Python

Eine Einführung in die Computer-Programmierung

Tobias Kohn

http://www.tobiaskohn.ch/jython/
Version vom 5. Januar 2014

INHALTSVERZEICHNIS

EINFÜHRUNG

Einführung In diesem Kurs lernst du, wie du einen Computer programmieren kannst. Dabei gehen wir davon aus, dass du noch kein Vorwissen mitbringst, und werden dir schrittweise alles erklären, was du dazu brauchst. Am Ende des Kurses hast du dann alles nötige zusammen, um z. B. ein kleines Chatprogramm oder Computerspiel zu programmieren. Vielleichst wirst du dann aber auch einen Kurs für Fortgeschrittene besuchen oder Simulationen für eine wissenschaftliche Arbeit schreiben. Wenn du die Grundlagen erst einmal hast, sind dir kaum noch Grenzen gesetzt.

Beim Programmieren musst du eine wichtige Regel beachten: **Probiere alles selber aus!** Je mehr Programme du selber schreibst, umso mehr wirst du verstehen und beherrschen. Der Computer wird nicht immer das tun, was du möchtest. Halte dich in diesen Fällen einfach an Konfuzius: «Einen Fehler zu machen heisst erst dann wirklich einen Fehler zu machen, wenn man nichts daraus lernt.»

Installation Das Programm *TigerJython* ist schnell installiert. Lade es von der Seite

`http://www.tobiaskohn.ch/jython/`

herunter und speichere es am besten in einem eigenen Ordner, dem du z. B. den Namen `Python` geben kannst. Achte beim Herunterladen darauf, dass du das Programm unter dem Namen `tigerjython.jar` abspeicherst (beim Download werden die Dateien manchmal umbenannt).

Damit ist die Installation eigentlich bereits abgeschlossen und du kannst das Programm `tigerjython.jar` starten. Die Abbildung 1.1 zeigt, wie das etwa aussehen sollte (allerdings wird das Editorfeld in der Mitte bei dir leer sein).

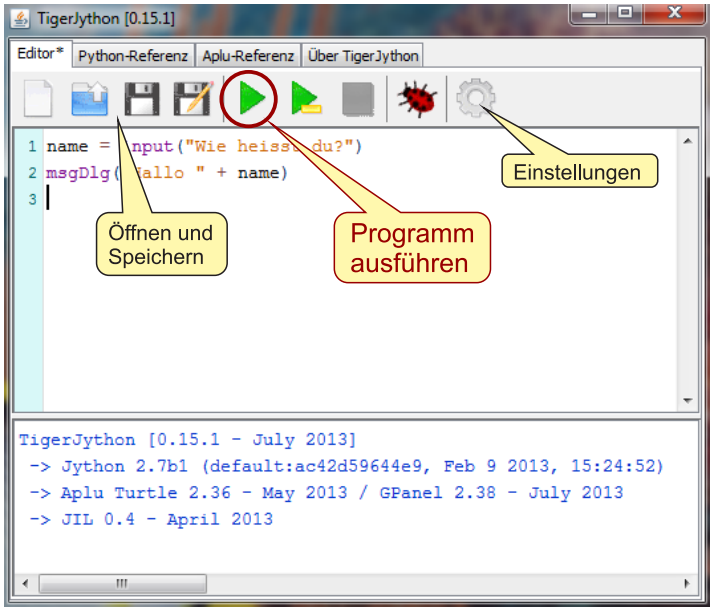


Abbildung 1.1: So präsentiert sich das Programm *TigerJython*.

Dein erstes Programm Tauchen wir doch gleich in die Welt des Programmierens ein! In der Abbildung 1.1 steht im Editorfenster bereits ein kleines Programm und zwar das folgende:

```
name = input("Wie heisst du?")
msgDlg("Hallo " + name)
```

Gib das Programm nun selber einmal ein (achte darauf, alles exakt so einzugeben wie im Code hier) und klicke dann auf die grüne Schaltfläche oben, um das Programm zu starten. Bevor TigerJython dein Programm ausführt, fragt es dich zuerst, ob du das Programm vorher speichern möchtest. Im Moment ist das nicht nötig und du kannst getrost auf «Nicht speichern» klicken.

Wenn dich der Computer jetzt nach deinem Namen fragt und dich dann begrüsst, dann hast du erfolgreich dein erstes Programm geschrieben: Gratulation!

Übrigens: Es ist ganz normal, dass du noch nicht alles hier verstehst. Wichtig ist im Moment nur, dass du weisst, wie du ein Programm eingibst und ausführst. Alles andere werden wir dir im Verlaufe dieses Kurses erklären.

K a p i t e l Z w e i

GRAFIK MIT DER TURTLE

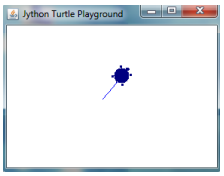
Die *Turtle* ist eine kleine Schildkröte, die eine Spur zeichnet, wenn sie sich bewegt. Du kannst ihr sagen, sie soll vorwärts gehen oder sich nach links oder rechts abdrehen. Indem du diese Anweisungen geschickt kombinierst, entstehen Zeichnungen und Bilder.

Für das Programmieren wesentlich und zentral ist, dass du dabei lernst, dieser Turtle neue Figuren beizubringen. Indem du z. B. einmal definierst, was ein Quadrat ist, kann die Turtle beliebig viele Quadrate in allen Grössen und Farben zeichnen. Mit der Zeit kannst du so immer komplexere Bilder und Programme aufbauen.

1 Die Turtle bewegen

Lernziele In diesem Abschnitt lernst du:

- Ein einfaches Programm zu schreiben und mit der Turtle beliebige Figuren auf den Bildschirm zu zeichnen.
- Die Turtle mit den Befehlen `left`, `right` und `forward` zu drehen und zu 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?

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 `makeTurtle()`. Um die Turtle zu bewegen verwendest du die drei Befehle `forward(länge)`, `left(winkel)` und `right(winkel)`.

Das Programm So sieht dein erstes Programm mit der Turtle aus. Schreib es ab und führe es aus, indem du auf den grünen Start-Knopf klickst. Die Turtle sollte dir dann ein rechtwinkliges Dreieck zeichnen.

```
1 from gturtle import *
2
3 makeTurtle()
4
5 forward(141)
6 left(135)
7 forward(100)
8 left(90)
9 forward(100)
```

Die Turtle ist in einer Datei (einem sogenannten *Modul*) «gturtle» gespeichert. In der ersten Zeile sagst du dem Computer, dass er alles aus dieser Datei laden soll. Die Anweisung `makeTurtle()` in der dritten Zeile erzeugt eine neue Turtle mit Fenster, die du programmieren kannst. Ab Zeile 5 stehen dann die Anweisungen für die Turtle selber.

Die wichtigsten Punkte 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 Anweisungen geben. Die drei Anweisungen, die die Turtle sicher versteht sind:

`forward(s)`	s Pixel vorwärts bewegen.
`left(w)`	Um den Winkel w nach links drehen.
`right(w)`	Um den Winkel w nach rechts drehen.

AUFGABEN

1. Zeichne mit der Turtle ein regelmässiges Fünfeck (Pentagon) mit einer Seitenlänge von 150 Pixeln.

2. Zeichne mit der Turtle zwei Quadrate ineinander wie in Abbildung 2.1(a).

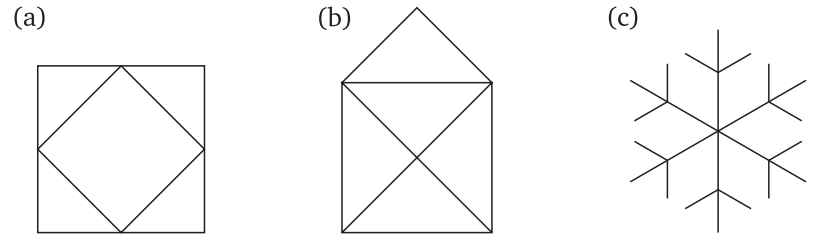


Abbildung 2.1: In der Mitte das «Haus von Nikolaus» und rechts eine einfache Schneeflocke.

3. Das «Haus vom Nikolaus» ist ein Zeichenspiel für Kinder. Ziel ist es, das besagte Haus (vgl. Abbildung 2.1(b)) in einem Linienzug aus genau 8 Strecken zu zeichnen, ohne dabei eine Strecke zweimal zu durchlaufen. Zeichne das Haus vom Nikolaus mithilfe der Turtle. Wähle dabei für die Seitenlänge des Quadrats 120 Pixel und nimm an, dass die beiden Dachflächen rechtwinklig aufeinander treffen.

4.* Lass die Turtle eine einfache Schneeflocke zeichnen wie in der Abbildung 2.1(c).

2 Den Farbstift steuern

Lernziele In diesem Abschnitt lernst du:

- Die Farbe und die Linienbreite einzustellen, mit der die Turtle zeichnet.
- Mit `penUp()` und `penDown()` zu steuern, wann die Turtle wirklich etwas zeichnet und wann nicht.

