereichs (die Wand) muss besonders behandelt werden. Dazu stehen verschiedene Möglichkeiten zur Auswahl. Es könnte eine torusartige Topologie verwendet werden, wo die auf der einen Seite aus dem Bereich hinausfliegenden Vögel auf der gegenüberliegenden Seite wieder hineinfliegen. Hier werden die Vögel gezwungen, am Rand umzukehren.

```
from gamegrid import *
import math
import random

# ==================== class Tree ======================
class Tree(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/tree1.png")

# ==================== class Bird ======================
class Bird(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/arrow1.png")
        self.r = GGVector(0, 0)  # Position
        self.v = GGVector(0, 0)  # Velocity
        self.a = GGVector(0, 0)  # Acceleration

    # Called when actor is added to gamegrid
    def reset(self):
        self.r.x = self.getX()
        self.r.y = self.getY()
        self.v.x = startVelocity * math.cos(math.radians(self.getDirection()))
        self.v.y = startVelocity * math.sin(math.radians(self.getDirection()))

    # ----------- cohesion ---------------
    def cohesion(self, distance):
        return self.getCenterOfMass(distance).sub(self.r)

    # ----------- alignment --------------
    def alignment(self, distance):
        align = self.getAverageVelocity(distance)
        align = align.sub(self.v)
        return align

    # ----------- separation -------------
    def separation(self, distance):
```

```
        repulse = GGVector()
        # ------ from birds ------
        for p in birdPositions:
            dist = p.sub(self.r)
            d = dist.magnitude()
            if d < distance and d != 0:
                repulse = repulse.add(dist.mult((d - distance) / d))

        # ------ from trees ------
        trees = self.gameGrid.getActors(Tree)
        for actor in trees:
            p = GGVector(actor.getX(), actor.getY())
            dist = p.sub(self.r)
            d = dist.magnitude()
            if d < distance and d != 0:
               repulse = repulse.add(dist.mult((d - distance) / d))
        return repulse

    # ------------ wall interaction -------
    def wallInteraction(self):
        width = self.gameGrid.getWidth()
        height = self.gameGrid.getHeight()
        acc = GGVector()
        if self.r.x < wallDist:
            distFactor = (wallDist - self.r.x) / wallDist
            acc = GGVector(wallWeight * distFactor, 0)
        if width - self.r.x < wallDist:
            distFactor = ((width - self.r.x) - wallDist) / wallDist
            acc = GGVector(wallWeight * distFactor, 0)
        if self.r.y < wallDist:
            distFactor = (wallDist - self.r.y) / wallDist
            acc = GGVector(0, wallWeight * distFactor)
        if height - self.r.y < wallDist:
            distFactor = ((height - self.r.y) - wallDist) / wallDist
            acc = GGVector(0, wallWeight * distFactor)
        return acc

    def getPosition(self):
        return self.r

    def getVelocity(self):
        return self.v

    def getCenterOfMass(self, distance):
        center = GGVector()
        sum = 0
        for p in birdPositions:
            dist = p.sub(self.r)
            d = dist.magnitude()
            if d < distance:
                center = center.add(p)
                sum += 1
        if sum != 0:
            return center.mult(1.0/sum)
        else:
            return center

    def getAverageVelocity(self, distance):
        avg = GGVector()
        sum = 0
        for i in range(len(birdPositions)):
            p = birdPositions[i]
            if (self.r.x - p.x) * (self.r.x - p.x) + \
               (self.r.y - p.y) * (self.r.y - p.y) < distance * distance:
                avg = avg.add(birdVelocities[i]);
                sum += 1
        return avg.mult(1.0/sum)

    def limitSpeed(self):
```

```
        m = self.v.magnitude()
        if m < minSpeed:
            self.v = self.v.mult(minSpeed / m)
        if m > maxSpeed:
            self.v = self.v.mult(minSpeed / m)

    def setAcceleration(self):
        self.a = self.cohesion(cohesionDist).mult(cohesionWeight)
        self.a = self.a.add(self.separation(separationDist).mult(separationWeight))
        self.a = self.a.add(self.alignment(alignmentDist).mult(alignmentWeight))
        self.a = self.a.add(self.wallInteraction())

    def act(self):
        self.setAcceleration()
        self.v = self.v.add(self.a) # new velocity
        self.limitSpeed()
        self.r = self.r.add(self.v) # new position
        self.setDirection(int(math.degrees(self.v.getDirection())))
        self.setLocation(Location(int(self.r.x), int(self.r.y)))


# ================== global section ==================
def populateTrees(number):
    blockSize = 70
    treesPerBlock = 10
    for block in range(number // treesPerBlock):
        x = getRandomNumber(800 // blockSize) * blockSize
        y = getRandomNumber(600 // blockSize) * blockSize
        for t in range(treesPerBlock):
            dx = getRandomNumber(blockSize)
            dy = getRandomNumber(blockSize)
            addActor(Tree(), Location(x + dx, y + dy))

def generateBirds(number):
    for i in range(number):
        addActorNoRefresh(Bird(), getRandomLocation(),
                                  getRandomDirection())
    onAct()  # Initialize birdPositions, birdVelocities

def onAct():
    global birdPositions, birdVelocities
    # Update bird positions and velocities
    birdPositions = []
    birdVelocities = []
    for b in getActors(Bird):
        birdPositions.append(b.getPosition())
        birdVelocities.append(b.getVelocity())

def getRandomNumber(limit):
    return random.randint(0, limit-1)

# coupling constants
cohesionDist = 100
cohesionWeight = 0.01
alignmentDist = 30
alignmentWeight = 1
separationDist = 30
separationWeight = 0.2
wallDist = 20
wallWeight = 2
maxSpeed = 20
minSpeed = 10
startVelocity = 10
numberTrees = 100
numberBirds = 50

birdPositions  = []
birdVelocities  = []
```

```
makeGameGrid(800, 600, 1, False)
registerAct(onAct)
setSimulationPeriod(10)
setBgColor(makeColor(25, 121, 212))
setTitle("Swarm Simulation")
show()
populateTrees(numberTrees)
generateBirds(numberBirds)
doRun()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

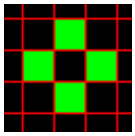
MEMO

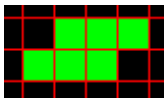
Die Simulation ist von verschiedenen **Kopplungskonstanten** abhängig, welche die "Stärke" der Wechselwirkung bestimmen. Ihre Werte sind sehr sensibel und du musst sie eventuell an die Leistungsfähigkeit deines Computers anpassen.

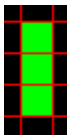
Wiederum wird der Callback *onAct()* mit *registerAct()* aktiviert, damit er automatisch in jedem Simulationszyklus aufgerufen wird. In der Methode *act()* der Klasse *Bird* werden die Vögel bewegt.

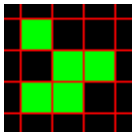
AUFGABEN

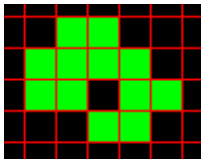
1. Untersuche im Game of Life das Verhalten folgender Muster:

a. 

b. 

c. 

d. 

e. 

2. Beschreibe drei typische Schwarmverhalten in der Tierwelt. Überlege dir in jedem Beispiel, welches die Gründe sein könnten, dass sich die Tiere in einem Schwarm zusammenfinden.

3*. Führe in der Schwarmsimulation drei Raubvögel ein, welche die Schwarmvögel verfolgen, aber von diesen gemieden werden. Anleitung: Verwende für die Raubvögel das Spritebild *arrow2.png*.

4*. Die Raubvögel in Aufgabe 3 sollen die Schwarmvögel bei einer Kollision auffressen.

ZUSATZSTOFF

LIST COMPREHENSION

Im Python können Listen elegant mit einer speziellen Schreibweise erzeugt werden. Sie lehnt sich an die mathematische Notation aus der Mengenlehre an.

Mathematik	*Python*
S = {x : x in {1...10}}	s = [x for x in range(1, 11)]
T = {x^2 : x in {0...10}}	t = [x**2 for x in range(11)]
V = {x | x in S und x gerade}	v = [x for x in s if x % 2 == 0]
$M = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	m = [[0 for x in range(3)] for y in range(3)]

Verwende die Konsole, um die Python-Ausdrücke auszutesten. Du kannst sie aus der Tabelle kopieren und in die Konsole einfügen.

```
>>> s = [x for x in range(1, 11)]
>>> s
 < [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> t = [x**2 for x in range(11)]
 < [0, 1, 4, 9, 16, 25, 35, ..., 100]
>>> v = [x for x in s iff x% 2 == 0]
 < [2, 4, 6, 8, 10]
>>> m = [[0 for x in range(3)] for y in range(3)]
>>> m
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
```

8.10 RANDOM WALK

EINFÜHRUNG

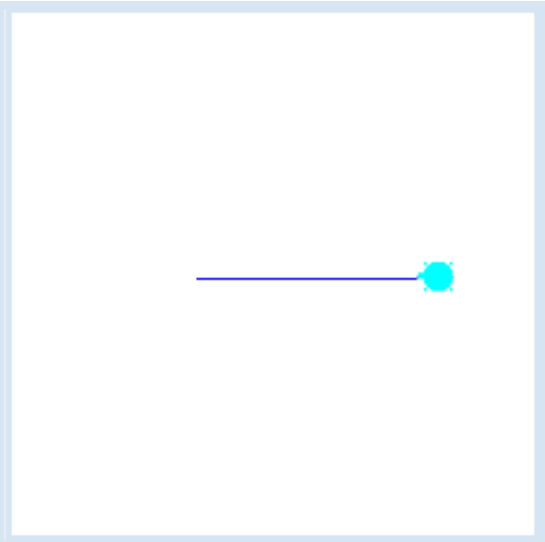
Im täglichen Leben sind viele Phänomene durch den Zufall bestimmt und du musst oft Entscheide auf der Basis von Wahrscheinlichkeiten fällen, beispielsweise ob du auf Grund einer Unfallwahrscheinlichkeit das eine oder andere Verkehrsmittel wählst. Der Computer kann in solchen Situationen ein wichtiges Hilfsmittel sein, dann mit ihm kannst du das Gefahrenpotential als Simulation untersuchen, ohne selbst gefährdet zu werden.

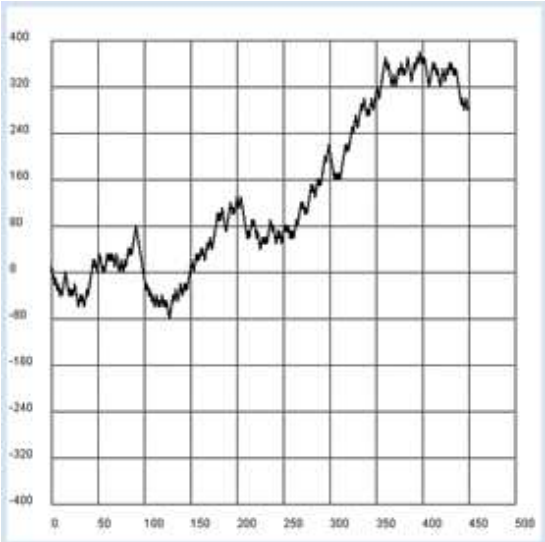
Im Folgenden gehst du davon aus, dass du dich verirrt hast und den Heimweg nicht kennst. Dabei betrachtest du eine Bewegung, Random Walk oder Irrfahrt genannt, die in einzelnen Zeitschritten erfolgt, wobei die Schritte in zufälliger Richtung gewählt werden. Obschon eine solche Bewegung nicht genau der Realität entspricht, lernst du wichtige Eigenschaften kennen, die sich nachher auf reale Systeme, beispielsweise in der Finanzmathematik zur Modellierung von Aktienkursen oder auf die Brownsche Molekularbewegung, anwenden lassen.

PROGRAMMIERKONZEPTE: *Random Walk, Brownsche Bewegung*

EINDIMENSIONALER RANDOM WALK

Wiederum eignet sich die Turtlegrafik hervorragend, um die Simulation grafisch zu untermalen. Die Turtle soll sich zur Zeit 0 an der Stelle x = 0 befinden und sich mit gleich langen Schritten auf der x-Achse bewegen. Zu jedem Zeitschritt "überlegt" sie, ob sie einen Rechts- oder Linksschritt macht. In deinem Programm fällt sie den Entscheid mit der gleichen Wahrscheinlichkeit p = ½. (symmetrischer Random Walk).






```
from gturtle import *
from gpanel import *
import random

makeTurtle()
makeGPanel(-50, 550, -480, 480)
windowPosition(880, 10)
drawGrid(0, 500, -400, 400)
```

```
title("Position versus Time")
lineWidth(2)
setTitle("Random Walk")

t = 0
while t < 500:
    if random.randint(0, 1) == 1:
        setHeading(90)
    else:
        setHeading(-90)
    forward(10)
    x = getX()
    draw(t, x)
    t += 1
print "All done"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Es erstaunt dich sicher, dass sich in den meisten Fällen die Turtle mit der Zeit vom Startpunkt wegbewegt, obschon sie bei jedem Schritt mit gleicher Wahrscheinlichkeit einen Rechts- oder Linksschritt macht. Allerdings kann sie sich dabei nach rechts oder links wegbewegen. Um dieses wichtige Ergebnis genauer zu untersuchen, bestimmst du im nächsten Programm für je 1000 Laufversuche den Mittelwert d der Distanz vom Startpunkt nach t Schritten.

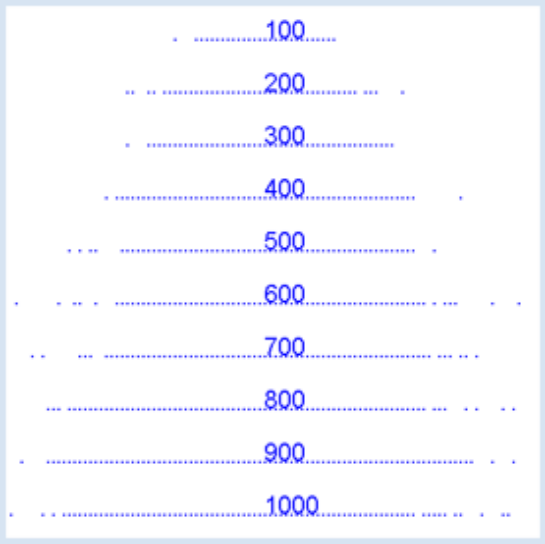
DAS WURZEL-AUS-T-GESETZ

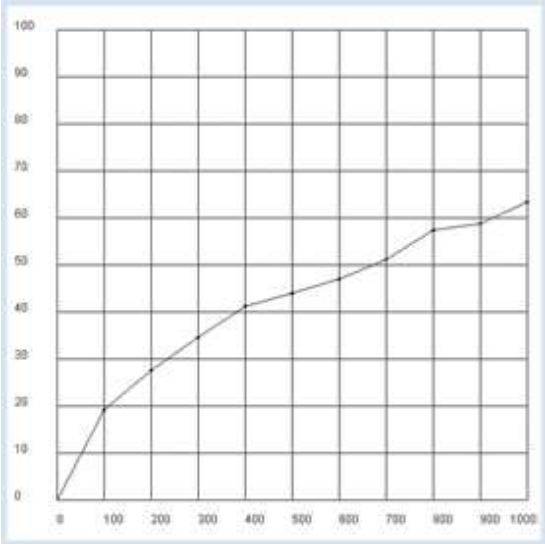
Du führst die Simulation für eine bestimmte feste Schrittzahl t an fester y-Position 1000 Mal durch und zeichnest die Endposition als Punkt ein. Damit du nicht zu lange auf die Resultate warten musst, versteckst du die Turtle und zeichnest auch keine Spuren. Bei jedem Simulationsversuch bestimmst du den Abstand r der Endposition vom Startpunkt. Für jede Versuchsreihe mit einen bestimmten t ergibt sich so der Mittelwert d der Abtände r, den du in einer GPanel-Grafik einträgst.

```
from gturtle import *
from gpanel import *
import math
import random

makeTurtle()
makeGPanel(-100, 1100, -10, 110)
windowPosition(850, 10)
drawGrid(0, 1000, 0, 100)
title("Mean distance versus time")
ht()
pu()

for t in range(100, 1100, 100):
    setY(250 - t / 2)
    label(str(t))
    sum = 0
    repeat 1000:
        repeat t:
           if random.randint(0, 1) == 1:
               setHeading(90)
           else:
               setHeading(-90)
           forward(2.5)
        dot(3)
        r = abs(getX())
        sum += r
```

```
        setX(0)
    d = sum / 1000
    print "t =", t, "d =", d, "q = d / sqrt(t) =", d / math.sqrt(t)
    draw(t, d)
    fillCircle(5)
print "all done"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)





MEMO

Wie du in der grafischen Darstellung siehst, nimmt der Mittelwert der Distanz zum Startpunkt mit zunehmender Schrittzahl bzw. Zeit t zu. In der Konsole berechnest du noch den Quotienten *q = d / sqrt(t)*. Da dieser nahezu konstant ist, liegt die Vermutung nahe, dass *d = q * sqrt(t)* exakt gilt, oder:

Die mittlere Distanz zum Startpunkt nimmt mit der Wurzel aus der Zeit zu.

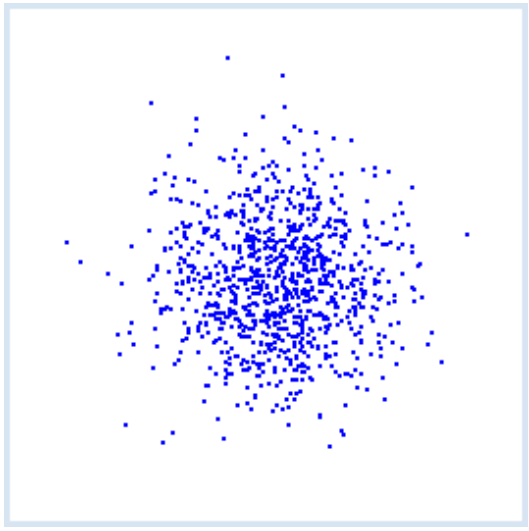
DRUNKEN MAN'S WALK

Noch etwas spannender wird es, wenn sich die Turtle in zwei Dimensionen bewegen kann. Man spricht dann von einem zweidimensionalen Random Walk. Dabei macht die Turtle immer noch gleich lange Schritte, wählt aber bei jedem Schritt eine zufällige Richtung.

Mathematiker und Physiker kleiden das Problem oft in folgende nicht allzu erstzunehmende Geschichte ein.

"Ein Betrunkener versucht nach einer Wirtshaustour nach Hause zurückzukehren. Da er die Orientierung verloren hat macht er dabei immer einen Schritt in zufälliger Richtung. Wie weit bewegt er sich dabei (im Mittel) vom Wirtshaus weg?

Du brauchst das vorhergehende Programm nur leicht abzuändern. Dabei untersuchst du wieder, wie der Mittelwert der Distanz von der Zeit abhängt.



```
from gturtle import *
from gpanel import *
import math

makeTurtle()
makeGPanel(-100, 1100, -10, 110)
windowPosition(850, 10)
drawGrid(0, 1000, 0, 100)
title("Mean distance versus time")
ht()
pu()

for t in range(100, 1100, 100):
    sum = 0
    clean()
    repeat 1000:
        repeat t:
           fd(2.5)
           setRandomHeading()
        dot(3)
        r = math.sqrt(getX() * getX() + getY() * getY())
        sum += r
        home()
    d = sum / 1000
    print "t =", t, "d =", d, "q = d / sqrt(t) =", d / math.sqrt(t)
    draw(t, d)
    fillCircle(5)
    delay(2000)
print "all done"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

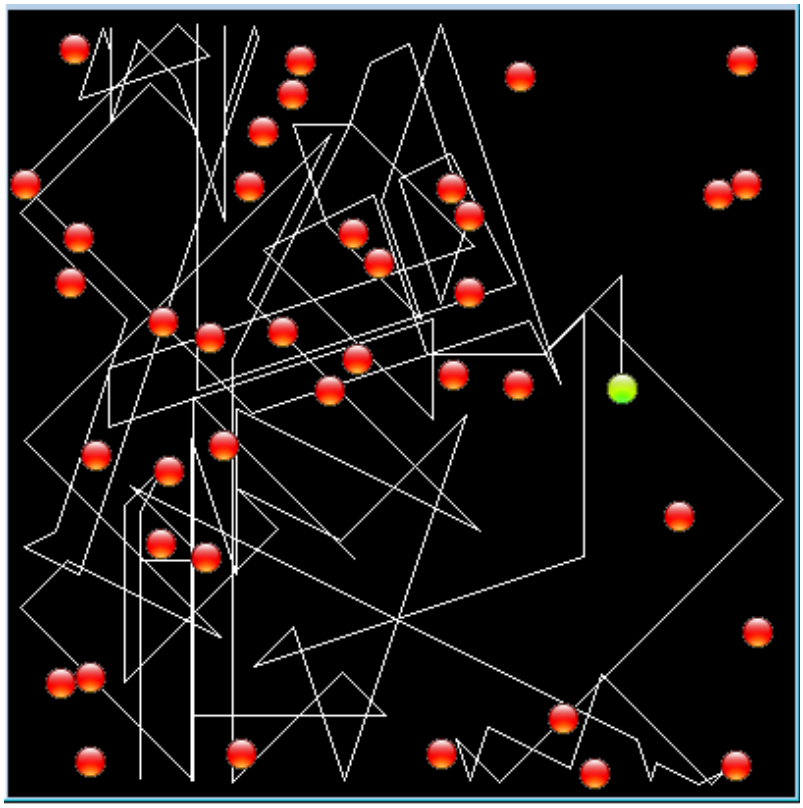
MEMO

Auch in zwei Dimensionen gilt das Wurzel-aus-t-Gesetz. Für Gasteilchen hat kein geringerer als Albert Einstein diese Gesetzmässigkeit 1905 in seiner berühmten Abhandlung "Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen" bewiesen und damit eine theoretische Erklärung für die Brownsche Bewegung gegeben. Ein Jahr später hat M. Smoluchowski mit einer anderen Überlegung dasselbe Resultat gefunden.

BROWNSCHE BEWEGUNG

Bereits 1827 hatte der Biologe Robert Brown beim Mikroskopieren beobachtet, dass Pollenkörner in einem Wassertropfen unregelmässige zuckende Bewegungen machen. Er vermutete als Ursache eine in den Pollen innewohnende Lebenskraft. Erst mit der Entdeckung der molekularen Struktur der Materie wurde klar, dass die thermische Bewegung der Wassermoleküle, die mit dem Pollen zusammenstossen, für das Phänomen verantwortlich ist.

Die Brownsche Bewegung lässt sich sehr schön in einem Computerexperiment demonstrieren, wo die Moleküle als kleine Kugeln modelliert werden, die beim Zusammenstoss ihre Geschwindigkeit austauschen. Mit *JGameGrid* lässt sich die Simulation einfach realisieren, da du eine Teilchenkollision als Ereignis auffassen kannst. Dazu leitest du die Klasse *CollisionListener* von *GGActorCollisionListener* ab und implementierst den Callback *collide()*. Jedem Partikel fügst du mit *addActorCollisionListener()* den Listener hinzu. Die Art und die Grösse der Kollisionsarea kannst du mit *setCollisionCircle()* einstellen. Du teilst die 40 Teilchen zu Beginn in 4 Geschwindigkeitsgruppen ein.



```
from gamegrid import *

# ==================== class Particle ====================
class Particle(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/ball.gif", 2)

    # Called when actor is added to gamegrid
    def reset(p):
        p.oldPt = p.gameGrid.toPoint(p.getLocationStart())

    def advance(p, distance):
        pt = p.gameGrid.toPoint(p.getNextMoveLocation())
        dir = p.getDirection()
        # Left/right wall
        if pt.x < 5 or pt.x > w - 5:
            p.setDirection(180 - dir)
        # Top/bottom wall
        if pt.y < 5 or pt.y > h - 5:
            p.setDirection(360 - dir)
        p.move(distance)

    def act(p):
        p.advance(3)
        if p.getIdVisible() == 1:
            pt = p.gameGrid.toPoint(p.getLocation())
            p.getBackground().drawLine(p.oldPt.x, p.oldPt.y, pt.x, pt.y)
            p.oldPt.x = pt.x
            p.oldPt.y = pt.y

# ==================== class CollisionListener =========
class CollisionListener(GGActorCollisionListener):
   # Collision callback: just exchange direction and speed
    def collide(self, a,  b):
        dir1 = a.getDirection()
        dir2 = b.getDirection()
        sd1 = a.getSlowDown()
        sd2 = b.getSlowDown()
        a.setDirection(dir2)
```

```
        a.setSlowDown(sd2)
        b.setDirection(dir1)
        b.setSlowDown(sd1)
        return 10  # Wait a moment until collision is rearmed

# ==================== Global section ====================
def init():
    collisionListener = CollisionListener()
    for i in range(nbParticles):
        particles[i] = Particle()
        # Put them at random locations, but apart of each other
        ok = False
        while not ok:
            ok = True
            loc = getRandomLocation()

        for k in range(i):
            dx = particles[k].getLocation().x - loc.x
            dy = particles[k].getLocation().y - loc.y
            if dx * dx + dy * dy < 300:
                ok = False
        addActor(particles[i], loc, getRandomDirection())
        # Select collision area
        particles[i].setCollisionCircle(Point(0, 0), 8)
        # Select collision listener
        particles[i].addActorCollisionListener(collisionListener)
        # Set speed in groups of 10
        if i < 10:
            particles[i].setSlowDown(2)
        elif i < 20:
            particles[i].setSlowDown(3)
        elif i < 30:
            particles[i].setSlowDown(4)
    #  Define collision partners
    for i in range(nbParticles):
        for k in range(i + 1, nbParticles):
            particles[i].addCollisionActor(particles[k])
    particles[0].show(1)

w = 400
h = 400
nbParticles = 40
particles = [0] * nbParticles

makeGameGrid(w, h, 1, False)
setSimulationPeriod(10)
setTitle("Brownian Movement")
show()
init()
doRun()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Moleküle sind in der Klasse *Particle* modelliert, die von *Actor* abgeleitet ist. Sie haben zwei Spritebilder, ein rotes und ein grünes für ein besonders ausgezeichnetes Molekül, dessen Bahn du verfolgst. Das grüne Bild entspricht der *Sprite-ID* = 1, auf die du in *act()* testest, um mit *drawLine()* die Spur zu zeichnen.

AUFGABEN

1. Bei einem eindimensionalen Random Walk mit 100 Schritten bewegt sich die Person ausgehend von x = 0 mit gleicher Wahrscheinlichkeit p = ½ nach rechts und q = ½ nach links. Führe die Simulation 10000 mal aus und bestimme die Häufigkeitsverteilung der Endposition. Stelle sie in einem GPanel dar.

2. Wie du in Aufgabe 1 siehst, ist die Wahrscheinlichkeit, sich nach hundert Schritten wieder am Startpunkt zu befinden, am grössten. Wie lässt sich dies mit dem Wurzel-aus-t-Gesetz vereinbaren?

3. Führe die gleiche Simulation durch, wenn die Wahrscheinlichkeit für einen Rechtschritt p = 0.8 und für einen Linksschritt q = 0.2 ist.

4*. Die Theorie liefert beim eindimensionalen Random Walk eine Binomialverteilung. Die Wahrscheinlichkeit, sich nach n Schritten an der Koordinate x zu befinden, beträgt für gerade n und x:

$$P(x,n) = \frac{n!}{((n+x)/2)!((n-x)/2)!} p^{(n+x)/2} q^{(n-x)/2}$$

Trage den theoretischen Verlauf als Kurve in das Histogramm aus Aufgabe 1 ein.

kapitel neun

DATENBANKEN

Lernziele

- Du kannst erklären, warum Dateien in der Informatik eine wichtige Rolle spielen.
- Du weisst, wie man Daten aus einer Textdatei lesen, konvertieren und wieder abspeichern kann.
- Du kennst einige wichtige Eigenschaften von Online-Datenbanken und kannst einen eigenen Datenbankserver installieren.
- Du bist dir bewusst, dass heutzutage in Online- Datenbanken und sozialen Netzwerken eine riesige Menge von Informationen elektronisch gespeichert werden.
- Du kannst auf einem Datenbankserver mit SQL Tabellen erzeugen, Datensätze erstellen, lesen und modifizieren.
- Du bist in der Lage, ein einfaches Online-Reservationssystem aufzubauen und zu managen.
- Du weisst, wie das Abfangen von Exceptions funktioniert und warum es ein wichtiges Prinzip ist.

Wer Personendaten bearbeitet, hat sich über deren Richtigkeit zu vergewissern. Er hat alle angemessenen Massnahmen zu treffen, damit die Daten berichtigt oder vernichtet werden, die im Hinblick auf den Zweck ihrer Beschaffung oder Bearbeitung unrichtig oder unvollständig sind. Jede betroffene Person kann verlangen, dass unrichtige Daten berichtigt werden

Schweiz. Bundesgesetz über den Datenschutz, Artikel 5

9.1 PERSISTENZ, DATEIEN

EINFÜHRUNG

In der heutigen High-Tech-Gesellschaft spielen computergespeicherte Informationen, kurz Daten genannt, eine zentrale Rolle. Obschon sie mit geschriebenem Text vergleichbar sind, gibt es mehrere wichtige Unterschiede:

- Daten können nur mit einem **Computersystem** gespeichert, gelesen und verarbeitet werden
- Daten werden immer als **0,1 Werte codiert**. Sie erhalten erst einen Informationsgehalt, bzw. einen Sinn, wenn sie richtig **interpretiert** (decodiert) werden
- Daten besitzen eine **Lebensdauer**. Temporäre Daten existieren als lokale Variablen kurzzeitig in einem gewissen Programmblock oder als globale Variablen während der ganzen Programmdauer. Persistente Daten hingegen überleben die Programmdauer und können später wieder abgerufen werden
- Daten besitzen eine **Sichtbarkeit (Verfügbarkeit)**. Während gewisse Daten, beispielsweise Personendaten in einem sozialen Netzwerk von jedermann gelesen werden können, gibt es geheime Daten oder solche, die sich auf Datenträgern befinden, die nicht allgemein zugänglich sind
- Daten können **geschützt** werden. Dies kann durch **Verschlüsselung** oder durch **Zugriffsbeschränkungen** (Zugangs- und Passwortschutz) erreicht werden
- Daten können auf **digitalen Kommunikationskanälen** leicht transportiert werden

Persistente Daten werden mit Computerprogrammen auf einen physikalischen Datenträger (typisch: Festplatte (HD), Solid State Disk (SSD), Speicherkarte (Memory Stick)) als Dateien geschrieben bzw. gelesen. Dabei handelt es sich um Speicherbereiche mit einer bestimmten Struktur und einem bestimmten **Dateiformat**. Da die Datenübertragung auch über grosse Distanzen schnell und billig geworden ist, werden Dateien immer häufiger auch auf weit entfernten Datenträgern (**Clouds**) abgelegt.

Dateien werden auf dem Computer in einer hierarchischen Verzeichnisstruktur verwaltet, d.h. in einem bestimmten Verzeichnis können sich Dateien, aber auch Unterverzeichnisse befinden. Der Zugriff erfolgt über den **Dateipfad** (kurz Pfad), der die Verzeichnisse und den Dateinamen enthält. Das Dateisystem ist aber **betriebssystemabhängig**, sodass es gewisse Unterschiede zwischen Windows, Mac und Linux-Systemen gibt.

PROGRAMMIERKONZEPTE: *Kodierung, Lebensdauer, Sichtbarkeit von Daten, Datei*

TEXTDATEIEN LESEN UND SCHREIBEN

Du hast bereits im Kapitel Internet gelernt, wie man in Python Textdateien lesen kann. In Textdateien sind die Zeichen zwar hintereinander (sequentiell) abgelegt, aber man erhält eine Zeilenstrukturierung wie auf einem Blatt Papier durch Einfügen von Zeilenendzeichen. Gerade hier unterscheiden sich die Betriebssysteme: Während bei Mac und Linux als Zeilenendzeichen (End of Line, EOL) das ASCII-Zeichen <line feed> verwendet wird, ist es bei Windows die Kombination von <carriage return><line feed>. In Python werden diese Zeichen mit \r bzw. \n codiert [**mehr...**].

In deinem Programm verwendest du deutschsprachige Wortlisten, die als Textdateien vorliegen. Du kannst sie als *tfwordlist.zip* von **hier** downloaden und in irgend einem Verzeichnis auspacken. Kopiere die Dateien *worte-1$.txt* und *verben-1$.txt* in das Verzeichnisses, in dem sich dein Programm befindet.

Du stellst dir hier die interessante Frage, welche Wörter **Palindrome** sind. Darunter versteht man Wörter, die von vorne und hinten gelesen gleich lauten, wobei die Gross-Kleinschreibung nicht berücksichtigt wird.

Mit *open()* erhältst du ein Fileobjekt *f* zurück, das dir den Zugang zur Datei vermittelt. Du kannst nachher mit einer einfachen for-Schleife alle Zeilen durchlaufen. Dabei musst du beachten, dass die einzelnen Zeilen ein Zeilenendzeichen enthalten, das du zuerst mit einer Slice-Operation wegschneiden musst, bevor du das Wort von hinten liest. Zudem solltest du noch alle Zeichen mit *lower()* auf Kleinschreibung konvertieren.

Das Umkehren eines Strings ist in Python etwas trickreich. Die Slice-Operation lässt nämlich auch negative Indizes zu, wobei in diesem Fall die Indizes-Zählung am Ende des Strings beginnt. Wählst du als step-Parameter -1, so wird der String von hinten her durchlaufen.

```
def isPalindrom(a):
    return a == a[::-1]

f = open("worte-1$.txt")

print "Searching for palindroms..."
for word in f:
    word = word[:-1] # remove trailing \n
    word = word.lower() # make lowercase
    if isPalindrom(word):
        print word
f.close()
print "All done"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Du kannst auch mit der Methode *readline()* Zeile um Zeile lesen. Dabei wird bei jedem Aufruf sozusagen ein Zeilenzeiger vorgeschoben. Wenn du am Ende der Datei angekommen bist, liefert die Methode einen leeren String zurück. Du speicherst das Resultat in einer Datei mit dem Namen *palindrom.txt*. Um in die Datei zu schreiben, musst du diese zuerst mit *open()* und dem Parameter "w" (für write) erstellen und mit der Methode *write()* hineinschreiben. Am Schluss darfst du *close()* nicht vergessen, damit sicher alle Zeichen in die Datei geschrieben und die Betriebssystem-Ressourcen wieder frei gegeben werden.

```
def isPalindrom(a):
    return a == a[::-1]

fInp = open("worte-1$.txt")
fOut = open("palindrom.txt", "w")

print "Searching for palindroms..."
while True:
    word = fInp.readline()
    if word == "":
        break
    word = word[:-1] # remove trailing \n
    word = word.lower() # make lowercase
    if isPalindrom(word):
        print word
        fOut.write(word + "\n")
fInp.close()
fOut.close()
print "All done"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Beim Öffnen von Textdateien mit *open(path, mode)* wird mit dem Parameter *mode* der Benützungsmodus angegeben.

Modus	**Beschreibung**	**Bemerkung**
"r" (read)	Nur lesen	Datei muss existieren. Parameter kann weggelassen werden
"w" (write)	Datei erstellen und schreiben	Eine existierende Datei wird zuerst gelöscht
"a" (append)	Am Ende der Datei anhängen	Datei erstellen, falls sie noch nicht existiert
"r+"	Lesen und anfügen	Datei muss existieren

Wenn du alle Zeilen einer Datei gelesen hast und sie nochmals lesen willst, so musst du entweder die Datei schliessen und wieder öffnen, oder einfacher die Methode *seek(0)* des Fileobjekts aufrufen. Du kannst auch den gesamten Inhalt der Textdatei mit

```
text = f.read()
```

in einen String einlesen und dann die Datei schliessen. Mit

```
textList = text.splitlines()
```

erstellst du eine Liste mit den Zeilenstrings (ohne Zeilenendzeichen).

Weitere wichtige Dateioperationen:

import os os.path.isfile(path)	Prüfen, ob Datei existiert
import os os.remove(path)	Eine Datei löschen

OPTIONEN/SPIELDATEN SPEICHERN UND WIEDERHERSTELLEN

Dateien werden oft dazu verwendet, einen Zustand zu speichern (man sagt auch, zu *retten*), damit er bei der nächsten Ausführung des Programms wieder hergestellt werden kann. Dies betrifft beispielsweise Programmeinstellungen (Optionen), die der Benutzer vorgenommen hat, um das Programm seinen Wünschen anzupassen. Vielleicht möchtest du aber auch den gegenwärtigen Spielzustand eines Spiels speichern, damit du später genau in dieser Situation weiter spielen kannst.

Optionen und Zustände lassen sich meist elegant als Schlüssel-Werte-Paare speichern, wobei der Schlüssel (key) ein Bezeichner für den Wert (value) ist. Beispielsweise gibt es für die TigerJython IDE bestimmte Konfigurationswerte (Setup-Parameter):

Key	**Value**
"WindowSize"	[800, 600]
"AutoSave"	True
"Language"	"de"

Wie du im Kapitel 6.3 gelernt hast, kann man solche Key-Value-Paare in einem Python *Dictionary* speichern, das du mit dem Modul *pickle* sehr einfach als (binäre) Datei abspeichern und wiederherstellen kannst. Im folgenden Beispiel speicherst du beim Schliessen des Gamefensters die aktuelle Position und Richtung des Hummers und die Stellung des Simulationszyklus-Reglers.

Beim nächsten Starten werden die abgespeicherten Werte wieder hergestellt.


```
import pickle
import os
from gamegrid import *

class Lobster(Actor):
   def __init__(self):
      Actor.__init__(self, True, "sprites/lobster.gif");

   def act(self):
      self.move()
      if not self.isMoveValid():
          self.turn(90)
          self.move()
          self.turn(90)

makeGameGrid(10, 2, 60, Color.red)
addStatusBar(30)
show()

path = "lobstergame.dat"
simulationPeriod = 500
startLocation = Location(0, 0)
if os.path.isfile(path):
    inp = open(path, "rb")
    dataDict = pickle.load(inp)
    inp.close()
    # Reading old game state
    simulationPeriod = dataDict["SimulationPeriod"]
    loc = dataDict["EndLocation"]
    location = Location(loc[0], loc[1])
    direction = dataDict["EndDirection"]
    setStatusText("Game state restored.")
else:
    location = startLocation
    direction = 0

clark = Lobster()
addActor(clark, startLocation)
clark.setLocation(location)
clark.setDirection(direction)
setSimulationPeriod(simulationPeriod)

while not isDisposed():
    delay(100)
gameData = {"SimulationPeriod": getSimulationPeriod(),
            "EndLocation": [clark.getX(), clark.getY()],
            "EndDirection": clark.getDirection()}
out = open(path, "wb")
pickle.dump(gameData, out)
out.close()
print "Game state saved"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Mit den Methoden **pickle.dump()** wird ein Dictionary in einer Datei gespeichert. Es handelt sich um eine binäre Datei, die du nicht direkt editieren kannst.

AUFGABEN

1. Suche in der Datei *worte-1$.txt* nach Anagrammen (zwei Wörter mit gleichen Buchstaben in verschiedener Reihenfolge, Gross-Kleinschreibung vernachlässigen). Schreibe die gefundenen Anagramme in eine Datei *anagram.txt*

2. Der folgende Text wurde durch Anagrammieren verschlüsselt, d.h. die ursprünglichen Wörter wurden durch solche mit vertauschten Buchstaben ersetzt.

   EIRKG NEDENBE HLOSECHSRSR LAFSSEREIN

   Versuche den Text mit Hilfe der Wortlisten aus den Dateien *worte-1$.txt* und *verben-1$.txt* zu entschlüsseln [**mehr...**].

3. Erstelle selbst einen Geheimtext, der sich mit den Wortlisten eindeutig dechiffrieren lässt.

ZUSATZSTOFF

WORTLISTEN SELBST ERSTELLEN

Wortlisten sind begehrt, da sie in vielen Zusammenhängen verwendet werden, beispielsweise in Korrekturprogrammen, Passwortgeneratoren, Wortspielen (z.B. Scrabble), usw. Wortlisten sind als Textdateien kommerziell erhältlich, aber teuer. Du kannst dir Wortlisten gratis selbst zusammenstellen, indem du die Wörter aus bestehenden Korrekturhilfedateien (Wörterbücher & Sprachpakete) von Gratisprogrammen herausfilterst, beispielsweise aus einem Mailprogramm (Thunderbird). Für eine Wortliste in Deutsch gehst du auf die Website **https://www.j3e.de/ispell/igerman98** und lädst eine Datei mit dem Namen *igerman98-xxx.tar.gz* herunter (xxx kann irgend etwas sein). Nachdem du sie ausgepackt hast, findest du einige Wortlisten als Textdateien. Kopiere diese in ein Verzeichnis und öffne sie mit einem Texteditor. Wie du siehst, sind die Wörter zwar enthalten, müssen aber noch etwas aufgearbeitet werden. Dazu eignet sich ein kleines Pythonprogramm hervorragend.