Einführung Um ihre Spur zu zeichnen hat die Turtle einen Farbstift (engl. *pen*). Für diesen Farbstift kennt die Turtle wiederum vier 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, pink, magenta, purple, navy, blue, sky blue, cyan, turquoise, lightgreen, green, darkgreen, chocolate, brown, black, gray, white

Schliesslich kennt die Turtle noch den Befehl `setLineWidth(Breite)`. Damit kannst du die Breite der gezeichneten Linie einstellen. Die Breite gibst du hier in Pixeln an.

Das Programm In diesem Programm zeichnet die Turtle zwei kurze blaue Linien übereinander. Die untere Linie ist dunkelblau (navy), die obere hellblau (light blue). Dazwischen ist ein Teil (Zeilen 9 bis 13), in dem sich die Turtle zwar bewegt, aber keine Linie zeichnet, weil der Stift «oben» ist.

```
1 from gturtle import *
2 makeTurtle()
3
4 setLineWidth(3)
5 right(90)
6 setPenColor("navy")
```

```
7  forward(50)
8
9  penUp()
10 left(90)
11 forward(30)
12 left(90)
13 penDown()
14
15 setPenColor("light blue")
16 forward(50)
```

Die wichtigsten Punkte Der Zeichenstift (pen) der Turtle kann mit `setPenColor(Farbe)` die Farbe wechseln. Indem du den Stift mit `penUp()` «hochhebst», hört die Turtle auf, eine Spur zu zeichnen. Mit `penDown()` wird der Stift wieder gesenkt und die Turtle zeichnet weiter.

Die Breite der Linie kannst du mit der Anweisung `setLineWidth(Breite)` steuern.

AUFGABEN

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

6. Lass die Turtle einen «Regenbogen» zeichnen wie in der Abbildung 2.2 angedeutet. Ganz innen ist dieser Regenbogen *rot*, danach *orange, gelb, grün* und schliesslich aussen *blau* und *violett*. Es genügt allerdings auch, wenn du nur drei Bogen zeichnen lässt.

Abbildung 2.2: Ein (eckiger) Regenbogen.

3 Der Turtle Neues beibringen

Lernziele In diesem Abschnitt lernst du:

- Für die Turtle neue Anweisungen zu definieren und damit ihren Wortschatz beliebig zu erweitern.

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

Es geht einfacher! Du kannst der Turtle nämlich neue Befehle beibringen und musst ihr dann nur noch sagen, sie soll ein Quadrat oder ein Dreieck zeichnen. Den Rest erledigt sie ab da selber. Neue Befehle definierst du mit `def NeuerName()`. Danach schreibst du alle Anweisungen auf, die zum neuen Befehl gehören. Damit der Computer weiss, was zum neuen Befehl gehört, müssen diese Anweisungen eingerückt sein.

Das Programm In diesem Programm definieren wir in den Zeilen 4 bis 12 mit `def` den Befehl `quadrat()`. Wie du ein Quadrat zeichnest, weisst du bereits: Es braucht vier gleich lange Seiten und dazwischen muss die Turtle um $90°$ gedreht werden. Wichtig: Alle acht Anweisungen, die das Quadrat ausmachen, sind um vier Leerschläge eingerückt.

```
1  from gturtle import *
2  makeTurtle()
3
4  def quadrat():
5      forward(100)
6      left(90)
7      forward(100)
8      left(90)
9      forward(100)
10     left(90)
11     forward(100)
12     left(90)
13
14 setPenColor("red")
15 quadrat()
16 penUp()
17 forward(50)
```

```
18 left(60)
19 forward(50)
20 penDown()
21 setPenColor("blue")
22 quadrat()
```

Ab der Zeile 14 weiss die Turtle, was ein Quadrat ist (sie hat aber noch keines gezeichnet). Du kannst `quadrat()` genauso brauchen wie die anderen Befehle: Die Turtle zeichnet dann immer an der aktuellen Position ein Quadrat mit der Seitenlänge 100. Hier zeichnen wir damit ein rotes und ein blaues Quadrat.

Tipp: Wenn die Turtle eine Figur (z. B. ein Quadrat) zeichnet, dann achte darauf, dass die Turtle am Schluss wieder gleich dasteht wie am Anfang. Das macht es viel einfacher, mehrere Figuren zusammenzuhängen. Im Programm oben wird die Turtle in Zeile 12 deshalb auch nochmals um 90° gedreht (was ja nicht nötig wäre).

Die wichtigsten Punkte Mit `def NeuerName():` definierst du einen neuen Befehl für die Turtle. Nach der ersten Zeile mit `def` kommen alle Anweisungen, die zum neuen Befehl gehören. Diese Anweisungen müssen aber *eingerückt* sein, damit der Computer weiss, was alles dazu gehört.

```
def NeuerName():
    Anweisungen
```

Vergiss die Klammern und den Doppelpunkt nach dem Namen nicht!

AUFGABEN

7. Definiere einen Befehl für ein Quadrat, das auf der Spitze steht und zeichne damit die Figur in der Abbildung 2.3. Die fünf Quadrate berühren sich nicht, sondern haben einen kleinen Abstand und die Farben *blau, schwarz, rot* (oben) und *gelb, grün* (unten).

Abbildung 2.3: Olympische Quadrate.

4 Parameter

Lernziele In diesem Abschnitt lernst du:

- Was ein Parameter ist und wofür du Parameter brauchst.
- Eigene Befehle mit Parametern zu definieren.

Einführung Bei der Anweisung `forward()` gibst du in Klammern an, *wieviel* die Turtle vorwärts gehen soll. Diese Länge ist ein *Parameter*: Ein Zahlenwert, der bei jeder Verwendung von `forward()` anders sein kann.

Je nach Zusammenhang werden Parameter auch als «Argumente» bezeichnet.

Im letzten Abschnitt hast du einen eigenen Befehl `quadrat()` definiert. Im Unterschied zu `forward()` ist die Seitenlänge dieses Quadrats aber immer 100 Pixel. Dabei wäre es doch praktisch, auch die Seitenlänge des Quadrats bei jeder Verwendung direkt angegeben und anpassen zu können. Wie geht das?

Das Programm Auch in diesem Programm definieren wir in Zeilen 4 bis 12 ein Quadrat. Im Unterschied zum letzten Abschnitt ist die Seitenlänge aber noch unbekannt. Anstatt `forward(100)` steht `forward(seite)`. In Zeile 4 steht `seite` auch in Klammern hinter dem Namen. Damit weiss der Computer, dass du bei jeder Verwendung von `quadrat` eine Zahl für `seite` angeben wirst.

```
1  from gturtle import *
2  makeTurtle()
3
4  def quadrat(seite):
5      forward(seite)
6      left(90)
7      forward(seite)
8      left(90)
9      forward(seite)
10     left(90)
11     forward(seite)
12     left(90)
13
14 setPenColor("red")
15 quadrat(80)
16 left(30)
17 setPenColor("blue")
18 quadrat(50)
```

In Zeile 15 zeichnet die Turtle ein rotes Quadrat mit der Seitenlänge von 80 Pixeln, in Zeile 18 ein blaues mit der Seitenlänge von 50 Pixeln. Weil hinter `quadrat` in Klammern die Zahlen 80 bzw. 50 stehen, setzt der Computer bei der Definition von Quadrat für `seite` überall 80 bzw. 50 ein.

Die wichtigsten Punkte *Parameter* sind Platzhalter für Werte, die jedes Mal anders sein können. Du gibst den Parameter bei der Definition eines Befehls hinter den Befehlsnamen in Klammern an.

```
def Befehlsname(Parameter):
    Anweisungen mit
    dem Parameter
```

Sobald du den Befehl verwenden möchtest, 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.

AUFGABEN

8. Definiere einen Befehl `jump(Distanz)`, mit dem die Turtle die angegebene Distanz überspringt. Der Befehl funktioniert also wie `forward()`, allerdings zeichnet die Turtle hier keine Spur.

9. Definiere einen Befehl `rechteck(grundseite)`, mit dem die Turtle ein Rechteck zeichnet, das doppelt so hoch wie breit ist.

10.* Definiere einen Befehl, der ein offenes Quadrat ⊓ zeichnet und verwende deinen Befehl, um ein Kreuz zu zeichnen, wobei du mit dem offenen Quadrat die vier Arme zeichnest.

5 Flächen füllen

Lernziele In diesem Abschnitt lernst du:

- Mit `fill()` beliebige Flächen mit einer Farbe auszufüllen.

Einführung Indem du die Turtle bewegst, kannst du beliebige Figuren zeichnen – oder zumindest den Umriss davon. Für ein hübsches Bild willst du aber sicher auch Flächen mit einer Farbe ausfüllen. Dazu kennt die Turtle den Befehl `fill()`.