Für die Konvertierung der Wortlisten aus *igerman98* ist folgende Konvertierung angebracht:

- Umlaute sind mit nachfolgendem Anführungszeichen gekennzeichnet (also a" für ä). Der Buchstabe wird durch den Umlaut ersetzt und das Anführungszeichen eliminiert
- Das Scharf-s wird sS geschrieben. Der zweite S wird in s umgewandelt (schweizerische Schreibweise)
- qq-Paare werden eliminiert

```
def toUmlaut(c):
    if c == "A":
        return "Ä"
    if c == "O":
        return "Ö"
    if c == "U":
        return "Ü"
    if c == "a":
        return "ä"
    if c == "o":
```

```
        return "ö"
    if c == "u":
        return "ü"

def convert(infile, outfile):
    inFile = open(infile, "r")
    outFile = open(outfile, "w")
    for line in inFile:
        # Cut at trailing /
        index = line.find("/")
        if index != -1:  # found
            line = line[0:index + 1]
        # Insert umlaute
        line1 = ""
        i = 0
        while i < len(line) - 1: # don't include trailing \n
            pair = line[i : i + 2]
            if pair[1] == "\"":  # indicates Umlaut
                line1 = line1 + toUmlaut(pair[0])
                i += 1
            elif pair == "sS":  # indicates sz
                line1 = line1 + "ss"
                i += 1
            elif  pair == "qq":
                i += 2 # skip "qq"
            else:
                line1 = line1 + pair[0]
            i += 1
        outFile.write(line1 + "\n")
    inFile.close()
    outFile.close()

convert("verben.txt", "verben-1$.txt")
print "All done"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die von Thunderbird heruntergeladenen Wörterbücher besitzen die Dateierweiterung *.xpi*. Du kannst sie durch *.zip* ersetzen und die Datei wie ein ZIP-Archiv öffnen.

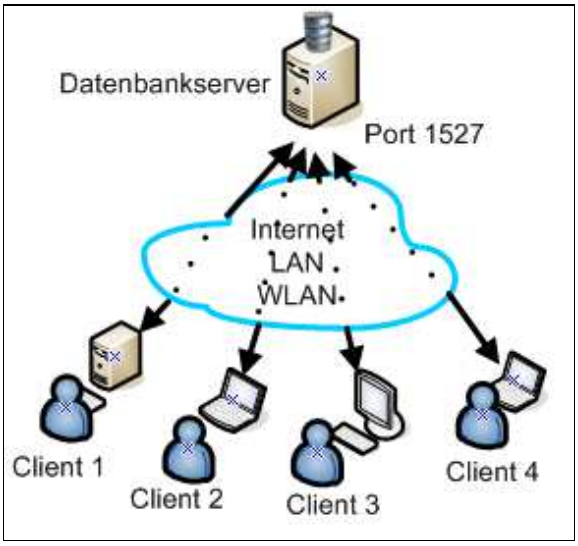
9.2 ONLINE-DATENBANKEN

EINFÜHRUNG

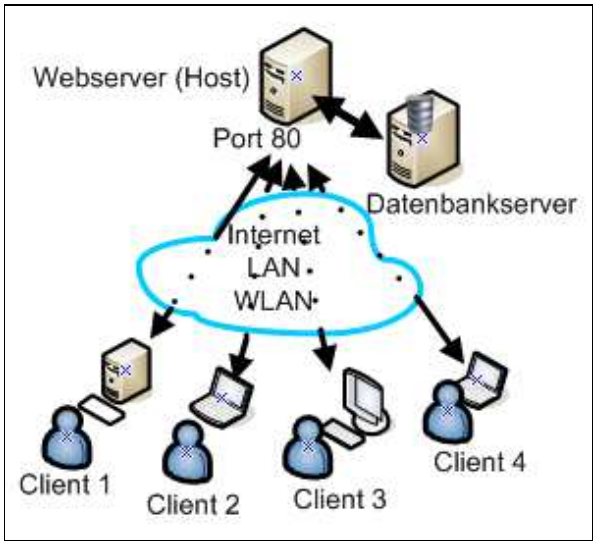
In der heutigen Welt spielen Datenbanken eine ausserordentlich grosse Rolle. Mit ihnen legt man Information strukturiert ab und kann sie mit Such- und Verknüpfungskriterien leicht wieder auffinden. Es ist davon auszugehen, dass Personendaten eines Normbürgers in mehreren hundert Datenbanken aufzufinden sind. Wegen der Vernetzung durch das Internet und der grossen Verbreitung der sozialen Netzwerke nimmt die gespeicherte Datenmenge riesige Ausmasse an. Es ist deshalb wichtig, dass du lernst, wie Daten in einer Datenbank abgelegt und verwaltet werden. Damit kannst du auch die Risiken bei der Verwendung von datenbankbasierten Systemen besser einschätzen.

In den meisten computergestützten Datenbanken werden die Daten in Form von **Tabellen** abgespeichert, die in einer gegenseitigen Beziehung stehen. Sie heissen darum **Relationale Datenbanken**. Um effizient mit den Tabellen umzugehen, muss ein Programmpaket zur Verfügung stehen, das sich **relationales Datenbank-Management-System (RDBMS)** nennt. Man versteht daher unter einer Datenbank nicht nur die Datensammlung selbst, sondern auch die dazugehörenden Verwaltungswerkzeuge.

Einfache Datenbanken können sich lokal auf einem PC befinden und von einer einzelnen Person bedient werden, beispielsweise für die Verwaltung von Musik-CDs oder Büchern. Meist sind Datenbanken aber über das Internet für viele Benutzer gleichzeitig zugänglich (**Online-Datenbanken**). Dabei handelt es sich um ein Client-Server-System mit einem Datenbankserver (auch Host genannt) und mehreren verteilten Clients. Die Datenkommunikation erfolgt wie bei Webservern mit dem TCP-Protokoll über einen TCP-Port (z.B. für MySQL 3306 oder für Derby 1527). Für den Datenaustausch läuft auf dem Client ein spezifisches Anwendungsprogramm, das in irgend einer höheren Programmiersprache geschrieben ist, z.B. in Python.



Direkte Verbindung



Indirekte Verbindung

Vielfach wird der Client nicht direkt, sondern indirekt über einen Webserver mit dem Datenbankserver verbunden. Auf dem Client läuft dann lediglich einer der bekannten Webbrowser. Das spezifische Programm für den Datenaustausch mit dem Datenbankserver befindet sich in diesem Fall auf dem Webserver und muss ebenfalls in einer dafür geeigneten Programmiersprache geschrieben sein (häufig PHP). In beiden Fällen sind die Verfahren ähnlich und setzen gute Programmierkenntnisse im Zusammenhang mit Datenbanken voraus, die du hier erarbeiten wirst [**mehr...**].

DEIN EIGENER DATENBANKSERVER

Der Zugang zu vorhandenen Datenbankservern über das Internet ist starken Sicherheitsbeschränkungen unterworfen, da man vermeiden will, dass unerlaubt Daten gespeichert oder verändert werden. Aus diesen Gründen installierst du auf deinem PC einen eigenen Datenbankserver, den du entweder vom PC selbst, aber auch von anderen Clients innerhalb eines LAN/WLAN verwenden kannst. Gewöhnlich muss man sich für den Zugriff auf eine Datenbank mit einem Usernamen/Passwort authentifizieren [**mehr...**].

Im Folgendem verwendest du als Datenbankserver **Derby**, ein kostenfreies Produkt der Apache-Organisation, die auch den weltberühmten Apache-Webserver entwickelt. (Du könntest aber auch irgend eine andern Datenbank-Software einsetzen, beispielsweise MySQL.) Für die Installation von Derby gehst du gemäss folgender Anleitung vor:

1. Lade von **hier** die Datei tjderby.zip herunter
2. Packe sie aus und kopiere die Dateien in das Unterverzeichnis *Lib* des Verzeichnisses, in dem sich tigerjython2.jar befindet.
3. Gehe mit einer Command-Shell in das Verzeichnis *Lib* und starte den Server mit
   `java -jar derbynet.jar start`
   Achtung! Du musst Administratorberechtigungen haben [**mehr...**].

Du findest im Verzeichnis *Lib* die Dateien *startderby.bat* (für Windows) und *startderby* (für Linux/Mac), mit denen du den Server starten kannst. Erstelle einen Link auf diese Dateien, damit ein Mausklick genügt [**mehr...**].

Der Datenbankserver kann mehrere verschiedene Datenbanken (databases) gleichzeitig verwalten. Der Zugang erfolgt über den Datenbanknamen und einem Username/Passwort-Paar. Zur Erstellung und Verwendung der Datenbank musst du dich mit *getDerbyConnection()* mit dem Server verbinden. Ist die Datenbank noch nicht vorhanden, so wird sie dabei erstellt.

Da du nachfolgend ein Reservationssystem für den Konzertsaal im frei erfundenen Konzertlokal Casino erstellen wirst, wählst du als Datenbanknamen *casino* und die Username/Passwort-Kombination *admin/solar*.

Nach der erfolgreichen Verbindungsaufnahme kriegst du von *getDerbyConnection()* ein **Connection-Objekt** *con* zurück, mit dem du durch Aufruf der Methode *cursor()* einen Zugriffsschlüssel *cursor* zur Datenbank anforderst. Nun stehen dir alle Türen zur Datenbank *casino* offen.

Die Datenbank kannst du als ein Objekt auffassen, dem du Befehle senden kannst, die es entsprechend ausführt. Die Befehle werden in einer einigermassen standardisierten **Datenbanksprache SQL** (Structured Query Language) abgefasst, die sich stark an die englische Umgangssprache anlehnt, damit möglichst jedermann damit umgehen kann. Einen SQL-Befehl packst du in einen String und lässt in dann mit der Methode *execute(SQL)* ausführen. Zur besseren Unterscheidung schreibst du SQL-Sprachelemente in Grossbuchstaben. Sie funktionieren aber auch mit Kleinschreibung oder einem Gemisch von Gross- und Kleinschreibung.

Als erstes erstellst du mit dem SQL-Befehl *CREATE TABLE* eine Datenbank-Tabelle. Wie üblich besteht diese aus Zeilen (rows) und Spalten (columns). Die Spalten, auch **Felder** genannt, legen fest, welche Informationen du in der Tabelle speichern willst. Du musst dazu den Feldern **einen Namen und einen Datentyp** geben. Die wichtigsten Typen entnimmst du der folgenden Tabelle:

SQL Datentyp (für Derby)	Beschreibung
VARCHAR	String, angepasste Länge (<=255)
CHAR(M)	String, M Zeichen, fixe Länge M <= 255
INTEGER	Ganzzahl (4 bytes integer)
FLOAT	Dezimalzahl (64 bit double precision)
DATE	Datum (java.sql.date)

Um die Sitzplatz-Reservationen im Casinosaal für ein bestimmtes Konzert zu verwalten, fügst du dem Tabellennamen *res* noch das Datum im Format yyyymmdd an (y: year, m: month, d: day) hinzu. Als Felder wählst du die Sitznummer *seat*, *booked* mit dem Buchstaben N oder Y und eine Kundennummer *cust*, über die der Sitzplatz-Besteller identifiziert wird.

```
#Db2a.py

from dbapi import *

username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
SQL = "CREATE TABLE res_20140115 (seat INTEGER, booked CHAR(1), cust INTEGER)"
cursor.execute(SQL)
con.commit()
cursor.close()
con.close()
print "Table created"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Läuft das Programm problemlos durch, so hast du auch gleich eine Bestätigung, dass du den Datenbankserver erfolgreich eingerichtet hast. Kriegst du eine Fehlermeldung, so lies dieses Kapitel nach einmal sorgfältig durch und versuche jeden Schritt zu wiederholen.

Es ist für Datenbanken typisch, dass sich die SQL-Befehle erst dann in der Datenbank auswirken, nachdem **commit()** aufgerufen wird. Damit wird sicher gestellt, dass mehrstufige Datenbank-Transaktion als eine Einheit betrachtet werden, die immer als Ganzes oder unter Abgabe einer Fehlermeldung gar nicht ausgeführt wird. Zuletzt müssen alle Ressourcen mit *close()* wieder freigegeben werden. Du solltest dir das "Aufräumen" mit

```
con.commit()
cursor.close()
con.close()
```

gut merken, denn es darf im Folgenden nie fehlen.

Falls du das Programm ausführst, wenn die Tabelle bereits existiert, so bricht das Programm mit einer Fehlermeldung ab. Das Unangenehme daran ist, das im Folgenden alle Datenbankverbindungen fehlschlagen, bis TigerJython neu gestartet wird.

FEHLER MIT TRY-EXCEPT ABFANGEN

Ist die Tabelle, die du erzeugen willst, bereits vorhanden, so führt der Aufruf von *execute()* zu einem Programmabbruch. Man sagt auch, dass **das Programm eine Exception wirft**. Nachfolgende Teile des Programms werden dann nicht mehr zur Ausführung gelangen, insbesondere auch nicht der wichtige Aufräumungsteil. Dadurch werden bestimmte Ressourcen nicht mehr freigegeben, was andere Programme oder gar den ganzen PC blockieren kann. Es ist darum sehr wichtig, solche Fehler **abzufangen**.

In Python ist der Abfang einer geworfenen Exception sehr einfach. Man setzt den Anweisungsteil, der eine Exception werden kann, in einen try-except-Block. Tritt eine Exception auf, so verzweigt das Programm sofort in den except-Teil. Nach dessen Abarbeitung wird das Programm mit der Anweisung unmittelbar nach dem try-except-Block weiterfahren. Fakultativ kann auch noch ein else-Teil angegeben werden, der dann ausgeführt wird, wenn keine Exception auftritt.

Du schreibst daher in Zukunft deine Programm korrekt mit einem Abfang von eventuellen Exception bei der Ausführung von SQL-Befehlen.

```
from dbapi import *

username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
try:
   SQL = "CREATE TABLE res_20140115 (seat INTEGER, booked CHAR(1), cust INTEGER)"
   cursor.execute(SQL)
except Exception, e:
   print "SQL executing failed.", e
else:
   print "Table created"
con.commit()
cursor.close()
con.close()
```

MEMO

Kritische Programmteile sollten in einen try-except-Block gesetzt werden, damit bei einem Programmabbruch die Aufräumarbeiten trotzdem noch ausgeführt werden. Der except-Befehl kann mit dem Parameter e wichtige Informationen über die Art des Fehlers abliefern.

DATEN IN DIE TABELLE EINFÜGEN

In der nächsten Phase willst du die Tabelle mit Initialisierungsdaten füllen, also alle Sitze als frei kennzeichnen und die Kundennummer 0 eintragen. Dazu verwendest du den SQL-INSERT Befehl, beispielsweise für den Sitz mit der Nummer 1:

```
INSERT INTO res_20140115 VALUES (1, 'N', 0)
```

Mit jedem INSERT-Befehl fügst du eine einzelne Zeile in die Tabelle ein. Eine einzelne Zeile der Tabelle nennt man auch **Datensatz** oder **record**.

```
#Db2c.py

from dbapi import *
```

```
table = "res_20140115"
username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"
con = getDerbyConnection(serverURL, dbname, username, password)

cursor = con.cursor()
try:
    for seatNb in range(1, 31):
        SQL = "INSERT INTO " + table + " VALUES (" + str(seatNb) + ",'N',0)"
        cursor.execute(SQL)
except Exception, e:
    print "SQL executing failed. ", e
else:
    print "Table initialized"
con.commit()
cursor.close()
con.close()
```

MEMO

Führst du das Programm zweimal aus, so werden die Datensätze zweimal eingefügt.

DATEN AUS EINER TABELLE AUSLESEN

Du bist jetzt natürlich gespannt, ob sich die Daten tatsächlich in der Tabelle befinden. Heutige Softwaresysteme sind allerdings so stabil, dass du davon ausgehen kannst, dass ohne eine Fehlermeldung die Datenbank-Transaktion tatsächlich erfolgreich war. Zum Auslesen einer Tabelle verwendest du den wohl berühmtesten SQL -Befehl

```
SELECT * FROM res_20140115
```

Mit dem Stern (*Wildcard*) verlangst du, dass alle Datensätze ausgelesen werden. Nach der Ausführung mit *execute()*, kannst du die erhaltenen Records mit der Cursormethode *fetchall()* herausholen. Dabei wird eine Liste zurückgeben, in der die einzelnen Records als *Tuples* (eine unveränderliche Liste) enthalten sind. Darin kannst du die einzelnen Felder mit Indizes auslesen.

Wichtig ist oft die totale Anzahl der Datensätze, die der *SELECT*-Befehl geliefert hat. Du kriegst die Anzahl Records aus der Variablen *rowcount*.

```
#Db2d.py

from dbapi import *

table = "res_20140115"
username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
try:
    SQL = "SELECT * FROM " + table
    cursor.execute(SQL)
except Exception, e:
    print "SQL execution failed.", e
else:
    nbRecord = cursor.rowcount
    print "Number of records:", nbRecord
```

```
    result = cursor.fetchall() # list of tuples
    for record in result:
        print "seatNb:", record[0], " booked:", record[1]," cust:", record[2]
con.commit()
cursor.close()
con.close()
```

MEMO

Neben *fetchall()* stehen dir für das Auslesen der Daten auch *fetchmany(n)* und *fetchone()* zur Verfügung. Bei jedem Aufruf einer der fetch-Methoden wird der Cursor sozusagen wie ein Zeiger um die Anzahl zurückgegebener Records vorwärts geschoben (daher der Name *Cursor*) und der nächste Aufruf von *fetchall()*, *fetchmany()* oder *fetchone()* liefert die Records von der neuen Stelle an. Ist der Cursor am Ende der Tabelle angelangt, so liefern diese Methoden den Wert *None* zurück.

TABELLE LÖSCHEN

Du beendest die Übung, indem du die eben erzeugte Tabelle wieder löschst. In der Datenbanksprache nennt man dies eine **Drop**-Operation (Fallenlassen). Der entsprechende SQL-Befehl lautet:

```
DROP TABLE res_20140115
```

Nachher hat deine Datenbank *casino* keine benutzerspezifischen Tabellen mehr.

```
from dbapi import *

table = "res_20140115"
username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
try:
    SQL = "DROP TABLE " + table
except Exception, e:
    print "SQL executing failed. ", e
else:
    print "Table removed"
con.commit()
cursor.close()
con.close()
```

MEMO

Nach dem Löschen der Tabelle führen deine Programme zum Auslesen der Daten zu einer Fehlermeldung. Du müsstest sie also für diesen Spezialfall noch anpassen.

Die Datenbank mit dem Namen *pythondb* bleibt auch nach dem Löschen der Tabelle erhalten. Es handelt sich um mehrere Dateien im Unterverzeichnis *pythondb* des Verzeichnisses, in das du die Derby-Software kopiert hast (hier also um *Lib/pythondb*). Willst du alle Spuren der Übung entfernen, so kannst du das Verzeichnis *pythondb* löschen.

AUFGABEN

1. Erstelle ein neues Reservationssysten, in dem du Platzreservationen vornehmen und annullieren kannst. Starte den Datenbankserver *Derby* und führe die Programme Db2a.py und Db2c.py aus. Dabei werden eine neue Tabelle erstellt und 30 Datensätze mit leeren Plätzen hinzugefügt.

   Mit folgendem aus Db2d.py vereinfachten Programm kannst du alle Datensätze deiner Tabelle anzeigen lassen.

```
from dbapi import *

username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
table = "res_20140915"

def showAll():
    SQL = "SELECT * FROM " + table
    cursor.execute(SQL)
    con.commit()
    result = cursor.fetchall()
    for record in result:
        print "seatNb:", record[0]," booked:",record[1],"customer:",record[2]

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
showAll()
cursor.close()
con.close()
```

2. Du möchtest für den Benutzer 33 den Sitz 6 reservieren. Dies erreichst du mit einer Aktualisierungsabfrage, die du im oben stehenden Programm vor Aufruf von *showAll()* einfügst.

```
SQL = "UPDATE " + table + " SET booked='Y', cust=33 WHERE seat=6"
cursor.execute(SQL)
```

   Reserviere für den gleichen Benutzer noch den Sitz 5 , für den Benutzer 34 die Sitze 10, 11 und 12 und für den Benutzer 35 die Plätze 17 und 18.

3. Mit der SQL-Abfrage

```
SQL = "SELECT * FROM " + table + " WHERE booked='Y'"
cursor.execute(SQL)
```

   kannst du alle reservierten Plätze anzeigen.
   a. Ändere die Funktion *showAll()* so, dass nur alle leeren Plätze angezeigt werden.
   b. Es sollen nur alle Reservationen des Benutzers 34 angezeigt werden.

4. Der Benutzer 34 annulliert alle Reservationen. Führe eine entsprechende Aktualisierungsabfrage aus.

5. In Folge einer Saal-Renovation stehen die Plätze 24 - 30 nicht mehr zur Verfügung. Lösche diese Datensätze aus der Tabelle.

   Dazu kannst du eine Löschabfrage nach folgendem Muster benutzen:

```
SQL = "DELETE * FROM " + table + " WHERE ..."
cursor.execute(SQL)
```

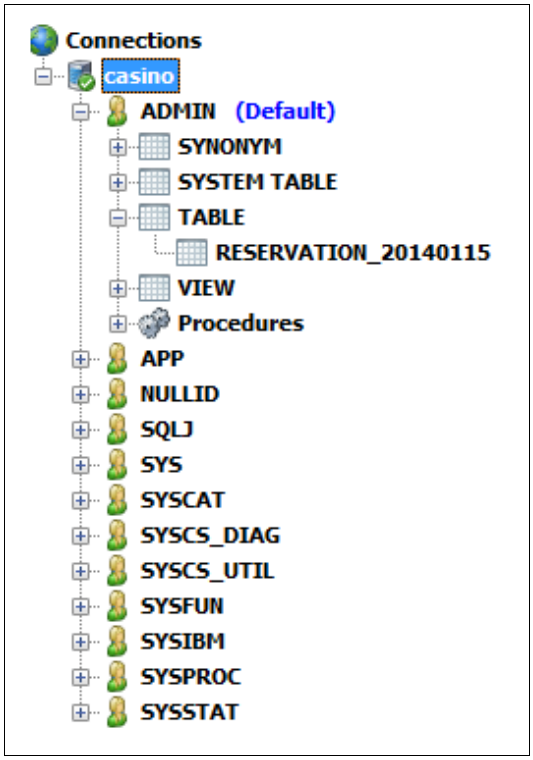
PROGRAMME ZUR DATENBANK-VERWALTUNG

Als Datenbank-Administrator möchtest du deine Datenbanken verwalten, ohne dabei selbst programmieren zu müssen. Dazu stehen dir verschiedene Verwaltungstools zur Verfügung, die du auf dem Internet findest. Oft sind diese allerdings nicht gratis. Man unterscheidet dabei zwischen Kommandozeilen-Tools und solchen mit einer grafischen Benutzeroberfläche. Während Profis sich oft in einer Command-Shell wohl fühlen, bevorzugen Gelegenheitsnutzer GUI-Programme.

Ein bekanntes Administrator-Tool mit einer grafischen Benutzeroberfläche ist **DBVisualizer**, von dem du eine Gratisversion downloaden kannst [**mehr...**]. Du kannst damit SQL-Befehle direkt ausführen und ihre Wirkung beobachten. Für die Installation gehst du wie folgt vor:

1. Hole dir von **http://www.dbvis.com/download** die für deine Plattform angepasste Distribution **als setup installer**.
2. Führe die heruntergeladene Datei aus. Wähle die Standardoptionen.
3. Vergewissere dich, dass du den Derby-Datenbankserver gestartet hast und eine Datenbank *casino* mit der Tabelle *res_20140115* erzeugt und mit Datensätzen initialisiert hast.
4. Starte DBVisalizer und wähle *Tools* | *Connection Wizzard*. Im Dialog wählst du irgend einen Connection Alias, z.B. *Casino*. Im Dialog Select Database Driver wählst du *JavaDB/Derby Server*.
5. Im nächsten Dialog gibst du Userid *admin*, Password *sonar* und Database *casino* ein und klickst auf c*asino.*

   Du siehst nun im Navigator-Fenster das folgende Bild und kannst durch Doppelklicken auf die Tabelle RES_2014015 ein Fenster öffnen. Ein Klick auf den Reiter *Data* zeigt den Inhalt der Tabelle.



9.3 RESERVATIONSSYSTEM

EINFÜHRUNG

Die Reservation von Sitzplätzen in Flugzeugen und Konzertsälen erfolgt heute fast ausnahmslos über Online-Reservationssysteme. Die Informationen werden dabei mit Online-Datenbanken verwaltet. Für die Entwicklung sind Datenbank-Spezialisten mit guten Programmierkenntnissen verantwortlich. Nach dem Studium dieses Kapitels kannst du dich bereits zu ihnen zählen.

PROGRAMMIERKONZEPTE: *Mehrbenutzersystem, Zugriffskonflikt, Sporadischer Fehler*

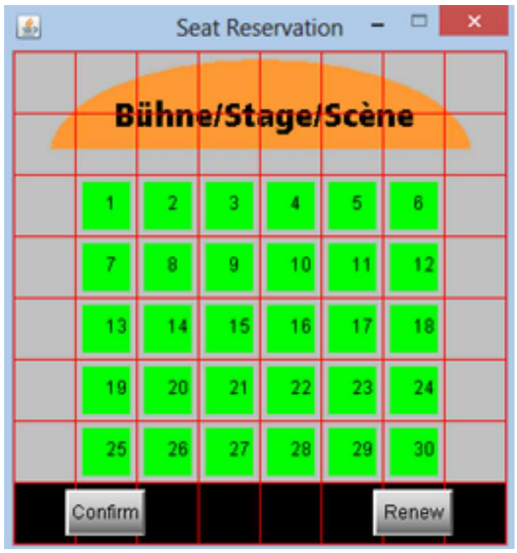
RESERVATIONSSYSTEM TEIL 1: BENUTZERINTERFACE

Das Reservationssystem präsentiert sich dem Benutzer mit einer attraktiven Benutzeroberfläche. Diese kann mit einem eigenständigen Programm oder als Webseite erzeugt werden. Bei Sitzplatz-Reservationssystemen für Konzertsäle und Flugzeuge wird üblicherweise der Grundriss des Raumes bzw. des Flugzeuges dargestellt, auf dem die Sitze eingezeichnet sind. Bereits reservierte Sitze werden mit einer besonderen Farbe gekennzeichnet, beispielsweise freie grün und bereits reservierte rot.

Der Benutzer wählt eine Sitzplatzoption durch einen Mausklick auf einen freien Sitz. Dieser wechselt dabei seine Farbe, er wird beispielsweise gelb. Diese Wahl wird noch nicht zum Datenbankserver übertragen, da der Benutzer oft mehrere zusammengehörende Sitze gleichzeitig reservieren möchte. Erst beim Klicken auf einen Confirm-Button, wird der Reservationsprozess durchgeführt.

Für die Implementierung des graphischen Benutzerinterfaces ist die Gamelibrary *JGameGrid* hervorragend geeignet, da sich die Sitze meist gitterartig anordnen lassen. Für die Speicherung des aktuellen Zustands (frei, option, reserviert) können die Sprite-Identifier (0, 1, 2) verwendet werden.

In deinem Beispiel modellierst du einen kleinen Konzert- oder Theatersaal mit nur 30 Plätzen, die von 1 bis 30 nummeriert sind. Du bettest sie in das nebenstehend gezeigte Gitter ein.



Für die beiden Buttons verwendest du die Klasse *GGButton*. Um eine Notifikation des Buttonklicks zu erhalten, musst du eine eigene Buttonklasse *MyButton* definieren, die sowohl von *GGButton* wie von *GGButtonListener* abgeleitet ist. In der Methode *buttonPressed(),* die aufgerufen wird, wenn ein Button gedrückt wird, kannst du mit dem Parameter *button* herausfinden, um welchen Button es sind handelt.

Mit diesem kurzen Programm hast du bereits ein voll funktionsfähiges Benutzerinterface realisiert. Es fehlt natürlich noch die Anbindung an die Datenbank [**mehr...**] .

```
from gamegrid import *

def toLoc(seat):
```

```
    i = ((seat - 1) % 6) + 1
    k = ((seat - 1) // 6) + 2
    return Location(i, k)

def toSeatNb(loc):
    if loc.x < 1 or loc.x > 6 or loc.y < 2 or loc.y > 6:
       return None
    i = loc.x - 1
    k = loc.y - 2
    seatNb = k * 6 + i + 1
    return seatNb

class MyButton(GGButton, GGButtonListener):
    def __init__(self, imagePath):
        GGButton.__init__(self, imagePath)
        self.addButtonListener(self)

    def buttonClicked(self, button):
        if button == confirmBtn:
            confirm()
        if button == renewBtn:
            renew()

def renew():
    setStatusText("View refreshed")

def confirm():
    for seatNb in range(1, 31):
       if seats[seatNb - 1].getIdVisible() ==  1:
           seats[seatNb - 1].show(2)
           refresh()
    setStatusText("Reservation successful")

def pressCallback(e):
    loc = toLocation(e.getX(), e.getY())
    seatNb = toSeatNb(loc)
    if seatNb == None:
        return
    seatActor = seats[seatNb - 1]
    if seatActor.getIdVisible() == 0:  # free
        seatActor.show(1) # option
        refresh()
    elif seatActor.getIdVisible() == 1:  # option
       seatActor.show(0) # free
       refresh()


makeGameGrid(8, 8, 40, None, "sprites/stage.gif", False,
             mousePressed = pressCallback)
addStatusBar(30)
setTitle("Seat Reservation")
setStatusText("Please select free seats and press 'Confirm'")
confirmBtn = MyButton("sprites/btn_confirm.gif")
renewBtn = MyButton("sprites/btn_renew.gif")
addActor(confirmBtn, Location(1, 7))
addActor(renewBtn, Location(6, 7))
seats = []
for seatNb in range(1, 31):
    seatLoc = toLoc(seatNb)
    seatActor = Actor("sprites/seat.gif", 3)
    seats.append(seatActor)
    addActor(seatActor, seatLoc)
    addActor(TextActor(str(seatNb)), seatLoc)
show()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Umrechnung von Sitznummern auf Locations und umgekehrt wird in zwei Transformationsfunktionen erledigt. Solche Mappingfunktionen werden oft mit **to..** eingeleitet.

RESERVATIONSSYSTEM TEIL 2: ANBINDUNG AN DIE DATENBANK

Online-Datenbanken sind Mehrbenutzer-Systeme. Weil mehrere Clients die Daten gleichzeitig manipulieren, können je nach Situation schwerwiegende **Zugriffskonflikte** auftreten. Es liegt in der Natur solcher Probleme, dass sie nur **sporadisch** auftreten und darum schwierig zu meistern sind. Das folgende Szenario beschreibt einen typischen Konfliktfall:

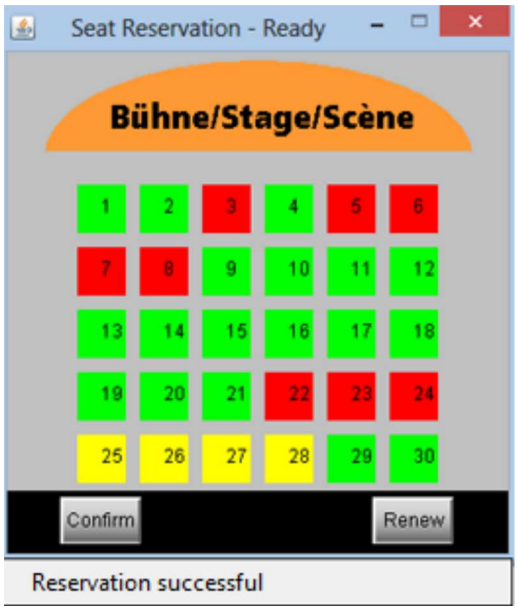
Auf einem Reservationssystem nehmen Kunde A und Kunde B gleichzeitig eine Reservation vor. A und B fragen zu Beginn kurz hintereinander den aktuellen Belegungsplan von der Datenbank ab. Beide erhalten also die gleichen Informationen über die Sitzbelegung (grün = freie, rot = reservierte Plätze)

A und B können mit einem Mausklick die gewählten Sitze als Option auswählen. Die optierten Sitze werden gelb gefärbt. Diese Optionen werden aber der Datenbank nicht übermittelt, da der Benutzer sie noch verändern oder mit weiteren Optionen ergänzen will. Erst nach einiger Zeit, die bei A und B unterschiedlich lange dauert, klickt der schneller entschlossene Benutzer A den Confirm-Button, um die Option definitiv zu machen. Die Datenbank setzt die Sitze in den Zustand *reserviert* (booked = 'Y').

Von dieser Veränderung der Datenbank merkt allerdings der Kunde B nichts, da die Datenbank von sich aus keine Rückmeldungen an B machen kann und A natürlich nicht im direkten Kontakt mit B ist. [**mehr...**]. Nach einer gewissen Zeit entscheidet sich auch Kunde B und drückt den Confirm-Button.

Wegen Murphy's Spruch "Wenn etwas schiefgehen kann, wird es auch schief gehen" haben A und B gleiche Sitzplätze ausgewählt. Was geschieht? Werden etwa die Sitze nun an B vergeben oder stürzt das Programm von B sogar ab?

Es gibt eine einfache **Lösung dieses Zugriffskonflikts**: Wenn ein Benutzer seine Option dem Datenbankserver mitteilt, muss kurz zuvor nochmals die Datenbank abgefragt werden, ob die Sitze tatsächlich immer noch frei sind. Wenn nicht, bleibt nichts anderes übrig, als dem Benutzer höflich mitzuteilen, dass die Plätze leider in der Zwischenzeit bereits vergeben wurden [**mehr...**]



Achtung: Die Ausführung dieses Programms setzt voraus, dass du den Datenbankserver gestartet, die Tabelle res_20140115 erzeugt und die Tabelle mit Initialisierungsdaten gefüllt hast (siehe vorhergehendes Kapitel).

```
from dbapi import *
from gamegrid import *
```

```
table = "res_20140115"
username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

def toLoc(seat):
    i = ((seat - 1) % 6) + 1
    k = ((seat - 1) // 6) + 2
    return Location(i, k)

def toSeatNb(loc):
    if loc.x < 1 or loc.x > 6 or loc.y < 2 or loc.y > 6:
       return None
    i = loc.x - 1
    k = loc.y - 2
    seatNb = k * 6 + i + 1
    return seatNb

def pressCallback(e):
    if not isReady:
        return
    loc = toLocation(e.getX(), e.getY())
    seatNb = toSeatNb(loc)
    if seatNb == None:
        return
    seatActor = seats[seatNb - 1]
    if seatActor.getIdVisible() == 0:  # free
        seatActor.show(1) # option
        refresh()
    elif seatActor.getIdVisible() == 1:  # option
       seatActor.show(0) # free
       refresh()

class MyButton(GGButton, GGButtonListener):
    def __init__(self, imagePath):
        GGButton.__init__(self, imagePath)
        self.addButtonListener(self)

    def buttonClicked(self, button):
        if not isReady:
            return
        if button == confirmBtn:
            confirm()
        if button == renewBtn:
            renew()

def renew():
    global isReady
    try:
        SQL = "SELECT * FROM " + table
        cursor.execute(SQL)
        con.commit()
    except:
        setStatusText("Fatal error. Restart and try again.")
        isReady = False
        return

    result = cursor.fetchall()
    for record in result:
        seatNb = record[0]
        isBooked = (record[1] != 'N')
        if isBooked:
            seats[seatNb - 1].show(2)
        else:
            seats[seatNb - 1].show(0)
    refresh()
    setStatusText("View refreshed")
```

```
def confirm():
    global isReady
    try:
        # check if seats is still available
        for seatNb in range(1, 31):
           if seats[seatNb - 1].getIdVisible() == 1:
               SQL = "SELECT * FROM "  + table + "  WHERE seat=" + str(seatNb)
               cursor.execute(SQL)
               result = cursor.fetchall()
               for record in result:
                   if record[1] == 'Y':
                        setStatusText("One of the seats are already taken.")
                        return
        isReserved = False
        for seatNb in range(1, 31):
           if seats[seatNb - 1].getIdVisible() == 1:
               SQL = "UPDATE "  + table + " SET booked='Y' WHERE seat=" + \
                     str(seatNb)
               cursor.execute(SQL)
               isReserved = True
           con.commit()
        renew()
        if isReserved:
            setStatusText("Reservation successful")
        else:
            setStatusText("Nothing to do")
    except Exception, e:
        setStatusText("Fatal error. Restart and try again.")
        isReady = False

isReady = False
makeGameGrid(8, 8, 40, None, "sprites/stage.gif", False,
             mousePressed = pressCallback)
addStatusBar(30)
setTitle("Seat Reservation - Loading...")
confirmBtn = MyButton("sprites/btn_confirm.gif")
renewBtn = MyButton("sprites/btn_renew.gif")
addActor(confirmBtn, Location(1, 7))
addActor(renewBtn, Location(6, 7))
seats = []
for seatNb in range(1, 31):
    seatLoc = toLoc(seatNb)
    seatActor = Actor("sprites/seat.gif", 3)
    seats.append(seatActor)
    addActor(seatActor, seatLoc)
    addActor(TextActor(str(seatNb)), seatLoc)
show()

con = getDerbyConnection(serverURL, dbname, username, password)
if con == None:
    setStatusText("Fatal error. Connection to database failed")
else:
    cursor = con.cursor()
    renew()

    setTitle("Seat Reservation - Ready")
    setStatusText("Select free seats and press 'Confirm'")
    isReady = True
    while not isDisposed():
        delay(100)
    cursor.close()
    con.close()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du lernst in dieser Übung auch, dass es nicht genügt, ein Programm so zu schreiben, das es bei idealen Voraussetzugen und vorsichtig bedient die richtigen Resultate liefert. Vielmehr soll es sich auch unter ungünstigen Voraussetzungen und bei Fehlmanipulationen richtig verhalten. Hat eine Applikation eine seriöse Benutzeroberfläche, so sollte der Benutzer über den aktuellen Zustand informiert werden. Auch sollte er wissen, ob seine Aktionen erfolgreich waren. Manipulationen, die in einem bestimmten Zustand nicht erlaubt sind, müssen deaktiviert sein.

Eine bekannte Möglichkeit, Aktionen zu deaktivieren, ist die Verwendung eines Flags *isReady*, das festlegt, ob Tastatur- und Mauseingaben erlaubt sind. Zu Beginn wird *isReady = False* gesetzt, bis die Verbindung zur Datenbank aufgebaut ist (dies kann bei entfernten Datenbanken eine gewisse Zeit dauern). Fehlfunktionen werden über einen try-except-Block abgefangen und im except-Teil *isReady = False* gesetzt.

Du kannst das Reservationssystem mit mehreren Benutzerfenster testen, indem du auf deinem Computer die TigerJython IDE mehrmals startest und das gleiche Programm ausführst.

AUFGABEN

1. Ergänze das Reservationssystem, dass beim Klicken auf *Confirm* die Kundennummer (als Integer) abgefragt und in der Datenbank gespeichert wird. Zum Einlesen kannst du die Funktion *inputInt(prompt, False)* verwenden, die beim Drücken der Abbrechen-Taste das Programm nicht beendet, sondern *None* zurück gibt.

2. Schreibe ein Administrator-Tool, dass die aktuelle Belegung und eine Liste der reservierten Sitze sowie der Kundennummern anzeigt. Ausserdem soll es möglich sein, eine Reservation durch Klicken auf einen roten Sitzplatz wieder rückgängig zu machen. Zur Anzeige der Liste kann ein Console-Fenster mit *console = Konsole.init()* geöffnet werden, in das du mit *console.println(text)* zeilenweise hineinschreiben kannst (*from ch.aplu.util import Konsole* nötig).

3. Schreibe ein Tool, mit dem du eine Tabelle mit Kundennummern und dem Familien-/Vornamen der Kunden anlegen kannst. Ergänze das Tool aus Aufgabe 2, dass in der angezeigten Liste der reservierten Sitze auch den Kundennamen erscheinen.

SQL-Abfragen

SELECT * [spalte] **FROM** tabelle [**WHERE** bedingung] [**ORDER BY** spalte [asc|desc]]

Die Optionen in den eckigen Klammern sind fakultativ. Einige Beispiele:

SELECT * FROM tab	liefert alle Datensätze der Tabelle *tab*
SELECT name, vorname FROM tab	zeigt nur die Felder *name* und *vorname* an
SELECT * FROM tab ORDER BY name	alle Datensätze aus *tab* geordnet nach *name*
SELECT * FROM tab WHERE anrede = 'Herr' ORDER BY name	filtert alle Datensätze mit Anrede Herr geordnet nach *name*
SELECT * FROM tab WHERE name = 'Meier' and vorname = 'Luka'	sucht Luka Meier (beide Bedingungen müssen erfüllt sein)
SELECT * FROM tab WHERE name = 'Meier' or name = 'Mayer'	eine der beiden Bedingungen muss erfüllt sein
SELECT * FROM tab WHERE name in ('Meier', 'Meyer', 'Müller')	*name* muss unter den aufgeführten Namen sein
SELECT * FROM tab WHERE name LIKE '%haus% '	liefert alle Datensätze in denen im Feld *name* "haus" vorkommt
SELECT * FROM buch WHERE jahr between 1999 and 2014	Zahlen (integer) werden ohne Anführungszeichen angegeben
SELECT count (*) FROM tab	gibt Anzahl Datensätze an
SELECT concat (vorname, ' ', name) as vname FROM tab	verbindet *name* und *vorname* im neuen Feld *vname*
SELECT sum(preis) FROM buch	ermittelt die Summe aller Werte im Feld *preis*

UPDATE tabelle **SET** spalte1 = wert1 , [spalte2 = wert2], [...] [**WHERE** bedingung]

UPDATE tab SET institut = 'PHBern'	ersetzt in alle Datensätzen den Eintrag im Feld *institut* durch "PHBern"
UPDATE tab SET booked='Y', cust=33 WHERE seat=6	aktualisiert mehrere Spalten
UPDATE tab SET anrede = 'Frau' WHERE anrede = 'f'	ersetzt alle *f* im Feld *anrede* durch "Frau"
UPDATE buch SET preis = preis * 1.52	ersetzt alle Werte in Feld *preis* mit 1.52 mal grösseren Werten

DELETE FROM tabelle **[WHERE** bedingung**]**

DELETE FROM tab	löscht alle Datensätze aus der Tabelle *tab*
DELETE FROM tab WHERE name = "Meier"	löscht die Datensätze mit *name* = "Meier"

kapitel zehn

EFFIZIENZ & GRENZEN

Lernziele

- Du kannst an einigen Beispielen zeigen, dass es algorithmisch formulierbare Probleme gibt, die sich mit dem Computer nicht lösen lassen.
- Du weisst, was man unter polynominaler und nicht-polynominaler Zeitkomplexität eines Programms versteht.
- Du kannst das Halteproblem am Beispiel des 3n+1-Algorithmus erklären.
- Du weisst, was man unter der kombinatorischen Explosion versteht.
- Du kannst mit Backtracking einen Graphen durchsuchen.
- Du kennst einige klassische Verschlüsselungsmethoden und kannst diese in einem Programm umsetzen.
- Du kennst den Begriff des endlichen Automaten und weisst, wie man einen einfachen Automaten implementiert.

10.1 KOMPLEXITÄT BEIM SORTIEREN

EINFÜHRUNG

Computer sind weit mehr als Numbercrunchers, also reine Rechenmaschinen zur Zahlenverarbeitung. Vielmehr ist bekannt, dass ein beträchtlicher Teil der Laufzeit aller weltweit in Betrieb stehender Computer für das Sortieren und Suchen von Daten verwendet wird. Es ist darum wichtig, dass ein Programm nicht nur die richtigen Daten liefert, sondern auch optimiert wird. Dies betrifft:

- seine Länge
- seine Struktur und Übersichtlichkeit
- seine Laufzeit
- seinen Speicherbedarf

Grundsätzlich gilt, dass die Optimierung bereits zu Beginn der Problemlösung mit einbezogen werden sollte, denn es ist meist schwierig, ein salopp geschriebenes Programm im Nachhinein zu optimieren.

In diesem Kapitel untersuchst du die Laufzeitoptimierung beim Sortieren von Daten. Dabei wirst du auch Grenzen der Informatik und des Computereinsatzes kennen lernen, denn ein Problem, für das es wohl einen algorithmischen Lösungsweg gibt, der schnellste Computer aber hunderte von Jahren zur Lösung benötigt, gilt als **unlösbares Problem.**

PROGRAMMIERKONZEPTE: *Komplexität, Laufzeit, Ordnung von Algorithmen, Sortierverfahren, Überladen von Operatoren*

SORTIEREN WIE KINDER: CHILDREN SORT

Das Sortieren bzw. Ordnen einer Menge von Objekten, für die es die Vergleichsoperationen *grösser*, *kleiner* und *gleich* gibt [**mehr...**] , ist und bleibt eine Standardaufgabe der Informatik. Obschon du in allen gängigen höheren Programmiersprachen Bibliotheksroutinen findest, mit denen du sortieren kannst, gehören die Konzepte des Sortierens zu deinem Standardwissen, denn es gibt immer wieder Situationen, wo du das Sortieren selbst implementieren oder optimieren musst.

Eine Ansammlung unsortierter Objekte wird als eine Menge bezeichnet. Im Computer werden die Objekte aber in einer eindimensionalen Datenstruktur gespeichert, wozu sich eine Liste besonders gut eignet [**mehr...**].

Im Programm betrachtest du Zwerge als Actors der Gamebibliothek *JGameGrid*. Du kannst sehr einfach ihre Spritebilder in einem Gitter darstellen [**mehr...**]. Die Höhe der Spritebilder (in Pixel) dienen dir als Mass für die Körpergrösse.

Oft werden Algorithmen direkt aus Verfahren übernommen, die man auch im täglichen Leben anwendet. Fragt man Kinder, wie sie eine Menge von Objekten der Grösse nach ordnen, so beschreiben sie das Verfahren oft so: "*Du nimmst dir das kleinste (oder grösste) Objekt und setzt es der Reihe nach hin*". Dieses Lösungsverfahren klingt sehr plausibel, ist aber für den Computer ein Problem, denn er kann das kleineste oder grösste Objekte nicht wie wir Menschen auf einen Blick erfassen. Er muss es in der unsortierten Liste zuerst suchen, indem er der Reihe nach alle Objekte durchläuft und die Objekte miteinander vergleicht. Um das Sortierverfahren, hier **Children Sort** genannt zu implementieren, benötigst du eine Funktion *getSmallest(row)*, die von der übergebenen Liste den kleinsten Zwerg zurückliefert. Dabei gehst du wie folgt vor:

Du speicherst das erste Listenelement in der Variablen *smallest* und durchläufst in einer for-Schleife alle nachfolgenden Elemente. Ist das gerade betrachtete Element kleiner als *smallest*, so ersetzt du *smallest* durch dieses Element.

Beim Chidren Sort verwendest du zwei Listen, eine Liste *startList* mit den gegebenen Objekten und die Liste *targetList*, die vorerst leer ist. Du suchst in *startList* das kleinste Element, nimmst es dort heraus und fügst es hinten in die *targetList* an, bis *startList* leer ist.

```
from gamegrid import *
import random

def bodyHeight(dwarf):
    return dwarf.getImage().getHeight()

def updateGrid():
   removeAllActors()
   for i in range(len(startList)):
       addActor(startList[i], Location(i, 0))
   for i in range(len(targetList)):
       addActor(targetList[i], Location(i, 1))

def getSmallest(li):
    global count
    smallest = li[0]
    for dwarf in li:
        count += 1
        if bodyHeight(dwarf) < bodyHeight(smallest):
            smallest = dwarf
    return smallest

n = 7

makeGameGrid(n, 2, 170, Color.red, False)
setBgColor(Color.white)
show()
startList = []
targetList = []

for i in range(0 , n):
    dwarf = Actor("sprites/dwarf" + str(i) + ".png")
    startList.append(dwarf)
random.shuffle(startList)
updateGrid()
setTitle("Children Sort. Press <SPACE> to sort...")
count = 0
while not isDisposed() and len(startList) > 0:
    c = getKeyCodeWait()
    if c == 32:
        smallest = getSmallest(startList)
        targetList.append(smallest)
        startList.remove(smallest)
        count += 1
        setTitle("Count: " + str(count) + " <SPACE> for next step...")
        updateGrid()
setTitle("Count: " + str(count) + " All done")
```

MEMO

Beim Children Sort brauchst du neben der gegebenen unsortierten Liste der Länge n eine zweite Liste, die schliesslich auch die Länge n hat. Ist n sehr gross, kann dies zu einem Speicherplatzproblem werden. [**mehr...**].

Du kannst dir leicht überlegen, wieviele elementare Schritte zur Lösung nötig sind: Unabhängig davon, wie die Objekte in der vorgegebenen Liste angeordnet sind, musst du sie zur Suche des Minimums zuerst n mal, dann n-1 mal, usw. durchlaufen; dazu kommt jedesmal die Verschiebungsoperation von der Startliste in die Zielliste. Die Anzahl Operationen c ist daher die Summe aller natürlichen Zahlen von 2 bis n + 1, wie du auch mit der Zählvariablen count mitverfolgen kannst. Aus der Summenformel für natürliche Zahlen ergibt sich :

$$c = \frac{(n+1)*(n+2)}{2} - 1 = \frac{n^2}{2} - \frac{3n}{2}$$

Beispielsweise ergeben sich für n = 1000 bereits

$$c = \frac{1000*1000}{2} + \frac{3*1000}{2} = 500000 + 1500 \approx 500000$$

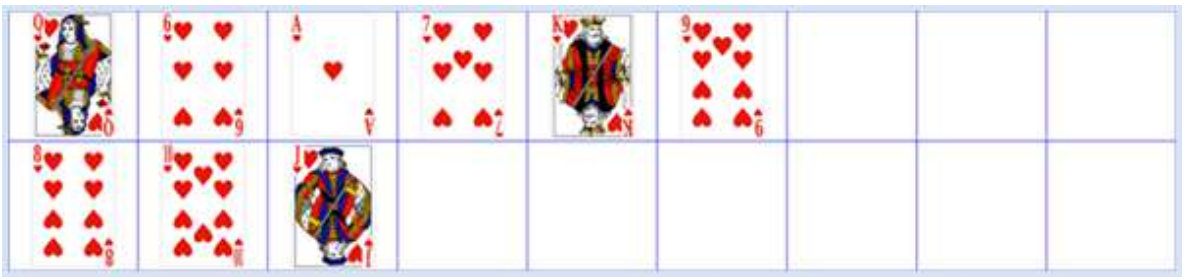
Schritte. Wie du siehst, überwiegt für grosse n das quadratische Glied und man sagt darum:
Die Komplexität des Algorithmus ist von der Ordnung n-Quadrat
und schreibt dafür

Komplexität = O(n^2)

SORTIEREN BEIM KARTENSPIEL: INSERTION SORT

Nimmst du beim Ausspielen von Spielkarten Karte um Karte fächerartig in die Hand, so verwendest du meist intuitiv ein anderes Sortierverfahren: Du fügst jede neu aufgenommene Karte dort in die Hand ein, wo sie gemäss ihrer Wertigkeit hineinpasst. Im deinem Programm, dass die ungeordneten Karten von der Startliste (dem Kartenstapel) in die Zielliste (deine Hand) einfügt, gehst du genau so vor:

Du nimmst Karte um Karte von links nach rechts aus der Startliste und durchläuft die bereits geordnete Zielliste ebenfalls von links nach rechts. Sobald die aufgenommene Karte höhere Wertigkeit als die zuletzt in der Hand betrachtete ist, fügst du sie in die Zielliste ein.



```
from gamegrid import *
import random

def cardValue(card):
    return card.getImage().getHeight()

def updateGrid():
```

```
    removeAllActors()
    for i in range(len(startList)):
        addActor(startList[i], Location(i, 0))
    for i in range(len(targetList)):
        addActor(targetList[i], Location(i, 1))

n = 9

makeGameGrid(n, 2, 130, Color.blue, False)
setBgColor(Color.white)
show()

startList = []
targetList = []

for i in range(0 , 9):
    card = Actor("sprites/" + "hearts" + str(i) + ".png")
    startList.append(card)

random.shuffle(startList)
updateGrid()
setTitle("Insertion Sort. Press <SPACE> to sort...")
count = 0

while not isDisposed() and len(startList) > 0:
    getBg().clear()
    c = getKeyCodeWait()
    if c == 32:
        pick = startList[0] # take first
        startList.remove(pick)
        i = 0
        while i < len(targetList) and cardValue(pick) > cardValue(targetList[i]):
            i += 1
            count += 1
        targetList.insert(i, pick)
        count += 1
        setTitle("Count: " + str(count) + " <SPACE> for next step...")
        updateGrid()
setTitle("Count: " + str(count) + " All done")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Dieses Sortierverfahren heisst **Sortieren durch Einfügen** (**insertion sort**). Die benötigte Anzahl Schritte hängt dabei von der Reihenfolge der Karten im aufgenommen Kartenstapel ab. Am meisten Schritte braucht es, wenn der Kartenstapel zufälligerweise gerade umgekehrt geordnet ist. Man kann sich überlegen oder mit einer Computersimulation herausfinden, dass die Zahl der Schritte im Mittel (für grosse n) mit n^2 / 4 zunimmt, die Komplexität im Mittel also wie beim Chidren Sort ebenfalls $O(n^2)$ ist.

SORTIEREN MIT LUFTBLASEN: BUBBLE SORT

Eine bekannte Art, Objekt in einer Liste zu sortieren, besteht darin, die Liste von links nach rechts mehrmals zu durchlaufen und immer zwei nebeneinander liegende Elemente zu vertauschen, falls sie in falscher Reihenfolge liegen.

Mit diesem Verfahren, bewegt sich zuerst das grösste Element sukzessive von links nach rechts, bis es dort angekommen ist. Im nächsten Durchlauf beginnst du wieder links, gehst aber nur bis zum zweitletzten Element, da sich ja das grösste bereits an der richtigen Stelle befindet. Bei diesem Verfahren ist keine zweite Liste nötig [**mehr...**].

```
from gamegrid import *
import random

def bubbleSize(bubble):
    return bubble.getImage().getHeight()

def updateGrid():
   removeAllActors()
   for i in range(len(li)):
       addActor(li[i], Location(i, 0))

def exchange(i, j):
    temp = li[i]
    li[i] = li[j]
    li[j] = temp

n = 7
li = []

makeGameGrid(n, 1, 150, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    bubble = Actor("sprites/bubble" + str(i) + ".png")
    li.append(bubble)
random.shuffle(li)
updateGrid()
setTitle("Bubble Sort. Press <SPACE> for next step...")
k = n - 1
i = 0
count = 0
while not isDisposed() and k > 0:
    getBg().fillCell(Location(i, 0), makeColor("beige"))
    getBg().fillCell(Location(i + 1, 0), makeColor("beige"))
    refresh()
    c = getKeyCodeWait()
    if c == 32:
        count += 1
        bubble1 = li[i]
        bubble2 = li[i + 1]
        refresh()
        if bubbleSize(bubble1) > bubbleSize(bubble2):
             exchange(i, i + 1)
             setTitle("Last Action: Exchange. Count: " + str(count))
        else:
             setTitle("Last Action: No Exchange. Count: " + str(count))
        getBg().clear()
        updateGrid()
        if i == k - 1:
            k = k - 1
            i = 0
        else:
            i += 1
getBg().clear()
refresh()
setTitle("All done. Count: " + str(count))
```

MEMO

Die grösseren Elemente bewegen sich dabei nach rechts (sozusagen wie Luftblasen in Wasser nach oben). Aus diesem Grund heisst dieser Sortieralgorithmus **Bubble Sort**. Wie du überlegen kannst oder am eingebauten Schrittzähler siehst, ist seine Komplexität unabhängig von der Anordnung der Element in der vorgegebenen Liste wieder von der Ordnung $O(n^2)$.

Zur Bereicherung der Demonstration werden die beiden Zellen, deren Blasen als letztes verglichen wurden, mit der Background-Methode *fillCell()* farbig hinterlegt. Mit *getBg().clear()* kann die Hintergrundfarbe wieder entfernt werden. Der Aufruf von *refresh()* ist nötig, damit das Bild neu auf dem Bildschirm gerendert wird.

MIT BIBLIOTHEKSROUTINEN SORTIEREN: TIMSORT

Da das Sortieren zu den wichtigsten Algorithmen gehört, stellen alle höheren Programmiersprachen eingebaute Bibliotheksfunktionen zum Sortieren zur Verfügung. In Python handelt es sich um die Funktion *sorted(liste, cmp)*, die sogar zu den eingebauten Funktionen gehört, also ohne import verwendet werden kann. Du kannst dir also deinen selbstgeschriebenen Sortieralgorithmus ersparen, dafür musst du aber lernen, wie die Bibliotheksfunktion verwendet wird. Offensichtlich benötigt sie als Parameter die zu sortierende Liste. Du musst ihr aber auch noch mit einem zweiten Parameter die Information mitgeben, nach welchem Ordnungskriterium sie die Objekte sortieren soll.

Das Sortierkriterium legst du in einer Funktion fest, die du hier mit *compare()* bezeichnest. Diese muss als Parameter die zwei Objekte erhalten und drei Werte 1, 0 und -1 zurückgeben, je nachdem ob das erste Objekt grösser, gleich oder kleiner dem zweiten Objekt ist. Der Bibliotheksfunktion *sorted()* übergibst du den frei gewählten Funktionsnamen als zweiten Parameter oder mit dem benannten Parameter *cmp*.

```
from gamegrid import *
import random

def bodyHeight(dwarf):
    return dwarf.getImage().getHeight()
def compare(dwarf1, dwarf2):
    if bodyHeight(dwarf1) < bodyHeight(dwarf2):
        return -1
    elif bodyHeight(dwarf1) > bodyHeight(dwarf2):
        return 1
    else:
        return 0

def updateGrid():
   removeAllActors()
   for i in range(len(li)):
       addActor(li[i], Location(i, 0))
n = 7
li = []
makeGameGrid(n, 1, 170, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    dwarf = Actor("sprites/dwarf" + str(i) + ".png")
    li.append(dwarf)
random.shuffle(li)
updateGrid()
setTitle("Timsort. Press any key to get result...")
getKeyCodeWait()
li = sorted(li, cmp = compare)
updateGrid()
setTitle("All done.")
```

MEMO

Willst du Bibliotheksfunktionen zum Sortieren verwenden, so muss du mit einer Vergleichsfunktion festlegen, wie zwei Elemente auf *grösser, gleich* und *kleiner* verglichen werden [**mehr...**].

Der in Python verwendete Algorithmus wurde erst 2002 von Tim Peters erfunden und heisst darum *Timsort*. Er hat (im Mittel) die Ordnung O(nlog(n)). Es sind also beispielsweise für n = 10^6 nur rund 10^7 Operationen nötig, statt der rund 10^{12} bei einem Sortieralgorithmus mit der Ordnung O(n^2).

AUFGABEN

1. Sortiere die 7 Zwerge mit einem Bubble Sort.

2. Füge das Spritebild *snowwhite.png* von Schneewittchen, das dieselbe Grösse wie der grösste Zwerg besitzt, in den Bubble Sort von Aufgabe 1 hinzu. Zeige, dass die Reihenfolge von Schneewittchen und dem grössten Zwerg immer ihrer Reihenfolge in der Startliste einspricht. (Einen solchen Sortieralgorithmus nennt man **stabil**.)

3. Mit *row = range(n)* und nachfolgendem *random.shuffle(row)* kannst du sehr einfach lange unsortierte Zahlenlisten erzeugen. Messe die Laufzeit für das Sortieren mit dem internen Sortieralgorithmus (*Timsort*) für verschiedene Werte von n und zeige, dass die Komplexität wesentlich besser als O(n2) ist. Anleitung: Um eine Zeitdifferenz zu messen, importierst du das Modul *time* und bildest die Differenz von zwei Aufrufen von *time.clock()*.

ZUSATZSTOFF

ÜBERLADEN DER VERGLEICHSOPERATOREN

Das Vergleichen von zwei Objekten ist eine wichtige Operation. Für Zahlen kannst du dazu die 5 Vergleichsoperationen <, <=, ==, >, = > verwenden. In Python ist es möglich, diese Operatoren auch für irgend einen anderen Datentyp einzusetzen, also beispielsweise für Zwerge. Dadurch gewinnt der Programmcode an Eleganz und Übersichtlichkeit.

Du gehst wie folgt vor:

Definiere in der Klassendefinition deines Datentyps die Methoden *__lt__(), __le__(), __eq__(), __ge__(), __gt__(),* die den booleschen Werte der Vergleichsoperation *less, less-and-equal, equal, greater-and-equal, greater* zurückgeben. Zusätzlich kannst du noch die Methode *__str()__* definieren, die beim Aufruf der *str()* Funktion verwendet wird.

In der Klasse *Zwerge*, die von *Actor* abgeleitet ist, speicherst du als Instanzvariable zusätzlich noch den Namen des Zwergs, den du bei *updateGrid()* als *TextActor* ausschreibst.

```
from gamegrid import *
import random
```

```
class Dwarf(Actor):
    def __init__(self, name, size):
        Actor.__init__(self, "sprites/dwarf" + str(size) + ".png")
        self.name = name
        self.size = size
    def __eq__(self, a):  # ==
        return self.size == a.size
    def __ne__(self, a): # !=
        return self.size != a.size
    def __gt__(self, a): # >
        return self.size > a.size
    def __lt__(self, a): # <
        return self.size < a.size
    def __ge__(self, a): # >=
        return self.size >= a.size
    def __le__(self, a): # <=
        return self.size <= a.size
    def __str__(self):  # str() function
        return self.name

def compare(dwarf1, dwarf2):
    if dwarf1 < dwarf2:
        return -1
    elif dwarf1 > dwarf2:
        return 1
    else:
        return 0

def updateGrid():
   removeAllActors()
   for i in range(len(row)):
       addActor(row[i], Location(i, 0))
       addActor(TextActor(str(row[i])), Location(i, 0))

n = 7
row = []
names = ["Montag", "Dienstag", "Mittwoch", "Donnerstag",
         "Freitag", "Samstag", "Sonntag"]

makeGameGrid(n, 1, 170, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    dwarf = Dwarf(names[i], i)
    row.append(dwarf)
random.shuffle(row)
updateGrid()
setTitle("Press any key to get result...")
getKeyCodeWait()
row = sorted(row, cmp = compare)
updateGrid()
setTitle("All done.")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Verwendung der Vergleichsoperatoren für beliebige Datentypen ist zwar nicht zwingend, aber elegant. Man sagt, dass man die Operatoren dabei **überladet** (operator overloading).

10.2 UNLÖSBARE PROBLEME

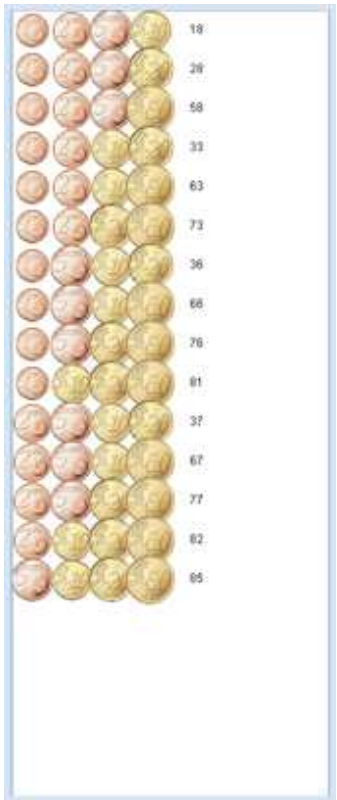
EINFÜHRUNG

Mit gescheiten Computerprogrammen kann man zwar viele und immer mehr Probleme lösen. In diesem Kapitel wirst du aber mit einfach zu formulierenden Fragen konfrontiert, die möglicherweise trotz der rasanten Entwicklung der Computer und enormem wissenschaftlichem Aufwand nie algorithmisch lösbar sein werden.

PROGRAMMIERKONZEPTE: *Unlösbares Problem, Teilsummenproblem, Aufzählungsverfahren, Kombinatorische Explosion, Polynomiale Ordnung, Unentscheidbares Problem*

UNLÖSBARE PROBLEME

Es harren noch einige Probleme der Lösung, die einfach zu formulieren und auch für die Praxis von grosser Bedeutung sind. Eines davon, bekannt unter dem Namen **Teilsummenproblem**, kann auf folgende Problemstellung zurückgeführt werden [**mehr...**]:



Du hast in deinem Geldbeutel eine Anzahl von Münzen und musst damit einen bestimmten Betrag (ohne Rückgeld) bezahlen. Ist dies mit den vorhandenen Münzen möglich und wenn ja, mit welchen Münzen musst du dabei bezahlen?

In deinem ersten Programm lernst du zuerst mit den Münzen umzugehen. Die Namen der Euro-Münzen mit den Werten 1, 2, 5, 10, 20, 50 cents speicherst du in der Liste *coins*. Die Funktion *value()* liefert den Wert einer Münze zurück. Der Geldbeutel wird mit einer Liste (oder einem Tupel) *moneybag*, welche die Namen der vorhandenen Münzen enthält. Die Funktion *getSum(moneybag)* liefert den Wert aller Münzen im Geldbeutel.

Der Geldbeutel soll zuerst alle Münzen genau einmal enthalten und du bildest alle möglichen Münzkombinationen mit 1, 2, 3, 4, 5 und 6 Münzen, die du in einem *JGameGrid*-Fenster darstellst. Dazu machst du in *showMoneybag(moneybag, y)* aus jeder Münze des *moneybags* einen Actor und stellst diese im Spielfenster in der Zeile y dar.

```
from gamegrid import *
import itertools

coins = ["one", "two", "five", "ten", "twenty", "fifty"]

def value(coin):
    if coin == "one":
        return 1
    if coin == "two":
        return 2
    if coin == "five":
        return 5
```

```
    if coin == "ten":
        return 10
    if coin == "twenty":
        return 20
    if coin == "fifty":
        return 50
    return 0

def getSum(moneybag):
    sum = 0
    for coin in moneybag:
        sum += value(coin)
    return sum

def showMoneybag(moneybag, y):
    x = 0
    for coin in moneybag:
        loc = Location(x, y)
        removeActor(getOneActorAt(loc))
        coinActor = Actor("sprites/" + coin + "cent.png")
        addActor(coinActor, loc)
        x += 1
    addActor(TextActor(str(getSum(moneybag))), Location(x, y))

makeGameGrid(8, 20, 40, False)
setBgColor(Color.white)
show()

n = 6
k = 1
while k <= n:
    combinations = list(itertools.combinations(coins, k))
    print type(combinations)
    setTitle("(n, k) = (" + str(n) + ", " + str(k) + ") nb = "
    + str(len(combinations)))
    y = 0
    for moneybag in combinations:
        showMoneybag(moneybag, y)
        y += 1
    getKeyCodeWait()
    removeAllActors()
    k += 1
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

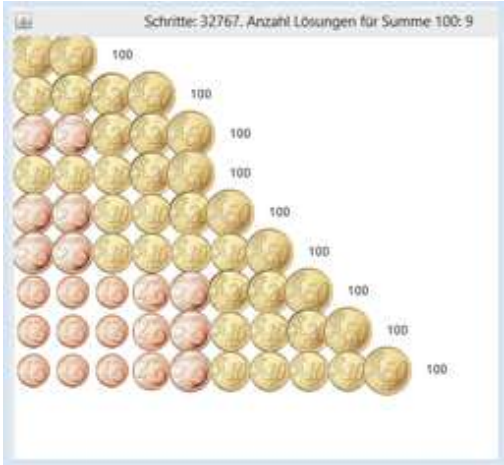
Die Kombinationen von k Elementen, die du aus einer Liste von n Elementen bilden kannst, lassen sich elegant mit der Funktion *combinations()* aus dem Modul *itertools* herausholen. Du musst den Rückgabewert in eine Liste umwandeln, in der sich die gefundenen Kombinationen dann (als Tupels) herausholen lassen.

Wie du siehst, sind die damit erhaltenen Kombinationen so geordnet, wie du es vernünftigerweise auch von Hand machen würdest. Die Anzahl der Kombinationen von n Elementen zur Ordnung k kannst du bekanntlich wie folgt berechnen:

$$c = \binom{n}{k} = \frac{n!}{k!*(n-k)!}$$

wo n! die Fakultät, also das Produkt aller Zahlen von 1 bis n bedeutet. Für unserem Fall mit n = 6 ergeben sich 6, 15, 20, 15, 6, 1, also insgesamt 63 Kombinationen.

Du kannst jetzt das Teilsummenproblem beim Geldbeutel wie folgt lösen: Du bestimmst alle Kombinationen der vorhandenen Münzen und untersuchst sie einzeln, ob ihre Summe den gewünschten Wert ergibt.



Dieses **Aufzählungsverfahren** ist wohl nicht das beste, ist aber sicher korrekt und liefert alle möglichen Lösungen. Für eine Geldbörse mit 3 Eincent, 1 Zweicent, 2 Fünfcent, 4 Zehncent, 2 Zwanzigcent und 3 Fünfzigcent-Münzen, also insgesamt 15 Münzen wäre es bereits schwierig, die Lösungen von Hand zu finden. Du schreibst nur unterschiedliche Münzzusammenstellungen aus, die zusammen 1 Euro ergeben.

```
from gamegrid import *
import itertools

coins = ["one", "one", "one", "two", "five", "five",
         "ten", "ten", "ten", "ten", "twenty", "twenty",
         "fifty", "fifty", "fifty"]

def value(coin):
    if coin == "one":
        return 1
    if coin == "two":
        return 2
    if coin == "five":
        return 5
    if coin == "ten":
        return 10
    if coin == "twenty":
        return 20
    if coin == "fifty":
        return 50
    return 0

def getSum(moneybag):
    sum = 0
    for coin in moneybag:
        sum += value(coin)
    return sum

def showMoneybag(moneybag, y):
    x = 0
    for coin in moneybag:
        loc = Location(x, y)
        removeActor(getOneActorAt(loc))
        coinActor = Actor("sprites/" + coin + "cent.png")
        addActor(coinActor, loc)
        x += 1
    addActor(TextActor(str(getSum(moneybag))), Location(x, y))

makeGameGrid(15, 20, 40, False)
setBgColor(Color.white)
show()

target = 100

k = 1
result = []
count = 0
while k <= len(coins):
```

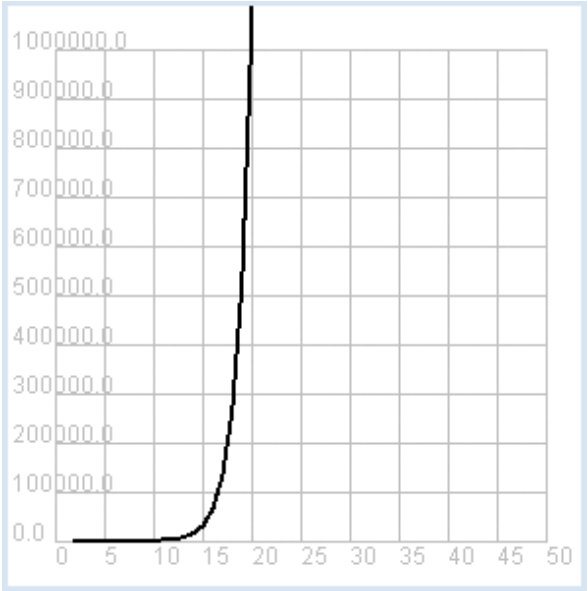
```
    combinations = tuple(itertools.combinations(coins, k))
    nb = len(combinations)
    for moneybag in combinations:
        count += 1
        sum = getSum(moneybag)
        if sum == target:
            if not moneybag in result:
               result.append(moneybag)
    k += 1

y = 0
for moneybag in result:
    showMoneybag(moneybag, y)
    y += 1
setTitle("Schritte: " + str(count) + ". Anzahl Lösungen für Summe "
          + str(target) + ": " + str(len(result)))
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Bereits mit nur 15 Münzen sind 32767 Schritte nötig, um das Teilsummenproblem mit dem Aufzählungsverfahren zu lösen.

Deine Freude an der Computerlösung wird leider getrübt, wenn du versuchst, einen etwas grösseren Geldsack, sagen wir mit 50 oder 100 Münzen, zu verwenden. Zählst du nämlich für einen Geldbeutel mit n Münzen die nötigen Schritte zusammen und trägst sie in einer Grafik auf, so gibt es bei n = 20 eine regelrechte **kombinatorische Explosion** und du stösst an eine Grenze des Machbaren [**mehr...**].




```
from gpanel import *
from math import factorial

z = 100

def nbCombi(n, k):
    return factorial(n) / factorial(k) / factorial(n - k)

makeGPanel(-5, 55, -1e5, 1.1e6)
drawGrid(0, 50, 0, 1e6, "gray")
setColor("black")
lineWidth(2)
for n in range(2, z + 1):
```

```
    sum = 0
    for k in range(1, n):
        sum += nbCombi(n, k)
    print "n =", n, ", nb =", sum
    if n == 2:
        move(n, sum)
    else:
        draw(n, sum)
print "Laufzeit mit 10^9 Operationen pro Sekunde:", sum / 3.142e16, "Jahre"
print "oder:", int(sum / 2e20), "Mal das Alter des Universums"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Das Teilsummenproblem ist bereits bei relativ kleiner Anzahl von Elementen mit dem Aufzählungsverfahren **unlösbar**, obschon das Lösungsverfahren bekannt ist. Es fragt sich daher, ob es **wesentlich bessere Algorithmen** zu seiner Lösung gibt, wenn möglich solche wie beim Sortieren, deren Schrittzahl oder Komplexität eine Potenz von n, also **polynomial** ist. Leider ist es bis heute nicht gelungen, einen solchen Algorithmus für das Teilsummenproblem zu finden, und man vermutet, dass es keinen solchen gibt. Allerdings gibt es auch keinen theoretischen Beweis für diese Vermutung.

Immerhin weiss man heute, dass es eine Vielzahl **ähnlich schwieriger Probleme** gibt, und dass man damit rechnen kann, auf einen Schlag alle diese Probleme mit polynomialer Komplexität zu lösen, wenn man für eines davon eine solche Lösung findet [**mehr...**].

UNENTSCHEIDBARE PROBLEME

Die Grenzen des menschlichen Verstands und der Computertechnik werden noch in einem anderen Zusammenhang als bei der Komplexität sichtbar. Der Mathematiker und Zahlentheoretiker Lothar Collatz hat bestimmte Zahlenfolgen natürlicher Zahlen untersucht und 1939 folgende Frage formuliert:

Gehe von irgend einer Startzahl n aus und bilde die Nachfolgezahlen nach folgender Regel:

- Ist n gerade, so teile n durch 2 (wieder eine natürliche Zahl)
- Ist n ungerade, so bilde die Nachfolgezahl 3n +1 (eine gerade Zahl)

Frage: Erreicht diese Folge mit jeder möglichen Startzahl n immer die Zahl 1?

Collatz und viele andere Zahlentheoretiker und Computerwissenschafter haben sich um eine Lösung bemüht, denn selbst mit den grössten und schnellsten Computern ergeben sich immer Folgen, die bei 1 landen. (Die Folge konvertiert nicht, denn fährt man weiter, so wird endlos die Sequenz 4,2 1 durchlaufen).

Darum liegt die **Vermutung** nahe, dass der folgende Satz gilt:

Die 3n+1-Folge erreicht für alle natürlichen Startzahlen n nach endlich vielen Schritten die Zahl 1.

Mit einem Computerprogramm kannst du für eine beliebig vorgebbare Startzahl die 3n+1-Folge durchlaufen.

```
from gpanel import *

def collatz(n):
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
```

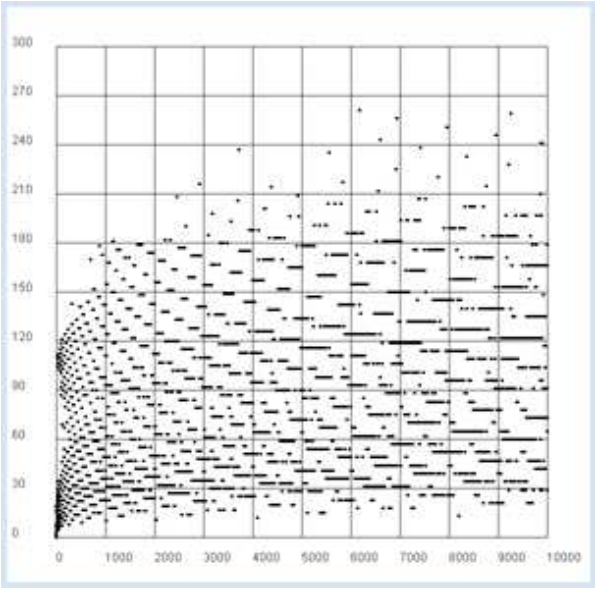
```
            n = 3 * n + 1
        print n,
    print "Result 1"
while True:
    n = inputInt("Enter a start number:")
    collatz(n)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

In Python kannst du sogar mit grossen Startzahlen die 3n+1-Folge durchlaufen und feststellen, dass du immer bei 1 landest. Damit hast du aber natürlich die Vermutung nicht bewiesen.

Interessant und ästhetisch ansprechend ist es, die Länge der 3n+1-Folge in Abhängigkeit von der Startzahl aufzutragen. Diese schwankt nämlich beträchtlich. Dazu entfernst du in der Funktion *collatz()* das Ausschreiben der Folgeglieder und gibst lediglich die Anzahl Schritte zurück.




```
from gpanel import *

def collatz(n):
    nb = 0
    while n != 1:
        nb += 1
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    return nb

z = 10000 # max n
yval = [0] * (z + 1)
for n in range(1, z + 1):
    yval[n] = collatz(n)
ymax = (max(yval) // 100  + 1) * 100

makeGPanel(-0.1 * z, 1.1 * z, -0.1 * ymax, 1.1 * ymax)
title("Collatz Vermutung")
drawGrid(0, z, 0, ymax, "gray")

for x in range(1, z + 1):
    move(x, yval[x])
    fillCircle(z / 200)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Vermutung von Collatz ist ein hartnäckiges Problem. Falls die Vermutung stimmt, so lässt sie sich nicht beweisen, in dem man Computertests mit immer grösseren Startzahlen durchführt. Es könnte allerdings sogar sein, dass die Vermutung zwar richtig ist, aber nie einen Beweis dafür gefunden wird, denn 1931 hat der Mathematiker Kurt Gödel mit dem **Unvollständigkeitssatz** gezeigt, dass es in einer Theorie durchaus richtige Sätze geben kann, deren Korrektheit aber nicht bewiesen werden kann.

Die Vermutung von Collatz lässt sich auch als **Entscheidungsproblem** formulieren:

Stoppt ein Algorithmus, der die Glieder der 3n+1-Folge berechnet und bei 1 anhält, mit Sicherheit für beliebig vorgebbare Anfangswerte?

Man könnte versuchen, diese Frage mit einem Computer zu lösen. Leider könnte auch dies hoffnungslos sein, denn der grosse Mathematiker und Informatiker Alain Turing hat mit dem **Halteproblem** bewiesen, dass es nie einen Algorithmus geben wird, mit dem man für alle Programme entscheiden kann, ob sie anhalten.

Die 3n+1-Vermutung von Collatz könnte also zwar stimmen, aber ein **unentscheidbares Problem** sein.

10.3 BACKTRACKING

EINFÜHRUNG

Bei der Entwicklung von Computergames wird es erst richtig interessant, wenn der Computer selbst zum intelligent handelnden Spielpartner wird. Dazu muss das Programm nicht nur die Spielregeln einhalten, sondern auch eine Gewinnstrategie verfolgen. Um eine Spielstrategie zu implementieren, wird das Spiel als Abfolge von Spielsituationen aufgefasst, die sich mit einer geeigneten Variablen s eindeutig identifizieren lassen. Man spricht auch von **Spielzuständen** und nennt darum s eine **Zustandsvariable**. Das Ziel einer Strategie ist es, von einem Anfangszustand zu einem Gewinnzustand zu gelangen, wobei das Spiel damit meist beendet ist.

Die Spielzustände lassen sich anschaulich als **Knoten** in einem **Spielgrafen** aufzeichnen. Bei jedem Zug gibt es einen Übergang von einem Knoten zu einem seiner Nachfolgeknoten. Die Spielregeln legen fest, welches ausgehend von einem bestimmten Spielzustand die möglichen Nachfolgeknoten oder **Nachbarn** (neighbours) diese Spielzustands sind. Den Übergang zeichnet man als eine gerichtete Verbindungslinie, auch **Kante** genannt, ein [**mehr...**].

Hier lernst du wichtige Verfahren kennen, die allgemeine Gültigkeit haben und dir helfen, Computergames zu schreiben, die sogar gegen sehr intelligente menschliche Spieler gewinnen können. Allerdings gibt es für die meisten Spiele neben diesem allgemeinen Verfahren noch viel Platz für deine eigenen Ideen, effizientere, einfachere und dem speziellen Spiel besser angepasste Spielstrategien zu schreiben, die unter Umständen auch bessere Gewinnchancen haben.

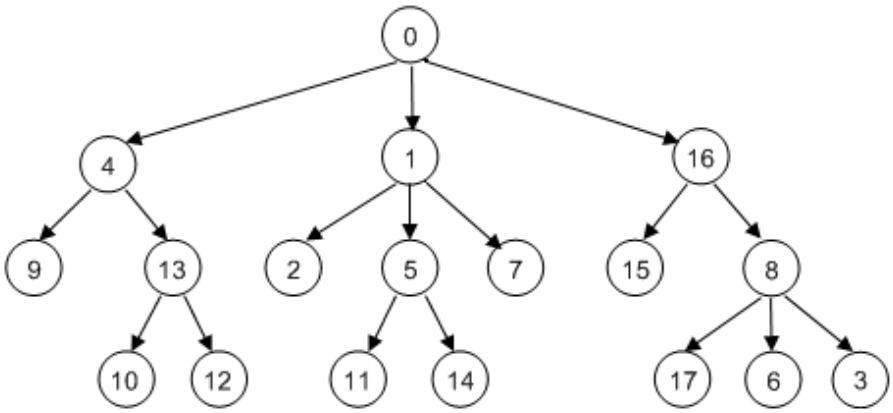
Auch viele politische, wirtschaftliche und soziale Systeme lassen sich als Spiel auffassen, sodass du deine hier erworbenen Kenntnisse auf eine weites praxisrelevantes Feld anwenden kannst.

PROGRAMMIERKONZEPTE: *Spielzustand, Spielgraf, Baumsuche, Tiefensuche, Backtracking*

LÖSUNGSSUCHE FÜR EIN SOLOSPIEL

Wie erwähnt fasst du die Spielzustände als Knoten (nodes) in einem Graphen auf, der von Zug zu Zug durchlaufen wird. Dazu musst du die Spielzustände durch bestimmte Kriterien, z.B. die Anordnung der Spielfiguren auf einem Spielbrett, eindeutig identifizieren. Die Spielregeln legen fest, welches die möglichen Nachfolgeknoten oder Nachbarn (neighbours) eines bestimmten Spielzustands sind. Diese werden mit dem Knoten durch Linien (Kanten) verbunden. Da es sich um Nachfolgeknoten handelt, haben die Kanten eine Richtung vom Knoten zu seinem Nachbarn. Manchmal heisst ein Knoten auch "Mutter" und sein Nachbarn "Töchter", wobei die Möglichkeit offen gelassen wird, ob es auch eine Kante von einer Tochter zurück zur Mutter gibt.

Du gehst hier zuerst von einem einfachen Spielgraphen für ein Spiel aus, das eine Person oder der Computer allein spielt. Das Einpersonenspiel oder Solospiel soll so aufgebaut sein, dass es keine Wege gibt, die wieder zurück führen. Damit ist sicher gestellt, dass du beim Durchlauf des Graphen nicht in einen Zyklus gerätst, der endlos durchlaufen wird. Ein solcher spezieller Graph heisst ein **Baum**. [**mehr...**] Die Knoten identifizierst du in irgend einer Reihenfolge mit Nummern zwischen 0 und 17. Der Graph hat folgende Struktur:



Der Baum soll als Ganzes in einer geeigneten Datenstruktur gespeichert werden. Dazu eignet sich eine Liste, in der die Nummern der Nachbarknoten als Teillisten enthalten sind, also beim Index 0 die Liste der Nachbarn von Knoten 0, beim Index 1 die Liste der Nachbarn von Knoten 1, usw. Enthält ein Knoten keine Nachbarn, so ist seine Nachbarliste leer [**mehr...**].

Wie du leicht siehst, wird aus dem vorgegebenen Baum die Liste

```
neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13],[11, 14], [], [], [17, 6, 3],
[], [], [], [], [10, 12], [], [], [15, 8], []]
```

Die Identifikation der Knoten durch eine Nummer ist ein Trick und ermöglicht dir, die Nachbarn eines Knoten mit einem Listenindex zu bestimmen. Der Algorithmus zum Auffinden des Weges von einem bestimmten Knoten zu einem in der Baumstruktur tiefer liegenden Knoten wird in der Funktion *search(node)* rekursiv durchgeführt. In **Pseudocode** formuliert lautet er:

```
suche(Knoten):
    Falls Knoten == Zielknoten:
        print "Ziel erreicht"
        return
    Bestimme Liste der Nachbarknoten
    Durchlaufe diese Liste und führe aus:
        suche(Nachbarknoten)
```

Zusätzlich werden die "besuchten" Knoten hinten in der Liste *visited* eingetragen. Falls nicht vorher das Ziel erreicht wurde, wird nach dem Durchlauf aller Nachbarn der Knoten wieder aus *visited* entfernt, damit wieder der ursprüngliche Zustand hergestellt ist [**mehr...**]. Start- und Zielknotennummer können zu Programmbeginn eingegeben werden.

```
neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13], [11, 14], [], [],
              [17, 6, 3], [], [], [], [], [10, 12], [], [], [15, 8], []]

def search(node):
    visited.append(node) # put (push) to stack

    # Check for solution
    if node == targetNode:
        print "Ziel ", targetNode, "erreicht. Pfad:", visited
        targetFound = True
        return

    for neighbour in neighbours[node]:
        search(neighbour) # recursive call
    visited.pop() # redraw (pop) from stack

startNode = -1
while startNode < 0 or startNode > 17:
   startNode = inputInt("Startknoten (0..17):")
targetNode = -1
while targetNode < 0 or targetNode > 17:
```

```
    targetNode = inputInt("Zielknoten (0..17):")
visited = []
search(startNode)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

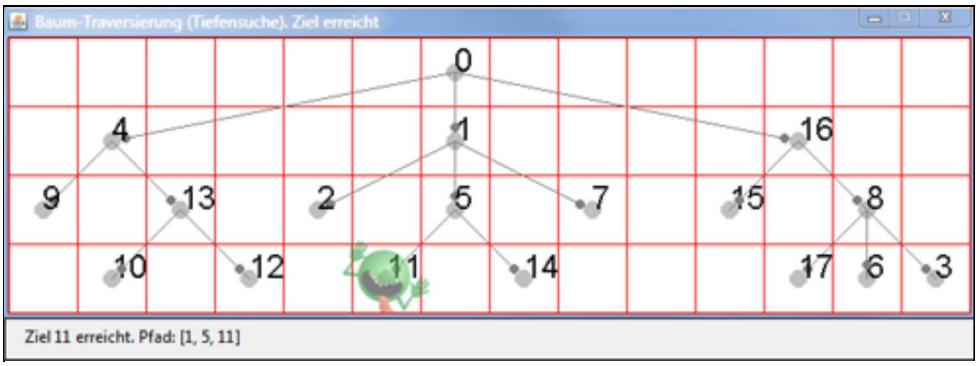
MEMO

Für den Startknoten 0 und den Zielknoten 14 wird der richtige Weg [0, 1, 5, 14] ausgeschrieben. Fügst du beim Knoten 13 als Nachbar zusätzlich den Knoten 0 ein, so ergibt sich eine verhängnisvolle Situation und das Programm wird mit einem Laufzeitfehler abgebrochen, der sagt, dass die maximale Rekursionstiefe erreicht wurde.

DURCHLAUF EINES ALIENS

Es ist hübsch, den Algorithmus anschaulich zu verfolgen, indem du den Spielbaum grafisch aufzeichnest und schrittweise (mit einem Tastendruck) durchläufst. Dazu verwendest du am einfachsten ein GameGrid-Fenster, in welchem die Knoten bei gewissen Zellenkoordinaten (Locations) als Kreise sichtbar gemacht werden.

In der aktuellen Zelle siehst du einen halbtransparenten Alien, der dir zuwinkt.