Um eine Figur mit Farbe zu füllen, muss die Turtle ingendwo im Innern der Figur stehen. Bei `fill()` schüttet sie dann Farbe aus, bis die ganze Figur mit Farbe gefüllt ist. Aber Vorsicht: Wenn die Turtle dabei auf einer Linie steht, dann glaubt sie, dieses Linienstück alleine mache die ganze Figur aus! Du brauchst also sicher `penUp()`, um die Turtle ins Innere einer Figur zu bewegen ohne eine Linie zu zeichnen.

Das Programm In den Zeilen 4 bis 9 zeichnet die Turtle ein gleichseitiges Dreieck, das auf der Spitze steht. In den Zeilen 10 bis 12 nehmen wir den Stift hoch und setzen die Turtle ins Innere des Dreiecks (das muss nicht die Mitte sein). Schliesslich setzen wir die Füllfarbe in Zeile 13 auf «Himmelblau» und füllen das Dreieck dann in Zeile 14 aus.

```
1  from gturtle import *
2  makeTurtle()
3
4  right(30)
5  forward(100)
6  left(120)
7  forward(100)
8  left(120)
9  forward(100)
10 penUp()
11 left(150)
12 forward(50)
13 setFillColor("sky blue")
14 fill()
```

Die wichtigsten Punkte Um eine (geschlossene) Figur mit Farbe zu füllen, muss die Turtle im Innern der Figur sein (verwende `penUp()`, damit die Turtle keine Linie ins Innere hineinzeichnet). Dann verwendest du den Befehl `fill()`. Zuvor kannst du mit `setFillColor("Farbe")` genauso wie bei `setPenColor()` eine Farbe auswählen.

AUFGABEN

11. Die Abbildung 2.4 zeigt vier Figuren mit farbigen Flächen. Lass deine Turtle diese vier Figuren zeichnen (in vier verschiedenen Programmen). Die Farben kannst du dabei selbst wählen.

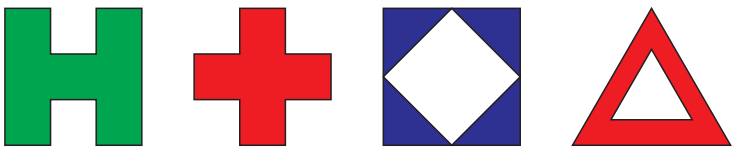


Abbildung 2.4: Figuren mit gefüllten Flächen.

12. Definiere einen neuen Befehl `fillQuadrat(seite)` um eine ausgefülltes Quadrat zu zeichnen. Verwende dann diesen neuen Befehl, um die ersten drei Figuren in der Abbildung 2.4 nochmals zu zeichnen.

6 Repetitionen

Lernziele In diesem Abschnitt lernst du:

- Der Turtle zu sagen, sie soll eine oder mehrere Anweisungen mehrfach wiederholen.

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

Eine mehrfache Wiederholung heisst in der Fachsprache «Schleife». Mit **`repeat`** *kannst du also Schleifen programmieren.*

Die Turtle kennt die Anweisung **`repeat`** `Anzahl`. Damit sagst du der Turtle, sie soll einige Befehle «*Anzahl*» Mal wiederholen. Damit die Turtle aber weiss, welche Befehle sie wiederholen soll, müssen diese wieder eingerückt sein – genauso wie bei **`def`** vorhin.

Das Programm Um ein regelmässiges Neuneck zu zeichnen muss die Turtle neunmal geradeaus gehen und sich dann um $40°$ drehen. Würdest du das alles untereinander schreiben, dann würde das Programm ziemlich lange werden. Hier verwenden wir in Zeile 4 aber die Anweisung **`repeat`** und sagen der Turtle damit, sie soll die zwei eingerückten Befehle in Zeilen 5 und 6 neunmal wiederholen.

In den Zeilen 8 bis 11 füllen wir das Neuneck auch gleich noch aus. Weil diese Befehle nicht mehr eingerückt sind, werden sie auch nicht wiederholt, sondern nur noch einmal ausgeführt.

```
1  from gturtle import *
2  makeTurtle()
3
4  repeat 9:
5      forward(50)
6      left(40)
7
8  penUp()
9  left(80)
10 forward(60)
11 fill()
```

Wenn du **`repeat`** in einem neuen Befehl verwenden möchtest, dann musst du die Anweisungen, die wiederholt werden sollen, noch stärker einrücken:

```
def neuneck():
    repeat 9:
        forward(50)
        left(40)
```

Die wichtigsten Punkte Mit `repeat Anzahl:` gibst du der Turtle an, sie soll einen oder mehrere Befehle «*Anzahl*» Mal wiederholen, bevor sie mit neuen Befehlen weitermacht. Alles, was wiederholt werden soll, muss unter `repeat` stehen und eingerückt sein.

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

AUFGABEN

13. Zeichne mit der Turtle die Treppenfigur (a) aus der Abbildung 2.5. Verwende dazu `repeat`.

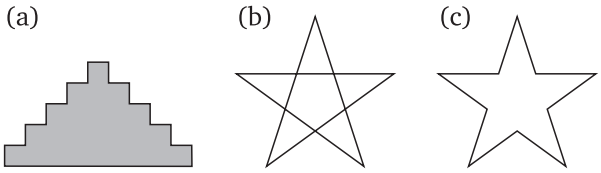


Abbildung 2.5: Treppen und Sterne

14. Zeichne den fünfzackigen Stern aus der Abbildung 2.5(b) oder (c). Dazu musst du zuerst die entsprechenden Winkel berechnen. Verwende wiederum `repeat` für alles, was sich wiederholen lässt.

15. Definiere einen Befehl `achteck(seite)`, um ein regelmässiges und ausgefülltes Achteck zeichnen zu lassen.

7 Kreise und Bogen

Lernziele In diesem Abschnitt lernst du:

- Mit der Turtle Kreise und Kreisbogen zu zeichnen.

Einführung Die Turtle kann keine «perfekten» Kreise zeichnen: Sie kann ja nur ein Stück weit geradeaus gehen und sich drehen. Wenn wir die Turtle aber ein Vieleck mit sehr vielen Ecken zeichnen lassen, dann entsteht eine ziemlich gute Annäherung an einen Kreis.

Der Computerbildschirm hat eine relativ grobe Auflösung. Ein 36-Eck lässt sich meistens schon nicht mehr von einem echten Kreis unterscheiden. 72 Ecken dürften auf jeden Fall genügen, aber wenn du besonders präzise sein willst, kannst du natürlich auch ein 360-Eck zeichnen.

Nach einem ganzen Kreis hat sich die Turtle immer um $360°$ gedreht. Bei z. B. 36 Ecken bedeutet das, dass sich die Turtle an jeder Ecke um $360° : 36 = 10°$ drehen muss.

Das Programm Dieses Programm zeichnet einen Kreis, indem die Turtle immer 6 Pixel vorwärts geht, sich dann um $5°$ nach links dreht und wieder vier Pixel vorwärts geht. Damit ein ganzer Kreis entsteht, muss die Turtle sich 72 Mal drehen ($72 \cdot 5° = 360°$).

```
1 from gturtle import *
2 makeTurtle()
3
4 repeat 72:
5     forward(6)
6     left(5)
```

In der Zeile 5 kannst du die 6 ändern, um einen grösseren oder kleineren Kreis zu erhalten. Mit der Zahl in Zeile 4 – hier 72 – steuerst du, welcher Teil des Kreises gezeichnet werden soll. Für 36 Wiederholungen (bei $5°$) entsteht zum Beispiel ein Halbkreis.

Radius und Umfang Wenn die Turtle einen Kreis zeichnet, dann kennst du den Umfang des Kreises sehr genau. Im Beispiel oben ging die Turtle 72 mal 6 Pixel vorwärts. Der Kreis hat also einen Umfang von $u = 72 \cdot 6 = 432$ Pixeln.

Ein Umfang von 432 Pixeln entspricht einem Radius von 68.75 Pixeln. Du kannst das selber ausrechnen mit der Formel: $u = 2\pi \cdot r$. Soll der

Radius des Kreises umgekehrt 100 Pixel sein, dann hat der Kreis einen Umfang von 628 Pixeln. Damit muss die Turtle jedes Mal $628 : 72 = 8.72$ Pixel vorwärts gehen:

```
repeat 72:
    forward(8.72)
    left(5)
```

Auf diese Weise kannst du sehr genau bestimmen, wie gross deine Kreise sein sollen.

Die wichtigsten Punkte Die Turtle kann keine echten Kreise zeichnen. Bei einem regelmässigen Vieleck mit sehr vielen Ecken ist der Unterschied zu einem Kreis aber nicht mehr sichtbar.

Für einen ganzen Kreis muss sich die Turtle insgesamt um $360°$ drehen. Indem du die Anzahl der Wiederholungen kleiner machst, dreht sich die Turtle nur um $180°$ oder $90°$ und zeichnet damit einen Halb- oder Viertelkreis.