Den Baum zeichnest du mit den Grafikmethoden von *GGBackground*. Auf den Kanten kannst du mit *getMarkerPoint()* statt einem Pfeil eine kleine Kreismarkierung anbringen, um die Richtung der Kante zu zeigen. Achte darauf, dass du den Bildschirm mit *refresh()* aktualisieren musst. In der Statusbar kannst du wichtige Informationen anzeigen.

```
from gamegrid import *

neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13], [11, 14], [], [],
              [17, 6, 3], [], [], [], [], [10, 12], [], [], [15, 8], []]

locations = [Location(6, 0), Location(6, 1), Location(4, 2), Location(13, 3),
             Location(1, 1), Location(6, 2), Location(12, 3), Location(8, 2),
             Location(12, 2), Location(0, 2), Location(1, 3), Location(5, 3),
             Location(3, 3), Location(2, 2), Location(7, 3), Location(10, 2),
             Location(11, 1), Location(11, 3)]

def drawGraph():
    getBg().clear()
    for i in range(len(locations)):
        getBg().setPaintColor(Color.lightGray)
        getBg().fillCircle(toPoint(locations[i]), 6)
        getBg().setPaintColor(Color.black)
        getBg().drawText(str(i), toPoint(locations[i]))
        for k in neighbours[i]:
            drawConnection(i, k)
    refresh()
```

```
def drawConnection(i, k):
    getBg().setPaintColor(Color.gray)
    startPoint = toPoint(locations[i])
    endPoint = toPoint(locations[k])
    getBg().drawLine(startPoint, endPoint)
    getBg().fillCircle(getMarkerPoint(endPoint, startPoint, 10), 3)

def search(node):
    global targetFound
    if targetFound:
        return
    visited.append(node) # put (push) to stack
    alien.setLocation(locations[node])
    refresh()
    if node == targetNode:
        setStatusText("Ziel " + str(targetNode) + " erreicht. Pfad: "
                      + str(visited))
        targetFound = True
        return
    else:
        setStatusText("Aktueller Knoten " + str(node) + " .  Besucht: "
        + str(visited))
    getKeyCodeWait(True) # exit if GameGrid is disposed

    for neighbour in neighbours[node]:
        search(neighbour)  # Recursive call
    visited.pop()

makeGameGrid(14, 4, 50, Color.red, False)
setTitle("Baum-Traversierung (Tiefensuche). Drücke eine Taste...")
addStatusBar(30)
show()
setBgColor(Color.white)
drawGraph()

startNode = -1
while startNode < 0 or startNode > 17:
   startNode = inputInt("Startknoten (0..17):")
targetNode = -1
while targetNode < 0 or targetNode > 17:
   targetNode = inputInt("Zielknoten (0..17):")

visited = []
targetFound = False
alien = Actor("sprites/alieng_trans.png")
addActor(alien, locations[startNode])

search(startNode)
setTitle("Baum-Traversierung (Tiefensuche). Ziel erreicht")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Wie du siehst, bewegt sich der Alien zu den Töchterknoten "in die Tiefe des Baums" und springt dann auf den letzten Mutterknoten zurück. Aus diesem Grund nennt man das Verfahren **Tiefensuche mit Backtracking** (depth-first search).

DER ALIEN AUF DEM WEG ZURÜCK

Willst du sichtbar machen, auf welchem Weg sich der Alien beim Rückzug bewegt, so musst du die Knotenfolge bei der Vorwärtsbewegung in einer Liste *steps* abspeichern. In jeder Rekursionstiefe gibt es wieder eine neue Liste und du speicherst diese in *stepsList*. Nach der

Rückkehr musst du den letzten Eintrag entfernen.

```
from gamegrid import *

neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13], [11, 14], [], [],
              [17, 6, 3], [], [], [], [], [10, 12], [], [], [15, 8], []]

locations = [Location(6, 0), Location(6, 1), Location(4, 2), Location(13, 3),
             Location(1, 1), Location(6, 2), Location(12, 3), Location(8, 2),
             Location(12, 2), Location(0, 2), Location(1, 3), Location(5, 3),
             Location(3, 3), Location(2, 2), Location(7, 3), Location(10, 2),
             Location(11, 1), Location(11, 3)]

def drawGraph():
    getBg().clear()
    for i in range(len(locations)):
        getBg().setPaintColor(Color.lightGray)
        getBg().fillCircle(toPoint(locations[i]), 6)
        getBg().setPaintColor(Color.black)
        getBg().drawText(str(i), toPoint(locations[i]))
        for k in neighbours[i]:
            drawConnection(i, k)
    refresh()

def drawConnection(i, k):
    getBg().setPaintColor(Color.gray)
    startPoint = toPoint(locations[i])
    endPoint = toPoint(locations[k])
    getBg().drawLine(startPoint, endPoint)
    getBg().fillCircle(getMarkerPoint(endPoint, startPoint, 10), 3)



def search(node):
    global targetFound
    if targetFound:
        return
    visited.append(node) # put (push) to stack
    alien.setLocation(locations[node])
    refresh()
    if node == targetNode:
        setStatusText("Ziel " + str(targetNode) + " erreicht. Pfad: "
                      + str(visited))
        targetFound = True
        return
    else:
        setStatusText("Aktueller Knoten " + str(node) + " .  Besucht: "
                      + str(visited))
    getKeyCodeWait(True) # exit if GameGrid is disposed

    for neighbour in neighbours[node]:
        steps = [node]
        stepsList.append(steps)
        steps.append(neighbour)
        search(neighbour)  # Recursive call
        steps.reverse()
        if not targetFound:
            for loc in steps[1:]:
                setStatusText("Gehe zurück")
                alien.setLocation(locations[loc])
                refresh()
                getKeyCodeWait()
        stepsList.pop()
    visited.pop()

makeGameGrid(14, 4, 50, Color.red, False)
setTitle("Baum-Traversierung (Tiefensuche). Drücke eine Taste...")
addStatusBar(30)
show()
```

```
setBgColor(Color.white)
drawGraph()

startNode = -1
while startNode < 0 or startNode > 17:
   startNode = inputInt("Startknoten (0..17):")
targetNode = -1
while targetNode < 0 or targetNode > 17:
   targetNode = inputInt("Zielknoten (0..17):")

visited = []
stepsList = []
targetFound = False
alien = Actor("sprites/alieng_trans.png")
addActor(alien, locations[startNode])

search(startNode)
setTitle("Baum-Traversierung (Tiefensuche). Ziel erreicht")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Der Alien bewegt sich nun tatsächlich im Baum wieder nach oben, wodurch es besonders deutlich wird, warum der Algorithmus **Backtracking** heisst. Das rekursive Backtracking spielt bei vielen Algorithmen eine wichtige Rolle und wird manchmal auch als "Schweizer Taschenmesser der Informatiker" bezeichnet.

STRATEGIE IM IRRGARTEN

Manchmal ist es schwierig, ja geradezu beängstigend, den Ausgang aus einem Irrgarten zu finden. Du kannst aber nun mit deinem Wissen über Backtracking ein Programm schreiben, dass den Ausgang mit Sicherheit findet. Es ist offensichtlich, dass du einen Irrgarten, der keine Kreise aufweist, als Baum modellieren kannst und darum das Auffinden des Ausgang einer Baumsuche entspricht.

Du verwendest hier nur ein kleines Zufallslabyrinth mit 11x11 Zellen. Per Tastendruck führt der Alien einen Schritt aus, drückst du aber die Enter-Taste, so sucht er den Ausgang völlig autonom.

Das Labyrinth erzeugst du mit der Klasse *Maze*. Dabei übergibst du die gewünschte Anzahl der Zeilen und Spalten als ungerade Zahlen. Es entsteht jedesmal ein anderes zufälliges Labyrinth mit einem Eingang oben an der linken und einem Ausgang unten an der rechten Seite. Mit *isWall(loc)* kannst du testen, ob sich an der Location *loc* eine Wandzelle befindet.

Oft ist es nicht zweckmässig, den vollständigen Spielgraphen vor dem Spiel zu bestimmen. In vielen Fällen ist dies sogar unmöglich, da es derart viele Spielsituationen gibt, dass du sie in vernünftiger Zeit gar nicht bestimmen kannst und zudem der Speicherplatz des Programms nicht ausreichen würde. Es ist deswegen meist besser, erst während des Backtracking zu dem gerade aktuellen Knoten die Nachbarknoten zu bestimmen.

In deinem Programm ermittelst du die Nachbarknoten so, dass du von den 4 angrenzenden Zellen diejenigen auswählst, die nicht auf der Wand und nicht ausserhalb des Gitters liegen.

```
from gamegrid import *

def createMaze():
    global maze
    maze = GGMaze(11, 11)
    for x in range(11):
        for y in range(11):
            loc = Location(x, y)
            if maze.isWall(loc):
                getBg().fillCell(loc, Color(0, 50, 0))
            else:
                getBg().fillCell(loc, Color(255, 228, 196))
    refresh()

def getNeighbours(node):
    neighbours = []
    for loc in node.getNeighbourLocations(0.5):
        if isInGrid(loc) and not maze.isWall(loc):
           neighbours.append(loc)
    return neighbours

def search(node):
    global targetFound, manual
    if targetFound:
        return
    visited.append(node)  # push
    alien.setLocation(node)
    refresh()
    delay(500)
    if manual:
        if getKeyCodeWait(True) == 10:  #Enter
            setTitle("Suche Ziel...")
            manual = False

    # Check for termination
    if node == exitLocation:
        targetFound = True

    for neighbour in getNeighbours(node):
        if neighbour not in visited:
            backSteps = [node]
            backStepsList.append(backSteps)
            backSteps.append(neighbour)

            search(neighbour) # recursive call

            backSteps.reverse()
            if not targetFound:
                for loc in backSteps[1:]:
                    setTitle("Muss  zurückgehen...")
                    alien.setLocation(loc)
                    refresh()
                    delay(500)
                if manual:
                    setTitle("Zurückgehen gemacht. Taste drücken...")
                else:
                    setTitle("Zurückgehen gemacht. Suche Ziel...")
            backStepsList.pop()
    visited.pop() # pop

manual = True
targetFound = False
visited = []
backStepsList = []
makeGameGrid(11, 11, 40, False)
setTitle("Taste drücken. (<Enter> für automatisch)")
```

```
show()
createMaze()
startLocation = maze.getStartLocation()
exitLocation = maze.getExitLocation()
alien = Actor("sprites/alieng.png")
addActor(alien, startLocation)
search(startLocation)
setTitle("Ziel gefunden")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

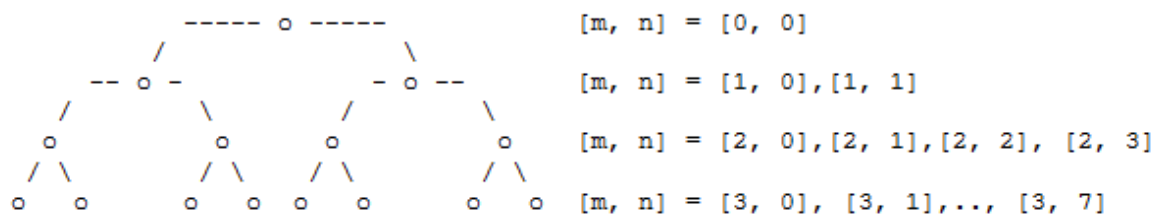
MEMO

Es ist interessant, die Lösungsstrategie des Menschen mit der des Computers zu vergleichen. Ein menschlicher Spieler erkennt auf einem Blick die Gesamtsituation und leitet daraus eine Strategie ab, wie er möglichst schnell zum Ausgang gelangt. Er handelt dabei mit einer für Menschen typischen globalen Übersicht, die deinem Computerprogramm fehlt. Dieses erfasst die momentane Situation nur lokal, *"erinnert"* sich aber dafür sehr genau an die bereits durchlaufenen Wege und durchsucht sehr systematisch neue Wege zum Ziel.

Die Situation ändert sich aber zu Gunsten des Computers, wenn man dem menschlichen Menschen die globale Übersicht vorenthält, z.B. wenn er sich selbst im Innern des Labyrinths befindet.

AUFGABEN

1. Ein **Binärbaum** besitzt zu jedem Knoten zwei Nachbarknoten, nämlich einen linken und einen rechten. Wähle als Knotenbezeichner eine Liste mit zwei Zahlen [m, n], wo m die Tiefe im Baums und n die Breite bezeichnen.



   Das Programm soll nach der Eingabe des Start- und Zielknotens den Weg ausschreiben.

2. Es ist erstaunlich, dass du mit der einfachen **Rechten-Hand-Regel** immer den Ausgang in einem Labyrinth findest, das sogar Kreise aufweisen kann. Du wanderst dabei immer mit der rechten Hand an der Wand entlang und hältst dich an die folgende Regel:

   - Wenn rechts frei ist, dann gehst du nach rechts
   - Wenn du nicht nach rechts gehen kannst, hingegen geradeaus frei ist, dann gehe vorwärts
   - Wenn du weder nach rechts noch vorwärts gehen kannst, dann drehe dich nach links.

   Implementiere diese Regel für ein GameGrid-Labyrinth mit einem rotierbaren Käfer-Actor *lady = Actor(True, "sprites/ladybug.gif").* Anleitung: Setze den Käfer mit *move()* gemäss der Regel in die nächste Zelle. Wenn es sich um eine Wandzelle handelt, so mache den Schritt rückgängig.

   Vergleiche diesen Lösungsalgorithmus mit der Lösung durch Backtracking.

ZUSATZSTOFF

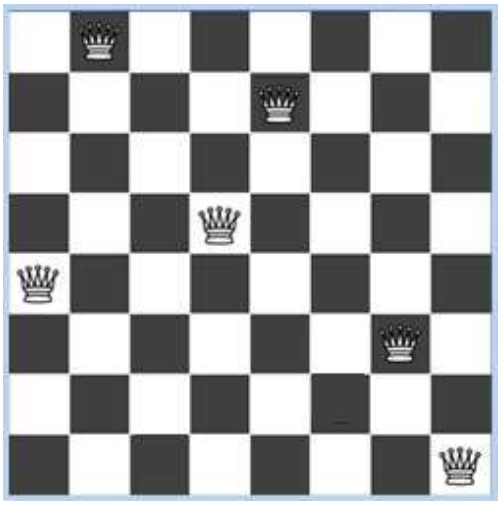
DAS N-DAMEN PROBLEM

Es handelt sich um ein schachtechnisches Problem, das bereits Mitte des 19. Jahrhunderts diskutiert wurde. Es gilt dabei, auf einem Schachbrett mit nxn-Feldern n Damen so zu platzieren, dass sie sich gemäss den Schachregeln gegenseitig nicht schlagen können. (Die Schachregeln besagen, dass sich eine Dame horizontal und vertikal, sowie auf der Diagonalen bewegen kann.) Es gibt zwei Fragestellungen mit ganz unterschiedlichem Schwierigkeitsgrad: Zum einen möchte man eine Lösung angeben, zum anderen möchte man herausfinden, wieviele Lösungen es insgesamt gibt. Bereits 1874 bewies der Mathematiker Glaisher, dass es für das klassische Schachbrett mit n = 8 insgesamt 92 Lösungen gibt.

Das Damenproblem gilt als Musterbeispiel für das Backtracking. Dabei setzt man Zug um Zug eine Dame nach der anderen derart auf das Brett, dass die neue Dame nicht in Konflikt mit den bereits vorhandenen gerät. Geht man ziellos vor, so gibt es meist vor Ende der Aufstellung einen Moment, wo dies nicht mehr möglich ist. Die beim Backtracking angewendete Strategie besteht darin, dass man den letzten Zug zurücknimmt und es mit einer Alternative versucht. Gibt es auch jetzt keine Lösung, so muss man auch diesen Zug zurücknehmen, usw. Der Mensch verliert bei diesem Vorgehen leicht die Übersicht, welche Stellungen bereits geprüft wurden, der Computer hat dagegen damit kein Problem.

Wie bei allen Aufgaben zum Backtracking kannst du die Spielzustände als Knoten in einem Spielgrafen auffassen. Die geeignete Wahl der Datenstruktur ist von entscheidender Bedeutung. Ein Vorgehen gemäss des "Brute-Force"-Prinzips, wo man die n Damen auf alle möglichen Arten auf das Brett setzt und nachher diejenigen Fälle aussondiert, in denen sie sich nicht schlagen können, ist ungeeignet, denn es gibt für n = 8 die grosse Anzahl rund 442 Millionen Stellungen.

Viel besser ist es, wenn du von Anfang an nur Stellungen betrachtest, in denen sich die gesetzten Damen auf verschiedenen Zeilen und Spalten befinden. Für n = 8 kannst du den Spielzustand mit einer Liste mit 8 Zahlen, wo die erste Zahl den Spaltenindex der Dame auf der ersten Zeile, die zweite Zahl den Spaltenindex der Dame auf der zweiten Zeile, usw. angeben. Für Zeilen, welche noch keine gesetzte Damen haben, schreibst du als Index -1.



Wenn du Zeilen- und Spaltenindizes von 0 bis n-1 nimmst, so identifiziert beispielsweise node = [1, 4, -1, 3, 0, 6, -1, 7] nebenstehende Stellung.

Im Backtracking-Algorithmus bestimmst du die Nachbarknoten des aktuellen Knotens *node* mit der Funktion *getNeighbours(node)*. Dabei gehst du von der eindimensionalen Datenstruktur auf *Locations* über, welche die x und y Koordinate der Felder verwendet. In der die Liste *squares* sammelst du die bereits besetzten Felder und in der Liste *forbidden* diejenigen, die auf Grund der Spielregeln nicht besetzt werden dürfen. (Dabei ist es praktisch, die Methode *getDiagonalLocations()* zu verwenden.) Schliesslich bildest du die Liste *allowed* der noch besetzbaren Felder. In den Nachbarknoten musst du nun die -1 durch den Spaltenindex ersetzen, auf den die neue Dame gesetzt wird

In *search()* implementierst du den dir bereits bekannten Backtracking-Algorithmus. Sobald eine Lösung gefunden wurde, brichst du die weitere Suche ab (Rekursionsstopp).

```
from gamegrid import *

n = 8 # number of queens

def getNeighbours(node):
    squares = [] # list of occupied squares
    for i in range(n):
        if node[i] != -1:
           squares.append(Location(node[i], i))

    forbidden = [] # list of forbidden squares
    for location in squares:
        a = location.x
        b = location.y
        for x in range(n):
            forbidden.append(Location(x, b))  # same row
        for y in range(n):
            forbidden.append(Location(a, y))  # same column
        for loc in getDiagonalLocations(location, True):   #diagonal up
            forbidden.append(loc)
        for loc in getDiagonalLocations(location, False):  #diagonal down
            forbidden.append(loc)

    allowed = [] # list of all allowed squares = all - forbidden
    for i in range(n):
        for k in range(n):
            loc = Location(i, k)
            if not loc in forbidden:
                allowed.append(loc)

    neighbourNodes = []
    for loc in allowed:
        neighbourNode = node[:]
        i = loc.y # row
        k = loc.x # col
        neighbourNode[i] = k
        neighbourNodes.append(neighbourNode)
    return neighbourNodes

def search(node):
    global found
    if found or isDisposed():
        return
    visited.append(node)  # node marked as visited

    # Check for solution
    if not -1 in node:
        found = True
        drawNode(node)

    for s in getNeighbours(node):
        search(s)
    visited.pop()

def drawBoard():
    for i in range(n):
        for k in range(n):
            if (i + k) % 2 == 0:
                getBg().fillCell(Location(i, k), Color.white)

def drawNode(node):
    removeAllActors()
    for i in range(n):
        addActorNoRefresh(Actor("sprites/chesswhite_1.png"), Location(node[i], i))
    refresh()

makeGameGrid(n, n, 600 // n, False)
setBgColor(Color.darkGray)
drawBoard()
```

```
show()
setTitle("Bin am Suchen. Bitte warten..." )

visited = []
found = False
startNode = [-1] * n  # all squares empty
search(startNode)
setTitle("Suche beendet. ")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Je nach Leistung deines Rechners musst du einige Sekunden bis einige Minuten warten, bis eine Lösung gefunden wird. Dauert es dir zulange, so kannst du n = 6 setzen.

AUFGABEN

1. Löse die gleiche Aufgabe ohne Verwendung der GameGrid-Bibliothek. Ersetze dabei Location durch Zellenlisten [i, k]. Du kannst die Lösung als node-Liste ausschreiben

2. Verallgemeinere für n = 6 das oben angegebene Programm oder dein Programm aus Aufgabe 1 so, dass alle Lösungen gefunden werden. Beachte, dass ein Lösungsknoten mehrmals gefunden wird und du dies aber mit einer Liste der bereits erhaltenen Lösungen berücksichtigen kannst.

10.4 KÜRZESTER WEG, 3 KRÜGE

EINFÜHRUNG

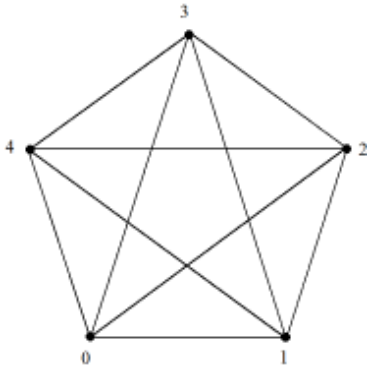
Viele wichtige algorithmische Lösungsverfahren haben ihren Ursprung in der Praxis des täglichen Lebens, so auch das Backtracking. Wie du bereits erfahren hast, wählt der Computer einen Spielzug aus den Möglichkeiten der erlaubten Züge und verfolgt ihn konsequent weiter. Gerät er in eine Sackgasse, so nimmt er die vorhergehenden Züge systematisch zurück. Diese Lösungsstrategie heisst im täglichen Leben auch ***Versuch und Irrtum*** (trial and error). Es ist bekannt, dass sie nicht immer optimal ist. Besser wäre es, den nächsten Zug möglichst günstig im Bezug auf die Zielsetzung zu wählen [**mehr...**].

PROGRAMMIERKONZEPTE: *Versuch und Irrtum, Graph mit Kreisen, Backtracking*

GRAPH MIT KREISEN

Beim Durchlaufen eines Graphen kann es sein, dass du nach einigen Schritten wieder im gleichen Zustand ankommst. Denke dabei zum Beispiel an das U-Bahn-Netz einer grossen Stadt, wo die Stationen eine verflochtene Struktur aufweisen. Willst du in dieser Stadt von A nach B fahren, so gibt es mehrere Möglichkeiten, und du kannst leicht auch in Kreisen herumfahren. Als Vorbereitung auf solche Navigationsaufgaben betrachtest du einen Graphen mit 5 Knoten, bei dem jeder Knoten mit jedem anderen verbunden ist.

Die Knoten werden mit einer Knotennummer 0..4 identifiziert. Wie du bereits gesehen hast, findet der einfache Backtracking-Algorithmus den Weg von einem bestimmten Knoten zu einem anderen unter Umständen nicht, weil er in einem Kreiszyklus hängen bleibt. Du benötigst aber nur eine kleine Ergänzung, um dies zu vermeiden: Bevor du den rekursiven Aufruf von *search()* machst, prüfst du in der Liste *visited*, ob der betreffende Nachbarknoten bereits besucht war. Wenn ja, dann überspringst du diesen Nachbarn. Im Programm werden nun alle 16 Wege zwischen den Knoten ausgeschrieben.




```
def getNeighbours(node):
    return range(0, node) + range(node + 1, 5)

def search(node):
    global nbSolution
    visited.append(node)  # node marked as visited
    if node == targetNode:
        nbSolution += 1
        print nbSolution, ". Weg:", visited
    for neighbour in getNeighbours(node):
        if neighbour not in visited: # Check if already visited
            search(neighbour) # recursive call
    visited.pop()

startNode = 0
targetNode = 4
nbSolution = 0
visited = []
search(startNode)
```

MEMO

Durch Prüfen, ob ein Nachbar bereits besucht wurde, kann man das Backtracking auch auf Graphen mit Kreisen anwenden. Vergisst du diese Prüfung, so wird sich dein Programm "aufhängen", was zu einem bösen Laufzeitfehler führt, da der Funktionsaufrufspeicher überläuft.

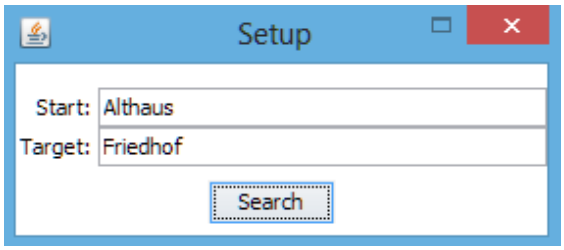
KÜRZESTER WEG, NAVIGATIONSSYSTEME

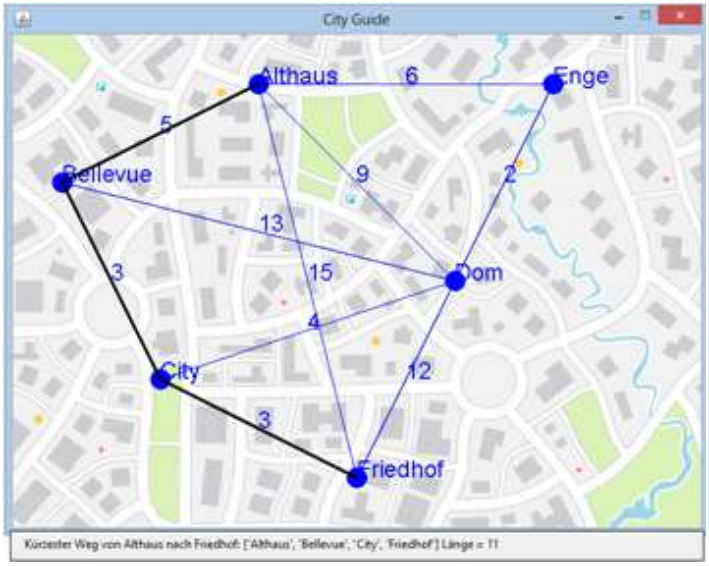
Das Auffinden eines Weges von einem Startort A zu einem Zielort B ist im täglichen Leben omnipräsent. Da oft viele Wege von A nach B (nicht nur nach Rom) führen, besteht meist noch die zusätzliche Aufgabe, den bezüglich eines bestimmten Kriteriums (Weglänge, Fahrzeit, Strassenqualität, Sehenswürdigkeiten, Kosten, usw.) optimalen Weg zu finden [**mehr...**].

In deinem Programm geht es nur um das Grundsätzliche. Darum wählst du ein Ortsnetz mit nur 6 Orten, die du als U-Bahn-Stationen in einer fiktiven Stadt auffassen kannst. Die Knoten des Graphen werden mit dem Namen der Stationen identifiziert. (Die Anfangsbuchstaben sind A, B, C, D, E, F, die Stationen könnten auch mit diesen Buchstaben oder mit Knotennummern identifiziert werden.) Die Angabe der Nachbarstationen ist eine Zuordnung eines Stationsnamens zu einer Liste von Namen, wozu sich ein Dictionary hervorragend eignet, das als Schlüssel (key) den Stationsnamen und als Wert (value) die Liste der Nachbarstationen hat. Statt einer Funktion *getNeighbours()* wird direkt das Dictionary *neighbours* verwendet.

Analog werden die Distanzen zwischen den Stationen ebenfalls in einem Dictionary *distances* abgespeichert, das als Schlüssel die zwei verbundenen Stationen und als Wert die Distanz hat.Um dies Stationen in einer GameGrid einzutragen, gibt es zudem noch ein Dictionary *locations* mit den Zellenkoordinaten (Locations) der Stationen.

Der zentrale Teil deines Programms ist eine genaue Wiederholung des oben verwendeten Backtracking-Algorithmus. Darüber hinaus benötigst du einige Hilfsfunktionen zur grafischen Darstellung.
Für die Benutzereingabe verwendest du einen Eingabedialog und Ausgaben schreibst du in eine Statusbar. Überdies zeichnest du den optimalen Weg im Stationengraphen ein.





```
from gamegrid import *

neighbours = {
 'Althaus':['Bellevue', 'Dom', 'Enge'],
 'Bellevue':['Althaus', 'City', 'Dom'],
 'City':['Bellevue', 'Dom', 'Friedhof'],
 'Dom':['Althaus', 'Bellevue', 'City', 'Enge', 'Friedhof'],
 'Enge':['Althaus', 'Dom'],
 'Friedhof':['Althaus', 'City', 'Dom']}

distances = {
   ('Althaus', 'Bellevue'):5, ('Althaus', 'Dom'):9,
   ('Althaus', 'Enge'):6, ('Althaus', 'Friedhof'):15,
   ('Bellevue', 'City'):3, ('Bellevue', 'Dom'):13,
   ('City', 'Dom'):4, ('City', 'Friedhof'):3,
   ('Dom', 'Enge'):2, ('Dom', 'Friedhof'):12}

locations = {
 'Althaus':Location(2, 0),
 'Bellevue':Location(0, 1),
 'City':Location(1, 3),
 'Dom':Location(4, 2),
 'Enge':Location(5, 0),
 'Friedhof':Location(3, 4)}

def getNeighbourDistance(station1, station2):
    if station1 < station2:
        return distances[(station1, station2)]
    return distances[(station2, station1)]

def totalDistance(li):
    sum = 0
    for i in range(len(li) - 1):
        sum += getNeighbourDistance(li[i], li[i + 1])
    return sum

def drawGraph():
    getBg().clear()
    getBg().setPaintColor(Color.blue)
    for station in locations:
        location = locations[station]
        getBg().fillCircle(toPoint(location), 10)
        startPoint = toPoint(location)
        getBg().drawText(station, startPoint)
        for s in neighbours[station]:
            drawConnection(station, s)
            if s < station:
                distance = distances[(s, station)]
            else:
                distance = distances[(station, s)]
            endPoint = toPoint(locations[s])
            getBg().drawText(str(distance),
                   getDividingPoint(startPoint, endPoint, 0.5))
    refresh()

def drawConnection(startStation, endStation):
    startPoint = toPoint(locations[startStation])
    endPoint = toPoint(locations[endStation])
    getBg().drawLine(startPoint, endPoint)

def search(station):
    global trackToTarget, trackLength
    visited.append(station)  # station marked as visited

    # Check for solution
    if station == targetStation:
        currentDistance = totalDistance(visited)
        if currentDistance < trackLength:
            trackLength = currentDistance
```

```
            trackToTarget = visited[:]

    for s in neighbours[station]:
        if s not in visited:  # if all are visited, recursion returns
            search(s) # recursive call
    visited.pop() # station may be visited by another path

def init():
    global visited, trackToTarget, trackLength
    visited = []
    trackToTarget = []
    trackLength = 1000
    drawGraph()

makeGameGrid(7, 5, 100, None, "sprites/city.png", False)
setTitle("City Guide")
addStatusBar(30)
show()
init()
startStation = ""
while not startStation in locations:
    startStation = inputString("Start station")
targetStation = ""
while not targetStation in locations:
    targetStation = inputString("Target station")
search(startStation)
setStatusText("Shortest way from " + startStation + " to " + targetStation
    + ": " + str(trackToTarget) + " Length = " + str(trackLength))
for i in range(len(trackToTarget) - 1):
    s1 = trackToTarget[i]
    s2 = trackToTarget[i + 1]
    getBg().setPaintColor(Color.black)
    getBg().setLineWidth(3)
    drawConnection(s1, s2)
refresh()
```

MEMO

Die Suche nach dem kürzesten Weg in einem Graphen gehört zu den Grundaufgaben der Informatik. Die hier gezeigte Lösung mit Backtracking führt zwar zum Ziel, ist aber sehr rechenintensiv. Es gibt viel bessere Lösungsalgorithmen, die auch davon Gebrauch machen, dass man nicht alle Wege systematisch absuchen muss. Es ist beispielsweise wenig wahrscheinlich, dass der kürzeste Weg von einem nördlich gelegenen Startort zu einem südlich gelegenen Zielort über eine weit entfernte Station nördlich des Startorts führt.

DREIKRÜGEPROBLEM

Seit vielen Jahrhunderten findet man in Kinderbüchern und Zeitschriften Denksportaufgaben, bei denen eine vorgegebene Menge durch Abmessen (Umgiessen, Wägen, usw.) in bestimmte Teilmengen aufgeteilt werden soll. Das seit dem 17. Jahrhundert bekannte Dreikrügeproblem wird dem französischen Mathematiker Bachet de Méziriac zugeschrieben und lautet wie folgt:

Zwei Freunde haben beschlossen, sich durch Umgiessen 8 Liter Wein zu teilen, der sich in einem 8-Liter-Krug befindet. Sie besitzen dazu neben dem 8-Liter-Krug noch einen 5-Liter und einen 3-Liter-Krug. Die Krüge habe keine Inhaltsmarkierung. Wie müssen sie vorgehen und wie viele Umgiessvorgänge sind im Minimum nötig?

Gemäss der Aufgabenstellung geht es also nicht nur darum, eine Lösung zu finden, was du wahrscheinlich auch mit ein bisschen Gedankenspielerei zu Stande bringst, sondern alle Lösungen zu suchen, um daraus die kürzeste zu bestimmen. Ohne Computer ist dies sehr anstrengend. Man nennt die Suche nach allen Lösungen auch **Erschöpfende Suche**.

Wieder gehst du gemäss der vorher erprobtem Lösungsstrategie mit Backtracking vor. Zuerst erfindest du eine geeignete Datenstruktur für die Spielzustände. Du verwendest dazu eine Liste mit einem Zahlentripel, das den aktuellen Füllzustand der drei Krüge beschreibt. [1, 4, 3] soll heissen, dass der 8-Liter-Krug gegenwärtig 1 Liter, der 5-Liter-Krug 4 Liter und der 3-Liter-Krug 3 Liter enthält.

Du kannst die Spielzustände wiederum als Knoten in einem Graphen modellieren und das Umgiessen als Übergang von einem Knoten zu einem seiner Nachbarknoten auffassen. Es ist hier, wie in vielen anderen Beispielen, nicht sinnvoll, den ganzen Spielbaum zu Beginn aufzubauen. Vielmehr bestimmst du die Nachbarknoten eines Knotens *node* in der Funktion *getNeighbours(node)* erst dann, wenn du sie im Laufe des Spiels tatsächlich benötigst. Dabei gehst du von folgender Überlegung aus:

Unabhängig davon, welche Menge sich in den Krügen befindet, gibt es für das Umgiessen grundsätzlich 6 Möglichkeiten: Man nimmt einen der Krüge und giesst seinen ganzen Inhalt oder soviel wie Platz vorhanden ist in einen der zwei anderen Krüge. In *getNeighbours()* sammelst du daher in der Liste *neighbours* die Nachbarknoten für diese 6 Fälle. Die Funktion *transfer(state, source, target)* hilft dir dabei herauszufinden, welches der Nachbarzustand zu einem bestimmten Zustand *state* und gegebenen Krugnummern *source* und *target* beim Umgiessen von *source* nach *target* ist. Dabei werden die Kruggrössen (maximaler Inhalt) und die darin bereits enthaltenen Mengen berücksichtigt.
In der rekursiven Funktion *search()* verwendest du wieder den Backtracking-Algorithmus, so wie er dir bereits bekannt ist.

```
def transfer(state, source, target):
    # Assumption: source, target 0..2, source != target
    s = state[:] # clone
    if s[source] == 0 or s[target] == capacity[target]:
        return s  # source empty or target full
    free = capacity[target] - s[target]
    if s[source] <= free: # source has enough space in target
        s[target] += s[source]
        s[source] = 0
    else:  # target is filled-up
        s[target] = capacity[target]
        s[source] -= free
    return s

def getNeighbours(node):
# returns list of neighbours
    neighbours = []
    t = transfer(node, 0, 1) # from 0 to 1
    if t not in neighbours:
       neighbours.append(t)
    t = transfer(node, 0, 2) # from 0 to 2
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 1, 0) # from 1 to 0
    if  t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 1, 2) # from 1 to 2
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 2, 0) # from 2 to 0
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 2, 1) # from 2 to 1
    if t not in neighbours:
        neighbours.append(t)
```

```
    return neighbours

def search(node):
    global nbSolution
    visited.append(node)

    # Check for solution
    if node == targetNode:
        nbSolution += 1
        print nbSolution, ". Weg:", visited, ". Länge:", len(visited)

    for s in getNeighbours(node):
        if s not in visited:
            search(s)
    visited.pop()

capacity = [8, 5, 3]
startNode = [8, 0, 0]
targetNode = [4, 4, 0]
nbSolution = 0
visited = []
search(startNode)
print "Geschafft. Suche die beste Lösung!"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Lösungen werden in das Ausgabefenster ausgeschrieben. Damit du unvoreingenommen an das Problem heran gehst und vielleicht versuchst, zuerst eine eigene Lösung mit Papier und Bleistift zu finden, werden sie hier nicht gezeigt. Immerhin sei verraten, dass es 16 Lösungen gibt und für den l längsten 16 Umgiessvorgänge nötig sind.

AUFGABEN

1. Vereinfache das Navigationsprogramm so, dass die Knoten mit Zahlen 0, 1, 2, 3, 4, 5 identifiziert werden und *neighbours* eine Liste mit Teillisten ist.

2. Mit einem 3-Liter und einem 5-Liter-Krug, sollst du genau 4 Liter Wasser aus einem See schöpfen. Beschreibe, wie du vorgehen würdest und gib den kürzesten Umgiessvorgang an. Beachte, dass du das Wasser auch wieder in den See zurückgiessen kannst.

3. Erfinde ein lösbares Umgiessproblem und stelle es als Denksportaufgabe für andere Personen in deiner Umgebung.

ZUSATZSTOFF

CITY-NAVIGATION MIT MAUSUNTERSTÜTZUNG

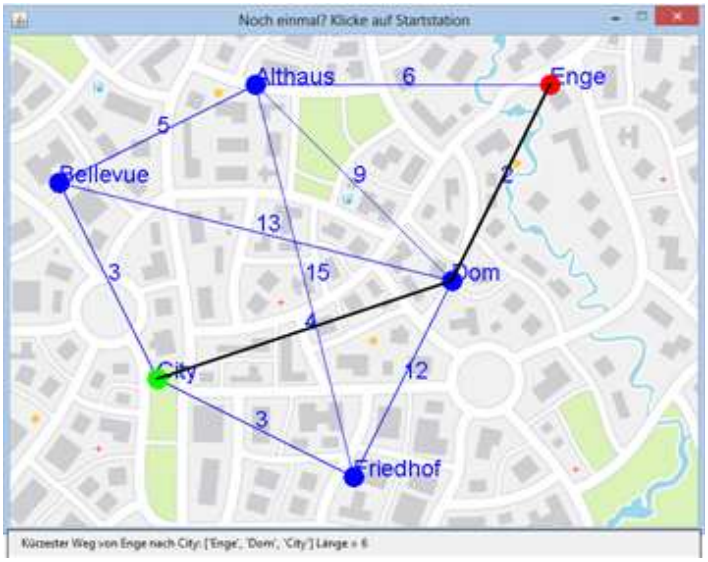
In einem professionellen Programm spielt die Benutzeroberfläche eine zentrale Rolle. Dabei muss sich der Programmierer weniger von programmtechnischen Überlegungen leiten lassen, sondern sich vielmehr in die Situation des unvoreingenommenen Anwenders versetzen, der mit möglichst wenig Aufwand und mit natürlicher menschlicher Logik das Programm verwendet. Man spricht dabei auch vom **Benutzerinterface**, das heutzutage meist eine grafische Benutzeroberfläche mit Mausbedienung umfasst. Der Aufwand für die Entwicklung des Benutzerinterfaces kann ein beträchtlicher Teil des Gesamtaufwands eines Informatikprojekts ausmachen.

Für Navigationssysteme werden immer mehr berührungsempfindliche Bildschirme (touch

screens) verwendet, die sich aber bezüglich der Programmlogik wenig von einer Mausbedienung unterscheiden. Daher wirst du hier die City-Navigation auf Mausbedienung umstellen, so dass der Benutzer Start- und Zielort mit einem Mausklick auswählen kann. Vom Programm abgegebene Informationen werden einerseits in die Titelzeile und andererseits in die Statusbar geschrieben.

Der Mausklick löst einen Event aus, der im Callback *pressEvent()* abgearbeitet wird. Diese registrierst du in *makeGameGrid()* mit dem benannten Parameter *mousePressed*. Dabei musst du berücksichtigen, dass sich das Programm in zwei unterschiedlichen Zuständen befinden kann, je nachdem ob der Benutzer als nächste Aktion die Startstation anklickt oder ob er dies bereits getan hat und als nächstes die Zielstation wählen muss. Für diese Zustandsumschaltung genügt eine boolesche Variable (ein Flag) *isStart*, die dann *True* ist, wenn als nächstes die Startstation zu wählen ist.

Das Programm soll so aufgebaut sein, dass der Benutzer die Wegsuche mehrmals durchführen kann, ohne dass er das Programm neu starten muss. Das Programm muss daher programmintern wieder in einen wohldefinierten Anfangszustand zurückgesetzt werden. Man spricht von der **Initialisierung**, die am besten mit einer Funktion *init()* durchgeführt wird. Da bei Programmstart gewisse Initialisierungen automatisch durchgeführt werden, ist es keineswegs trivial, das laufende Programm durch eine eigene Funktion immer wieder in einen wohldefinierten Anfangszustand zurückzusetzen. **Initialisierungsfehler** sind darum weit verbreitete, gefährliche und schwierig aufzufindende Programmierfehler, da sich oft das Programm bei der Erprobung richtig und erst später im Einsatz falsch verhält.




```
from gamegrid import *

locations = {
 'Althaus':Location(2, 0),
 'Bellevue':Location(0, 1),
 'City':Location(1, 3),
 'Dom':Location(4, 2),
 'Enge':Location(5, 0),
 'Friedhof':Location(3, 4)}

neighbours = {
 'Althaus':['Bellevue', 'Dom', 'Enge'],
 'Bellevue':['Althaus', 'City', 'Dom'],
 'City':['Bellevue', 'Dom', 'Friedhof'],
 'Dom':['Althaus', 'Bellevue', 'City', 'Enge', 'Friedhof'],
 'Enge':['Althaus', 'Dom'],
 'Friedhof':['Althaus', 'City', 'Dom']}

distances = {('Althaus', 'Bellevue'):5, ('Althaus', 'Dom'):9,
             ('Althaus', 'Enge'):6, ('Althaus', 'Friedhof'):15,
             ('Bellevue', 'City'):3, ('Bellevue', 'Dom'):13,
             ('City', 'Dom'):4, ('City', 'Friedhof'):3,
             ('Dom', 'Enge'):2, ('Dom', 'Friedhof'):12}
```

```
def getNeighbourDistance(station1, station2):
    if station1 < station2:
        return distances[(station1, station2)]
    return distances[(station2, station1)]

def totalDistance(li):
    sum = 0
    for i in range(len(li) - 1):
        sum += getNeighbourDistance(li[i], li[i + 1])
    return sum

def drawGraph():
    getBg().clear()
    getBg().setPaintColor(Color.blue)
    for station in locations:
        location = locations[station]
        getBg().fillCircle(toPoint(location), 10)
        startPoint = toPoint(location)
        getBg().drawText(station, startPoint)
        for s in neighbours[station]:
            drawConnection(station, s)
            if s < station:
                distance = distances[(s, station)]
            else:
                distance = distances[(station, s)]
            endPoint = toPoint(locations[s])
            getBg().drawText(str(distance),
                 getDividingPoint(startPoint, endPoint, 0.5))
    refresh()

def drawConnection(startStation, endStation):
    startPoint = toPoint(locations[startStation])
    endPoint = toPoint(locations[endStation])
    getBg().drawLine(startPoint, endPoint)

def search(station):
    global trackToTarget, trackLength
    visited.append(station)  # station marked as visited

    # Check for solution
    if station == targetStation:
        currentDistance = totalDistance(visited)
        if currentDistance < trackLength:
            trackLength = currentDistance
            trackToTarget = visited[:]

    for s in neighbours[station]:
        if s not in visited:  # if all are visited, recursion returns
            search(s) # recursive call
    visited.pop() # station may be visited by another path

def getStation(location):
    for station in locations:
        if locations[station] == location:
            return station
    return None # station not found

def init():
    global visited, trackToTarget, trackLength
    visited = []
    trackToTarget = []
    trackLength = 1000
    drawGraph()

def pressEvent(e):
    global isStart, startStation, targetStation
    mouseLoc = toLocationInGrid(e.getX(), e.getY())
    mouseStation = getStation(mouseLoc)
```

```
    if mouseStation == None:
        return
    if isStart:
        isStart = False
        init()
        setTitle("Klicke auf Zielstation")
        startStation = mouseStation
        getBg().setPaintColor(Color.red)
        getBg().fillCircle(toPoint(mouseLoc), 10)
    else:
        isStart = True
        setTitle("Noch einmal? Klicke auf Startstation")
        targetStation = mouseStation
        getBg().setPaintColor(Color.green)
        getBg().fillCircle(toPoint(mouseLoc), 10)
        search(startStation)
        setStatusText("Kürzester Weg von " + startStation + " nach "
            + targetStation + ": " + str(trackToTarget) + " Länge = "
            + str(trackLength))
        for i in range(len(trackToTarget) - 1):
            s1 = trackToTarget[i]
            s2 = trackToTarget[i + 1]
            getBg().setPaintColor(Color.black)
            getBg().setLineWidth(3)
            drawConnection(s1, s2)
            getBg().setLineWidth(1)
    refresh()

isStart = True
makeGameGrid(7, 5, 100, None, "sprites/city.png", False,
             mousePressed = pressEvent)
setTitle("City Guide. Klicke auf Startstation")
addStatusBar(30)
show()
init()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Der algorithmische Teil mit dem Backtracking bleibt praktisch unverändert. Das Benutzerinterface mit der Maussteuerung ist trotz guter Unterstützung durch Callbacks ziemlich aufwendig.

Die Verwendung von **global** führt in Python leicht zu Initialisierungsfehlern, da globale Variablen in Funktionen erzeugt werden können und man später vergisst, ihren Wert zurückzusetzen.

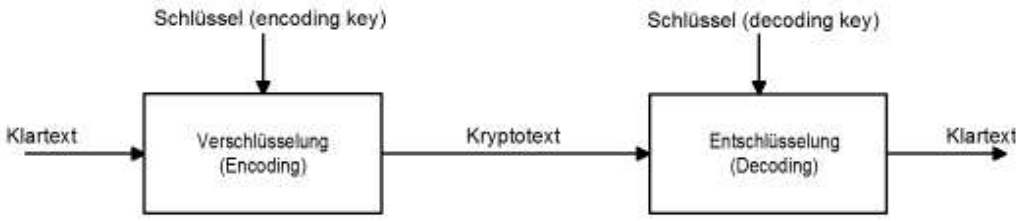
10.5 KRYPTOSYSTEME

EINFÜHRUNG

Das Geheimhalteprinzip spielt in unserer modernen Welt eine immer wichtigere Rolle. Zum Schutz der Privatsphäre, aber auch zur Geheimhaltung wichtiger staatlicher, industrieller und militärischer Informationen ist es nötig, Daten derart zu verschlüsseln, dass die verschlüsselten Daten wohl in die Hände von Unbefugten fallen können, es aber ohne Bekanntgabe der Entschlüsselungsmethode unmöglich oder zumindest sehr schwierig ist, die Originalinformation herauszufinden.

Bei der **Verschlüsselung** (**encoding**) werden die Originaldaten in verschlüsselte (chiffrierte) Daten umgewandelt. Bei der **Entschlüsselung** (**decoding**) werden die Originaldaten wieder hergestellt. Verwendet man für die Daten das Buchstabenalphabet, so spricht man auch von **Klartext** und **Kryptotext**.

Die Beschreibung des Verfahrens zum Entschlüsseln wird ***Schlüssel*** genannt. Es kann sich auch nur um eine einzige Zahl, eine Zahlen- oder eine Buchstabenfolge (ein Schlüsselwort) handeln. Wird beim Verschlüsseln und Entschlüsseln derselbe Schlüssel verwendet, so spricht man von einem **symmetrischen Kryptoverfahren**, andernfalls von einem **asymmetrischen Kryptoverfahren**.



PROGRAMMIERKONZEPTE: *Verschlüsselung, Entschlüsselung, Symmetrisches/Asymmetrisches Kryptoverfahren, Caesar-, Vigenère-, RSA-Verschlüsselung, Privater/Öffentlicher Schlüssel*

CAESAR-VERSCHLÜSSELUNG

Nach der Überlieferung soll bereits Julius Cäsar (100 v.Chr. - 44 v.Chr.) folgendes Verfahren für seine militärische Korrespondenz angewendet haben: Jeder Buchstabe des Klartexts wird dabei alphabetisch um eine bestimmte feste Schlüsselzahl nach rechts verschoben, wobei man nach Z wieder bei A weiterfährt.

Das Alphabet wird bei dieser Methode also in einem **Ringbuffer** angelegt. Mit dem Schlüssel 3 werden die Buchstaben A in D, B in E, C in F, D in G, usw. verschlüsselt.

In deinem Programm verwendest du Textdateien für die Daten, damit du sie leicht verändern und weitergeben kannst. Den Klartext schreibst du mit irgendeinem Texteditor in die Datei *original.txt*, die du im Verzeichnis, in dem sich dein Programm befindet, speichern musst. Verwende für den Text nur Grossbuchstaben und das Leerzeichen. Du kannst mehrere Zeilen schreiben, also

beispielsweise

```
HEUTE TREFFEN WIR UNS UM ACHT
LIEBER GRUSS
TANIA
```

Der Encoder verschlüsselt den von der Datei eingelesene Textstring *msg* mit der Funktion *encode(msg)*, wobei ausser für den Zeilenumbruch \n jedes Zeichen durch das entsprechende Kryptozeichen ersetzt wird.

```
import string
key = 4
alphabet = string.ascii_uppercase + " "

def encode(text):
    enc = ""
    for ch in text:
        if ch != "\n":
            i = alphabet.index(ch)
            ch = alphabet[(i + key) % 27]
        enc += ch
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Dein verschlüsselter Text sieht wie folgt aus:

```
LIYXIDXVIJJIRD MVDYRWDYQDEGLX
PMIFIVDKVYWW
XERME
```

Der Decoder ist ganz analog aufgebaut, nur dass die Zeichen im Alphabet rückwärts verschoben werden.

```
import string
key = 4
alphabet = string.ascii_uppercase + " "

def decode(text):
    dec = ""
    for ch in text:
        if ch != "\n":
            i = alphabet.index(ch)
            ch = alphabet[(i - key) % 27]
        dec += ch
    return dec

fInp = open("secret.txt")
krypto = fInp.read()
fInp.close()

print "Krypto:\n", krypto
msg = decode(krypto)
print "Message:\n", msg
```

```
fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Beachte, dass du im Kryptotext auch alle Leerzeichen beibehalten musst, auch wenn diese am Anfang oder am Ende einer Zeile stehen. Es ist klar, dass die Verschlüsselung leicht geknackt werden kann. Es genügt beispielsweise, die Schlüsselzahlen 1..26 auszuprobieren

VERSCHLÜSSELUNG NACH DEM VIGENÈRE-VERFAHREN

Du kannst die Caesar-Verschlüsselung sicherer machen, indem du auf jedes Zeichen des Klartexts eine unterschiedliche alphabetische Verschiebung anwendest. Diese sogenannte polyalphabetische Substitution könnte als Schlüssel irgendeine Permutation von 27 Zahlen verwenden. Davon gibt es eine riesige Zahl, nämlich

$$27! = 10'888'869'450'418'352'160'768'000'000 \approx 10^{27}$$

Etwas einfacher ist die Verwendung eines Schlüsselworts, dem die Liste der entsprechenden Zeichen im Alphabet zugeordnet ist, also beispielsweise dem Schlüssel ALICE die Liste [0, 11, 8, 2, 4]. Bei der Verschlüsselung werden dann die Zeichen der Reihe nach um 0, 11, 8, 2, 4 und dann wiederholt um 0, 11,... Zeichen alphabetisch verschoben.

Blaise Vigenère (1523-1596)

```
import string
key = "ALICE"
alphabet = string.ascii_uppercase + " "

def encode(text):
    keyList = []
    for ch in key:
        i = alphabet.index(ch)
        keyList.append(i)
    print "keyList:", keyList
    enc = ""
    for n in range(len(text)):
        ch = text[n]
        if ch != "\n":
            i = alphabet.index(ch)
            k = n % len(key)
            ch = alphabet[(i + keyList[k]) % 27]
        enc += ch
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
```

```
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Der Decoder ist wiederum praktisch identisch.

```
import string
key = "ALICE"
alphabet = string.ascii_uppercase + " "

def decode(text):
    keyList = []
    for ch in key:
        i = alphabet.index(ch)
        keyList.append(i)
    print "keyList:", keyList
    enc = ""
    for n in range(len(text)):
        ch = text[n]
        if ch != "\n":
            i = alphabet.index(ch)
            k = n % len(key)
            ch = alphabet[(i - keyList[k]) % 27]
        enc += ch
    return enc

fInp = open("secret.txt")
krypto = fInp.read()
fInp.close()

print "Krypto:\n", krypto
msg = decode(krypto)
print "Message:\n", msg

fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

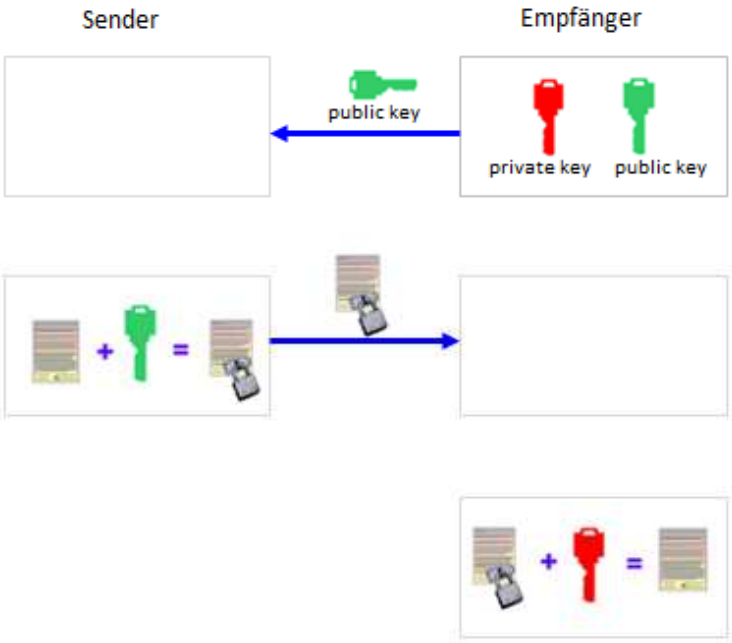
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Vigenère-Verschlüsselung wurde bereits im 16. Jh. von Blaise de Vigenère erfunden und galt Jahrhunderte lang als sehr sicher. Kennt man die Länge 5 des Schlüsselworts, so muss man immerhin noch 26^5 = 11'881'376 Schlüsselzahlen durchprobieren, ausser man weiss etwas über das verwendete Wort, beispielsweise, dass es sich um den Vornamen einer Frau handelt.

VERSCHLÜSSELUNG NACH DEM RSA-VERFAHREN

Bei diesem Verfahren, das nach ihren Erfindern Rivest, Shamir und Adleman benannt ist, wird ein Schlüsselpaar verwendet, nämlich ein privater (private key) und ein öffentlicher Schlüssel (public key). Die Originaldaten werden mit dem öffentlichen Schlüssel verschlüsselt und mit dem privaten Schlüssel entschlüsselt. Es handelt sich also um ein asymmetrisches Kryptoverfahren.



Schritt 1:
Der Empfänger erzeugt den *private key* und den *public key* und schickt den *public key* an den Sender.

Schritt 2:
Der Sender verschlüsselt seine Nachricht mit dem *public key* und schickt den verschlüsselten Text zurück.

Schritt 3:
Der Empfänger entschlüsselt den Text mit dem *private key*.

Die Schlüssel werden mit folgendem Algorithmus erzeugt, der auf der Zahlentheorie beruht [**mehr...**].

Zuerst werden zwei Primzahlen p und q gewählt, die für ein sicheres System mehrere hundert Stellen haben sollten. Man multipliziert diese und bildet m = p*q. Aus der Zahlentheorie weiss man, dass die Eulersche Funktion φ(m) = (p-1)*(q-1) die Zahl der teilerfremden Zahlen zu m ist (a, b sind teilerfremd, wenn der grösste gemeinsame Teiler ggT(a,b) = 1 ist).

Als nächstes wählt man eine Zahl e, die kleiner als φ und teilerfremd zu φ ist. Damit ist der öffentliche Schlüssel bereits erstellt, er besteht aus dem Zahlenpaar:

Öffentlicher Schlüssel: [m, e]

Hier ein Beispiel mit den kleinen Primzahlen p = 73 und q = 151:

m = 73 * 151 = 11023, φ = 72 * 150 = 10800, e = 11 (gewählt teilerfremd zu φ)

Öffentlicher Schlüssel: [m, e] = [11023, 11]

Der private Schlüssel besteht dann aus dem Zahlenpaar:

Privater Schlüssel: [m, d]

wobei für die Zahl d muss gelten: (d * e) mod φ = 1

(da e und φ teilerfremd sind, sagt das Lemma von Bézout aus der Zahlentheorie, dass die Gleichung mindestens eine Lösung hat).

Du kannst mit deinen Werten für e und φ die Zahl d mit einem einfachen Programm bestimmen, indem du in einer for-Schleife 100 000 Werte für d ausprobierst.

```
e = 11
phi = 10800

for d in range(100000):
    if  (d * e) % phi == 1:
        print "d", d
```

Du erhältst mehrere Lösungen (5891, 16691, 27491, 49091, usw.). Im Prinzip brauchst du aber nur die erste, um den privaten Schlüssel festzulegen.

Privater Schlüssel: [m, d] = [11023, 5891]

Die Berechnung des privaten Schlüssels ist hier nur deswegen so einfach, da du die Zahlen p und q und damit auch die Zahl φ kennst. Ohne Kenntnis dieser Zahlen lässt sich der private Schlüssel nur mit einem grossen Aufwand berechnen

Mit dem RSA-Algorithmus werden **Zahlen** verschlüsselt. Um einen Text zu verschlüsseln, verwendest du den ASCII-Code jedes Zeichens und bildest mit dem öffentlichen Schlüssel [m, e] den Verschlüsselungswert s für den Geheimtext gemäss der Formel

$s = r^e$ (modulo m).

Diese Verschlüsselungszahlen schreibst du zeilenweise in die Datei *secret.txt*.

```
publicKey = [11023, 11]

def encode(text):
    m = publicKey[0]
    e = publicKey[1]
    enc = ""
    for ch in text:
        r = ord(ch)
        s = int(r**e % m)
        enc += str(s) + "\n"
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Im Decoder liest du die Zahlen von der Datei *secret.txt* zuerst in eine Liste. Zum Entschlüsseln berechnest du mit dem privaten Schlüssel aus der Verschlüsselungszahl s die ursprüngliche Zahl gemäss der Formel

$r = s^d$ (modulo m).

Dies ist der ASCII-Code des ursprünglichen Zeichens.

```
privateKey = [11023, 5891]

def decode(li):
    m = privateKey[0]
    d = privateKey[1]
    enc = ""
    for c in li:
        s = int(c)
        r = s**d % m
        enc += chr(r)
    return enc

fInp = open("secret.txt")
krypto = []
while True:
   line = fInp.readline().rstrip("\n")
   if line == "":
```

```
        break
    krypto.append(line)
fInp.close()

print "Krypto:\n", krypto
msg = decode(krypto)
print "Message:\n", msg

fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Der grosse Vorteil des RSA-Verfahrens besteht darin, dass keine geheime Schlüsselinformation zwischen dem Sender und dem Empfänger ausgetauscht werden muss. Vielmehr generiert der Empfänger sowohl den öffentlichen und den privaten Schlüssel und teilt nur den öffentlichen Schlüssel dem Sender mit, behält aber den privaten Schlüssel bei sich geheim. Der Sender kann nun seine Daten verschlüsseln, aber nur der Empfänger kann sie entschlüsseln [**mehr...**].

In der Praxis wählt man die Primzahlen p und q sehr gross (mehrere Hundert Stellen lang). Die Generierung des öffentlichen Schlüssels erfordert nur die Produktbildung m = p * q, was sehr einfach ist. Will ein Hacker aus dem öffentlichen Schlüssel den privaten Schlüssel herausfinden, so muss er umgekehrt aus m die beiden geheimen Primfaktoren bestimmen. Das Faktorisieren einer langen Zahl ist aber bisher nur mit einem enormen Rechenaufwand möglich. Kryptosysteme nutzen also die Grenzen der Berechenbarkeit.

Grundsätzlich gibt es aber kein absolut sichereres Verschlüsselungsverfahren. Die Verschlüsselung gilt aber bereits dann als sicher, wenn die Zeit für die Entschlüsselung wesentlich länger dauert als die Zeit, während der die Information von Wichtigkeit ist.

AUFGABEN

1. Versuch den Geheimtext
   AV SFX EXSWAXSXAFDTWMFZSZXJFXSTF
   CTFFSTUXJSXJKLSMES TDUFXMFSCGEEXF
   YJXMXSEAV
   ETP
   zu entschlüsseln. Es handelt sich um eine Caesar-Verschlüsselung.

   Bemerkung: Eine Lösungsmöglichkeit besteht darin, davon auszugehen, dass der Buchstabe E in deutschen Texten weitaus am häufigsten vorkommt. Du kannst aber auch alle Verschiebungsmöglichkeiten durchprobieren.

2. Orientiere dich auf dem Internet, was man unter dem Verschlüsselungsverfahren mit einer Skytale versteht und implementiere einen Encoder/Decoder nach diesem Prinzip.

3. Begründe, warum die Caesar-Verschlüsselung ein Spezialfall des Vigenère-Verfahrens ist.

4. Erzeuge mit zwei Primzahlen p und q (beide kleiner als 100) einen öffentlichen und privaten Schlüssel gemäss dem RSA-Verfahren und verschlüssle/entschlüssle damit einen Text.

10.6 ENDLICHE AUTOMATEN

EINFÜHRUNG

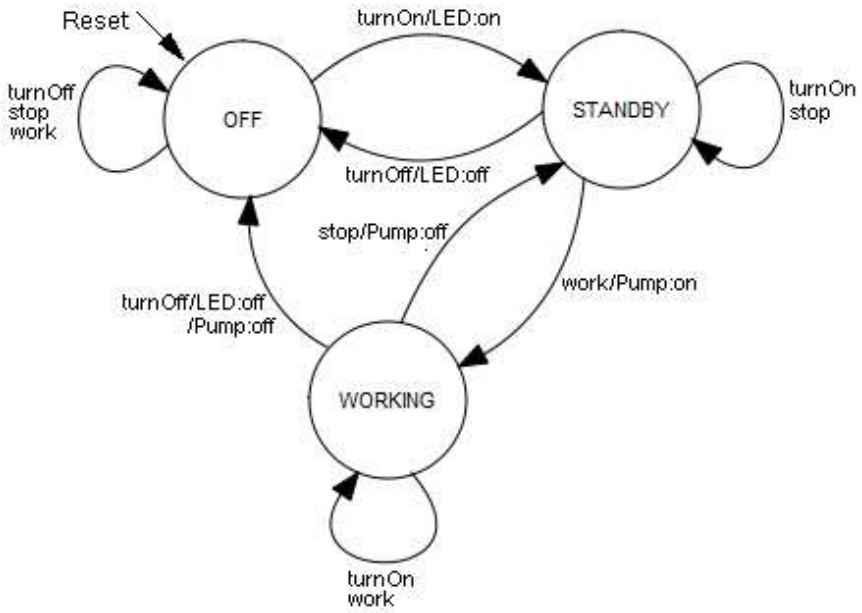
Will man untersuchen, welche Probleme ein Computer grundsätzlich lösen kann und welches seine Grenzen sind, so muss man zuerst exakt definieren, was man unter einer Rechenmaschine versteht. Der berühmte Mathematiker und Informatiker Alan Turing veröffentlichte bereits 1936 eine Untersuchung zu diesem Thema, lange bevor es überhaupt einen programmierbaren Digitalrechner gab. Die nach ihm benannte **Turingmaschine** durchläuft programmgesteuert und auf Grund von Eingabewerten, die sie von einem Band liest, schrittweise einzelne Zustände und schreibt dabei Ausgabewerte auf das Band. Diese grundsätzliche Vorstellung über die Funktionsweise des Computers ist auch heute noch gültig, denn jeder Prozessor ist eigentlich eine Turingmaschine, die im Takt einer Clock Zustand um Zustand durchläuft. Besser an die Praxis angepasst sind allerdings Zustandsautomaten, die sich mit Zustandsgraphen modellieren lassen. Man nennt sie **Endliche Automaten**.

PROGRAMMIERKONZEPTE: *Turingmaschine, Endlicher Automat, Automatengraf, Formale Sprache*

DIE ESPRESSO-MASCHINE ALS MEALY-AUTOMAT

Im täglichen Leben machst du Bekanntschaft mit vielen Geräten und Maschinen, die du als Automaten auffassen kannst. Dazu gehören Getränkeautomaten, Waschautomaten, Geldautomaten, usw. Als Ingenieur und Informatiker entwickelst du einen solchen Automaten mit der klaren Vorstellung, dass dieser ausgehend von einem aktuellen **Zustand** schrittweise in einen **Nachfolgezustand** übergeht, der von Sensordaten und der Betätigung von Tasten und Schaltern abhängt. Dies nennst du die **Eingaben** des Automaten. Bei jedem **Übergang** betätigt der Automat bestimmte Aktoren, wie Motoren, Pumpen, Lampen usw. Dies nennst du die **Ausgaben** des Automaten.

Du entwickelst hier einen Espresso-Automaten, der 3 Zustände besitzt: Er kann ausgeschaltet (OFF), betriebsbereit (STANDBY) und am Kaffeepumpen (WORKING) sein. Zur Bedienung stehen 4 Drucktasten für die Funktionen *Einschalten* (turnOn), *Ausschalten* (turnOff), *Kaffeepumpe einschalten* (work), *Kaffeepumpe ausschalten* (stop) zur Verfügung. Du kannst zwar die Funktionsweise des Espresso-Automaten in Worten beschreiben. Viel anschaulicher ist es aber, den **Automatengraph** zu zeichnen.



Dabei stellst du die **Zustände** mit einem **Kreis** und die **Übergänge** als **Übergangspfeile** dar, die du mit den **Eingaben/Ausgaben** anschreibst. Zudem ist es wichtig festzulegen, in welchem **Anfangszustand** der Automat ist, wenn du ihn ans Netz anschliesst. Da in jedem Zustand irgendeine Taste gedrückt werden kann, müssen bei jedem Zustand alle möglichen Eingaben vorkommen. Falls keine Aktion erfolgt, wird die Ausgabe weggelassen. Du kannst das Verhalten auch in einer Tabelle festhalten, in der du zu jedem Zustand s und jeder Eingabe t den Nachfolgezustand s' angibst. Mit einem Stern bezeichnest du den Anfangszustand.

Übergangstabelle:

t = s =	**OFF(*)**	**STANDBY**	**WORKING**
turnOff	OFF	OFF	OFF
turnOn	STANDBY	STANDBY	WORKING
stop	OFF	STANDBY	STANDBY
work	OFF	STANDBY	WORKING

Mathematisch ausgedrückt kannst du sagen, dass der Nachfolgezustand s' eine Funktion des aktuellen Zustands *s* und der Eingabe t ist: *s' = F(s, t)*. Du nennst F die **Übergangsfunktion**.

Die Ausgaben, die zu jedem Zustand und einer Eingabe gehören, kannst du ebenfalls tabellarisch festhalten:

Ausgabetabelle:

t = s =	**OFF(*)**	**STANDBY**	**WORKING**
turnOff	-	LED off	LED off, Pump off
turnOn	LED on	-	-
stop	-	-	Pump off
work	-	Pump on	-

Mathematisch kannst du auch hier sagen, dass die Ausgabe g eine Funktion des aktuellen Zustands s und der Eingabe ist: *g = G(s, t).* Du nennst G die **Ausgabefunktion**.

MEMO

Die Zustände (mit Auszeichnung des Anfangszustandes), die Eingabe- und Ausgabewerte, sowie die Übergangs- aus Ausgabefunktion bilden zusammen einen sogenannten **Mealy-Automaten**.

IMPLEMENTIERUNG DES ESPRESSO-AUTOMATEN MIT STRINGS

Der Auslöser für den Übergang von einem Zustand zum nächsten soll ein Tastendruck sein. Die betreffende Taste legt den Eingabewert fest, und zwar werden die 4 Cursor-Tasten verwendet. Die Implementierung ist typisch: In einer endlosen Ereignisschleife (event loop) wartet das Programm mit *getKeyEvent()* auf eine Tastatureingabe. Mit dem Rückgabewert wird der aktuelle Zustand gemäss der Übergangstabelle geändert und die Ausgaben gemäss der Ausgabetabelle gemacht.

```
from gconsole import *

def getKeyEvent():
    keyCode = getKeyCodeWait(True)
    if keyCode == KeyEvent.VK_UP:
        return "stop"
```

```
    if keyCode == KeyEvent.VK_DOWN:
        return "work"
    if keyCode == KeyEvent.VK_LEFT:
        return "turnOff"
    if keyCode == KeyEvent.VK_RIGHT:
        return "turnOn"
    return ""

state = "OFF"  # Start state
makeConsole()
while True:
    gprintln("Zustand: " + state)
    entry = getKeyEvent()
    if entry == "turnOff":
       if state == "STANDBY":
            state = "OFF"
            gprintln("LED ausschalten")
       if state == "WORKING":
            state = "OFF"
            gprintln("LED und Pumnpe ausschalten")
    elif entry == "turnOn":
        if state == "OFF":
            state = "STANDBY"
            gprintln("LED einschalten")
    elif entry == "stop":
        if state == "WORKING":
            state = "STANDBY"
            gprintln("Pumpe ausschalten")
    elif entry == "work":
        if state == "STANDBY":
            state = "WORKING"
            gprintln("Pumpe einschalten")
```

MEMO

In der Eventloop werden nur diejenigen Events behandelt, die zu einem Zustandswechsel führen oder die eine Ausgabe erzeugen.

ENUMERATIONEN ALS ZUSTANDS- UND EVENTBEZEICHNER

Da der Automat mit bestimmten Zuständen und bestimmten Eingabe- und Ausgabewerten arbeitet, ist es sinnvoll, dafür eine spezielle Datenstruktur einzuführen. Viele Programmiersprachen kennen dafür einen speziellen **Aufzählungstyp** (**enumeration**). In der Standardsyntax von Python fehlt dieser Datentyp leider, er wurde aber in TigerJython mit dem zusätzlichen Schlüsselwort ***enum()*** hinzugefügt. Bei der Definition der Aufzählungswerte verwendet man Strings. Diese müssen sich an die erlaubte Namensgebung für Variablen halten.

```
from gconsole import *

def getKeyEvent():
    keyCode = getKeyCodeWait(True)
    if keyCode == KeyEvent.VK_UP:
        return Events.stop
    if keyCode == KeyEvent.VK_DOWN:
        return Events.work
    if keyCode == KeyEvent.VK_LEFT:
        return Events.turnOff
    if keyCode == KeyEvent.VK_RIGHT:
        return Events.turnOn
    return None

State = enum("OFF", "STANDBY", "WORKING")
state = State.OFF
Events = enum("turnOn", "turnOff", "stop", "work")
```

```
makeConsole()
while True:
    gprintln("State: " + str(state))
    entry = getKeyEvent()
    if entry == Events.turnOn:
        if state == State.OFF:
            state = State.STANDBY
    elif entry == Events.turnOff:
            state = State.OFF
    elif entry == Events.work:
        if state == State.STANDBY:
            state = State.WORKING
    elif entry == Events.stop:
        if state == State.WORKING:
            state = State.STANDBY
```

MEMO

Es ist Geschmackssache, ob man den zusätzlichen Datentyp *enum* verwenden will. Die Programme werden dadurch nicht kürzer, hingegen übersichtlicher und sicherer, da nur im *enum* definierte Aufzählungswerte vorkommen dürfen.

MAUSGESTEUERTE IMPLEMENTIERUNG DES ESPRESSO-AUTOMATEN

Mit nur wenig zusätzlichem Aufwand kannst du mit der GameGrid-Bibliothek den Espresso-Automat grafisch simulieren, wodurch das Programm stark an Anschaulichkeit gewinnt und das Programmieren mehr Spass macht. Statt der Tastatur werden die 4 Eingaben durch Mausklicks auf simulierte Druckknöpfe ausgelöst und die Ausgaben der LED und der Pumpe unmittelbar mit Spritebildern sichtbar gemacht. An Stelle der Eventloop tritt hier der Callback *pressEvent()*, der immer dann aufgerufen wird, wenn mit der Maus auf das Bild geklickt wird. Da du als *GameGrid* ein Gitter mit 7 x 11 Zellen verwendet, kannst du die Klicks auf die Druckknöpfe mit Gitterkoordinaten erfassen.

```
from gamegrid import *

def pressEvent(e):
    global state
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc == Location(1, 2): # off
        state = State.OFF
        led.show(0)
        coffee.hide()
    elif loc == Location(2, 2): # on
        if state == State.OFF:
            state = State.STANDBY
            led.show(1)
    elif loc == Location(4, 2): # stop
        if state == State.WORKING:
            state = State.STANDBY
            coffee.hide()
    elif loc == Location(5, 2): # work
        if state == State.STANDBY:
            state = State.WORKING
```

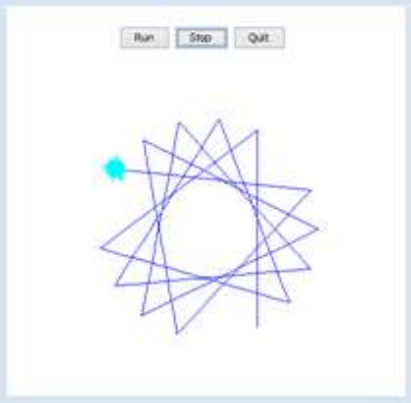
```
            coffee.show()
    setTitle("State: " + str(state))
    refresh()

State = enum("OFF", "STANDBY", "WORKING")
state = State.OFF
makeGameGrid(7, 11, 50, None, "sprites/espresso.png", False,
             mousePressed = pressEvent)
show()
setTitle("State: " + str(state))
led = Actor("sprites/lightout.gif", 2)
addActor(led, Location(3, 3))
coffee = Actor("sprites/coffee.png")
addActor(coffee, Location(3, 6))
coffee.hide()
refresh()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

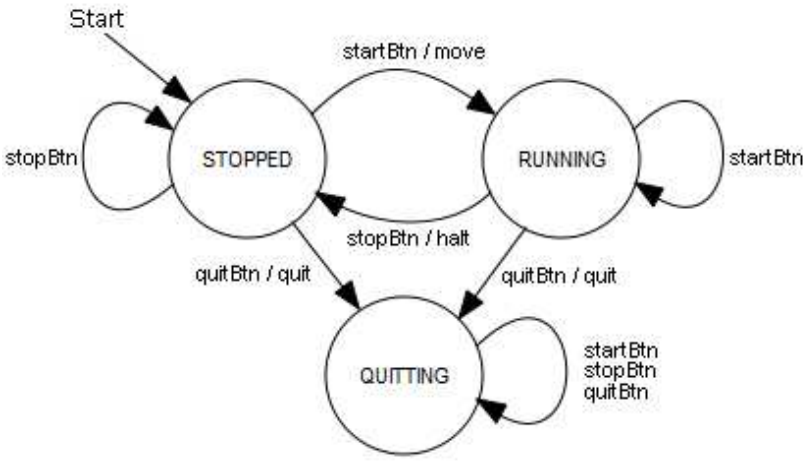
MEMO

Eine Simulation gewinnt durch eine grafische Benutzeroberfläche viel an Anschaulichkeit und Attraktivität.

DENKEN IN ZUSTÄNDEN BEI GRAFISCHEN BENUTZEROBERFLÄCHEN



Auf den ersten Blick scheinen Mealy-Automaten eine eher theoretische Angelegenheit zu sein. Dies ist aber keineswegs so. Vielmehr muss man bei modernen, eventgesteuerten Programmen mit einer grafischen Benutzeroberfläche **immer in Zuständen denken**. Als Beispiel schreibst du ein Turtle-Programm, das über 3 Buttons gesteuert wird: Der *Startbutton* setzt die Turtle in Bewegung, der *Stopbutton* hält die Bewegung an und der *Quitbutton* beendet das Programm. Um das Programm richtig zu implementieren, musst du den Automatengraph im Kopf haben:



Wie du weisst, dürfen in Callbacks von GUI-Events keine Animationen und nur kurz dauernder Code ausgeführt werden, da der Bildschirm nur am Ende der Funktion neu gerendert wird. Darum schaltest du in den Callbacks der Buttonklicks nur den Zustand um und führst die Bewegung der Turtle im Hauptteil des Programms aus.

Mehr zu diesem Problem erfährst du im Anhang 4: **Parallelverarbeitung**

```
from javax.swing import JButton
from gturtle import *

def buttonCallback(evt):
    global state
    source = evt.getSource()
    if source == runBtn:
        state = State.RUNNING
        setTitle("State: RUNNING")
    if source == stopBtn:
        state = State.STOPPED
        setTitle("State: STOPPED")
    if source == quitBtn:
        state = State.QUITTING
        setTitle("State: QUITTING")

State = enum("STOPPED", "RUNNING", "QUITTING")
state = State.STOPPED

runBtn = JButton("Run", actionPerformed = buttonCallback)
stopBtn = JButton("Stop", actionPerformed = buttonCallback)
quitBtn = JButton("Quit", actionPerformed = buttonCallback)
makeTurtle()
setTitle("State: STOPPED")
back(100)

pg = getPlayground()
pg.add(runBtn)
pg.add(stopBtn)
pg.add(quitBtn)
pg.validate()

while state != State.QUITTING and not isDisposed():
    if state == State.RUNNING:
          forward(200).left(127)
dispose()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Diese Programmstruktur ist für eventgesteuerte Programme typisch und du solltest sie dir gut merken.

Du musst das Package *JButton* importieren, um die Buttons zu verwenden. Mit *add()* fügst du sie in das Turtlefenster (den Playground). Damit sie sichtbar werden, musst du das Turtlefenster mit *validate()* neu rendern.

AUFGABEN

1. Ein Parkscheinautomat akzeptiert nur 1 € und 2 € Geldstücke, die einzeln hintereinander eingeworfen werden. Sobald der Automat mindestens die Parkgebühr erhalten hat, gibt er den Parkschein und das Restgeld aus. Die Parkgebühr betrage 3 €.

   Der Automat durchlaufe ausgehend vom Startzustand S0 die Zustände S1 oder S2, je nachdem ob 1 € oder 2 € eingeworfen werden. Seine Ausgabewerte sind - (nichts), K (Karte) oder K,R (Karte und Rückgeld).

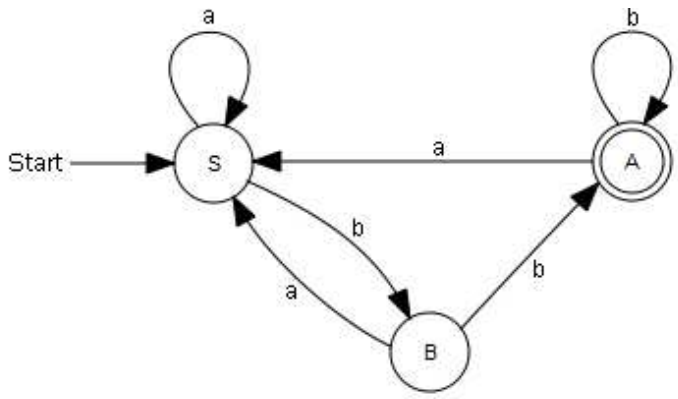
a. Erstelle die Übergangs- und Ausgabetabellen
b. Zeichne den Automatengraphen
c. Erstelle ein Programm mit GConsole, welches das Drücken der Zahlentaste 1 als Einwurf von 1 € und das Drücken der Zahlentaste 2 als Einwurf von 2 € interpretiert und den Folgezustand, sowie die Ausgabewerte in das Konsolenfenster ausschreibt.

ZUSATZSTOFF

AKZEPTOR FÜR REGULÄRE SPRACHEN

Eine formale Sprache besteht aus einem Alphabet von Zeichen und einem Regelsystem, mit dem man eindeutig entscheiden kann, ob eine bestimmte Zeichensequenz mit Zeichen aus diesem Alphabet zur Sprache gehört. Kann man das Regelsystem mit einem Automaten realisieren, so spricht man von einer **regulären formalen Sprache**.

Du betrachtest als Beispiel eine sehr einfache Sprache mit einem Alphabet, das nur aus den Buchstaben a und b besteht. Das Regelsystem kannst du als Spezialfall eines Mealy-Automaten auffassen, der keine Ausgabewerte erzeugt. Dabei liest der Automat ausgehend von einem Startzustand Zeichen um Zeichen und geht entsprechend des gelesenen Zeichens in einen Nachfolgezustand über. Befindet er sich nach dem Lesen des letzten Zeichens in einem der vorgegebenen Endzustände, so gehört das Wort zur Sprache. Du betrachtest den folgenden Automatengraphen (S: Startzustand, A: Endzustand):



In der Implementierung wird der Zustandswechsel durch Drücken der Buchstabentasten a oder b ausgelöst. Danach schreibt dein Programm den aktuellen Zustand und das bisher eingegebene Wort aus.

```
from gconsole import *

def getKeyEvent():
    global word
    keyCode = getKeyCodeWait(True)
    if keyCode == KeyEvent.VK_A:
        return Events.a
    if keyCode == KeyEvent.VK_B:
        return Events.b
    return None

State = enum("S", "A", "B")
state = State.S
Events = enum("a", "b")
makeConsole()
word = ""
gprintln("State: " + str(state))
while True:
    entry = getKeyEvent()
```

```
    if entry == Events.a:
        if state == State.A:
            state = State.S
        elif state == State.B:
            state = State.S
        word += "a"
        gprint("Word: " + word + " -> ")
        gprintln("State: " + str(state))
    elif entry == Events.b:
        if state == State.S:
            state = State.B
        elif state == State.B:
            state = State.A
        word += "b"
        gprint("Word: " + word + " -> ")
        gprintln("State: " + str(state))
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Ein Akzeptor prüft, ob ein Wort zu einer Sprache gehört. Er ist ein Spezialfall eines Mealy-Automaten ohne Ausgabewerte. Das Wort gehört dann zur Sprache, wenn man ausgehend vom Startzustand S nach dem Lesen aller Buchstaben bei einem Endzustand A ankommt. Beispielsweise gehört *abbabb* zur Sprache, hingegen *baabaa* nicht.

AUFGABEN

1. Ein Lachautomat soll nur die Wörter *ha.* oder *haha.* oder *hahaha.* usw. (letztes Zeichen ein Punkt) akzeptieren. Zeichne den Automatengraph und implementiere ihn.
   Anleitung: Du kannst einen Fehlerzustand E einführen, von dem man bei beliebiger Eingabe nicht mehr wegkommt.

10.7 INFORMATION & ORDNUNG

EINFÜHRUNG

Obschon das Wort *Information* oft verwendet wird, ist es gar nicht so leicht, den Begriff exakt zu fassen und messbar zu machen. Im täglichen Leben wird *mehr Information* mit *mehr Wissen* verbunden und gesagt, dass eine informierte Person A in einer gewissen Sache mehr weiss als eine nicht informierte Person B.

Um die Mehrinformation von A gegenüber B (oder dem Team B) bzw. den Informationsmangel von B gegenüber A zu messen, stellst du dir am besten ein TV-Fragespiel vor. Dabei muss eine Person B oder ein Team etwas herausfinden, was nur die Person A weiss, beispielsweise ihren Beruf. Dabei stellt B Fragen, die A stets mit Ja oder Nein beantworten muss. Man definiert:

Unter dem Informationsmangel I von B gegenüber A (in bit) versteht man die Anzahl Fragen mit Ja/Nein-Antwort, die B (im Mittel und bei optimaler Fragestrategie) stellen muss, um über eine bestimmte Sache das gleiche Wissen wie A zu haben.

PROGRAMMIERKONZEPTE: *Information, Informationsgehalt, Entropie*

ZAHLEN-RATESPIEL

Ein solches Ratespiel kannst du (auch nur in Gedanken) in deiner Schulklasse wie folgt inszenieren: Eure Klassenkameradin Judith wird vor die Tür geschickt. Einer der verbleibenden W = 16 Kameraden, erhält ein "begehrliches" Objekt, beispielsweise einen Schokoladenriegel. Nachdem Judith wieder in die Schulstube zurückgerufen wird, wissen du und deine Kameraden, wer der Schokoladenbesitzer ist, Judith aber nicht. Wie gross ist ihr Informationsmangel?

Der Einfachheit halber werden die Kameraden von 0 bis 15 numeriert und Judith kann euch nun Fragen stellen, um die geheime Zahl des Schokoladenbesitzers herauszufinden. Je weniger Fragen sie dazu braucht, je besser. Eine Möglichkeit wäre, irgendwelche zufälligen Zahlen abzufragen: "Ist es 13?" und wenn eine Nein-Antwort kommt, nach einer anderen Zahl zu fragen.

Judith könnte auch systematisch vorgehen und von 0 an fragen: "Ist es die 0?", dann "Ist es die 1?", usw. In deiner Computersimulation bestimmst du, wie viele Fragen sie mit dieser Frageart benötigt, um **im Mittel** (d.h. wenn man das Spiel oft spielt) die Geheimzahl herauszufinden. Dabei überlegst du Folgendes:

Wenn die Geheimzahl 0 ist, so braucht man 1 Frage, ist sie 1, so sind 2 Fragen nötig, usw. Wenn die Geheimzahl 14 ist, so sind 15 Fragen nötig, aber wenn sie 15 ist, so sind ebenfalls 15 Fragen nötig, da man bei der 15. Frage "Ist es 14?" Nein als Antwort erhält und weiss, dass es die 15 ist. Das Programm spielt das Spiel 100000 mal und zählt jeweils die Anzahl Fragen, bis die Geheimzahl ermittelt wurde. Die Zahl der Fragen wird aufsummiert, um zuletzt den Mittelwert zu bestimmen.

```
import random

sum = 0
z = 100000
repeat z:
    n = random.randint(0, 15)
    if n != 15:
```

```
            q = n + 1 # number of questions
        else:
            q = 15
        sum += q
    print "Mean:", sum / z
```

Mit dieser Fragestrategie würde Judith im Mittel 8.43 Fragen benötigen. Das ist sehr viel. Da aber Judith gescheit ist, wählt sie eine viel bessere Fragestrategie. Sie fragt euch: *"Ist es eine Zahl zwischen 0 und 7?"* (Grenzen eingeschlossen). Falls ihr *Ja* sagst, so teilt sie den Bereich wieder in zwei gleich grosse Teile und fragt: *"Ist die Zahl zwischen 0 und 3? "*, falls ihr *Ja sagt*, so fragt sie *"Ist es 0 oder 1?".* Falls ihr *Ja sagt*, so fragt sie *"Ist es 0?"* und sie kennt die Zahl. Mit dieser "binären" Fragestrategie muss Judith bei W = 16 immer genau 4 Fragen stellen, also ist 4 auch die mittlere Anzahl Fragen. Die binäre Fragestrategie ist optimal und die Information über den Riegelbesitzers hat daher den Wert I = 4 bit.

MEMO

Wie du leicht überlegen kannst, beträgt die Information bei W = 32 Zahlen I = 5 bit und bei W = 64 Zahlen I = 6 bit. Es gilt also offenbar $2^I = W$ oder $I = ld(W)$. Es braucht sich auch nicht um eine Zahl zu handeln, nach der man sucht, sondern es kann sich um irgend einen Zustand handeln, den man aus W (gleichwahrscheinlichen) Zuständen herausfinden muss.

Unter der **Information** bei Kenntnis eines Zustands aus W gleichwahrscheinlichen Zuständen versteht man

$$\boldsymbol{I = ld(W)}$$ (ld ist der Logarithmus dualis, Logarithmus zur Basis 2)

INFORMATIONSGEHALT EINES WORTS

Da die W Zustände gleichwahrscheinlich sind, ist die Wahrscheinlichkeit für einen der Zustände p = 1/W und du kannst die Information auch so schreiben:

$$I = ld(1/p) = -ld(p)$$

Es ist dir sicher bekannt, dass die Wahrscheinlichkeit für einen bestimmten Buchstaben in einer gesprochenen Sprache ganz unterschiedlich ist. Die folgende Tabelle zeigt die Wahrscheinlichkeiten für die deutsche Sprache [**mehr...**].

A	5.58%	I	8.02%	Q	0.02%	Y	0.05%
B	1.96%	J	0.24%	R	6.89%	Z	1.21%
C	3.16%	K	1.32%	S	6.42%	Ä	0.54%
D	4.98%	L	3.60%	T	5.79%	Ö	0.30%
E	16.93%	M	2.55%	U	3.83%	Ü	0.65%
F	1.49%	N	10.53%	V	0.84%	ß	0.37%
G	3.02%	O	2.24%	W	1.78%		
H	4.98%	P	0.67%	X	0.05%		

Es ist natürlich nicht so, dass die Wahrscheinlichkeit für einen Buchstaben in einem Wort unabhängig davon ist, welche Buchstaben des Worts man bereits kennt und in welchem Zusammenhang das Wort steht. Macht man aber diese vereinfachende Annahme, so gilt für die Wahrscheinlichkeit p einer Buchstabenkombination aus zwei Buchstaben mit den Einzelwahrscheinlichkeiten p_1 und p_2 gemäss der Produktregel $p = p_1 * p_2$ und für die Information

$$I = -ld(p) = -ld(p_1 * p_2) = -ld(p_1) - ld(p_2)$$

oder für beliebig viele Buchstaben:

$$I = - ld(p_1) - ld(p_2) - ld(p_3) - ... - ld(p_n) = - \Sigma ld(p_i)$$

In deinem Programm kannst du ein Wort eingeben und du erhältst als Ausgabe den Informationsgehalt des Worts. Dabei solltest du die Dateien mit den Buchstabenhäufigkeiten in deutsch, englisch und französisch von **hier** downloaden und in das Verzeichnis kopieren, in dem dein Python-Programm ist.