AUFGABEN

16. Zeichne mit der Turtle einen Kreis mit einem Radius von $r = 20$ Pixeln.

17. Definiere einen eigenen Befehl `circle(s)`, um einen Kreis zu zeichnen und einen Befehl `fillcircle(s)`, um einen ausgefüllten Kreis zu zeichnen. Der Parameter s gibt dabei an, um wieviel die Turtle bei jedem Schritt vorwärts gehen soll.

18. Lass deine Turtle den PacMan (a), das Yin-Yang-Symbol (b) und ein beliebiges Smiley (c) zeichnen wie in der Abbildung 2.6. Bei den kleinen Punkten genügt es meist, ein kurzes, breites Linienstück zu zeichnen.

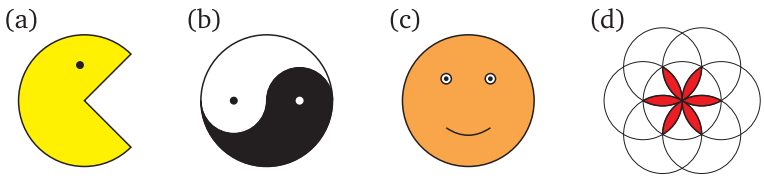


Abbildung 2.6: Kreisfiguren.

19. Die «Blume» in der Abbildung 2.6(d) setzt sich aus sieben gleich grossen Kreisen zusammen. Verwende einen eigenen Befehl `circle()` und zeichne damit diese Blume.

8 Parameter strecken

Lernziele In diesem Abschnitt lernst du:

- Parameter mit einer Zahl zu multiplizieren oder zu dividieren.

Einführung Eigene Befehle mit Parametern sind unglaublich nützlich und bewähren sich in vielen Situationen. Eine grosse Stärke der Parameter kommt aber erst jetzt zum Zug: Du kannst in deinem eigenen Befehl nicht nur die Parameter selbst verwenden, sondern auch Vielfache oder Teiler davon. In den zwei Programmen unten zeigen wir dir, wie du den Parameter mit einer Zahl multiplizieren oder dividieren kannst.

Das Programm (I) Bei einem gleichschenklig rechtwinkligen Dreieck muss die längste Seite immer $\sqrt{2}$ mal so lang sein wie eine der kürzeren Seiten (vgl. Abbildung 2.7). Die Turtle kennt zwar $\sqrt{2}$ nicht, aber in diesem Programm sagen wir ihr, dass die letzte Seite des Dreiecks 1.4142 mal so lang sein soll wie die beiden anderen Seiten.

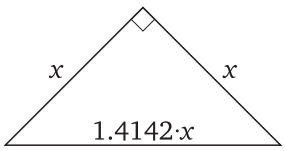


Abbildung 2.7: Mit dem Satz des Pythagoras ergibt sich, dass die Hypotenuse des Dreiecks $\sqrt{2}$ mal so lang sein muss wie die Katheten x.

```
1  from gturtle import *
2  makeTurtle()
3
4  def rewiDreieck(seite):
5      forward(seite)
6      left(90)
7      forward(seite)
8      left(135)
9      forward(seite * 1.4142)
10     left(135)
11
12 rewiDreieck(50)
```

In Zeile 9 multiplizieren wir die Seitenlänge 50 mit einem Näherungswert von $\sqrt{2} \approx 1.41412136$.

Das Programm (II) Bei jedem geschlossenen Vieleck (oder Kreis) muss sich die Turtle für die ganze Figur einmal um $360°$ drehen. Für ein Quadrat dreht sich die Turtle jedes Mal um $360° : 4 = 90°$, für ein gleichseitiges Dreieck $360° : 3 = 120°$ (das sind *nicht* die Innenwinkel der Vielecke).

Dieses Wissen nutzen wir jetzt aus, um ein allgemeines Vieleck zu definieren. Der Parameter gibt dabei an, wie viele Ecken bzw. Seiten das Vieleck haben soll. Danach wird damit ein Fünfeck gezeichnet.

```
1 from gturtle import *
2 makeTurtle()
3
4 def vieleck(n):
5     repeat n:
6         forward(70)
7         left(360 / n)
8 vieleck(5)
```

Die wichtigsten Punkte In einem eigenen Befehl kannst du Vielfache eines Parameters verwenden, indem du den Parameter mit einer Zahl multiplizierst (mit `*`) oder dividierst (mit `/`).

AUFGABEN

20. Schreibe einen Befehl, der ein Quadrat mit einem Kreuz darin zeichnet (vgl. Abbildung 2.8 (a)). Dabei wird die Seitenlänge des Quadrats mit einem Parameter angegeben und kann beliebig gewählt werden.

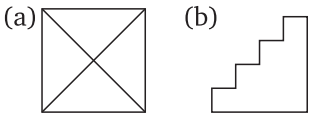


Abbildung 2.8: Figuren zu den Aufgaben 2.20 und 2.21.

21. Definiere einen Befehl `treppe(stufen)`, der die Treppe aus Abbildung 2.8(b) zeichnet. Die Treppe hat dabei immer eine Breite und eine Höhe von 120 Pixeln. Die Anzahl der Stufen kann man über den Parameter steuern.

22. Definiere einen Befehl `kreis(umfang)`, der einen Kreis mit dem angegebenen Umfang zeichnet. Schwierigere Variante: Definiere den Kreis so, dass der Parameter den Radius angibt.

9 Mehrere Parameter

Lernziele In diesem Abschnitt lernst du:

- Befehle mit mehreren Parametern zu definieren.

Einführung Parameter sind sehr praktisch, weil du mit ihnen steuern kannst, wie gross etwa dein Quadrat werden soll. Bei einigen Figuren reicht ein einzelner Parameter aber nicht aus: Ein Rechteck hat nicht nur eine Seitenlänge, sondern eine Breite und eine eine Höhe. In solchen Fällen ist es sinnvoll, einen Befehl mit mehreren Parametern zu definieren.

Parameter gibst du immer in Klammern hinter dem Befehlsnamen an. Wenn du mehrere Parameter brauchst, dann trennst du sie mit Kommata.

Das Programm In der Zeile 4 definieren wir einen neuen Befehl für ein Rechteck. Dieser Befehl hat zwei Parameter: `breite` und `hoehe` (beachte, dass du beim Programmieren nur englische Buchstaben verwenden darfst und wir deshalb «ö» als «oe» schreiben). In Zeile 15 sagen wir der Turtle dann, sie soll das Rechteck zeichnen und geben für beide Parameter Zahlenwerte an.

```
1  from gturtle import *
2  makeTurtle()
3
4  def rechteck(breite, hoehe):
5      forward(hoehe)
6      right(90)
7      forward(breite)
8      right(90)
9      forward(hoehe)
10     right(90)
11     forward(breite)
12     right(90)
13
14 setPenColor("salmon")
15 rechteck(150, 120)
```

Die Reihenfolge der Parameter spielt eine Rolle. Weil wir in der Definition zuerst die Breite und dann die Höhe haben, wird in Zeile 15 auch für die Breite der Wert 150 und für die Höhe der Wert 120 eingesetzt.

Die wichtigsten Punkte In Python können Befehle beliebig viele Parameter haben. Die verschiedenen Parameter werden immer mit Komma voneinander getrennt. Natürlich braucht auch jeder Parameter einen eigenen Namen.

```
def Befehl(param1, param2, param3):
    Anweisungen mit
    den Parametern

Befehl(123, 456, 789)
```

Wenn du für deinen neuen Befehl beim Ausführen Zahlenwerte angibst, dann ist die Reihenfolge immer die gleiche wie bei der Definition.

AUFGABEN

23. Definiere einen Befehl `vieleck(n, seite)`, bei dem du die Anzahl der Ecken n und die Seitenlänge angeben kannst.

24. Der Rhombus (Abb. 2.9(a)) hat vier gleich lange Seiten, im Gegensatz zum Quadrat sind die Winkel allerdings in der Regel keine rechten Winkel. Definiere einen Befehl `rhombus(seite, winkel)`, bei dem man die Seitenlänge und den Winkel angeben kann, und der dann einen Rhombus zeichnet.

25.* Erweitere den Rhombus-Befehl zu einem Parallelogramm mit unterschiedlicher Breite und Höhe. Dieser Befehl hat drei Parameter.

26. Definiere einen Befehl `gitter(breite, hoehe)`, der ein Gitter wie in Abbildung 2.9(b) zeichnet. Jedes Quadrätchen soll eine Seitenlänge von 10 Pixeln haben. Die beiden Parameter geben die Anzahl dieser Häuschen an (in der Abbildung wäre also `breite = 6` und `hoehe = 4`).

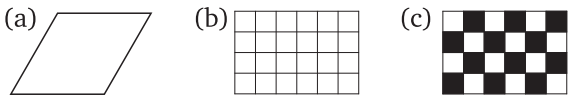


Abbildung 2.9: Rhombus, Gitter und Schachbrettmuster.

27.* Definiere einen Befehl `schachbrett(breite, hoehe)`, um ein Schachbrett wie in Abbildung 2.9(c) zu zeichnen.