```
import math

f = open("letterfreq_de.txt")
s = "{"
for line in f:
    line = line[:-1] # remove trailing \n
    s += line + ", "
f.close()
s = s[:-2]  # remove trailing ,
s += "}"
occurance = eval(s)

while True:
    word = inputString("Gib ein Wort ein")
    I = 0
    for letter in word.upper():
        p = occurance[letter] / 100
        I -= math.log(p, 2)
    print word, "-> I =", round(I, 2), "bit"
```

MEMO

Die Daten sind in den Textdateien zeilenweise im Format 'Buchstabe' : Prozentzahl abgespeichert. Als Python-Datenstruktur ist ein Dictionary gut geeignet. Um die Dateiinformation in ein Dictionary umzuwandeln, werden die Zeilen eingelesen und in einen String im üblichen Dictionary-Format {key : value, key : value,...} gepackt. Mit *eval()* wird dieser String als Programmzeile interpretiert und ein Dictionary erstellt.

Wie du mit deinem Programm herausfinden kannst, ist der Informationsgehalt für Wörter mit seltenen Buchstaben grösser. Der so bestimmte Informationsgehalt hat allerdings nichts damit zu tun, wie wichtig ein bestimmtes Wort in einem bestimmten Zusammenhang für dich sein kann, ob also die Information, die mit diesem Wort verbunden ist, für dich persönlich belanglos oder von entscheidender Bedeutung ist. Ein Mass dafür zu bestimmen liegt weit ausserhalb der Möglichkeiten von heutigen Informationssystemen.

AUFGABEN

1. Beim Ratespiel mit 16 Kameraden könnte die clevere Schulkameradin auch anders fragen, indem sie sagt, dass du dir die Nummer des Riegelbesitzers als Binärzahl mit 4 Ziffern vorstellen sollst. Schreib auf, wie die Fragen heissen.

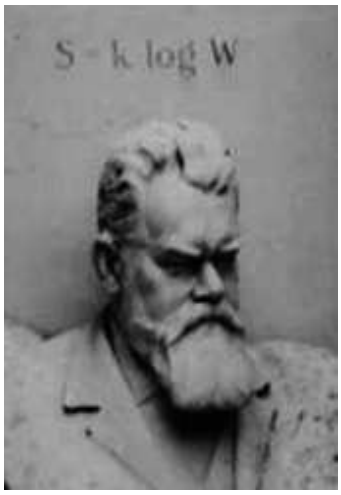
2. Verwende die Liste der deutschen Wörter aus der Datei worte-1.txt und bestimme bei einer Wortlänge von 5 das Wort mit dem kleinsten und grössten Informationsgehalt. Kommentiere das Resultat. Download der Wortlisten von **hier**.

3*. Bestimme die mittlere Anzahl Fragen beim Zahlenraten mit den Zahlen 0..15 und der Fragestrategie "Ist es die 0, ist es die 1, usw." aus einer theoretischen Überlegung..

DER ZUSAMMENHANG ZWISCHEN UNORDNUNG UND ENTROPIE

Es gibt einen äusserst interessanten Zusammenhang zwischen der Information, die wir über ein System haben und der Ordnung des Systems. Sitzen beispielsweise in einem Klassenzimmer alle Schülerinnen und Schüler an ihren Pulten, so ist die Ordnung sicher viel grösser als wenn sie sich wahllos im Zimmer herumbewegen. Im ungeordneten Zustand ist unser Informationsmangel, wo sich jede einzelne Person befindet, sehr gross, darum kann man diesen Informationsmangel als Mass für die Unordnung heranziehen. Dazu definiert man den Begriff der Entropie:

*Unter der Entropie (in bit) verstehen wir den Informationsmangel I, den wir bei einer makroskopischen Beschreibung des Systems gegenüber jemandem besitzen, der den mikroskopischen Zustand kennt. Für die Entropie in (J/K) setzen wir S = k * ln2 * I*



Bis auf einen Vorfaktor sind also Entropie und Informationsmangel dasselbe. Gehen wir von einem System mit W gleichwahrscheinlichen Zuständen aus, so haben wir

$S = k * ln2 * ldW$ oder **$S = k * \log W$**

Diese fundamentale Beziehung stammt vom grossen Physiker Ludwig Boltzmann (1844-1900) und steht auf seiner Grabinschrift.

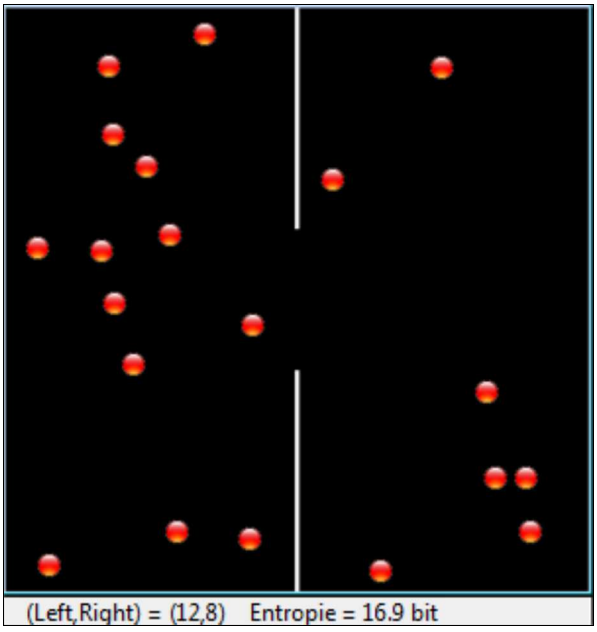
Wie du feststellen kannst, haben sich selbst überlassene Systeme die Tendenz, von einem geordneten in einen ungeordneten Zustand überzugehen. Beispielsweise:

- Passagiere in einem Eisenbahnwagen verteilen sich über den ganzen Wagen
- Zigarettenrauch verteilt sich im Zimmer
- Tintenklecks verteilt sich im Wasserglas
- Temperatur zwischen Kaffee und Tasse gleicht sich aus

Da der ungeordnete Zustand eine höhere Entropie als der geordnete hat, kann man dies als Naturgesetz (2. Hauptsatz der Thermodynamik) formulieren:

In einem abgeschlossenen System nimmt die Entropie zu oder bleibt gleich, nimmt aber nie ab.

Mit deinem Programm simulierst du ein Teilchensystem, bei dem die Teilchen wie Gasatome miteinander zusammenstossen und dabei ihre Richtung und ihre Bewegungsenergie austauschen. Zuerst befinden sich alle Teilchen im linken Teil eines Behälters, der eine Trennwand mit einem Loch aufweist. Was passiert?



Für die Animation wird das Modul *GameGrid* herangezogen. Die Teilchen sind in der Klasse *Particle* modelliert, die aus der Klasse *Actor* abgeleitet ist. In der Methode *act()*, die in jedem Simulationszyklus automatisch aufgerufen wird, bewegst du die Teilchen. Die Kollisionen werden mit Kollisionsevents behandelt. Dazu leitest du die Klasse *CollisionListener* aus *GGActorCollisionListener* ab und überschreibst die Methode *collide()*. Das Vorgehen ist das

Gleiche wie im Kapitel 8.10 **Brownsche Bewegung**. Die 20 Teilchen werden in 4 verschiedene Geschwindigkeitsgruppen eingeteilt. Da beim Zusammenstoss die Geschwindigkeiten ausgetauscht werden, bleibt die Gesamtenergie des Systems konstant, d.h. das System ist abgeschlossen.

```
from gamegrid import *
from gpanel import *
import math
import random

# ================== class Particle ====================
class Particle(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/ball_0.gif")

    # Called when actor is added to gamegrid
    def reset(self):
        self.isLeft = True

    def advance(self, distance):
        pt = self.gameGrid.toPoint(self.getNextMoveLocation())
        dir = self.getDirection()
        # Left/right wall
        if pt.x < 0 or pt.x > w:
            self.setDirection(180 - dir)
        # Top/bottom wall
        if pt.y < 0 or pt.y > h:
            self.setDirection(360 - dir)
        # Separation
        if (pt.y < h // 2 - r or pt.y > h // 2 + r) and \
            pt.x > self.gameGrid.getPgWidth() // 2 - 2 and \
            pt.x < self.gameGrid.getPgWidth() // 2 + 2:
                self.setDirection(180 - dir)

        self.move(distance)
        if self.getX() < w // 2:
            self.isLeft = True
        else:
            self.isLeft = False

    def act(self):
        self.advance(3)

    def atLeft(self):
        return self.isLeft

# ================== class CollisionListener =========
class CollisionListener(GGActorCollisionListener):
    # Collision callback: just exchange direction and speed
    def collide(self, a,  b):
        dir1 = a.getDirection()
        dir2 = b.getDirection()
        sd1 = a.getSlowDown()
        sd2 = b.getSlowDown()
        a.setDirection(dir2)
        a.setSlowDown(sd2)
        b.setDirection(dir1)
        b.setSlowDown(sd1)
        return 5  # Wait a moment until collision is rearmed


# ================== Global sections =================
def drawSeparation():
    getBg().setLineWidth(3)
    getBg().drawLine(w // 2, 0, w // 2, h // 2 - r)
    getBg().drawLine(w // 2, h, w // 2, h // 2 + r)

def init():
```

```
    collisionListener = CollisionListener()
    for i in range(nbParticles):
        particles[i] = Particle()

        # Put them at random locations, but apart of each other
        ok = False
        while not ok:
            ok = True
            loc = getRandomLocation()
            if loc.x > w / 2 - 20:
                ok = False
            continue

        for k in range(i):
            dx = particles[k].getLocation().x - loc.x
            dy = particles[k].getLocation().y - loc.y
            if dx * dx + dy * dy < 300:
                ok = False
        addActor(particles[i], loc, getRandomDirection())
        delay(100)

        # Select collision area
        particles[i].setCollisionCircle(Point(0, 0), 8)
        # Select collision listener
        particles[i].addActorCollisionListener(collisionListener)

        # Set speed in groups of 5
        if i < 5:
            particles[i].setSlowDown(2)
        elif i < 10:
            particles[i].setSlowDown(3)
        elif i < 15:
            particles[i].setSlowDown(4)

    # Define collision partners
    for i in range(nbParticles):
        for k in range(i + 1, nbParticles):
            particles[i].addCollisionActor(particles[k])

def binomial(n, k):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    k = min(k, n - k) # take advantage of symmetry
    c = 1
    for i in range(k):
        c = c * (n - i) / (i + 1)
    return c

r = 50  # Radius of hole
w = 400
h = 400
nbParticles = 20
particles = [0] * nbParticles
makeGPanel(Size(600, 300))
window(-6, 66, -2, 22)
title("Entropy")
windowPosition(600, 20)
drawGrid(0, 60, 0, 20)
makeGameGrid(w, h, 1, False)
setSimulationPeriod(10)
addStatusBar(20)
drawSeparation()
setTitle("Entropy")
show()
init()
doRun()
```

```
t = 0
while not isDisposed():
    nbLeft = 0
    for particle in particles:
        if particle.atLeft():
            nbLeft += 1
    entropy = round(math.log(binomial(nbParticles, nbLeft), 2), 1)
    setStatusText("(Left,Right) = (" + str(nbLeft) +
           "," + str(nbParticles - nbLeft) + ")" +
           "    Entropie = " + str(entropy) + " bit")
    if t % 60 == 0:
        clear()
        lineWidth(1)
        drawGrid(0, 60, 0, 20)
        lineWidth(3)
        move(0, entropy)
    else:
        draw(t % 60, entropy)
    t += 1
    delay(1000)
dispose() # GPanel
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Ein Zustand wird von Aussen gesehen (makroskopisch) durch die Anzahl k der Teilchen im rechten Teil und damit N - k im linken Teil bestimmt. Bei makroskopischer Betrachtung fehlt die genaue Kenntnis, welche k der N als nummeriert gedachten Teilen sind rechts befinden. Gemäss der Kombinatorik gibt es

$$W = \binom{N}{k} = \frac{N!}{k! \cdot (N-k)!}$$

solche Möglichkeiten. Der Informationsmangel ist daher

$$I = ld(W) = ld\binom{N}{k}$$ bzw. die Entropie $$S = \log(w) = k \cdot log\binom{N}{k}$$

Der zeitliche Verlauf wird in einer GPanel-Grafik aufgetragen. Man sieht deutlich, dass sich die Teilchen im Laufe der Zeit über den ganzen Container verteilen, es aber durchaus sein kann, dass sich wieder alle Teilchen in der linken Hälfte befinden. Die Wahrscheinlichkeit dazu nimmt mit zunehmender Teilchenzahl aber rapide ab. Der 2. Hauptsatz beruht also auf einer statistischen Eigenschaft von Vielteilchensystemen.

Da der umgekehrte Ablauf nicht beobachtet wird, nennt man diese Prozesse **irreversibel**. Es gibt aber durchaus Prozesse, die von Unordnung zu Ordnung übergeben. Dazu braucht es aber einen "ordnenden Zwang" [**mehr...**].

Beispiele sind:

- Die Materie des Kosmos bildet Galaxien, Sterne und Planetensysteme
- Das Leben entsteht aus toter Materie
- Bei genügend Zwang wird die Unordnung in der Küche, im Schulzimmer kleiner
- Beim Abkühlen von Wasserdampf (Gas) entstehen Wolken (Flüssigkeit), dann Eis (Festkörper). Phasenübergänge verändern also die Ordnung
- Nach der Pause strömen die Konzertbesucher wieder an ihre Plätze

Bei diesen Prozessen nimmt die Unordnung, also die Entropie ab und ausgehend von einem Chaos werden Strukturen sichtbar.

AUFGABEN

1. Modifiziere die Gassimulation so, dass nach einer Zeit von 50 Sekunden ein "Dämon" dafür sorgt, dass die Teilchen nur noch von rechts nach links, aber nicht mehr von links nach rechts durch das Loch fliegen. Zeige, dass nun die Entropie abnimmt.

2. Finde weitere Beispiele von Systemen, die

   a. von Ordnung zu Unordnung übergehen

   b. von Unordnung zu Ordnung übergeben. Gib den ordnenden Zwang an.

kapitel elf

ANHANG

Lernziele

- Du erweiterst deine Kenntnisse über Algorithmen und deren Implementierung in Python.
- Du weisst, in welchem Zusammenhang besonders häufig Programmierfehler gemacht werden.
- Du kennst wichtige Verfahren zur Fehlerbehebung (Debugging).
- Du weisst, was man unter Parallelverarbeitung versteht und kannst einfache Programme mit eigenen Threads schreiben.
- Du kannst beschreiben, was man unter Race Conditions und Deadlock versteht.

11.1 VERGNÜGLICHE DENKSPIELE

EINFÜHRUNG

Denkspiele mit mathematischem Hintergrund sind sehr beliebt und weit verbreitet. Es geht in diesem Kapitel nicht darum, dir die Freude am eigenständigen Lösen solcher Rätsel "von Hand", also mit Papier und Schreibstift zu verderben. Vielmehr sollst du erleben, dass der Computer durch systematisches Lösen mit Backtracking die Lösung zwar auch finden kann, allerdings mit zwei wichtigen Unterschieden: Falls im Lösungsprozess nicht weitere Einschränkungen und Strategien eingebaut werden, kann zur Lösung trotz eines schnellen Computers enorm viel Zeit benötigt werden, im Gegenzug kann aber der Computer im Prinzip **alle** Lösungen finden, was von Hand meist sehr schwierig ist. Dadurch ist es möglich, mit dem Computer den Beweis zu liefern, dass eine bestimmte Lösung die einfachste (kürzeste) ist.

SUDOKU

Das Spiel hat sich seit 1980 boomartig verbreitet und ist heute sehr populär. Du findest Spielvorgaben in der Rätselecke von vielen Tages- und Wochenzeitungen. Es handelt sich um ein typisches Zahlenrätsel mit einfachen Spielregeln. In der Standardvariante müssen in einem 9x9 Gitter die Zahlen 1 bis 9 derart in die Felder gesetzt werden, dass in jeder Zeile und jeder Spalte alle Zahlen genau einmal vorkommen. Zudem wird das Gitter in 9 Untergitter mit 3x3 Feldern aufgeteilt, wobei auch in diesen die Zahlen genau einmal auftreten müssen. Zu Beginn des Spiels ist bereits eine bestimmte Anzahl der Zellen besetzt. Bei idealer Spielvorgabe sollte es genau eine Lösung geben.

Je nach Spielvorgabe ist das Rätsel leichter oder schwieriger zu lösen. Der geübte Sudoku-Spieler geht mit gewissen allgemein bekannten oder auch persönlichen Lösungsstrategien vor. Beim Backtracking mit dem Computer werden systematisch die offenen Felder der Reihe nach mit Zahlen von 1 bis 9 so gefüllt, dass sich kein Widerspruch zu den Spielregeln ergibt. Ist dies nicht möglich, so wird der letzte Zug zurückgenommen.

5	6	3	7	9	8	4	1	2
7	9	4	1	2	5	3	6	8
2	8	1	4	6	3	9	5	7
3	4	7	2	1	9	5	8	6
9	5	6	3	8	7	2	4	1
8	1	2	5	4	6	7	3	9
6	3	9	8	7	4	1	2	5
1	7	5	6	3	2	8	9	4
4	2	8	9	5	1	6	7	3

Der Computer verwendet im Folgenden den Backtracking-Algorithmus. Für die grafische Darstellung eignet sich ein *GameGrid* hervorragend, da das Spiel eine Gitterstruktur aufweist. Die Vorgabezahlen zeichnest du als *TextActor* schwarz ein, die eingesetzten Zahlen sind rot. Wie du aus dem **Kapitel 10.3.** über Backtracking weisst, musst du zuerst eine günstige Datenstruktur für die Spielzustände finden. Da es sich hier um ein 9x9-Gitter handelt, wählst du eine Liste mit den 9 Zeilenlisten. Die Zahl 0 kennzeichnet ein leeres Feld. Das hier gezeigte Spiel hat also zu Spielbeginn folgenden Zustand:

```
startState = [ \
[0, 6, 0, 7, 9, 8, 0, 1, 2],
[7, 9, 4, 1, 0, 5, 0, 6, 8],
[2, 0, 1, 4, 0, 0, 9, 5, 7],
[0, 0, 0, 2, 1, 0, 5, 0, 0],
[0, 5, 6, 3, 0, 0, 2, 4, 1],
[0, 1, 2, 5, 4, 0, 7, 3, 9],
[6, 3, 0, 8, 7, 4, 0, 0, 0],
```

```
[1, 0, 5, 6, 0, 2, 8, 0, 0],
[4, 2, 8, 9, 0, 1, 0, 7, 0]]
```

Wie gewohnt wirst du das Programm schrittweise entwickeln. Deine erste Aufgabe besteht darin, den Spielzustand im *GameGrid* darzustellen und dem Benutzer die Möglichkeit zu geben, mit einem Mausklicks die leeren Zellen zu belegen. Dies ist zwar für die automatische Lösungssuche überflüssig, erlaubt dir aber, interaktiv auf das Spiel einzuwirken, was besonders in der Testphase hilfreich ist. Zudem kannst du so das Rätsel am Bildschirm statt auf Papier lösen.

```
from gamegrid import *

def pressEvent(e):
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in fixedLocations:
        setStatusText("Location fixed")
        return
    x = loc.x
    y = loc.y
    value = startState[y][x]
    value = (value + 1)  % 10
    startState[y][x] = value
    showState(startState)

def showState(state):
    removeAllActors()
    for y in range(9):
        for x in range(9):
            loc = Location(x, y)
            value = state[y][x]
            if  value != 0:
                if loc in fixedLocations:
                    c = Color.black
                else:
                    c = Color.red
                digit = TextActor(str(value), c, Color.white,
                        Font("Arial", Font.BOLD, 20))
                addActorNoRefresh(digit, loc)
    refresh()
makeGameGrid(9, 9, 50, Color.red, False, mousePressed = pressEvent)
show()
setTitle("Sudoku")
setBgColor(Color.white)
getBg().setLineWidth(3)
getBg().setPaintColor(Color.red)
for x in range(4):
    getBg().drawLine(150 * x, 0, 150 * x, 450)
for y in range(4):
    getBg().drawLine(0, 150 * y, 450, 150 * y)

startState = [
[0, 6, 0, 7, 9, 8, 0, 1, 2],
[7, 9, 4, 1, 0, 5, 0, 6, 8],
[2, 0, 1, 4, 0, 0, 9, 5, 7],
[0, 0, 0, 2, 1, 0, 5, 0, 0],
[0, 5, 6, 3, 0, 0, 2, 4, 1],
[0, 1, 2, 5, 4, 0, 7, 3, 9],
[6, 3, 0, 8, 7, 4, 0, 0, 0],
[1, 0, 5, 6, 0, 2, 8, 0, 0],
[4, 2, 8, 9, 0, 1, 0, 7, 0]]

fixedLocations = []
for x in range(9):
    for y in range(9):
        if startState[y][x] != 0:
            fixedLocations.append(Location(x, y))
showState(startState)
```

Als nächstes baust du eine Funktion *isValid(state)* ein, die überprüft, ob ein bestimmter Spielzustand *state* die Spielregeln befolgt. Dies ist eine reine Knochenarbeit, da du einfach die 9 Zeilen und 9 Spalten, sowie die 9 quadratischen Blöcke überprüfen musst. Du gehst so vor, dass du jeweils die bereits vorhandenen Zahlen in eine Liste *values* schreibst. Kommt die Zahl bereits darin vor, so handelt es sich um einen illegalen Spielzustand. Das Resultat schreibst du in der Statusbar aus.


```
from gamegrid import *

def pressEvent(e):
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in fixedLocations:
        setStatusText("Location fixed")
        return
    xs = loc.x // 3
    ys = loc.y // 3
    x = loc.x % 3
    y = loc.y % 3
    value = startState[ys][xs][y][x]
    value = (value + 1)  % 10
    startState[ys][xs][y][x] = value
    showState(startState)
    if isValid(startState):
        setStatusText("State valid")
    else:
        setStatusText("Invalid state")

def showState(state):
    removeAllActors()
    for ys in range(3):
        for xs in range(3):
            for y in range(3):
                for x in range(3):
                    loc = Location(x + 3 * xs, y + 3 * ys)
                    value = state[ys][xs][y][x]
                    if  value != 0:
                        if loc in fixedLocations:
                            c = Color.black
                        else:
                            c = Color.red
                        digit = TextActor(str(value), c, Color.white,
                                Font("Arial", Font.BOLD, 20))
                        addActorNoRefresh(digit, loc)
    refresh()

def isValid(state):
    # Check lines
    for ys in range(3):
        for y in range(3):
            line = []
            for xs in range(3):
                for x in range(3):
                    value = state[ys][xs][y][x]
                    if value > 0 and value in line:
                        return False
                    else:
                        line.append(value)
```

```
    # Check rows
    for xs in range(3):
        for x in range(3):
            row = []
            for ys in range(3):
                for y in range(3):
                    value = state[ys][xs][y][x]
                    if value > 0 and value in row:
                        return False
                    else:
                        row.append(value)

    # Check subgrids
    for ys in range(3):
        for xs in range(3):
            subgrid = state[ys][xs]
            square = []
            for y in range(3):
                for x in range(3):
                    value = subgrid[y][x]
                    if value > 0 and value in square:
                        return False
                    else:
                        square.append(value)
    return True

makeGameGrid(9, 9, 50, Color.red, False, mousePressed = pressEvent)
show()
setTitle("Sudoku")
addStatusBar(30)
visited = []
setBgColor(Color.white)
getBg().setLineWidth(3)
getBg().setPaintColor(Color.red)
for x in range(4):
    getBg().drawLine(150 * x, 0, 150 * x, 450)
for y in range(4):
    getBg().drawLine(0, 150 * y, 450, 150 * y)

stateWiki = [[[[0, 3, 0], [0, 0, 0], [0, 0, 8]],
              [[0, 0, 0], [1, 9, 5], [0, 0, 0]],
              [[0, 0, 0], [0, 0, 0], [0, 6, 0]]],
             [[[8, 0, 0], [4, 0, 0], [0, 0, 0]],
              [[0, 6, 0], [8, 0, 0], [0, 2, 0]],
              [[0, 0, 0], [0, 0, 1], [0, 0, 0]]],
             [[[0, 6, 0], [0, 0, 0], [0, 0, 0]],
              [[0, 0, 0], [4, 1, 9], [0, 0, 0]],
              [[2, 8, 0], [0, 0, 5], [0, 7, 0]]]]
startState = stateWiki

fixedLocations = []
for xs in range(3):
    for ys in range(3):
        for x in range(3):
            for y in range(3):
                if startState[ys][xs][y][x] != 0:
                    fixedLocations.append(Location(x + 3 * xs, y + 3 * ys))

showState(startState)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Für das Backtracking musst mit der Funktion *getNeighbours(state)* die Nachfolgeknoten eines Zustands bestimmen. Dabei wählst du mit *getEmptyCell()* eine leere Zelle aus und setzt dort der Reihe nach alle Zahlen ein, die zu einem legalen Spielzustand gehören. Mindestens eine davon wir sicher die richtige sein.

In *getNeighbours(state)* kopierst du zuerst die gegebene Zustandsliste in einen **Clone**, da du

den übergegebenen Zustand nicht verändern darfst. Nachher füllst du die leere Zelle nacheinander mit den Zahlen 1 bis 9 und fügst Kopien der **erlaubten** Zustände in die Nachbarliste, die du zuletzt zurückgibst [**mehr...**]

Die rekursive definierte Funktion *search()*, die das eigentliche Backtracking durchführt, kannst praktisch unverändert von anderen Backtracking-Problemen übernehmen. Du suchst hier nur nach einer einzigen Lösung und beendest mit dem Flag *found* den rekursiven Aufruf.


```
from gamegrid import *

def pressEvent(e):
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in fixedLocations:
        setStatusText("Location fixed")
        return
    x = loc.x
    y = loc.y
    value = startState[y][x]
    value = (value + 1)  % 10
    startState[y][x] = value
    showState(startState)
    if isValid(startState):
        setStatusText("State valid")
    else:
        setStatusText("Invalid state")

def getBlockValues(state, x, y):
    return [state[y][x], state[y][x + 1], state[y][x + 2],
            state[y + 1][x], state[y + 1][x + 1], state[y + 1][x + 2],
            state[y + 2][x], state[y + 2][x + 1], state[ y + 2][x + 2]]

def showState(state):
    removeAllActors()
    for y in range(9):
        for x in range(9):
            loc = Location(x, y)
            value = state[y][x]
            if  value != 0:
                if loc in fixedLocations:
                    c = Color.black
                else:
                    c = Color.red
                digit = TextActor(str(value), c, Color.white,
                        Font("Arial", Font.BOLD, 20))
                addActorNoRefresh(digit, loc)
    refresh()

def isValid(state):
    # Check lines
    for y in range(9):
```

```
        values = []
        for x in range(9):
            value = state[y][x]
            if value > 0 and value in values:
                return False
            else:
                values.append(value)
    # Check rows
    for x in range(9):
        values = []
        for y in range(9):
            value = state[y][x]
            if value > 0 and value in values:
                return False
            else:
                values.append(value)

    # Check blocks
    for yblock in range(3):
        for xblock in range(3):
            values = []
            li = getBlockValues(state, 3 * xblock, 3 * yblock)
            for value in li:
                if value > 0 and value in values:
                    return False
                else:
                    values.append(value)
    return True

def getEmptyCell(state):
    emptyCells = []
    for y in range(9):
        for x in range(9):
            if state[y][x] == 0:
                return [x, y]
    return []

def cloneState(state):
    li = []
    for y in range(9):
        line = []
        for x in range(9):
           line.append(state[y][x])
        li.append(line)
    return li

def getNeighbours(state):
    clone = cloneState(state)
    cell = getEmptyCell(state)
    validStates = []
    for value in range(1, 10):
        clone[cell[1]][cell[0]] = value
        if isValid(clone):
            validStates.append(cloneState(clone))
    return validStates

def search(state):
    global found, solution
    if found:
        return
    visited.append(state)  # state marked as visited

    # Check for solution
    if getEmptyCell(state) == []:
        solution = state
        found = True
        return

    for neighbour in getNeighbours(state):
```

```
        if neighbour not in visited: # Check if already visited
            search(neighbour) # recursive call
    visited.pop()

makeGameGrid(9, 9, 50, Color.red, False, mousePressed = pressEvent)
show()
setTitle("Sudoku")
addStatusBar(30)
visited = []
setBgColor(Color.white)
getBg().setLineWidth(3)
getBg().setPaintColor(Color.red)
for x in range(4):
    getBg().drawLine(150 * x, 0, 150 * x, 450)
for y in range(4):
    getBg().drawLine(0, 150 * y, 450, 150 * y)

startState = [
[0, 6, 0, 7, 9, 8, 0, 1, 2],
[7, 9, 4, 1, 0, 5, 0, 6, 8],
[2, 0, 1, 4, 0, 0, 9, 5, 7],
[0, 0, 0, 2, 1, 0, 5, 0, 0],
[0, 5, 6, 3, 0, 0, 2, 4, 1],
[0, 1, 2, 5, 4, 0, 7, 3, 9],
[6, 3, 0, 8, 7, 4, 0, 0, 0],
[1, 0, 5, 6, 0, 2, 8, 0, 0],
[4, 2, 8, 9, 0, 1, 0, 7, 0]]

fixedLocations = []
for x in range(9):
    for y in range(9):
        if startState[y][x] != 0:
            fixedLocations.append(Location(x, y))

showState(startState)
setStatusText("Press any key to search solution.")
getKeyCodeWait(True)
setStatusText("Searching. Please wait...")
found = False
solution = None
search(startState)
if solution != None:
    showState(solution)
    setStatusText("Solution found")
else:
    setStatusText("No solution")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du kannst auch Sudoku-Vorgaben verwenden, die du im Internet oder in einer Rätselecke gefunden hast. Wenn du genügend Geduld hast, so sollte dein Programm immer eine Lösung finden.

Man kann die Rechenzeit drastisch senken, falls man Lösungsstrategien in das Programm einbeziehst, die man auch bei der Lösung von Hand anwendet. Es bleibt dir überlassen, den Algorithmus mit solchen **heuristischen Verfahren** zu verbessern.

Das Lösungsverfahren lässt sich auch auf Sudoku-Vorgaben anwenden, die nicht-quadratische Blöcke aufweisen. Du musst dazu lediglich die Funktion *getBlockValues()* entsprechend anpassen.

DIE EIFERSÜCHTIGEN EHEMÄNNER

Bereits um 1613. hat der Mathematiker C. G. Bachet, Sieur de Méziriac eine Untersuchung mit dem Titel *Problèmes plaisants et détectables qui se font par les nombres* veröffentlicht, in dem er das Rätsel *Les vilains maris jaloux* wie folgt beschreibt:

Claude Gaspard Bachet de Méziriac (1581-1638) (© Wiki)

"Trois maris jaloux se trouvent le soir avec leurs femmes au passage d'une rivière, et rencontrent un bateau sans batelier; le bateau est si petit, qu'il ne peut porter plus de deux personnes à la fois. On demande comment ces six personnes passeront de tel sorte qu'aucune femme ne demeure en la compagnie d'un ou de deux hommes, si son mari n'est présent, soit sur l'une des deux rives, soit sur le bateau." (Lit. Édouard Lucas, L'arithmétique amusante" 1885, reprint 2006)

"Drei eifersüchtige Ehemänner befinden sich am Abend zusammen mit ihren Ehefrauen an einem Flussübergang, wo sie ein Boot ohne Bootsführer vorfinden. Das Boot ist so klein, dass darin nur zwei Personen Platz haben. Es stellt sich die Frage, wie die 6 Personen den Fluss überqueren können, ohne dass sich jemals eine der Frauen in Anwesenheit eines oder zwei Männer befindet, ohne dass ihr eigener Ehemann dabei ist, sei es auf der einen oder anderen Seite des Flusses, sei es im Boot."

Das Problem löst du wieder schrittweise und erstellst als erstes die Benutzeroberfläche, wobei sich ein GameGrid gut eignet, da man die Personen leicht durch Spritebilder veranschaulichen kann. Du fügst die GameGrid-Actors in ein unsichtbares Gitter der Grösse 7x3, damit du im Mittelstreifen den Fluss einzeichnen kannst. Da es nur wesentlich ist, ob sich eine Person links oder rechts vom Fluss befindet, ist als Datenstruktur eine Dualzahl geeignet, bei der jedes Bit zu einer bestimmten Person gehört. Der Bitwert 0 sagt, dass sich die Person links, der Wert 1, dass sie sich rechts vom Fluss befindet. Für das Boot verwendest du das Bit mit der höchsten Wertigkeit.

In der Simulation gibt es ein blaues, grünes und rotes Ehepaar, die du folgenden Stellen der Dualzahl zuordnest:

b6	b5	b4	b3	b2	b1	b0
boat	man_red	female_red	man_green	female_green	man_blue	female_blue

b0 ist das Bit mit der kleinsten Wertigkeit. Interpretierst du den Zustand im Dezimalsystem, so kommen also alle Zahlen zwischen 0 und 127 vor, d.h. es gibt 128 verschiedene Zustände.

Es ist sehr zu empfehlen, dass du auch hier einen kleinen Umweg nimmst, damit du in einem Testprogramm die Kodierung der Zustände ausprobieren kannst. Mit einem Mausklick auf eine Person oder auf das Boot springt das Bild von der einen Seite auf die andere Seite des Flusses und der Zustand wird dezimal und binär in der Titelzeile ausgeschrieben.

Bereits hier baust du mit *isStateAllowed(state)* einen Test ein, ob die aktuelle Situation gemäss den Vorgaben legal ist. (Du könntest auch zuerst eine Vorversion ohne diesen Test erstellen.)

```
from gamegrid import *

def pressEvent(e):
    global state
    loc = toLocationInGrid(e.getX(), e.getY())
```

```
    if loc in left_locations:
        actor = getOneActorAt(loc)
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    if loc in right_locations:
        actor = getOneActorAt(loc)
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    state = 0
    for i in range(7):
        loc = right_locations[i]
        actor = getOneActorAt(loc)
        if actor != None:
            state += 2**(6 - i)
    showState(state)

def stateToString(state):
    return str(bin(state)[2:]).zfill(7)

def showState(state):
    sbin = stateToString(state)
    for i in range(7):
        if sbin[i] == "0":
           actors[i].setLocation(left_locations[i])
        else:
           actors[i].setLocation(right_locations[i])
    setTitle("State: " + str(state) + ", bin: " + stateToString(state))
    if isStateAllowed(state):
        setStatusText("Situation erlaubt")
    else:
        setStatusText("Situation verboten")
    refresh()

def isStateAllowed(state):
    print state
    stateStr = stateToString(state)
    mred = stateStr[1] == "1"
    fred = stateStr[2] == "1"
    mgreen = stateStr[3] == "1"
    fgreen = stateStr[4] == "1"
    mblue = stateStr[5] == "1"
    fblue = stateStr[6] == "1"

    if mred and not fred or not mred and fred: # mred/fred not together
        if not fred and (not mgreen or not mblue) or fred and (mgreen or mblue):
            return False
    if mgreen and not fgreen or not mgreen and fgreen:#mgreen/fgreen not together
        if not fgreen and (not mred or not mblue) or fgreen and (mred or mblue):
            return False
    if mblue and not fblue or not mblue and fblue:  # mblue/fblue not together
        if not fblue and (not mred or not mgreen) or fblue and (mred or mgreen):
            return False
    return True

makeGameGrid(7, 3, 50, None, False, mousePressed = pressEvent)
setBgColor(Color.white)
addStatusBar(30)
show()
actors = [Actor("sprites/boat.png"),
   Actor("sprites/man_0.png"), Actor("sprites/woman_0.png"),
   Actor("sprites/man_1.png"), Actor("sprites/woman_1.png"),
   Actor("sprites/man_2.png"), Actor("sprites/woman_2.png")]

left_locations =  [Location(2, 0),
                   Location(2, 1), Location(2, 2),
```

```
                    Location(1, 1), Location(1, 2),
                    Location(0, 1), Location(0, 2)]
right_locations = [Location(4, 0),
                    Location(4, 1), Location(4, 2),
                    Location(5, 1), Location(5, 2),
                    Location(6, 1), Location(6, 2)]

for i in range(7):
    addActorNoRefresh(actors[i], left_locations[i])
for i in range(3):
   getBg().fillCell(Location(3, i), Color.blue)
refresh()

startState = 0
showState(startState)
```

Als nächstes implementierst du wieder den Backtracking-Algorithmus, wie du ihn aus **Kapitel 10.3.** bestens kennst. Dabei muss du als erstes in *getNeighbours(state)* zu einem bestimmen Zustand alle möglichen Folgezustände bestimmen. Dabei gehst du so vor: Zuerst unterscheidest du, ob sich das Boot links (*state < 64*) oder rechts (*state >=64*) befindet. Dann bestimmst du in den Listen *li_one* und *li_two* alle Personen, die für einen Transfer als Einzelperson oder im Zweierteam in Frage kommen. Dabei musst du noch mit *removeForbiddenTransfers()* berücksichtigen, dass eine Frau wirklich nie mit einem fremden Mann im Boot sitzt.

In *search()* implementierst du das Backtracking in bekannter Form. Die Lösungen kopierst du in eine Liste *solutions*, auf die du nach Ende des Suchvorgangs zugreifen kannst, um die Lösungen zu untersuchen. Dabei schreibst du zuerst die Anzahl gefundener Lösungen aus und wählst dann die kürzeste aus, die du mit Tastenbetätigungen durchlaufen kannst.

```
from gamegrid import *
import itertools

def pressEvent(e):
    global state
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in left_locations:
        actor = getOneActorAt(loc)
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    if loc in right_locations:
        actor = getOneActorAt(loc)
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    state = 0
    for i in range(7):
        loc = right_locations[i]
        actor = getOneActorAt(loc)
        if actor != None:
            state += 2**(6 - i)
    setTitle("State: " + str(state) + ", bin: " + stateToString(state))
    if isStateAllowed(state):
        setStatusText("Situation erlaubt")
    else:
        setStatusText("Situation verboten")
    refresh()

def stateToString(state):
    return str(bin(state)[2:]).zfill(7)

def showState(state):
    sbin = stateToString(state)
    for i in range(7):
```

```
        if sbin[i] == "0":
            actors[i].setLocation(left_locations[i])
        else:
            actors[i].setLocation(right_locations[i])
    refresh()

def getTransferInfo(state1, state2):
    state1 = state1 & 63
    state2 = state2 & 63
    mod = state1 ^ state2
    passList = []
    for n in range(6):
        if mod % 2 == 1:
            if n // 2 == 0:
                couple = "blue"
            elif n // 2 == 1:
                couple = "green"
            elif n // 2 == 2:
                couple = "red"
            if n % 2 == 0:
                passList.append("f" + couple)
            else:
                passList.append("m" + couple)
        mod = mod // 2
    return passList

def getTransferSequence(solution):
    transferSequence = []
    oldState = solution[0]
    for state in solution[1:]:
        transferSequence.append(getTransferInfo(oldState, state))
        oldState = state
    return transferSequence

def isStateAllowed(state):
    stateStr = stateToString(state)
    mred = stateStr[1] == "1"
    fred = stateStr[2] == "1"
    mgreen = stateStr[3] == "1"
    fgreen = stateStr[4] == "1"
    mblue = stateStr[5] == "1"
    fblue = stateStr[6] == "1"

    if mred and not fred or not mred and fred:  # mred/fred not together
        if not fred and (not mgreen or not mblue) or fred and (mgreen or mblue):
            return False
    if mgreen and not fgreen or not mgreen and fgreen:#mgreen/fgreen not together
        if not fgreen and (not mred or not mblue) or fgreen and (mred or mblue):
            return False
    if mblue and not fblue or not mblue and fblue:  # mblue/fblue not together
        if not fblue and (not mred or not mgreen) or fblue and (mred or mgreen):
            return False
    return True

def removeForbiddenTransfers(li):
    forbiddenPairs = [(0, 3), (0, 5), (1, 2), (2, 5), (1, 4), (3, 4)]
    allowedPairs = []
    for pair in li:
        if pair not in forbiddenPairs:
            allowedPairs.append(pair)
    return allowedPairs

def getNeighbours(state):
    neighbours = []
    li_one = []  # one person in boat
    bin = stateToString(state)
    if state < 64:  # boat at left
        for i in range(6):
            if bin[6 - i] == "0":
```

```
                li_one.append(i)
        li_two = list(itertools.combinations(li_one, 2)) #two persons in boat
        li_two = removeForbiddenTransfers(li_two)
    else: # boat at right
        for i in range(6):
            if bin[6 - i] == "1":
                li_one.append(i)
        li_two = list(itertools.combinations(li_one, 2))
        li_two = removeForbiddenTransfers(li_two)

    li_values = []
    if state < 64:  # boat at left, restrict to two persons transfer
        for li in li_two:
            li_values.append(2**li[0] + 2**li[1] + 64)
    else:  # boat at right, one or two persons transfer
        for i in li_one:
            li_values.append(2**i + 64)
        for li in li_two:
            li_values.append(2**li[0] + 2**li[1] + 64)

    for value in li_values:
        v = state ^ value
        if isStateAllowed(v):  # restrict to allowed states
            neighbours.append(v)
    return neighbours

def search(state):
    visited.append(state)  # state marked as visited

    # Check for solution
    if state == targetState:
        solutions.append(visited[:])

    for neighbour in getNeighbours(state):
        if neighbour not in visited: # Check if already visited
            search(neighbour) # recursive call
    visited.pop()

nbSolution = 0
makeGameGrid(7, 3, 50, None, False, mousePressed = pressEvent)
addStatusBar(30)
setBgColor(Color.white)
setTitle("Searching...")
show()
visited = []
actors = [Actor("sprites/boat.png"),
   Actor("sprites/man_0.png"), Actor("sprites/woman_0.png"),
   Actor("sprites/man_1.png"), Actor("sprites/woman_1.png"),
   Actor("sprites/man_2.png"), Actor("sprites/woman_2.png")]

left_locations =  [Location(2, 0),
                   Location(2, 1), Location(2, 2),
                   Location(1, 1), Location(1, 2),
                   Location(0, 1), Location(0, 2)]
right_locations = [Location(4, 0),
                   Location(4, 1), Location(4, 2),
                   Location(5, 1), Location(5, 2),
                   Location(6, 1), Location(6, 2)]

for i in range(7):
    addActorNoRefresh(actors[i], left_locations[i])
for i in range(3):
   getBg().fillCell(Location(3, i), Color.blue)
refresh()

startState = 0
targetState = 127
solutions = []
search(startState)
```

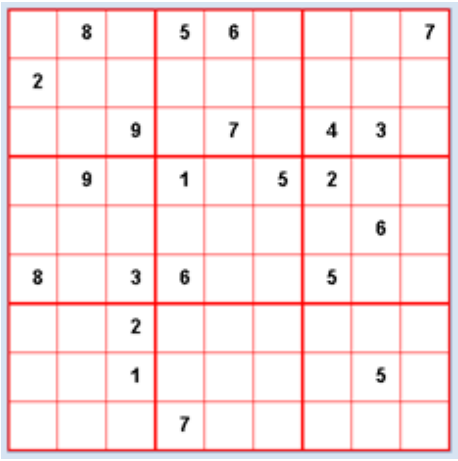
```
maxLength = 0
maxSolution = None
minLength = 100
minSolution = None
for solution in solutions:
    if len(solution) > maxLength:
        maxLength = len(solution)
        maxSolution = solution
    if len(solution) < minLength:
        minLength = len(solution)
        minSolution = solution
setStatusText("#Solutions: " + str(len(solutions)) + ", Min Length: "
            + str(minLength) + ", Max Length: " + str(maxLength))

setTitle("Press key to cycle")
oldState = startState
for state in minSolution[1:]:
    getKeyCodeWait(True)
    showState(state)
    info = getTransferInfo(oldState, state)
    setTitle("#Transferred: " + str(info))
    oldState = state
setTitle("Done. #Boat Transfers: " + str((len(minSolution) - 1)))
```

MEMO

Rätsel dieser Art haben die Eigenschaft, dass man zum vornherein nicht weisst, ob sie keine, genau eine oder mehrere Lösungen haben. Dabei zeigt sich, dass es für uns Menschen viel einfacher ist, eine Lösung zu finden, als zu beweisen, dass es keine Lösung gibt. Für den Computer, der mit der *Erschöpfenden Suche* systematisch nach allen Lösungen sucht, ist das Aufsuchen aller Lösungen grundsätzlich kein Problem, ausser dass der Lösungsprozess sehr lange dauern kann. Dies ist wegen der kombinatorischen Explosion leider schon bei relativ einfachen Spielanlagen der Fall, sodass wir auch hier wieder an Grenzen des Computereinsatzes stossen. Es ist interessant, dass bereits Bachet die Minimallösung mit 11 Überfahrten publizierte, die auch dein Computerprogramm findet. Dabei hat er von Überfahrt zu Überfahrt so geschickt argumentiert, dass der Lösungsgang evident erscheint. Ob er die Lösung allerdings zuerst doch nach der Methode von "Versuch und Irrtum" gefunden und erst nachträglich die Argumente dazu aufgeschrieben hat, bleibe dahingestellt.