10 Den Programmablauf beobachten

Lernziele In diesem Abschnitt lernst du:

- Wie du den Programmablauf verfolgen kannst.

Einführung Du kannst der Turtle zuschauen, wie sie ihre Figuren zeichnet und siehst dabei, in welcher Reihenfolge sie ihre Linien zeichnet. Was du dabei nicht siehst: Welche Anweisungen in deinem Programmcode führt die Turtle im Moment gerade aus? Welche Zeile in deinem Programmcode macht genau was?

In TigerJython kannst du nicht nur der Turtle beim Zeichnen zuschauen. Du kannst auch mitverfolgen, welche Zeilen im Programmcode gerade ausgeführt werden. Das Werkzeug dazu heisst «*Debugger*». Klicke in TigerJython auf den Käfer oben, um den Debugger zu aktivieren:

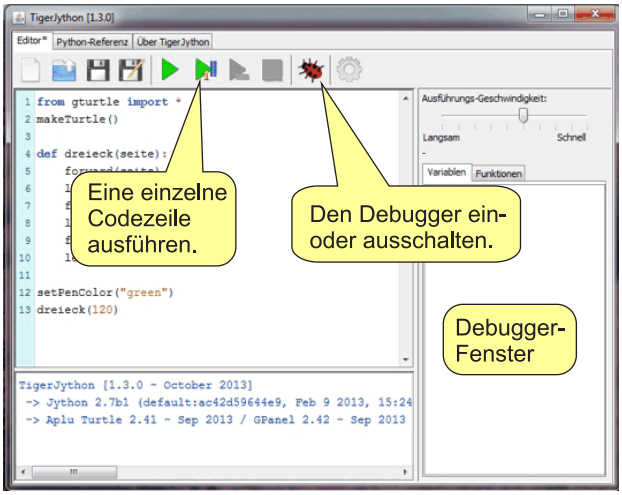


Abbildung 2.10: Der Debugger zeigt sich rechts im Fenster, sobald du auf den Käfer klickst. Solange das Debuggerfenster offen ist, kannst du den Programmablauf beobachten.

Das Programm Das Programm zeichnet ein Dreieck, so wie du es selbst schon gemacht hast. Tippe es ein und starte dann den Debugger (Abb. 2.10). Danach klickst du auf «Einzelschritt» im Debugger, um das Programm zu starten. Jedes Mal, wenn du auf «Einzelschritt» klickst, wird eine Codezeile ausgeführt. Dabei kannst du beobachten, wie die aktuelle Codezeile gelb unterlegt wird.

```
1  from gturtle import *
2  makeTurtle()
3
4  def dreieck(seite):
5      forward(seite)
6      left(120)
7      forward(seite)
8      left(120)
9      forward(seite)
10     left(120)
11 setPenColor("green")
12 dreieck(80)
```

Ist dir aufgefallen, dass der Computer von der Zeile 4 direkt zur Zeile 11 springt? Bei der Definition von `dreieck(seite):` merkt er sich nämlich nur, dass ab der Zeile 4 steht, wie die Turtle das Dreieck zeichnen soll. Aber: Gezeichnet wird das Dreieck hier noch nicht! Erst in Zeile 12 sagst du der Turtle, sie soll ein Dreieck zeichnen. Und weil sich der Computer gemerkt hat, dass in Zeile 4 steht, was ein Dreieck ist, springt er wieder hoch und arbeitet jetzt die Schritte ab, um das Dreieck wirklich zu zeichnen.

AUFGABEN

28. Schreibe das Programm unten ab und beobachte mit dem Debugger, wie das Programm ausgeführt wird.

```
from gturtle import *
makeTurtle()
repeat 4:
    forward(100)
    left(90)
penUp()
left(45)
forward(10)
setFillColor("sky blue")
fill()
```

29. Verändere das Programm oben so, dass du einen Befehl `quadrat` definierst, um das Quadrat zu zeichnen. Betrachte auch dann den Programmablauf mit dem Debugger. Was ist gegenüber der ersten Version grundlegend anders?

```
def quadrat(seite):
    repeat 4:
        forward(seite)
        left(90)
```

11 Programme mit Fehlern

Lernziele In diesem Abschnitt lernst du:

- In einem Programm Fehler zu finden und zu korrigieren.

Einführung Beim Programmieren entstehen immer Fehler (auch bei Profis). Dabei gibt es zwei wesentliche Arten von Fehlern: Ein «*Syntaxfehler*» bedeutet, dass die *Schreibweise* nicht stimmt. Der Computer ist hier sehr empfindlich: Wenn eine Klammer fehlt oder etwas falsch geschrieben ist, dann kann der Computer das Programm nicht ausführen oder hört mitendrin auf. Die zweite Fehlerart sind «*Logikfehler*». Bei solchen Fehlern läuft das Programm zwar, aber es macht nicht das, was es sollte. Bei grösseren Programmen sind solche Logikfehler sehr schwierig zu finden, obwohl man weiss, dass es solche Fehler enthält.

So sieht eine Fehlermeldung in TigerJython aus. Bevor du weiterarbeiten kannst, musst du immer zuerst auf «OK» klicken.

Nachdem du bereits etwas Erfahrung im Programmieren gesammelt hast, solltest du die Schreibweise (Syntax) von Python langsam kennen und Fehler selber finden können. Auf der anderen Seite solltest du aber auch die Fehlermeldungen verstehen lernen, die dir der Computer ausgibt, wenn etwas nicht stimmt.

Das Programm Dieses Programm sollte ein gleichseitiges Dreieck zeichnen, enthält aber in jeder Zeile einen Fehler (ausgenommen sind natürlich die leeren Zeilen). Der Fehler in Zeile 7 ist ein «Logikfehler»: Da stimmt der Winkel nicht. Alles andere sind sogenannte «Syntaxfehler», bei denen die Schreibweise falsch ist.

```
1  from turtle import *
2  makeTurtle
3
4  def dreieck(seite)
5      repeat:
6          forward seite
7          left(60)
8
9  setPenColor(red)
10 Dreieck(160)
```

Versuche einmal selbst, alle Fehler zu finden! Arbeite dazu zuerst ohne Computer und teste dein Wissen über die Schreibweise von Python. Auf der nächsten Seite findest du die Auflösung.

Wenn ein Programm Syntaxfehler enthält, dann wird dir TigerJython das auch sagen. Gib das fehlerhafte Programm ein und schaue dir an, welche Fehlermeldungen zu den einzelnen Zeilen ausgegeben werden.

Und hier die Auflösung:

```
from gturtle import *     # Das g bei gturtle fehlte
makeTurtle()              # Klammern vergessen

def dreieck(seite):       # Doppelpunkt vergessen
    repeat 3:             # Die Drei vergessen
        forward(seite)    # Parameter müssen in Klammern sein
        left(120)         # Der Winkel muss 120 sein

setPenColor("red")        # Farbnamen gehören in Anführungszeichen
dreieck(160)              # Die Gross-/Kleinschreibung ist wichtig
```

AUFGABEN

30. Auch dieses Programm ist voller Fehler. Finde und korrigiere sie!

```
from gturtle import

def fünfeck(seite):
    repeat5:
        forwart(seite)
    left(72)

fünfeck(100)
```

Tipp: Auch wenn wieder grundsätzlich jede Zeile einen Fehler enthält, kommen die gleichen Fehler oft mehrfach vor.

31. Beim diesem kurzen Prográmmchen sind zwei Fehler drin.

```
makeTurtle()
repeat 6:
    forward(100)
    right(120)
```

(a) Wenn du versuchst, das Programm auszuführen, dann sagt TigerJython «*Unbekannter Name: makeTurtle()*». Woran liegt das? Was ist falsch und wie lässt es sich korrigieren.

(b) Nachdem du das Programm soweit korrigiert hast, zeichnet die Turtle eine Figur. Allerdings scheint der Programmierer hier etwas anderes gewollt zu haben. Auf welche zwei Arten lässt sich die Absicht des Programmierers interpretieren und das Programm korrigieren?

Quiz

1. Nach welcher Anweisung zeichnet die Turtle ihre Linien in grün?

   ☐ a. `setColor("green")`

   ☐ b. `setFillColor("green")`

   ☐ c. `setLineColor("green")`

   ☐ d. `setPenColor("green")`

2. Die Turtle soll bei einem Dreieck einen Winkel von $45°$ nach links zeichnen. Mit welchen Drehungen entsteht der richtige Winkel?

   ☐ a. `left(45)`

   ☐ b. `left(135)`

   ☐ c. `right(225)`

   ☐ d. `right(-45)`

3. Was macht die Turtle bei der folgenden Befehlssequenz?

```
repeat 36:
    forward(2)
right(5)
```

   ☐ a. Sie zeichnet einen Halbkreis,

   ☐ b. Sie zeichnet eine gerade Linie und dreht sich dann,

   ☐ c. Sie zeichnet ein 36-Eck,

   ☐ d. Sie macht gar nichts, weil der Code falsch ist.

4. Was zeichnet die Turtle bei dieser Befehlssequenz?

```
def foo(x, y):
    left(y)
    forward(x)
repeat 9:
    foo(120, 60)
```

   ☐ a. Ein Sechseck,

   ☐ b. Ein Neuneck,

   ☐ c. Eine Schneeflocke,

   ☐ d. Gar nichts, weil `left` und `forward` vertauscht sind.

K a p i t e l D r e i

RECHNEN MIT PYTHON

Rechnen am Computer ist ein Heimspiel, zumal die ersten Computer tatsächlich zum Rechnen gebaut wurden. Du wirst sehen, dass du die Grundoperationen wie bei einem Taschenrechner eingeben kannst. Dabei kannst du auch vordefinierte Konstanten wie beispielsweise π verwenden.

Für das Programmieren zentral ist dabei der Umgang mit Variablen. Damit kannst du Formeln einprogrammieren und mit Werten rechnen, die du während des Programmierens noch gar nicht kennst (z. B. die Resultate von komplizierten Berechnungen oder Eingaben des Benutzers). Stell dir zum Beispiel vor, dein Programm soll herausfinden, ob ein bestimmtes Jahr ein Schaltjahr ist oder nicht. Dann weisst du während du das Programm schreibst noch nicht, für welches Jahr oder welche Jahre der Computer rechnen soll. Das wird erst dann entschieden, wenn das Programm läuft und ausgeführt wird. Und weil der Computer zwischen einfachen Fällen unterscheiden kann, findet er tatsächlich auch heraus, welches Jahr nun ein Schaltjahr ist. Aber dazu später mehr.

Mit den Techniken aus diesem Kapitel kannst du beispielsweise auch die Teilbarkeit von Zahlen untersuchen oder das Osterdatum im Jahr 2032 bestimmen. Selbst die Lösungen von quadratischen Gleichungen lassen sich auf diese Weise berechnen.

1 Grundoperationen und das Zahlenformat

Lernziele In diesem Abschnitt lernst du:

- Die vier Grundrechenarten in Python auszuführen.
- Den Unterschied zwischen ganzen und gebrochenen bzw. wissenschaftlichen Zahlen kennen.
- Die wissenschaftliche Schreibweise von Zahlen zu verwenden.

Einführung: Zahlen Der Computer kennt zwei verschiedene Zahlentypen: Ganze Zahlen (engl. *integer*) und wissenschaftliche Zahlen (engl. *floating point number*).

Vielleicht kennst du die *wissenschaftliche Schreibweise* bereits aus dem Mathematik- oder Physikunterricht. Mit dieser Schreibweise lassen sich sehr grosse (oder sehr kleine) Zahlen viel kürzer und übersichtlicher schreiben. Die Masse der Erde ist z. B.:

$$5\,\underbrace{980\,000\,000\,000\,000\,000\,000\,000}_{\text{24 Stellen}}\,\text{kg} = 5.98 \cdot 10^{24}\,\text{kg}$$

An diesem Beispiel siehst du bereits, wie ungemein praktisch diese kürzere wissenschaftliche Schreibweise ist. Die wesentlichen Ziffern sind hier nur «598» und nach der ersten dieser Ziffern (der Fünf) kommen nochmals 24 Stellen. Daraus entsteht die wissenschaftlichen Schreibweise mit der *Mantisse* 5.98 und dem *Exponenten* 24.

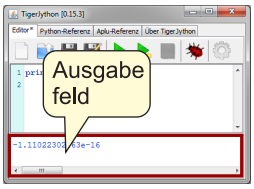
Computer arbeiten auch mit dieser wissenschaftlichen Schreibweise, konnten früher aber das $\cdot 10^{\square}$ nicht schreiben. Deshalb hat man sich vor langer Zeit auf die Darstellung mit einem `e` geeinigt: `5.98e+24` oder einfach `5.98e24`.

Sehr kleine Zahlen kannst du ebenfalls so schreiben:

$$0.\underbrace{000\,000\,000\,000\,04}_{\text{14 Stellen}}9\,05 = 4.905 \cdot 10^{-14} = \texttt{4.905e-14}$$

Im Prinzip entsteht die wissenschaftliche Schreibweise dadurch, dass du den Dezimalpunkt (das Komma) hinter die erste wesentliche Stelle verschiebst. Das wird durch den englischen Namen «*floating point number*» ausgedrückt, meist abgekürzt zu «*float*». Beachte, dass Computer in diesem Format nur mit 12 bis 20 Stellen arbeiten und längere Zahlen einfach runden. Das führt schnell zu *Rundungsfehlern*, wie du beim Programm auf der nächsten Seite sehen wirst.

Beim Rechnen mit wissenschaftlichen Zahlen auf dem Computer entstehen oft Rundungsfehler. Wegen dieser Rundungsfehler spielt auf dem Computer die Reihenfolge der einzelnen Rechnungen eine Rolle: $x + y \neq y + x$.



Ausgaben mit print Damit dir der Computer überhaupt etwas ausgibt, musst du ihn anweisen, das zu tun. Die Anweisung dafür heisst `print`. Es genügt also nicht, einfach die Rechnung einzugeben. Du musst dem Computer mit einem `print` auch sagen, dass er das Resultat der Rechnung ausgeben muss, etwa `print 13/3`, um das Resultat `4.3333333333` zu erhalten.

Das Programm In diesem Programm soll uns Python zuerst die Zahl $1\,234\,567\,890\,123\,456\,789$ zweimal ausgeben. Im ersten Fall ist es für den Computer eine ganze Zahl (*integer*). Weil im zweiten Fall am Schluss ein Dezimalpunkt steht, ist das automatisch eine wissenschaftliche Zahl und wird auch gleich gerundet: Die Ausgabe von Zeile 2 ist `1.23456789012e+18`.

```
1 print 1234567890123456789
2 print 1234567890123456789.0
3
4 print (1/2 + 1/3 + 1/6) - 1
5 print (1/6 + 1/3 + 1/2) - (1/2 + 1/3 + 1/6)
```

In Zeile 4 und 5 berechnet Python schliesslich zuerst das Resultat der Rechnung und gibt das dann aus. Siehst du, dass $\frac{1}{2} + \frac{1}{3} + \frac{1}{6} = 1$ ist? Wenn Python richtig rechnet, sollte die Ausgabe `0` sein – ist sie aber nicht! Kannst du diesen Unterschied erklären?

Die wichtigsten Punkte Für Rechnungen verwendest du in Python die Grundoperationen `+`, `-`, `*` und `/`. Zahlen werden entweder als ganze Zahlen (integer) oder in wissenschaftlicher Schreibweise (float) angegeben, also z. B. `5.98e24` für $5.98 \cdot 10^{24}$. In der wissenschaftlichen Schreibweise rundet Python immer auf 12 bis 20 Stellen.

Wenn du grosse ganze Zahlen ausgibst, dann hängt Python ein «`L`» an, das für «long» steht. Das ist ein Überbleibsel von früher und hat heute keine Bedeutung mehr.

Python schreibt die Resultate und Zahlen nur auf den Bildschirm, wenn du das mit `print` angibst.

AUFGABEN

1. Du kannst $\frac{1}{3}$ auch als Dezimalzahl eingeben: $0.333\ldots$. Wie viele Dreien musst du eingeben, bis die folgenden Rechnungen das richtige Resultat liefern?
(a) `print 3 * 0.333`, (b) `print (3 * 0.333) - 1`

2. Die Entfernung der Erde von der Sonne beträgt rund $1.496 \cdot 10^8\,\mathrm{km}$. Die Lichtgeschwindigkeit beträgt ca. $300\,000\,000\,\mathrm{m/s}$. Berechne mit Python, wie lange das Sonnenlicht braucht, um die Erde zu erreichen und gib das Resultat in Minuten an.

2 Variablen

Lernziele In diesem Abschnitt lernst du:

- Variablen zu definieren und im Programm zu verwenden.
- Rechnungen mit Variablen auszuführen.

Einführung Rechnen ist in Python recht einfach und funktioniert – abgesehen von `print` – gleich wie bei vielen Taschenrechnern. Du kommst allerdings auch immer wieder in die Situation, in der du die gleiche Rechnung mit verschiedenen Zahlen auswerten willst. Damit du nicht jedes Mal alles neu eingeben musst, verwendest du in einem solchen Fall *Variablen*.

Bevor du eine Variable verwendest, legst du immer zuerst den Zahlenwert der Variable fest. Python kann nicht algebraisch mit Variablen rechnen: Es ersetzt in einer Rechnung einfach alle Variablen durch den entsprechenden Wert. Im Gegensatz zur Mathematik dürfen die Variablen dafür beliebig lange Namen haben.