AUFGABEN

1. Suche ein Lösung für das folgende *X-Sudoku*, bei dem auch auf den beiden Hauptdiagonalen die 9 Zahlen genau einmal vorkommen müssen.



2. Zeige, dass es keine Lösung des Problems der "Eifersüchtigen Ehemänner" gibt, falls man nur Einpersonentransfers vom rechten zum linken Ufer zulässt.

11.2 FALLGRUBEN, REGELN & TRICKS

EINFÜHRUNG

Wie in jeder Programmiersprache, so gibt es auch in Python einige Fallgruben, in die selbst erfahrene Programmierer fallen können. Du kannst sie umgehen, wenn du sie als potentielle Gefahrenquelle kennst.

KOPIEREN, CLONES UND SEITENEFFEKTE

Grosse Vorsicht ist geboten, wenn du Variablenwerte in eine andere Variable umkopierst. Verwendest du dabei das Gleichheitszeichen, so ist die Versuchung gross davon auszugehen, dass die beiden Variablen nach dem Kopieren völlig unabhängig voneinander sind, was heissen würde, dass du den Variablenwert der einen Variable beliebig verändern kannst, ohne dass davon der Wert der anderen Variablen betroffen ist. Dem ist aber meist nicht so!

Hast du eine Variable a mit dem Wert 2 mit der Zuweisung a = 2 erzeugt und erstellst von ihr mit b = a eine Kopie, so gehst du davon aus, dass bei einer Veränderung von b durch eine neue Zuweisung b = 3 die Variable a nicht verändert wird, also immer noch den Wert 2 hat. Prüfst du dies in der Python-Konsole nach, so siehst du, das dies tatsächlich stimmt:

```
>>> a = 2
>>> b = a
>>> b = 3
>>> a
 < 2
>>> b
 < 3
```

Wendest du nun genau dasselbe Vorgehen auf eine Liste an, bemerkst du mit Erstaunen, dass bei einer Veränderung von b auch a verändert wird.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b[0] = 7
>>> a
 < [7, 2, 3]
>>> b
 < [7, 2, 3]
```

Der Grund liegt darin, dass in Python Variablen auf Objekte **verweisen**, also nicht einfach Namen für die Objekte selbst sind [**mehr...**]. In der Regel werden bei der Zuweisung mit dem Gleichheitszeichen lediglich der Verweis und nicht die darin enthaltenen Daten kopiert. Von dieser Regel gibt es eine Ausnahme: Gewisse Datentypen gelten als **unveränderlich** (**immutable**). Es sind dies **Zahlen, String, byte, tuple**. Bei der Zuweisung eines solchen Objekts wird ein neues Objekt erstellt und die inneren Daten übernommen. Damit sind die beiden Objekte voneinander unabhängig. Da man schnell übersieht, ob es sich um einen unveränderlichen oder veränderlichen Datentyp handelt, fällt man bei der Zuweisung leicht in die Fallgrube, ausser man merkt sich folgende Regel:

Regel 1a:
Die Kopieroperation mit dem Gleichheitszeichen ist für Zahlen, Strings, bytes und tuples unproblemtisch. Für andere Datentypen ist sie meist falsch.

Eine Kopie einer Variablen wird auch ein **Clone** genannt. Du kannst dich nochmals überzeugen, wie problematisch das Kopieren einer Liste ist, obschon sie unveränderliche Datentypen, nämlich Strings, enthält:

```
>>> myGarden = ["Rose", "Lotus"]
>>> yourGarden = myGarden
>>> yourGarden[0] = "Hibiskus"
>>> myGarden
 < ["Hibiskus", "Lotus"]
>>> yourGarden
 < ["Hibiskus", "Lotus"]
```

Willst du auch für veränderliche Datentypen, zum Beispiel Listen, eine unabhängige Kopie erstellen, so musst du entweder die Elemente mit eigenem Code explizite in eine neue Variable kopieren oder du kannst dazu die Funktion *deepcopy()* aus dem Modul *copy* verwenden:

```
>>> import copy
>>> myGarden = ["Rose", "Lotus"]
>>> yourGarden = copy.deepcopy(myGarden)
>>> yourGarden[0] = "Hibiskus"
>>> myGarden
 < ["Rose", "Lotus"]
>>> yourGarden
 < ["Hibiskus", "Lotus"]
```

Regel 1b:
Für veränderliche Datentypen sollte man beim Kopieren die Funktion *copy.deepcopy()* verwenden.

Regel 1 wird oft im Zusammenhang mit der Parameterübergabe missachtet. Übergibt man einer Funktion einen veränderlichen Datentyp, so kann die Funktion die Variablenwerte problemlos verändern (auch wenn man das Schlüsselwort *global* nicht verwendet).

```
>>> def show(garden)
>>>     print "garden:", garden
>>>     garden[0] = "Hibiskus"
>>> myGarden = ["Rose", "Lotus"]
>>> show(myGarden)
>>> myGarden
 < ["Hibiskus", "Lotus"]
```

Nach dem Aufruf von *show()* haben sich die Werte des übergebenen Parameters verändert! Wie in der Medizin handelt es sich meist um eine unerwartete und unerwünschte Nebenwirkung oder um einen **Seiteneffekt**.

Regel 2:
Es gehört zum guten Programmierstil, dass man in einer Funktion den übergebenen Parameter nicht mittels Seiteneffekten verändert.

Handelt es sich beim Parameter um einen unveränderlichen Datentyp, so ist die Gefahr gebannt, denn eine Zuweisung an den Parameter erstellt eine neue lokale Variable.

```
>>> def show(flower)
>>>     print "flower:", flover
>>>     flower = "Hibiskus"
>>> myFlower = "Rose"
>>> show(myFlower)
>>> myFlower
```

```
< "Rose"
```

Auf den ersten Blick scheinen sich **Tuples** nicht wesentlich von Listen zu unterscheiden. In der Tat handelt es sich bei Tuples grundsätzlich um unveränderliche Listen und damit sind alle Listenoperation, die zu keiner Veränderung der Liste führen, auch für Tuples anwendbar. Mit Tuples gibt es aber spezielle Schreibweisen unter Verwendung des Kommas.

Bei der Erzeugung von Tuples kann das runde Klammerpaar weggelassen werden:

```
>>>  t = 1, 2, 3
>>>  t
>>>  (1, 2, 3)
```

Hier wird das Komma also als Syntaxzeichen verwendet, um die Elemente voneinander zu trennen. Man spricht von automatischen **Verpacken (packing)**. Du kannst elegant davon Gebrauch machen, um mehrere Funktionswerte als Tuple zurückzugeben:

```
>>> import math
>>> def sqrt(x):
>>>     y = math.sqrt(x)
>>>     return y, -y
>>> sqrt(4)
 < (2,0, -2,0)
```

Du kannst den Kommaoperator auch bei der Variablendefinition verwenden, falls hinter dem Gleichheitszeichen ein Tuple steht. In diesem Fall spricht man vom automatischen **Entpacken (unpacking)**:

```
>>> import math
>>> def sqrt(x):
>>>     y = math.sqrt(x)
>>>     return y, -y
>>> y1, y2 = sqrt(2)
>>> y1
 < 1.41421356237330951
>>> y2
 < -1.41421356237330951
```

Das Verpacken ist praktisch, um mehrere Variablen gleichzeitig zu definieren:

```
>>> a, b, c = 1, 2, 3
>>> a
 < 1
>>> b
 < 2
>>> c
 < 3
```

Dabei werden auf der rechten Seite die drei Zahlen verpackt und auf der linken Seite wieder entpackt, so wie es hier explizite gemacht wird:

```
>>> t = 1, 2, 3
>>> a, b, c = t
>>> a
 < 1
>>> b
 < 2
>>> c
 < 3
```

Das Entpacken funktioniert übrigens auch mit Listen:

```
>>> li = [1, 2, 3]
>>> a, b, c = li
>>> a
 < 1
>>> b
 < 2
>>> c
 < 3
```

Mit dieser Syntax können zwei Zahlen elegant vertauscht werden, ohne dass eine Hilfsvariable nötig ist:

```
>>> li = [1, 2, 3]
>>> a, b = b, a
>>> a
 < 2
>>> b
 < 1
```

ZWEIDIMENSIONALE LISTEN, MATRIZEN

Matrizen werden in vielen Programmiersprachen aus Arrays konstruiert. Die Zeilen der Matrix sind Arrays und die Matrix selbst ein Array aus diesen Zeilenarrays. Es ist naheliegend, in Python statt Arrays Listen zu verwenden. Dabei ist aber besondere Vorsicht geboten, denn Listen verhalten sich nicht wie elementare (unveränderliche) Datentypen, sondern wie Referenztypen. Bereits bei der Erstellung der Matrix fällst du in eine bekannte Fallgrube. Ohne Argwohn erzeugst du in der Python-Konsole eine 3x3 Matrix mit Nullen.

```
>>> A = [[0] * 3] * 3
>>> A
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
```

Veränderst du nun mit einer Zuweisung den letzten Wert der ersten Zeile, so bemerkst du mit Erstaunen, dass alle Zeilen verändert wurden.

```
>>> A[0][2] = 1
>>> A
 < [[0, 0, 1],
    [0, 0, 1],
    [0, 0, 1]]
```

Was ist da passiert? Etwas Nachdenken hilft dir auf die Sprünge. Die Erzeugung von A hätte auch in zwei Schritten erfolgen können:

```
>>> z = [0] * 3
>>> A = [z] * 3
>>> A
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
```

Zuerst wird eine Liste z mit drei Nullen erzeugt. Dann wird eine Liste A mit dreimal derselben Zeilenreferenz gebildet. Alle verweisen also auf dieselbe Liste! Änderst du eine davon ab, so sind auch die anderen betroffen.

Regel 3:
Verwende bei verschachtelten Listen nie das Listenmultiplikationszeichen.

Um die Fallgrube zu umgehen, kannst du die List Comprehension einsetzen. Wie du nachprüfen kannst, verhält sich die Matrix nun richtig:

```
>>> A = [[0 for x in range(3)] for y in range(3)]
>>> A
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
>>> A[0][2] = 1
>>> A
 < [[0, 0, 1],
    [0, 0, 0],
    [0, 0, 0]]
```

FUNCTION DECORATORS

In Python kann man mit einer Zeile, die mit einem At-Symbol @ beginnt, eine Funktion speziell auszeichnen oder "ausschmücken". Eine solche Zeile nennt man einen **Decorator**. Du verwendest den *Function Decorator*, um einer Funktion zusätzliche Eigenschaften zu geben. Man spricht auch von einem Funktionswrapper, da die Ersatzfunktion üblicherweise im Inneren die bestehende Funktion verwendet. Im folgenden Programm gehst du von einer Dreiteilungsfunktion *trisect(x)* aus, die du so "dekorierst", dass sie für x = 0 den Wert 0 zurückgibt und alle Werte auf 2 Stellen rundet.

```
def tri(func):
    # inner function
    def _tri(x):
        if x == 0:
            return 0
        return round(func(x), 2)
    return _tri

@tri
def trisect(x):
    return x / 3

for x in range(0, 11):
   value = trisect(x)
   print value
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Im **Kapitel 3.7** hast du gesehen, dass du Decorators verwenden kannst, um eine Funktion so auszuzeichnen, dass sie automatisch als Callback registriert wird. Im **Kapitel 7.2** wurde der Decorator *@staticmethod* dazu verwendet, um eine Funktion als statische Methode auszuzeichnen.

11.3 BUGS & DEBUGGING

EINFÜHRUNG

Es gibt keine absolute Perfektion. Du kannst davon ausgehen, dass praktisch jede Software Fehler aufweist. Statt von Fehlern spricht man bei Software meistens von **Bugs**. Diese manifestieren sich beispielsweise dadurch, dass das Programm unter gewissen Umständen falsche Resultate produziert oder sogar abstürzt. Die Fehlersuche, **Debugging** genannt, ist daher fast ebenso wichtig wie das Programmieren selbst. Es ist allerdings selbstverständlich, dass jedermann, der Programmcode schreibt, sein Bestmögliches tun sollte, Bugs zu vermeiden. Im Wissen, dass Bugs allgegenwärtig sind, solltest du daher **vorsichtig** und **defensiv** programmieren. Bist du nicht ganz sicher, ob ein Algorithmus oder ein Codeteil richtig ist, so ist es besser, dass du dich mit ihm besonders intensiv beschäftigst, statt möglichst rasch darüber hinwegzugehen. Es gibt heutzutage viel Software, von deren richtigen Funktionieren grosse Geldsummen oder sogar Menschenleben auf dem Spiel stehen. Als Programmierer von solcher **missionskritischer Software** (mission critical software), musst du bei jeder Zeile Code die Verantwortung übernehmen, dass sie korrekt arbeitet. Schnelle Hacks und das Prinzip von Trial and Error sind hier fehl am Platz.

Für die Entwicklung von Algorithmen und grossen Software-Systemen spielt die Zielsetzung, möglichst fehlerlose Programme zu erzeugen, eine wichtige Rolle. Es gibt viele Ansätze dazu: Man kann versuchen, die Korrektheit von Programmen ohne Einsatz des Computers mathematisch exakt zu beweisen, was allerdings nur mit kurzen Programmen gelingt. Eine andere Möglichkeit besteht darin, die Syntax der Programmiersprache so einzuschränken, dass der Programmierer bestimmte Fehler gar nicht erst machen kann. Wichtigstes Beispiel ist die Elimination von Zeigervariablen (pointers), die bei unvorsichtiger Verwendung auf nicht definierte oder falsche Objekte verweisen. Es gibt darum Programmiersprachen mit vielen Einschränkungen dieser Art und solche, die dem Programmierer grosse Freiheiten lassen und dafür als weniger sicher gelten. Python gehört zur Klasse freiheitlicher Programmiersprachen und orientiert sich am Motto:
"Wir sind alle erwachsen und entscheiden selbst, was wir tun und lassen müssen" ("After all, we are all consenting adults here").

Ein wichtiges Prinzip zum Erstellen von möglichst fehlerfreier Software ist das **Design by contract** (DBC). Es geht auf den an der ETH Zürich wirkenden Bertrand Meyer zurück, dem Vater der Programmiersprache Eiffel. Er versteht Software als eine *Vereinbarung* zwischen dem Programmierer A und dem Anwender B, wobei B auch selbst Programmierer sein kann, der die von A entwickelten Module und Bibliotheken verwendet. In einem *Vertrag* legt A fest, unter welchen Bedingungen (**preconditions**) seine Module die richtigen Resultate liefern und beschreibt sie genau (**postconditions**). Anders gesagt: Hält sich Anwender B an die Preconditions, so hat er von A die Garantie, dass er ein Resultat gemäss den Postconditions erhält. Die Implementierung der Module braucht B nicht zu kennen, A kann diese auch jederzeit ändern, ohne dass B davon betroffen ist.

ZUSICHERUNGEN (ASSERTIONS)

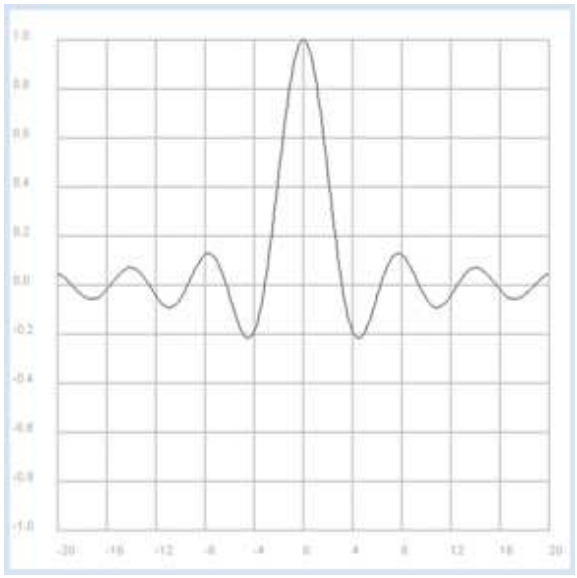
Da der Modulfabrikant A allerdings dem Anwender B etwas misstraut, baut er in seiner Software Tests ein, welche die geforderten Preconditions überprüfen. Diese werden **Zusicherungen (assertions)** genannt. Werden die Zusicherungen nicht eingehalten, so bricht das Programm gewöhnlich mit einer Fehlermeldung ab. (Es ist auch vorstellbar, dass der Fehler lediglich abgefangen wird, ohne dass das Programm abbricht.) Dabei kommt folgendes Prinzip der Softwareentwicklung zum Tragen:

Ein Programmabbruch mit einer möglichst klaren Beschreibung des Ursache (Fehlermeldung) ist besser als ein falsches Resultat.

In deinem Beispiel schreibst du als Programmierer A eine Funktion *sinc(x)* (sinus cardinalis), die in der Signalverarbeitung eine wichtige Rolle spielt. Sie lautet:

$$f(x) = \frac{\sin x}{x}$$

Als Anwender B willst du die Funktionswerte im Bereich x = -20 .. 20 grafisch darstellen.



```
from gpanel import *
from math import pi, sin

def sinc(x):
    y = sin(x) / x
    return y

makeGPanel(-24, 24, -1.2, 1.2)
drawGrid(-20, 20, -1.0, 1, "darkgray")
title("Sinus Cardinalis: y = sin(x) / x")

x = -20
dx = 0.1
while x <= 20:
    y = sinc(x)
    if x == -20:
        move(x, y)
    else:
        draw(x, y)
    x += dx
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Auf den ersten Blick scheint es keine Probleme zu geben, änderst du aber als Anwender das Inkrement von x auf 1, so gibt es einen bösen Crash.

```
Division durch Null ist nicht möglich!
    y = sin(x) / x
```

Eine Lösung des Problems besteht darin, dass A als Precondition verlangt, dass x nicht 0 ist und eine Assertion ausgibt, die den Fehler beschreibt.

```
from gpanel import *
from math import pi, sin

def sinc(x):
    if x == 0:
        return 1.0
    y = sin(x) / x
    return y
```

```
makeGPanel(-24, 24, -1.2, 1.2)
drawGrid(-20, 20, -1.0, 1, "darkgray")
title("Sinus Cardinalis: y = sin(x) / x")

x = -20
dx = 1
while x <= 20:
    y = sinc(x)
    if x == -20:
        move(x, y)
    else:
        draw(x, y)
    x += dx
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Die Fehlermeldung ist jetzt bedeutend besser, da genau gesagt wird, wo er auftritt.

Viel besser wäre es allerdings, wenn A die Funktion so schreiben würde, dass für x = 0 der Grenzwert 1 der Funktion zurückgegeben wird.

```
from gpanel import *
from math import pi, sin

def sinc(x):
    if x == 0:
        return 1.0
    y = sin(x) / x
    return y

makeGPanel(-24, 24, -1.2, 1.2)
drawGrid(-20, 20, -1.0, 1, "darkgray")
title("Sinus Cardinalis: y = sin(x) / x")

x = -20
dx = 1
while x <= 20:
    y = sinc(x)
    if x == -20:
        move(x, y)
    else:
        draw(x, y)
    x += dx
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Dieser Code widerspricht allerdings der Grundregel, dass Gleichheitstests mit Floats wegen möglicher Rundungsfehler gefährlich sind. Noch besser wäre es also, mit einer Epsilon-Schranke zu testen:

```
def sinc(x):
    epsilon = 1e-100
    if abs(x) < epsilon:
        return 1.0
```

Deine Funktion *sinc(x)* ist aber immer noch nicht sicher, dann eine weitere Precondition ist sicher, dass x eine Zahl sein muss. Ruft man *sinc(x)* mit dem Wert *x = "python"* auf, so ergibt sich wiederum ein böser Laufzeitfehler.

```
from math import sin

def sinc(x):
    y = sin(x) / x
```

```
    return y

print sinc("python")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Hier zeigt sich der Vorteil von Programmiersprachen mit einer Variablendeklarationen, denn dieser Fehler würde bereits vor der Programmausführung als Syntaxfehler entdeckt.

```
Hier braucht Python einen Wert vom Typ float.
    y = sin(x) / x
```

DEBUGGING-INFORMATION AUSSCHREIBEN

Da davon auszugehen ist, dass jedermann beim Schreiben von Programmen Fehler macht, ist es wichtig, Fehler möglichst rasch zu finden und zu beheben. Erfolgreiche Programmierer zeichnen sich dadurch aus, Fehler rasch zu eliminieren [**mehr...**].

```
def exchange(x, y):
    y = x
    x = y
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Eine bekannte und einfache Strategie, um Fehler aufzufinden, besteht darin, an geeigneter Stelle die Werte von Variablen in die Konsole auszuschreiben:

```
def exchange(x, y):
    print "exchange() with params", x, y
    y = x
    x = y
    print "exchange() returning", x, y
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Damit wird offensichtlich, wo der Fehler passiert und man kann ihn leicht beheben.

```
def exchange(x, y):
    print "exchange() with params", x, y
    temp = y
    y = x
    x = temp
    print "exchange() returning", x, y
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Jetzt wo der Fehler behoben ist, sind die zusätzlichen Debug-Zeilen überflüssig. Statt sie zu löschen, kannst du sie auch nur auskommentieren, da du sie vielleicht später noch einmal benötigst.

```
def exchange(x, y):
#    print "exchange() with params", x, y
    temp = y
    y = x
    x = temp
#    print "exchange() returning", x, y
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Elegant ist die Verwendung eines Debug-Flags, mit dem du Debug-Informationen an verschiedenen Stellen aktivieren oder deaktivieren kannst.

```
def exchange(x, y):
    if debug: print "exchange() with params", x, y
    temp = y
    y = x
    x = temp
    if debug: print "exchange() returning", x, y
    return x, y

debug = False
a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Wie du wahrscheinlich bereits weisst, kannst du in Python elegant Variablenwerte ohne Verwendung einer Hilfsvariablen vertauschen. Du verwendest dabei das automatische Verpacken/Entpacken von Tuples

```
def exchange(x, y):
    x, y = y, x
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

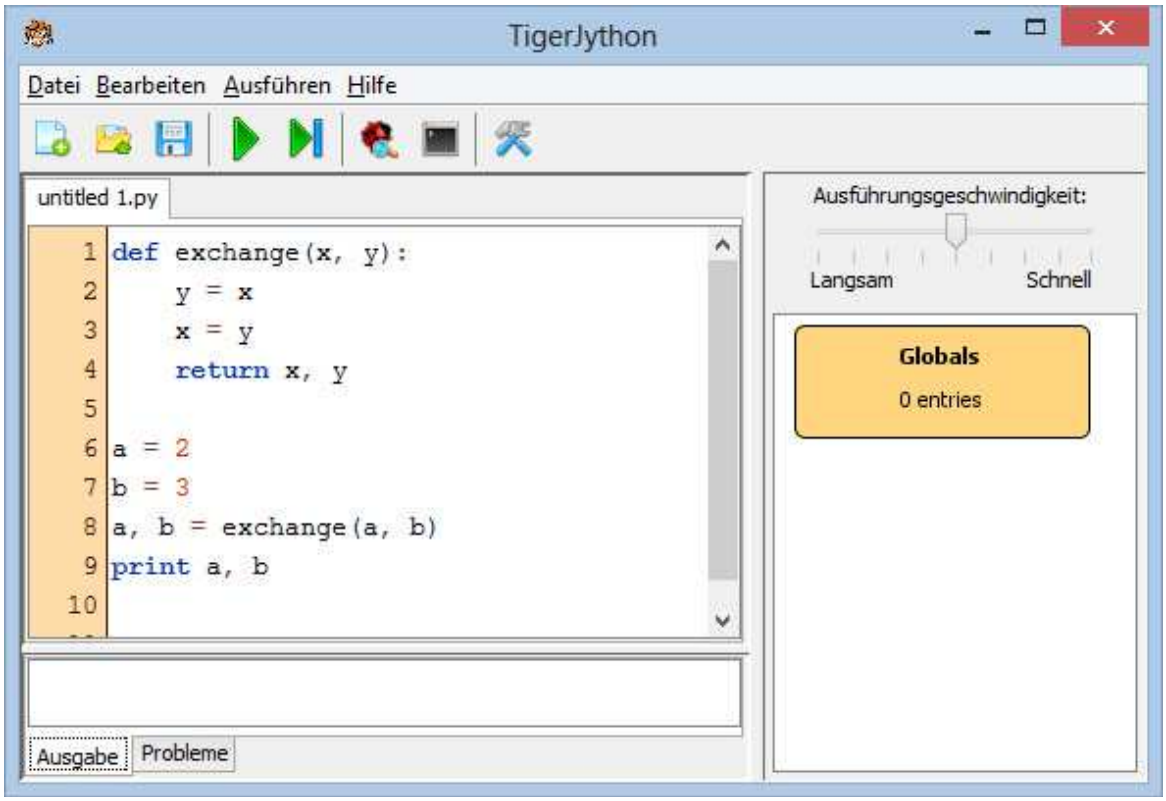
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

VERWENDUNG DES DEBUGGERS

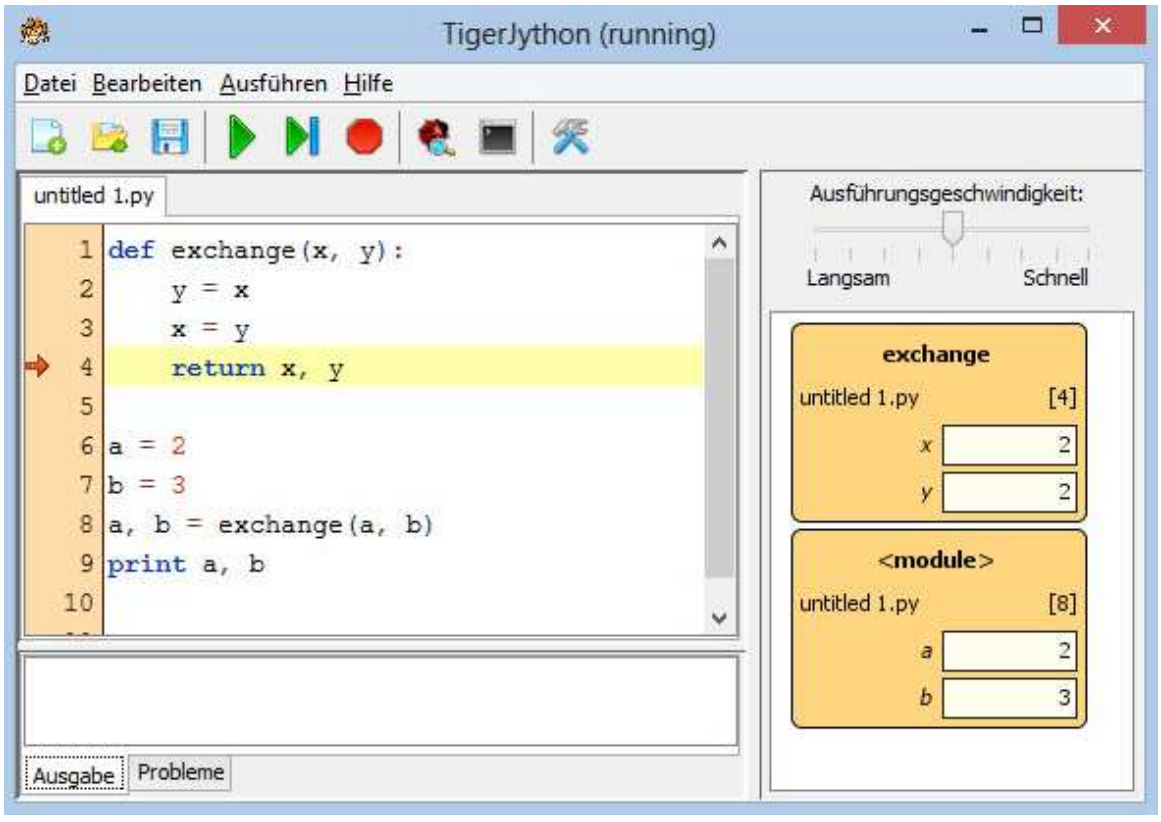
Debugger sind wichtige Hilfsmittel bei der Programmentwicklung von grossen Programmsystemen. Mit ihnen kannst du ein Programm langsam und sogar in Einzelschritten ausführen. Dabei wird sozusagen die enorme Ausführungsgeschwindigkeit des Computers dem beschränkten menschlichen Auffassungsvermögen angepasst. In TigerJython ist ein einfacher Debugger eingebaut, der dir auch helfen kann, den Programmablauf in einem korrekten Programm besser zu verstehen. Du untersuchst nun das oben verwendete, fehlerhafte Programm mit dem Debugger.

Nachdem du es in den Editor genommen hast, klickst du auf den Debugger-Knopf

Mit dem Einzelschritt-Button kannst du das Programm Schritt um Schritt ausführen und dabei im Debugger-Fenster die Variablen beobachten. Nach 8x Klicken siehst du, dass vor der Rückkehr aus der Funktion *exchange()* die beiden Variablen x und y den Wert 2 haben.

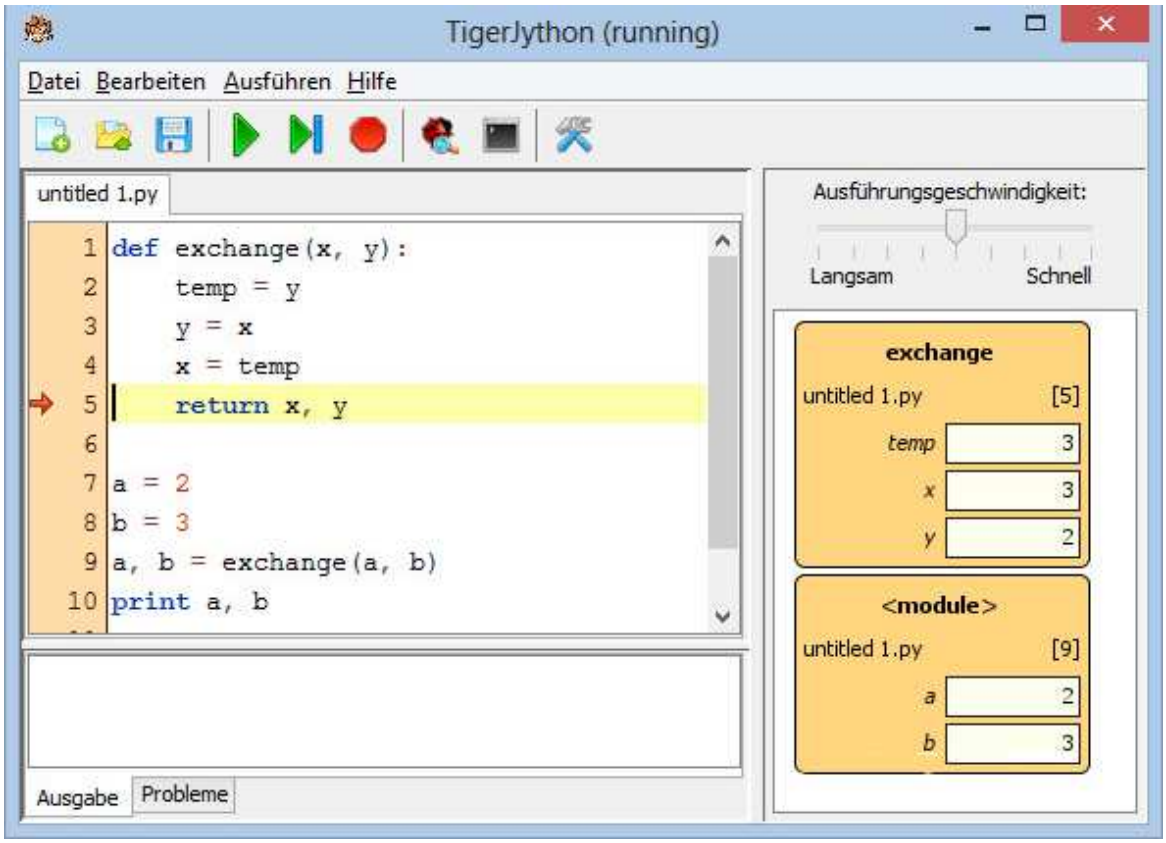


Führst du nun den Bugfix aus, so kannst du beobachten, wie in *exchange()* die Variablen einander zugewiesen werden, was schliesslich zum richtigen Resultat führt. Gut sichtbar ist auch die kurze Lebensdauer der lokalen Variablen x und y gegenüber den globalen Variablen a und b.



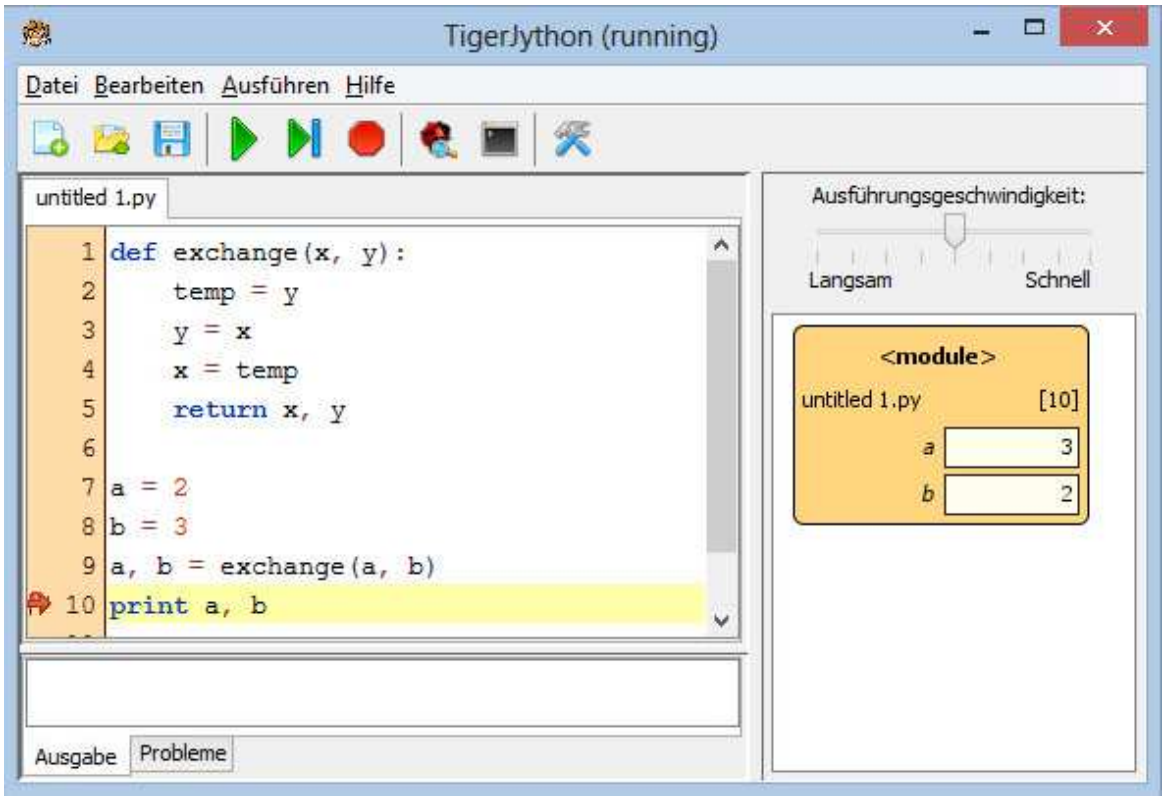
Im Programm kannst du auch Haltepunkte (Breakpoints) setzen, damit du nicht mühsam im

Einzelnschrittmodus unkritische Programmteile durchlaufen musst. Dazu klickst du ganz links auf die Zeilennummer-Spalte. Es erscheint eine kleine Flaggenikone, welche den Haltepunkt markiert.



Beim Drücken des Run-Buttons läuft nun das Programm bis zu dieser Stelle und hält an. Du kannst nun den momentanen Zustand der Variablen inspizieren und mit der Run-Button das Programm weiter führen oder mit dem Einzelschritt-Button schrittweise untersuchen.

Du kannst auch mehrere Haltepunkte setzen. Um einen Haltepunkt zu löschen, klickst du einfach auf die Flaggenikone.



FEHLERABFANG MIT EXCEPTIONS

Klassisch ist das Verfahren, Fehler mit Exceptions abzufangen. Man setzt dazu den kritischen Programmcode in einen try-Block. Tritt der Fehler auf, so wird der Block verlassen und das Programm fährt im except-Block weiter, wo du angepasst auf den Fehler reagieren musst. Im schlimmsten Fall, stoppst du die Programmausführung mit dem Aufruf von *sys.exit()*.

Du kannst beispielsweise den Fehler abfangen, falls in *sinc(x)* der Parameter kein numerischer Datentyp ist.

```
from sys import exit
from math import sin

def sinc(x):
    try:
        if x == 0:
            return 1.0
        y = sin(x) / x
    except TypeError:
        print "Error in sinc(x). x =", x, "is not a number"
        exit()
    return y

print sinc("python")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Die Postcondition für *sinc(x)* könnte auch heissen, dass der Rückgabewert *None* ist, falls der Parameter einen falschen Typ hat. Es ist dann Aufgabe des Anwenders, diesen Fehler angepasst zu behandeln.

```
from math import sin

def sinc(x):
    try:
        if x == 0:
            return 1.0
        y = sin(x) / x
    except TypeError:
        return None
    return y

y = sinc("python")
if y == None:
   print "Illegal call"
else:
   print y
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

11.4 PARALLELVERARBEITUNG

EINFÜHRUNG

Man kann einen Computer gemäss dem von Neumann-Modell als eine sequentielle Maschine auffassen, die auf Grund eines Programms in Zeitschritten Anweisung um Anweisung abarbeitet. In diesem Modell gibt es keine gleichzeitig ablaufenden Aktionen, also keine **Parallelverarbeitung** bzw. **Nebenläufigkeit**. Im täglichen Leben sind aber parallele Abläufe allgegenwärtig: So existiert und handelt jedes Lebewesen als eigenständiges Individuum und im menschlichen Körper laufen viele Prozesse gleichzeitig ab.

Die Vorteile der Parallelverarbeitung sind evident: Sie bringt eine enorme Leistungssteigerung, da Aufgaben in gleichen Zeitschlitzen gelöst werden. Zudem erhöht sich die Redundanz und Überlebenschance, da der Ausfall eines Teils nicht automatisch zum Versagen des ganzen Systems führt.

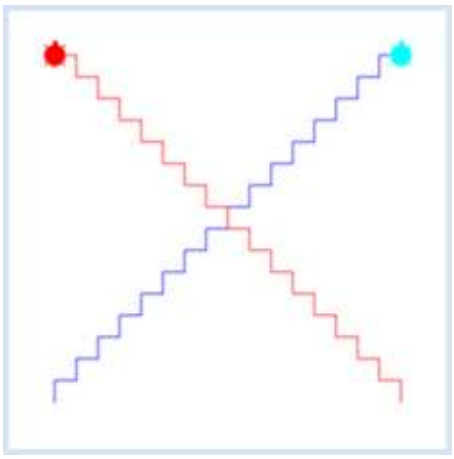
Das Parallelisieren von Algorithmen ist aber eine anspruchsvolle Herausforderung, die trotz grossen Anstrengungen noch immer in den Kinderschuhen steckt. Das Problem liegt vor allem daran, dass die Teilprozesse meist geteilte Ressourcen verwenden und auf Ergebnisse anderer Prozesse warten müssen.

Unter einem **Thread** versteht man parallel laufender Code innerhalb desselben Programms und unter einem **Prozess** versteht man Code, der durch das Betriebssystem gesteuert parallel ausgeführt wird. Python bietet eine gute Unterstützung von beiden Arten der Parallelität. Hier betrachten wir aber nur die Verwendung von mehreren Threads, also das **Multithreading**.

MULTITHREADING IST EINFACHER ALS ES SCHEINT

In Python ist es sehr einfach, den Code einer deiner Funktionen von einem eigenen Thread ausführen zu lassen: Dazu importierst du das Modul *threading* und übergibst *start_new_thread()* den Funktionsnamen sowie eventuelle Parameterwerte, die du in ein Tupel verpackst. Der Thread beginnt sofort zu laufen und führt den Code deiner Funktion aus.

In deinem ersten Programm mit Threads sollen zwei Turtles unabhängig voneinander eine Treppe zeichnen. Dazu schreibst du eine beliebig benannte Funktion, hier mit *paint()* bezeichnet, die auch Parameter haben darf, beispielsweise hier die Turtle und ein Flag, das angibt, ob die Turtle eine Links- oder Rechtstreppe zeichnet. Der Funktion *thread.start_new_thread()* übergibst du als Parameter den Funktionsnamen und ein Tupel mit den Parameterwerten. Und schon geht's los!



```
from gturtle import *
import thread

def paint(t, isLeft):
    for i in range(16):
        t.forward(20)
        if isLeft:
```

```
            t.left(90)
        else:
            t.right(90)
        t.forward(20)
        if isLeft:
            t.right(90)
        else:
            t.left(90)

tf = TurtleFrame()
john = Turtle(tf)
john.setPos(-160, -160)
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(160, -160)
thread.start_new_thread(paint, (john, False))
thread.start_new_thread(paint, (laura, True))
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Damit sich die beiden Turtles im gleichen Fenster bewegen, verwendest du ein *TurtleFrame tf* und übergibt es dem Turtle-Konstruktor.

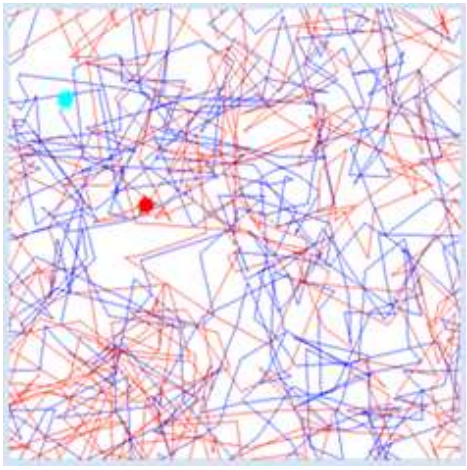
Mit *start_new_thread()* wird ein neuer Thread erzeugt und auch gleich gestartet. Der Thread terminiert, sobald die übergebene Funktion zurückkehrt.

Die Parameterliste muss als Tupel angegeben werden. Beachte, dass für eine Funktion ohne Parameter ein leeres Tupel () übergeben werden muss und dass man ein Tupel mit einem einzigen Element x nicht mit *(x)*, sondern mit *(x,)* angibt.

THREAD ALS KLASSENINSTANZ ERZEUGEN, STARTEN

Etwas mehr Spielraum erhältst du, wenn du eine eigene Klasse definierst, die von der Klasse *Thread* abgeleitet ist. In dieser Klasse überschreibst du die Methode *run(),* die den auszuführenden Code enthält.

Um den Thread zu starten, erzeugst du zuerst eine Instanz und rufst die Methode *start()* auf. Das System wird dann die Methode *run()* automatisch in einem neuen Thread ausführen und den Thread beenden, sobald *run()* zu Ende läuft.



```
from threading import Thread
import random
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
        Thread.__init__(self)
        self.t = turtle

    def run(self):
        while True:
```

```
            self.t.forward(150 * random.random())
            self.t.left(-180 + 360 * random.random())

tf = TurtleFrame()
john = Turtle(tf)
john.wrap()
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.wrap()
thread1 = TurtleAnimator(john)
thread2 = TurtleAnimator(laura)
thread1.start()
thread2.start()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Selbst bei einem Multiprozessor-System wird der Code nicht echt parallel ausgeführt, sondern in nacheinander folgenden **Zeitschlitzen** (time slices). Es handelt sich also in den meisten Fällen nur um eine **quasi-parallele** Datenverarbeitung. Wichtig ist aber, dass die Zuteilung des Prozessors auf die Threads zu unvorhersehbaren Zeitpunkten erfolgt, also irgendwo mitten in deinem Code. Zwar werden dabei die Unterbrechungsstelle und die lokalen Variablen automatisch gerettet und bei der Weiterführung wieder hergestellt, aber es kann Probleme geben, wenn in der Zwischenzeit andere Threads gemeinsam genutzte globale Daten verändern. Dazu gehört auch der Inhalt eines Grafikfensters. Daher ist es nicht selbstverständlich, dass sich die beiden Turtles nicht in die Quere kommen [**mehr...**].

Wie du feststellen kannst, läuft der Hauptteil des Programm zu Ende, aber die beiden Threads führen ihre Arbeit immer noch aus, bis das Fenster geschlossen wird.

THREAD BEENDEN

Ein einmal angestossener Thread kann nicht direkt mit einer Methode von aussen, also durch einen anderen Thread gestoppt werden. Um einen Thread anzuhalten, muss dafür gesorgt werden, dass die Methode *run()* zu Ende läuft. Darum ist eine nicht abbrechbare while-Schleife ist in der Methode *run()* eines Threads nie eine gute Idee. Statt dessen verwendest du für die while-Schleite eine globale boolesche Variable *isRunning*, die normalerweise auf *True* steht, aber von einem anderen Thread auf *False* gesetzt werden kann.

In deinem Programm führen beide Turtles eine Zufallsbewegung aus, bis sich eine der beiden über eine Kreisfläche hinausbewegt.

```
from threading import Thread
import random, time
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
        Thread.__init__(self)
        self.t = turtle

    def run(self):
        while isRunning:
            self.t.forward(50 * random.random())
            self.t.left(-180 + 360 * random.random())

tf = TurtleFrame()
john = Turtle(tf)
laura = Turtle(tf)
```

```
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(-200, 0)
laura.rightCircle(200)
laura.setPos(0, 0)
thread1 = TurtleAnimator(john)
thread2 = TurtleAnimator(laura)
isRunning = True
thread1.start()
thread2.start()

while isRunning and not tf.isDisposed():
    if laura.distance(0, 0) > 200 or john.distance(0, 0) > 200:
        isRunning = False
    time.sleep(0.001)
tf.setTitle("Grenze überschritten")
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

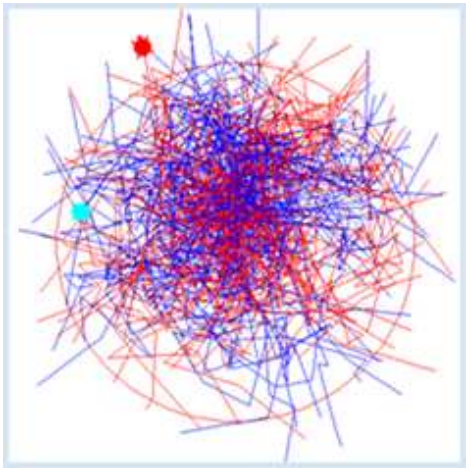
MEMO

Du solltest nie eine "enge" Schleife verwenden, die im Körper keine Aktion durchführt, da du damit viel Prozessorzeit vergeudest. Setze mit *time.sleep()*, *Turtle.sleep*() oder *GPanel.delay()* immer mindestens eine kleine Wartezeit von einigen Millisekunden ein.

Ein einmal beendeter Thread kann nicht nochmals angestossen werden. Versuchst du nochmals *start()* aufzurufen, gibt es eine Fehlermeldung.

THREAD ANHALTEN UND WEITERFÜHREN

Um einen Thread nur während einer bestimmten Zeit anzuhalten, kannst du mit einem globalen Flag *isPaused* die Aktionen in *run()* überspringen und später mit *isPaused* = *False* wieder weiterführen.



```
from threading import Thread
import random, time
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
        Thread.__init__(self)
        self.t = turtle

    def run(self):
        while True:
            if isPaused:
                Turtle.sleep(10)
            else:
                self.t.forward(100 * random.random())
                self.t.left(-180 + 360 * random.random())
```

```
tf = TurtleFrame()
john = Turtle(tf)
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(-200, 0)
laura.rightCircle(200)
laura.setPos(0, 0)
thread1 = TurtleAnimator(john)
thread2 = TurtleAnimator(laura)
isPaused = False
thread1.start()
thread2.start()

tf.setTitle("Running")
while not isPaused and not tf.isDisposed():
    if laura.distance(0, 0) > 200 or john.distance(0, 0) > 200:
        isPaused = True
        tf.setTitle("Paused")
        Turtle.sleep(2000)
        laura.home()
        john.home()
        isPaused = False
        tf.setTitle("Running")
    time.sleep(0.001)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Eleganter ist es, den Thread mit *Monitor.putSleep()* anzuhalten und später mit *Monitor.wakeUp()* weiterzuführen.

```
from threading import Thread
import random, time
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
        Thread.__init__(self)
        self.t = turtle

    def run(self):
        while True:
            if isPaused:
                Monitor.putSleep()
            self.t.forward(100 * random.random())
            self.t.left(-180 + 360 * random.random())

tf = TurtleFrame()
john = Turtle(tf)
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(-200, 0)
laura.rightCircle(200)
laura.setPos(0, 0)
thread1 = TurtleAnimator(john)
thread2 = TurtleAnimator(laura)
isPaused = False
thread1.start()
thread2.start()

tf.setTitle("Running")
while not isPaused and not tf.isDisposed():
    if laura.distance(0, 0) > 200 or john.distance(0, 0) > 200:
        isPaused = True
        tf.setTitle("Paused")
        Turtle.sleep(2000)
        laura.home()
```

```
        john.home()
        isPaused = False
        Monitor.wakeUp()
        tf.setTitle("Running")
    time.sleep(0.001)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Ein Thread kann sich selbst mit der blockierenden Methode *Monitor.putSleep()* so anhalten, dass er keine Rechenzeit mehr verbraucht. Ein anderer Thread kann ihn mit *Monitor.wakeUp()* wieder aktivieren, d.h. die blockierende Methode *Monitor.putSleep()* kehrt zurück.

AUF THREADRESULTATE WARTEN

In diesem Programm beschäftigst du eine Arbeitskraft damit, die Summe von natürlichen Zahlen von 1 bis 1000000 durch einfaches Aufsummieren zu berechnen. Im Hauptprogramm wartest du, bis die Arbeit erledigt ist und bestimmst die dafür benötigte Zeit. Da diese etwas schwankt, lässt du die Arbeit von einem Worker-Thread 10 Mal durchführen. Um auf das Ende des Threads abzuwarten, verwendest du *join()*.

```
from threading import Thread
import time

class WorkerThread(Thread):
     def __init__(self, begin, end):
         Thread.__init__(self)
         self.begin = begin
         self.end = end
         self.total = 0

     def run(self):
         for i in range(self.begin, self.end):
             self.total += i

startTime = time.clock()
repeat 10:
    thread = WorkerThread(0, 1000000)
    thread.start()
    thread.join()
    print thread.total
print "Time elapsed:", time.clock() - startTime, "s"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Wie auch im täglichen Leben kannst du die mühsame Arbeit auf mehrere Arbeitskräfte verteilen. Wenn du zwei Worker-Threads dazu einsetzest, um je die Hälfte der Arbeit zu verrichten, so musst du auf das Ende der beiden warten, bevor du die Summe bildest.

```
from threading import Thread
import time

class WorkerThread(Thread):
     def __init__(self, begin, end):
         Thread.__init__(self)
         self.begin = begin
         self.end = end
         self.total = 0

     def run(self):
         for i in range(self.begin, self.end):
```

```
            self.total += i

startTime = time.clock()
repeat 10:
    thread1 = WorkerThread(0, 500000)
    thread2 = WorkerThread(500000, 1000000)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    result = thread1.total + thread2.total
    print result
print "Time elapsed:", time.clock() - startTime, "s"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du könntest auch beim Terminieren der Threads ein globales Flag *isFinished()* auf *True* setzen und im Hauptteil in einer Warteschleife dieses Flag testen. Diese Lösung ist aber weniger elegant als die Verwendung von *join()*, denn du vergeudest Rechenzeit, weil du ständig das Flag testen musst.

Mit zwei Threads ist die Rechenzeit zwar etwas kleiner. Der Unterschied ist aber gering, da das Programm nicht echt parallel, sondern in sequentiellen Zeitschlitzen abläuft und für die Thread-Umschaltung auch eine gewisse Zeit benötigt wird [**mehr...**]

KRITISCHE BEREICHE UND LOCKS

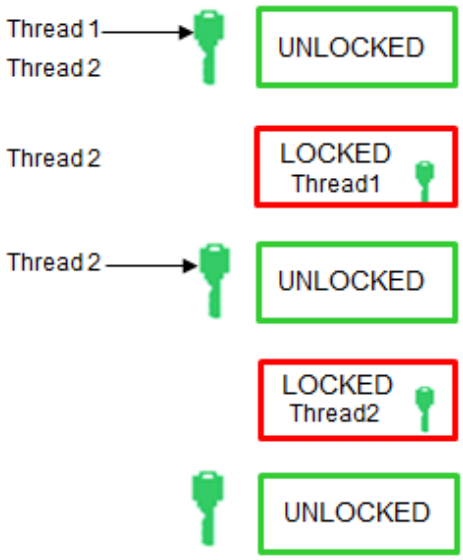
Da Threads quasi unabhängig voneinander Code ausführen, ist es heikel, falls mehrere Threads gemeinsame Daten verändern. Um Kollisionen zwischen Threads zu vermeiden, werden zusammengehörende Aktionen in einem sogenannten **kritischen Programmblock** zusammengefasst und mit einem Schutz versehen, so dass der Block nur ununterbrochen als Ganzes (*atomar*) ausgeführt wird. Versucht ein anderer Thread den Block auszuführen, so muss er warten, bis der aktuelle Thread den Block verlassen hat. Diesen Schutz realisiert man in Python mit einer **Sperre** (**Lock**). Eine Sperre ist eine Instanz der Klasse *Lock* und besitzt zwei Zustände *gesperrt (locked)* und *entsperrt (unlocked)* , sowie zwei Methoden *acquire()* und *release()* mit folgenden Regeln:

Zustand	Aufruf	Folgezustand/Wirkung
unlocked	acquire()	locked
locked	acquire()	blockiert, bis ein anderer Thread release() aufruft
unlocked	release()	Fehlermeldung (RuntimeException)
locked	release()	unlocked()

Man sagt anschaulich, dass ein Thread mit *acquire()* den Lock (die Sperre) *erhält* und mit *release()* den Lock (die Sperre) wieder *abgibt* [**mehr...**].

Zum Schutz eines kritischen Blocks gehst du konkret wie folgt vor: Du erzeugst zuerst mit *lock = Lock()* ein globales Lock-Objekt, das natürlich am Anfang im Zustand *unlocked* ist. Beim Eintritt in den kritischen Block versucht jeder Thread mit *acquire()* den Lock zu erhalten. Gelingt dies nicht, weil der Lock bereits vergeben wurden, so wird der Thread automatisch in einen Wartezustand versetzt, bis der Lock wieder frei wird. Hat ein Thread den Lock erhalten, so durchläuft er den kritischen Block und muss beim Verlassen den Lock mit *release()* wieder abgeben, damit andere Threads ihn bekommen [**mehr...**].

Wenn du das Durchlaufen des kritischen Blocks wie eine Ressource in einem mit einer Tür geschlossenen Raum auffasst, so kannst du dir einen Lock wie einen Schlüssel vorstellen, den ein Thread benötigt, um die Tür zum Raum zu öffnen. Er nimmt beim Eintritt den Schlüssel mit und verschliesst von Innen die Tür. Alle Threads, die nun in den Raum eintreten wollen, müssen in einer Warteschlange vor der Tür auf den Schlüssel warten. Wenn der Thread seine Arbeit verrichtet hat, verlässt er den Raum, schliesst die Tür hängt den Schlüssel auf. Der erste wartende Thread nimmt den Schlüssel und kann damit seinerseits die Türe zum Raum öffnen. Wartet kein Thread, so bleibt der Schlüssel aufgehängt, bis ihn ein neu ankommender Thread benötigt.



In deinem Programm besteht der kritische Block aus dem Zeichnen und Löschen eines gefüllten Quadrats, wobei beim Löschen das Quadrat mit der weissen Hintergrundfarbe übermalt wird. Der Hauptthread erzeugt ein blinkendes Quadrat, indem er das rot gefüllte Quadrat zeichnet und nach einer bestimmten Wartezeit wieder löscht. In einem zweiten Thread *MyThread* wird mit *getKeyCode()* die Tastatur laufend abgefragt. Drückt der Benutzer die Leertaste, so wird das blinkende Quadrat an eine zufällige Position verschoben.

Es versteht sich von selbst, dass der kritische Block mit einem Lock geschützt werden muss. Erfolgt nämlich das Verschieben des Quadrats noch während dem Zeichnen und Löschen, so ergibt sich ein chaotisches Verhalten.

```
from gpanel import *
from threading import Thread, Lock
import random

class MyThread(Thread):
    def run(self):
        while not isDisposed():
            if getKeyCode() == 32:
                print "----------- Lock requested by MyThread"
                lock.acquire()
                print "----------- Lock acquired by MyThread"
                move(random.randint(2, 8), random.randint(2, 8))
                delay(500)  # for demonstration purposes
                print "----------- Lock releasing by MyThread..."
                lock.release()
            else:
                delay(1)

def square():
    print "Lock requested by main"
    lock.acquire()
    print "Lock acquired by main"
    setColor("red")
    fillRectangle(2, 2)
    delay(1000)
    setColor("white")
    fillRectangle(2, 2)
    delay(1000)
    print "Lock releasing by main..."
    lock.release()

lock = Lock()
makeGPanel(0, 10, 0, 10)

t = MyThread()
t.start()
```

```
move(5, 5)
while not isDisposed():
    square()
    delay(1) # Give up thread for a short while
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

In der Konsole kannst du verfolgen, wie jeder Thread folgsam wartet, bis der Lock frei ist:

```
Lock requested by main
Lock acquired by main
---------- Lock requested by MyThread
Lock releasing by main...
---------- Lock acquired by MyThread
Lock requested by main
---------- Lock releasing by MyThread...
Lock acquired by main
```

Deaktivierst du durch Auskommentieren den Lock, so stellst du fest, dass die Quadrate nicht mehr korrekt gezeichnet und gelöscht werden.

Beachte auch, dass du in kurzen Schleifen immer eine kleine Wartezeit einbauen solltest, um nicht unnötige Prozessorzeit zu verbrauchen.

GUI-WORKERS

Callbacks, die durch GUI-Komponenten ausgelöst werden, laufen in einem bestimmten systemeigenen Thread (manchmal Event Dispatch Thread (EDT) genannt). Dieser ist dafür verantwortlich, dass das gesamte Grafikfenster mit allen Komponenten (Buttons, usw.) korrekt auf dem Bildschirm gerendert wird. Da das Rendern am Ende des Callbacks erfolgt, erscheint das **GUI eingefroren**, bis der Callback zurückkehrt. Es sind darum in einem GUI-Callback keine grafischen Animationen möglich. Du musst dich unbedingt an folgende Regel halten:

GUI-Callbacks müssen rasch zurückkehren, d.h in GUI-Callbacks dürfen keine lange dauernden Operationen ausgeführt werden.

Unter *lange dauernd* versteht man Zeiten von mehr als einige 10 ms. Dabei musst du aber vom schlimmsten Fall, d.h. von einer langsamen Hardware und grosser Systembelastung ausgehen. Dauert eine Aktion länger, so führst du sie in einem eigenen Thread aus, den man **GUI-Worker** nennt.

In deinem Programm zeichnest du mit einem Klick auf einen der zwei Buttons eine Rhodonea-Rosette. Das Zeichnen ist animiert und dauert eine gewisse Zeit. Du musst das Zeichnen daher in einem Worker-Thread ausführen, was mit deinen bisherigen Kenntnissen ja kein Problem ist.

Es gibt aber noch ein anderes Problem zu beachten: Da jeder Buttonklick einen neuen Thread erzeugt, können mehrere Zeichnungen kurz nacheinander gestartet werden, was zu einem Chaos führt. Du kannst dieses verhindern, wenn du die Buttons während der Ausführung der Zeichnung *grau* (*inaktiv*) machst [**mehr...**]

```
from gpanel import *
from javax.swing import *
import math
import thread
```

```
def rho(phi):
    return math.sin(n * phi)

def onButtonClick(e):
    global n
    enableGui(False)
    if e.getSource() == btn1:
        n = math.e
    elif e.getSource() == btn2:
        n = math.pi
#    drawRhodonea()
    thread.start_new_thread(drawRhodonea, ())

def drawRhodonea():
    clear()
    phi = 0
    while phi < nbTurns * math.pi:
        r = rho(phi)
        x = r * math.cos(phi)
        y = r * math.sin(phi)
        if phi == 0:
          move(x, y)
        else:
          draw(x, y)
        phi += dphi
    enableGui(True)

def enableGui(enable):
    btn1.setEnabled(enable)
    btn2.setEnabled(enable)

dphi = 0.01
nbTurns = 100
makeGPanel(-1.2, 1.2, -1.2, 1.2)
btn1 = JButton("Go (e)", actionListener = onButtonClick)
btn2 = JButton("Go (pi)", actionListener = onButtonClick)
addComponent(btn1)
addComponent(btn2)
validate()
```

MEMO

In GUI-Callbacks darf nur kurz dauernder Code ausgeführt werden, da sonst das Grafiksystem eingefroren wird. Länger dauernden Code (mehr als einige 10 ms) musst du in einen eigenen Worker-Thread auslagern.

Auf einer grafischen Benutzeroberfläche dürfen in jedem Moment nur diejenigen Komponenten aktiv sein, deren Bedienung erlaubt und sinnvoll ist.

ZUSATZSTOFF

RACE CONDITIONS, DEADLOCKS

Der Mensch funktioniert zwar hochgradig parallel, sein logisches Denken ist aber weitgehend sequentiell. Aus diesem Grund ist es für uns Menschen schwierig, bei Programmen mit mehreren Threads den Überblick zu behalten. Darum sollte die Verwendung von Threads wohl überlegt werden, so elegant und herausfordernd sie auf den ersten Blick scheinen mag.

Abgesehen von Statistikprogrammen sollte ein Programm bei gleichen Anfangsbedingungen (Preconditions) auch immer die gleichen Resultate (Postconditions) abgeben. Dies ist bei

Programmen mit mehreren Threads, die auf gemeinsame Daten zugreifen, keineswegs gewährleistet, selbst wenn die kritischen Bereiche mit Locks geschützt sind. In deinem Programm hast du zwei Threads *thread1* und *thread2*, die eine Addition und eine Multiplikation mit zwei globalen Zahlen *a* und *b* vornehmen. *a* und *b* werden durch einen *lock_a* bzw. *lock_b* geschützt. Im Hauptteil erzeugst und startest du die beiden Threads nacheinander und wartest, bis sie zu Ende gelaufen sind. Zum Schluss schreibst du die Werte von *a* und *b* aus.

Für die Erzeugung der Threads verwendest du hier eine etwas andere Schreibweise, bei der du die Methoden *run()* als benannten Parameter im Konstruktor der Klasse *Thread* angibst.

```
from threading import Thread, Lock
from time import sleep

def run1():
    global a, b
    print "----------- lock_a requested by thread1"
    lock_a.acquire()
    print "----------- lock_a acquired by thread1"
    a += 5
#    sleep(1)
    print "----------- lock_b requested by thread1"
    lock_b.acquire()
    print "----------- lock_b acquired by thread1"
    b += 7
    print "----------- lock_a releasing by thread1"
    lock_a.release()
    print "----------- lock_b releasing by thread1"
    lock_b.release()

def run2():
    global a, b
    print "lock_b requested by thread2"
    lock_b.acquire()
    print "lock_b acquired by thread2"
    b *= 3
#    sleep(1)
    print "lock_a requested by thread2"
    lock_a.acquire()
    print "lock_a acquired by thread2"
    a *= 2
    print "lock_b releasing by thread2"
    lock_b.release()
    print "lock_a releasing by thread2"
    lock_a.release()

a = 100
b = 200
lock_a = Lock()
lock_b = Lock()

thread1 = Thread(target = run1)
thread1.start()
thread2 = Thread(target = run2)
thread2.start()
thread1.join()
thread2.join()
print "Result: a =", a, ", b =", b
```

MEMO

Lässt man das Programm mehrmals laufen, so ergibt sich als Resultat manchmal a = 205, b = 607 und manchmal a = 210, b =621. Wie ist dies möglich? Die Erklärung ist die folgende:

Obschon im Hauptteil thread1 vor thread2 erzeugt und gestartet wird, ist es nicht sicher, welcher der Thread tatsächlich mit der Abarbeitung zuerst beginnt. Als erste Zeile kann also

```
lock_a requested by thread1
```

oder

```
lock_b requested by thread2
```

ausgeschrieben werden. Auch der weitere Verlauf ist nicht eindeutig, da der Threadwechsel irgendwo geschehen kann. Je nachdem wird also mit den Zahlen a und b zuerst die Addition oder die Multiplikation ausgeführt, was die unterschiedlichen Resultate erklärt. Da die beiden Threads wie in einem Wettbewerb quasi miteinander laufen, ergibt sich eine **Wettbewerbssituation** (**race condition**).

Es kann aber noch viel schlimmer sein, denn das Programm kann auch **total blockieren.** Vor dem "Sterben" wird noch Folgendes ausgeschrieben:

```
---------- lock_a requested by thread1
lock_b requested by thread2
lock_b acquired by thread2
lock_a requested by thread2
---------- lock_a acquired by thread1
---------- lock_b requested by thread1
```

Es braucht etwas kriminalistische Fähigkeiten um herauszufinden, was da geschehen ist. Wir versuchen es: Offenbar beginnt der *thread1* als erster zu laufen und versucht, *lock_a* zu erhalten. Bevor er den Erhalt ausschreiben kann, versucht *thread2 lock_b* zu erhalten und erhält ihn auch. Gleich darauf versucht *thread2* auch *lock_a* zu erhalten, was offenbar misslingt, weil ihn in der Zwischenzeit *thread1* gekriegt hat. *thread2* wird also blockieren. *thread1* läuft weiter und versucht, *lock_b* zu erhalten, was ebenfalls misslingt, da *thread2* ihn ja noch nicht zurückgegeben hat. Also blockiert auch *thread1* und damit das ganze Programm. Sinnigerweise nennt man diese Situation einen **Deadlock. (**Wenn du die beiden auskommentierten Zeilen mit *sleep(1)* aktivierst, so ergibt sich immer ein Deadlock. Überlege warum.)

Wie du siehst, treten Deadlocks dann auf, wenn zwei Threads *thread1* und *thread2* auf zwei gemeinsame Ressourcen a und b angewiesen sind und diese einzeln blockieren. Dadurch kann es geschehen, dass *thread2* auf *lock_a* und *thread1* auf *lock_b* wartet und damit beide blockiert sind, sodass sie die Locks auch nie mehr freigegeben werden.

Um Deadlocks zu vermeiden, solltest du dich deshalb an folgende Regel halten:

Gemeinsame Ressourcen sollten wenn immer möglich mit einem einzigen Lock geschützt werden. Zudem muss gewährleistet sein, dass der Lock auch sicher wieder zurückgegeben wird.

THREADSICHER UND ATOMAR

Sind mehrere Threads im Spiel, so weiss man als Programmierer nie so genau, zu welchem Zeitpunkt oder an welcher Stelle des Codes die Umschaltung der Threads erfolgt. Wie du bereits vorher gesehen hast, kann dies dann zu unerwartetem und falschem Verhalten führen, wenn die Threads mit denselben Ressourcen arbeiten. Dies ist insbesondere dann der Fall ist, wenn mehrere Threads ein Bildschirmfenster verändern. Wenn du also in einem Callback einen eigenen Worker-Thread erzeugst, um länger laufenden Code auszuführen, so musst du fast immer damit rechnen, dass es ein Chaos geben kann. Im vorhergehenden Programm hast du dies vermieden, indem du während dem Callback die Buttons inaktiviert hast. Durch besondere Vorsichtsmassnahmen kann man erreichen, dass mehrere Threads denselben Code ausführen können, ohne sich in die Quere zu kommen. Solchen Code nennt man **threadsicher** (**threadsafe**). Es ist eine Kunst, threadsicheren Code zu schreiben, der sich also in einer Umgebung mit mehreren Threads konfliktlos verwenden lässt [**mehr..**].

Es gibt wenige **threadsichere Bibliotheken**, da diese weniger Performat sind und dei Gefahr von **Deadlocks** besteht. Wie du oben erlebt hast, ist die GPanel-Bibliothek nicht threadsicher, die Turtlegrafik hingegen schon. Du kannst also mehrere Turtles mit mehreren Threads quasi-gleichzeitig bewegen. In deinem Programm entsteht bei jedem Mausklick ein neuer Thread, der an der Mausposition eine neue Turtle erzeugt. Diese zeichnet selbsttändig einen Stern und füllt ihn nachher aus.

```
from gturtle import *
import thread

def onMousePressed(event):
#    createStar(event)
    thread.start_new_thread(createStar, (event,))

def createStar(event):
    t = Turtle(tf)
    x = t.toTurtleX(event.getX())
    y = t.toTurtleY(event.getY())
    t.setPos(x, y)
    t.startPath()
    repeat 9:
        t.forward(100)
        t.right(160)
    t.fillPath()

tf = TurtleFrame(mousePressed = onMousePressed)
tf.setTitle("Klick To Create A Working Turtle")
```

MEMO

Erzeugst du keinen neuen Thread (auskommentierte Zeile), so siehst du erst die fertig gezeichneten Sterne. Du kannst aber das Programm ohne eigenen Thread schreiben, wenn du an Stelle von *mousePressed* den benannten Parameter *mouseHit* verwendest, so wie du es im **Kapitel 2.11** gemacht hast. Dabei wird der Thread automatisch in der Turtlebibliothek erzeugt.

Es ist wichtig, dass du weisst, dass die Umschaltung der Threads sogar mitten in einer Codezeile geschehen kann. Beispielsweise kann sogar in der Codezeile a = a + 1 bzw. a += 1 zwischen dem Lesen und Schreiben von a ein Threadswitch erfolgen, der den Variablenwert verändert.

Im Gegensatz dazu nennt man einen Ausdruck **atomar**, wenn er **nicht unterbrochen** werden kann. Wie in den meisten anderen Programmiersprachen ist auch in Python fast nichts atomar. Es kann also beispielsweise vorkommen, dass ein print-Befehl durch print-Befehle anderer Threads unterbrochen wird, was zu einem chaotischen Ausdruck führt. Es ist Aufgabe des Programmierers, durch Verwendung von Locks Funktionen, Ausdrücke und Codeteile threadsicher bzw. atomar zu machen.

AUFGABE

1. Verwende im oben stehenden Programm nur einen einzigen Lock und richte es so ein, dass keine Race Conditions und Deadlocks mehr gibt.

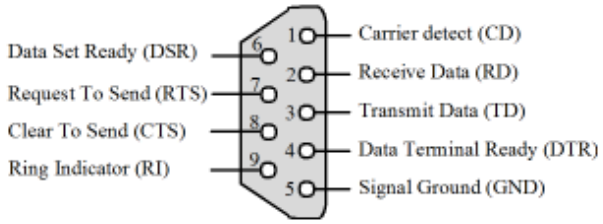
11.5 SERIELLE SCHNITTSTELLE

EINFÜHRUNG

Obschon für die Kommunikation zwischen einem Computer und Peripheriegeräten häufig die Bluetooth, Ethernet- oder USB-Schnittstelle eingesetzt wird, ist auch in Zukunft die Kommunikation über die serielle Schnittstelle (nach RS-232C) manchmal die bessere Lösung, da der schaltungstechnische Aufwand beim Peripheriegerät kleiner ist. Darum wird die serielle Schnittstelle immer noch zum Anschluss von Messgeräten (Voltmeter, Kathodenstrahl-Oszillografen, usw.), zur Steuerung von Apparaten und Robotern und zur Kommunikation mit Microcontrollern eingesetzt. Moderne Computer besitzen zwar keine serielle Schnittstelle mehr, mit preisgünstigen USB-Serial-Adaptern kann dieser Mangel aber leicht behoben werden.

Für das Verständnis der seriellen Schnittstelle ist es wichtig zu wissen, dass es je eine Datenleitung für das Senden und Empfangen der Daten (TD/RD), zwei Paare von Handshake-Leitungen RTS/CTS bzw. DTR/DSR, zwei Statusleitungen CD/RI und einen Ground gibt. Vom Computer aus gesehen sind die Leitungen TD, RTS, DTR Ausgänge, die Leitungen RD, CTS, DSR, CD, RI Eingänge. RTS und DTR können also programmgesteuert aktiviert und deaktiviert werden, CTS, DSR, CD und RI können nur gelesen werden.

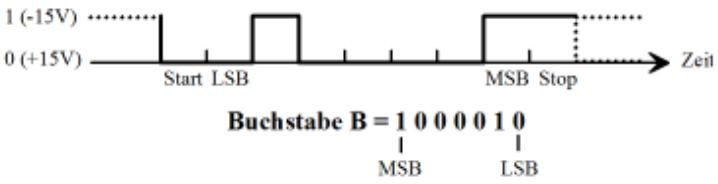
Anschlüsse beim 9-poligen RS-232-Stecker



Das Format der übertragenen Daten ist einfach. Es werden zeitlich hintereinander Datenbytes übertragen. Die Übertragung beginnt mit einem Startbit, das den Empfänger auf die bevorstehende Datenübertragung aufmerksam macht. Es folgen die Daten selbst, die 5, 6, 7 oder (gewöhnlich) 8 bits umfassen. Um eine Fehlerkorrektur zu ermöglichen, folgt nachher ein Paritätsbit, das angibt, ob im aktuell übertragenden Datenbyte eine gerade oder ungerade Zahl von Datenbits gesetzt sind, wobei das Paritätsbit auch entfallen kann. Die Übertragung wird mit einem oder zwei Stopbits beendet. Das sendende und empfangende Gerät sind nicht miteinander synchronisiert, d.h. die Datenübertragung kann irgend einmal beginnen und wieder enden. Immerhin ist es nötig, dass die beiden Geräte dieselbe zeitliche Dauer eines einzelnen Bits vereinbaren. Diese wird durch die Baudrate (in baud, bits/s) angegeben und kann in der Regel nur die standardisierten Werte 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 baud annehmen. Die beiden Geräte können zudem noch ein Handshake (flow control) vereinbaren, mit dem sie einander mitteilen, ob sie für den Datentransfer bereit sind. Man unterscheidet zwischen Hard- und Software-Handshake, je nachdem ob die Handshake-Leitungen benützt oder ob der Handshake mit speziellen ASCII-Zeichen (XON/XOFF) erfolgt, die in den Datenstrom eingebettet werden.

Eine typische Port-Konfiguration umfasst daher: Baudrate, Anzahl Datenbits, Anzahl Stopbits, Parity none, odd or even, Handshake none, hard- oder software.

Spannungsverlauf bei der Übertragung des Buchstabens 'B', mit der Konfiguration *7 databit/no parity/1 stopbit*



INSTALLATION

Die hier beschriebene Verwendung des Moduls **pySerial** funktioniert unter einem 32- oder 64-bit Betriebssystem, allerdings nur mit einem 32-bit-Java Runtime Environment (JRE). Man kann dieses von **hier** downloaden. Folgende weitere Installationsschritte sind nötig (*Lib* ist ein Unterverzeichnis des Verzeichnisses, in dem sich *tigerjython2.jar* befindet):

1. **Download** von *pyserial.zip*. Auspacken und ganze Verzeichnisstruktur in das Verzeichnis *Lib* kopieren

   <tigerjythonhome>
   Lib
   Serial
   tools
   urhandler

2. **Download** von *javacomm.zip*. Auspacken und *comm.jar* in *Lib* kopieren.

3. (Unter Windows): Aus *javacomm.zip* die Datei *win32com.dll* in *c.\windows\system32* kopieren. Unter *c:\Program Files (x86)* findet man das Verzeichnis *Java* und das Homeverzeichnis der JRE. Aus *javacomm.zip* die Datei *javax.com.properties* in das Unterverzeichnis *lib* kopieren. (Achtung: Bei Updates der JRE geht diese Datei eventuell verloren.)

4. Falls nötig, den Treiber des verwendeten USB-Serial-Adapters installieren. Adapter anschliessen und in den Adapter-Eigenschaften (im Geräte-Manager) den verwendeten COM-Port herausfinden/neu setzen, z.B. COM1.

5. Test der Installation durch Ausführen von *PortEnumerator.jar* (aus *javacomm.zip*). Der Com-Port muss angezeigt werden.

EINFACHES TERMINAL

Du solltest dich vorerst etwas in der Dokumentation von *pySerial* umsehen. Du findest auf **http://pyserial.sourceforge.net** unter *pySerial API* eine vollständige Beschreibung der Klassen. Einige davon sind allerdings plattformspezifisch. Mit deinem Programm kannst du Zeichen, die auf der Tastatur eingegeben werden, zeichenweise an einen externen Device senden und von ihm empfangene Zeichen in einer Konsole ausschreiben. Es handelt sich also um die einfachste Form eines Terminal-Emulators.

Zum Test kannst du entweder zwei Computer über eine Link-Kabel, das insbesondere RD (Receive Data) und TD (Transmit Data) vertauscht, miteinander verbinden und auf beiden das Programm laufen lassen. Du kannst auch nur diese beiden Pins miteinander verbinden (mit einer Klammer oder ähnlichem Gegenstand kurzschliessen). Dann werden alle gesendeten Zeichen sofort wieder empfangen.

```
import serial
from gconsole import *

makeConsole()
```

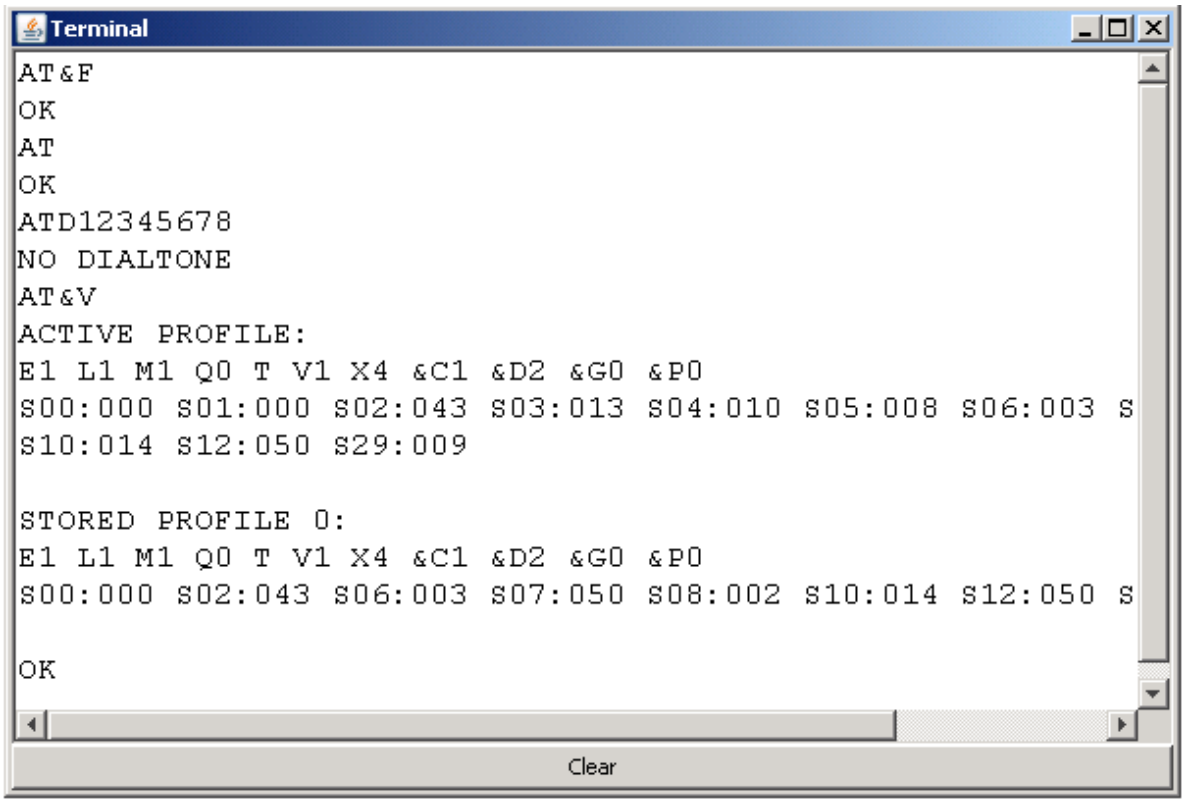
```
setTitle("Terminal")
ser = serial.Serial(port = "COM1", baudrate = 2400, timeout = 0)
while not isDisposed():
    delay(1)
    ch = getKey()
    if ch != KeyEvent.CHAR_UNDEFINED:  # a key is typed
        ser.write(ch)
    nbChars = ser.inWaiting()
    if nbChars > 0:
        text = ser.read(nbChars)
        for ch in text:
            if ch == '\n':
                gprintln()
            else:
                gprint(ch)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Zum Lesen der empfangenen Zeichen musst du eine nicht-blockierende Funktion verwenden, da ja das Programm auch laufend prüfen muss, ob eine Taste gedrückt wurde. Die Methode *ser.read()* blockiert nicht, wenn du im Konstruktor den *timeout*-Parameter auf 0 setzt.

Falls du ein Notebook mit einem eingebauten Modem hast, so kann das Terminalprogramm mit diesem Hayes-Befehlen mit ihm kommunizieren.



LITERATUR & LINKS

- **http://www.tigerjython.ch**
  Programmierkonzepte mit Python und der Lernumgebung TigerJython
  Online-Lernplattform

- **http://pdf.tigerjython.ch**
  Das Lehrmittel als PFD Downloaden.

- **http://examples.tigerjython.ch**
  Sourcecodes aller Programme aus der Lernumgebung

- **http://jython.tobiaskohn.ch**
  Distribution TigerJython und Tutorials von Tobias Kohn

- **http://www.jython.ch**
  Turtlegrafik, Robotik und Spiele mit Python von Jarka Arnold

- **http://www.aplu.ch/jython**
  Bibliotheken mit Programmbeispielen von Aegidius Plüss (in Englisch)

- **http://www.tigerjython.ch/download/ACMandIEreport.pdf**
  Informatics education: Europe cannot afford to miss the boat

- **http://www.tigerjython.ch/download/ForteGuzdialCommNotCalc.pdf**
  Computers for Communication, Not Calculation:
  Media as a Motivation and Context for Learning

- **http://de.padlet.com/myschool/python**
  Einführung in die Programmierung mit Python von Günter Öller, Linz (Österreich)

- **http://fit-in-it.ch/sites/default/files/downloads/informatik_d.pdf** informatik@gymnasium
  Ein Entwurf für die Schweiz (J. Kohlas, J. Schmid, C.A. Zehnder, Hrsg.), Hasler Stiftung

Literaturreferenzen:

- Böhm C., Jacopini G., *Flow diagrams, turing machines and languages with only two formation rules*, Communications of the ACM 9(5), 366-371 (1966)

- Wirth Niklaus, *Algorithms and Data Structures*, Pearson Education (1985)

- Wong Baoswan Dzung, *Bézierkurven: gezeichnet und gerechnet*, Orell Füssli (2003)

KONTAKT

help@tigerjython.com

Entwicklungsteam:

Jarka Arnold, Pädagogische Hochschule Bern
www.java-online.ch

Tobias Kohn, Kantonsschule Zürich Oberland
www.tobiaskohn.ch

Dr. Aegidius Plüss, Universität Bern
www.aplu.ch

Über die Autoren

Jarka Arnold

Jarka Arnold besitzt langjährige Erfahrung als Dozentin für Informatik an der Pädagogischen Hochschule Bern. Im Rahmen von mehreren Forschungsprojekten wurden unter ihrer Leitung webbasierte Lernumgebungen für den Programmierunterricht entwickelt, die an vielen Ausbildungsinstitutionen erfolgreich im Einsatz sind (**http://www.java-online.ch** bzw. **http://www.jython.ch**).

Tobias Kohn

Tobias Kohn (**http://www.tobiaskohn.ch/**) Tobias Kohn hat 2008 an der ETH Zürich sein Mathematikstudium abgeschlossen und arbeitet seither als Mathematik- und Informatiklehrer an der Kantonsschule Zürcher Oberland in Wetzikon. Im Herbst 2012 hat er neben seiner Lehrtätigkeit ein Doktoratsstudium an der ETH Zürich aufgenommen und sucht dabei nach Wegen, den Einstieg in die Computerprogrammierung zu vereinfachen.

Aegidius Plüss

Aegidius Plüss (**http://www.aplu.ch**) ist ehemaliger Professor für Informatik und deren Didaktik an der Universität Bern. Er hat ein Lehrbuch "Java exemplarisch" verfasst und war an zahlreichen Weiterbildungskursen für Informatik-Lehrpersonen beteiligt. Er entwickelt umfangreiche Bibliotheken und Programmierumgebungen für den Informatikunterricht.