Es ist wichtig, dass du dir bewusst bist, dass Python *nicht* algebraisch rechnen kann und auch die mathematische Notation ein wenig anders interpretiert. In der Mathematik kannst du die Multiplikationszeichen oft weglassen, beim Programmieren dürfen sie aber auf keinen Fall fehlen. Vergleiche:

$$3ab(2a+1) = 3 \cdot a \cdot b \cdot (2 \cdot a + 1) = \texttt{3*a*b*(2*a+1)}$$

Vor allem ist ab für Python nicht $a \cdot b$, sondern eine einzige Variable mit dem Namen «*ab*».

Das Programm Du kannst die Summe der ersten n Quadratzahlen mit einer Formel berechnen. Die Summe aller Quadratzahlen von 1^2 bis n^2 ergibt:

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

In diesem Programm haben wir die Formel einprogrammiert und verwenden dabei zwei Variablen: Die erste Variable n gibt an, wie viele Quadratzahlen wir zusammenzählen wollen (in diesem Fall 7). Die zweite Variable heisst `summe_der_quadrate` und enthält das Resultat der Rechnung. Dieses Resultat schreiben wir in Zeile 3 auf den Bildschirm.

```
1 n = 7
2 summe_der_quadrate = (n * (n+1) * (2*n+1)) / 6
3 print summe_der_quadrate
```

Wir hätten die Rechnung auch so programmieren können:

```
print (7*(7+1)*(2*7+1))/6
```

Die Variante mit Variablen hat zwei grosse Vorteile: Es ist sehr einfach, den Wert von n zu ändern und damit die Formel für verschiedene Zahlen auszuwerten. Zum zweiten ist die Variante mit Variablen auch übersichtlicher und verständlicher: Das Programm beschreibt, was es tut.

Die wichtigsten Punkte Eine Variable wird automatisch definiert, wenn du ihr mit = einen Wert zuweist. Dabei steht der Name der Variablen immer *links* vom Gleichheitszeichen = und der Wert (oder die Rechnung) rechts davon:

```
VARIABLE = WERT ODER RECHNUNG
```

Namen von Variablen dürfen in Python nur aus den lateinischen Buchstaben (ohne Umlaute äöü), den Ziffern und Unterstrichen _ bestehen. Beachte, dass Python streng zwischen Gross- und Kleinschreibung unterscheidet: $x \neq X$*!*

Wir nennen das in der Informatik eine *Zuweisung*.

Wähle den Variablennamen auf der linken Seite der Zuweisung möglichst sinnvoll! Er soll die Variable beschreiben und ausdrücken, wozu sie dient. Variablennamen dürfen aber keine Leerzeichen enthalten. Statt «`ein name`» schreibst du daher «`ein_name`».

Auf der rechten Seite einer Zuweisung kann ein Zahlenwert oder eine Rechnung mit Zahlen oder anderen Variablen[1] stehen.

AUFGABEN

3. In den USA werden Temperaturen in Grad Fahrenheit angegeben. Die Umrechnung solcher Temperaturangaben von Grad Fahrenheit (T_F) in Grad Celsius (T_C) erfolgt nach der folgenden einfachen Formel:

$$T_C = (T_F - 32) \cdot \frac{5}{9}$$

Programmiere diese Formel in Python und verwende zwei Variablen `temperatur_F` und `temperatur_C`. Bestimme damit, wie vielen $^\circ$ C die Temperatur 86° F entspricht.

4. Schreibe ein Programm, das Längenangaben von Zoll (z. B. $27''$ für die Diagonale eines Displays) in cm umrechnet. (Tipp: $1'' = 2.54\,\mathrm{cm}$)

5.* Schreibe ein Programm, das ausgehend von der Eckenzahl n eines regulären Vielecks den Innenwinkel berechnet.

[1] In der Mathematik wird ein solcher Rechenausdruck *Term* genannt.

3 Die ganzzahlige Division

Lernziele In diesem Abschnitt lernst du:

- Die ganzzahlige Division durchzuführen.
- Den Divisionsrest zu berechnen.

Einführung Die ganzzahlige Division mit Rest brauchst du beispielsweise, wenn du eine Zeitangabe wie 172 Minuten in Stunden und Minuten umrechnen willst. Du dividierst dazu 172 durch 60 und erhältst 2 Rest 52. In Python benutzst du für die ganzzahlige Division den Operator `//`. Mit `print 172 // 60` erhältst du also `2`. Doch wie kommst du zum Divisionsrest?

Der Divisionsrest wird manchmal auch als «remainder» oder «modulo» bezeichnet.

Zur Berechnung des Divisionsrestes verwendest du in Python den Operator `%`. Dieser Operator hat überhaupt nichts mit Prozentrechnung zu tun. Wenn du `print 172 % 60` eingibst, erhältst du `52`, den Rest der ganzahligen Division von 172 durch 60.[2]

Das Programm In diesem Programm wird eine Zeit in Sekunden ins gemischte Format mit Stunden, Minuten und Sekunden umgerechnet. Dazu setzen wir die Variable `sekunden` zunächst auf den Wert 15322 s. In den Zeilen 3 und 4 wird dann die ganzzahlige Division durch 60 durchgeführt. Das Resultat und den Rest legen wir in zwei neuen Variablen `minuten` und `rest_Minuten` ab.

Nach dem gleichen Prinzip wird in den Zeilen 6 und 7 die Anzahl Stunden und die Anzahl der übrigbleibenden Minuten ermittelt und in entsprechenden Variablen abgelegt. Am Ende wird die Zeit mit `print` im gemischten Format «Stunden Minuten Sekunden» ausgegeben.

```
1 sekunden = 15322
2
3 minuten = sekunden // 60
4 rest_Sekunden = sekunden % 60
5
6 stunden = minuten // 60
7 rest_Minuten = minuten % 60
8
9 print stunden, rest_Minuten, rest_Sekunden
```

[2] Weil es auf der Tastatur kein eigentliches Zeichen für den Rest der Division gibt, verwendet man einfach etwas «ähnliches» – und das Prozentzeichen hat ja auch einen Schrägstrich wie bei der Division.

Beachte, dass der `print`-Befehl hier am Ende die Werte von drei Variablen ausgibt. Die Variablen müssen dabei durch ein Komma getrennt werden. Im Ausgabefenster erscheint bei unserem Beisiel wie erwartet `4 15 22`. Wir haben also mit Python berechnet, dass $15\,322$ Sekunden umgerechnet 4 Stunden, 15 Minuten und 22 Sekunden entsprechen.

Die wichtigsten Punkte Das eigentliche Resultat der ganzzahligen Division wird mit dem Operator `//` berechnet, der Rest der ganzzahligen Division mit dem Operator `%`.

Mit dem Befehl `print` kannst du beliebig viele Zahlen, Resultate von Rechnungen und eben auch den Inhalt von mehreren Variablen ausgeben. Die einzelnen Ausgaben müssen dabei durch ein Komma getrennt werden.

AUFGABEN

6. Bei der Division durch 4 können im Prinzip die Reste $0, 1, 2, 3$ auftreten. Bei Quadratzahlen kommen aber nicht alle vier Möglichkeiten vor. Rechne mit einigen Quadratzahlen durch, welche der vier möglichen Reste bei der Division durch 4 auftreten.

7. Bei Geldautomaten gibst du einen Betrag ein. Der Automat muss dann ausrechnen, wie viele Noten von jedem Typ er dazu ausgeben soll. Der Automat in unserer Aufgabe kennt die Notentypen «200», «100» und «20». Schreibe ein Programm, das für den Gesamtgeldbetrag ausrechnet, wie viele Noten von jedem Typ ausgegeben werden sollen – dabei sollen möglichst grosse Noten verwendet werden. Das funktioniert natürlich nur für Beträge, die auch aufgehen, z. B. $480 \to 2 \cdot 200 + 4 \cdot 20$.

8.* Eine Herausforderung für Profis: Nimm beim Geldautomaten noch «50» als Notentyp hinzu und lass dein Programm auch für 210 die korrekte Antwort geben: $100 + 50 + 3 \cdot 20$.

9. Schreibe ein Programm, das eine dreistellige Zahl in Hunderter, Zehner und Einer zerlegt.

10.* Schreibe ein Programm, das natürliche Zahlen bis 255 ins Binärsystem umrechnet. Für 202 soll also beispielsweise `1 1 0 0 1 0 1 0` ausgegeben werden.

4 Text ausgeben

Lernziele In diesem Abschnitt lernst du:

- Zwischen Variablen und Text zu unterscheiden.

Einführung Bei ganz einfachen Prögrämmchen mag es durchaus ausreichen, wenn der Computer am Schluss eine, oder vielleicht zwei Zahlen auf den Bildschirm schreibt. Du wirst aber schnell an den Punkt kommen, an dem die Ausgabe etwas verständlicher oder strukturierter sein soll. Zeit also, einen genaueren Blick auf `print` zu werfen und was es damit auf sich hat.

Computer können zwar sehr gut Einzelheiten analysieren, sind aber unglaublich schlecht darin, einen Zusammenhang oder gar unsere Absichten zu erkennen. Die folgende Codezeile ist für uns zwar klar, hat für den Computer aber keinen Sinn:

```
print Das Resultat ist x
```

Wo liegt das Problem? Aus Computersicht sind *alle* Wörter entweder Anweisungen oder Variablen. `print` ist eine Anweisung, das erkennt er auch sofort. Aber bei `Das` weiss er nicht mehr weiter: Er kann keine Variable mit dem Namen «Das» finden und deshalb beschwert er sich.

Du musst dem Computer also klar machen, dass er «Das Resultat ist» nicht verstehen muss, sondern einfach Buchstabe für Buchstabe ins Ausgabefenster schreiben soll. Dazu setzst du den Text in Gänsefüsschen. In TigerJython wird der Text dann braun. Richtig müsste die Ausgabe von oben also so aussehen:

```
print "Das Resultat ist", x
```

Allerdings funktioniert das so natürlich nur, wenn du eine Variable `x` hast, die er hier ausgeben kann.

Das Programm Das Programm ist hier sehr kurz gehalten. Dafür verwenden wir einen hübschen Trick, um die Nachkommastellen einer Zahl zu «berechnen». Du kennst bereits die Ganzzahldivision `//` und den Operator für den Rest `%`. Diese funktionieren in Python auch für gebrochene Zahlen. Und wenn du eine Zahl durch 1 teilst, dann sind genau die Nachkommastellen der Rest dieser Division.

```
1 zahl = 12.345
2 nachkommastellen = zahl % 1
3 print "Die Nachkommastellen von", zahl,
4 print "sind:", nachkommastellen
```

Bei `print` siehst du sehr schön, wie wir zwischen Textstücken unterscheiden, die der Computer direkt ausgeben soll, und Variablen, die der Computer durch den entsprechenden Wert ersetzen soll. Du siehst auch: Einmal soll er das Wort «Nachkommastellen» auf den Bildschirm schreiben und einmal ist `nachkommastellen` eine Variable. Dank den Gänsefüsschen weiss der Computer, wann was gemeint ist.

Die wichtigsten Punkte Der Computer interpretiert grundsätzlich alle Wörter als Anweisungen, Variablen oder Funktionen. Wenn er ein Textstück bei `print` buchstabengetreu ausgeben und *nicht* interpretieren soll, dann muss dieses Textstück zwischen Gänsefüsschen stehen:

Übrigens kannst du mit der Anweisung `clrScr()` das ganze Ausgabefeld leeren und den Text darin löschen.

```
print "Textstück"
```

Im Gegensatz zu Variablen darf ein solches Textstück auch Leerschläge und Umlaute (äöü) enthalten. Lediglich Gänsefüsschen selber kannst du so nicht ausgeben.

Du darfst Textstücke auch mit Variablen mischen, musst aber dazwischen immer ein Komma setzen:

```
print "Textstück", Variable, "Textstück"
```

AUFGABEN

11. Mit `print 3+4` gibt dir Python einfach das Resultat 7 aus. Schreibe ein Programm, das nicht nur das Resultat berechnet und ausgibt, sondern auch die Rechnung auf den Bildschirm schreibt: `3+4 = 7`

12. Du hast bereits ein Programm gesehen, um eine Zeitangabe von Sekunden in Stunden, Minuten und Sekunden umzurechnen (z. B. $172'' \rightarrow 2' 52''$). Diese erste Version hat aber einfach drei Zahlen ausgegeben. Ergänze das Programm jetzt so, dass die Ausgabe mit Einheiten erfolgt:
`4 Stunden, 15 Minuten, 22 Sekunden`

13. Verwende den Trick mit der Ganzzahldivision durch 1, um zu einer beliebigen (gebrochenen) Zahl anzugeben, zwischen welchen zwei natürlichen Zahlen sie liegt. Die Ausgabe sähe dann zum Beispiel so aus:
`14.25 liegt zwischen 14.0 und 15.0.`

14. *Ascii-Art* ist die Kunst, nur mit den Buchstaben und Zeichen des Computers Bilder darzustellen[3]. Verwende `print`, um solche Ascii-Art-Bilder zu «zeichnen», z. B. die Eule:

```
                 _____  __   __  __       _____
.___.          /  ___/ / /  / / / /      /  ___/
{o,o}         /  __/  / /  / / / /      /  __/
/)__)        /  /__  / /__/ / / /___   /  /__
-'-'-       /_____/ /______/ /_____/ /_____/
```

[3] vgl. `http://de.wikipedia.org/wiki/ASCII-Art`

5 Potenzen, Wurzeln und die Kreiszahl

Lernziele In diesem Abschnitt lernst du:

- Potenzen und Quadratwurzeln auszurechnen.
- Die Kreiszahl π anzuwenden.

Einführung Natürlich kannst du eine Potenz wie 3^5 in Python ausrechnen, indem sie als Multiplikation ausschreibst: $3^5 = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$. Python kann aber Potenzen auch direkt berechnen und hat dafür einen eigenen Operator: `**`. Für 3^5 gibst du also `3**5` ein. `print 3**5` liefert somit wie erwartet 243.

Auch das Gegenstück zu den Potenzen, die Wurzel, lässt sich mit Python berechnen. Für die Quadratwurzel gibt es dafür die Funktion `sqrt` (engl. *square root*). Die Funktion `sqrt` ist in einem «Erweiterungspack», einem sogenannten *Modul* definiert, und zwar im «*math*»-Modul. Bevor du also mit `sqrt` eine Wurzel berechnen kannst, musst du sie aus dem *math*-Modul laden:

```
from math import *
```

Das *math*-Modul ist viel zu umfangreich, um alle Funktionen hier zu besprechen. Neben den Funktionen gibt es aber noch eine nützliche Konstante, die in diesem Modul definiert ist: Die *Kreiszahl* π als `pi`.

Das Programm Das folgende Programm berechnet ausgehend vom Radius r das Volumen einer Kugel. Die Formel dazu lautet:

$$V_{\text{Kugel}} = \frac{4}{3}\pi r^3$$

In unserem Beispiel setzen wir den Radius auf $r = \sqrt{2}$.

```
1 from math import *
2
3 radius = sqrt(2)
4 volumen = 4/3 * pi * radius**3
5
6 print round(volumen, 2)
```

Auf der Zeile 1 wird wie oben angekündigt das Modul `math` geladen. Damit lässt sich auf Zeile 4 das Kugelvolumen berechnen, das schliesslich auf Zeile 6 ausgegeben wird. Bei der Ausgabe wird das Volumen mit dem Befehl `round(Zahl, Anzahl_Stellen)` auf zwei Nachkommastellen gerundet.

Die wichtigsten Punkte Potenzen berechnest du mit `**`, z. B. `5**2` für 5^2.

Für die Quadratwurzel $\sqrt{x}$ gibt es die Funktion `sqrt(x)` im *math*-Modul. Ebenfalls in diesem *math*-Modul ist die Kreiszahl π als `pi` definiert. Bevor du die Quadratwurzel berechnen oder π verwenden kannst, musst du sie aus dem Modul laden:

```
from math import *
```

Mit `round(Zahl, Anzahl_Stellen)` wird eine Zahl auf die angegebene Anzahl Stellen gerundet. Im Gegensatz zu `sqrt` und `pi` kennt Python die `round`-Funktion auch ohne das *math*-Modul zu laden.

AUFGABEN

15. Schreibe ein Programm, das den Umfang und den Flächeninhalt eines Kreises mit vorgegebenem Radius berechnet. Verwende für den Radius eine Variable wie im Programm oben und lass das Programm untereinander den Flächeninhalt und das Volumen auf drei Stellen genau ausgeben.

16. Die Kreiszahl π lässt sich zwar nicht genau angeben. Es gibt aber eine Reihe von Brüchen und Wurzelausdrücken, um π anzunähern. Einige davon sind:

$$\pi \approx \quad \frac{22}{7}, \quad \frac{355}{113}, \quad \sqrt{2}+\sqrt{3}, \quad \sqrt{7+\sqrt{6+\sqrt{5}}}, \quad \frac{63\left(17+15\sqrt{5}\right)}{25\left(7+15\sqrt{5}\right)}$$

Berechne diese Näherungswerte mit Python und vergleiche sie mit π. Auf wie viele Stellen stimmen die Werte jeweils?

17. Der goldene Schnitt ist ein Verhältnis, das in der Kunst und Architektur gerne verwendet wird. Zeige numerisch, dass der goldene Schnitt $a : b = \dfrac{\sqrt{5}+1}{2}$ die Eigenschaft $b : a = a : b - 1$ erfüllt.

18. Nach dem Satz des Pythagoras gilt für die drei Seiten a, b und c eines rechtwinkligen Dreiecks $a^2 + b^2 = c^2$. Berechne mit Python für die zwei Katheten $a = 48$ und $b = 55$ die Hypotenuse c.

19.* Schreibe ein Programm, mit dem du Winkel aus dem Gradmass ins Bogenmass umrechnen kannst. (Genauigkeit: 3 Nachkommastellen)

Zur Erinnerung: Das Bogenmass entspricht der Bogenlänge des entsprechenden Sektors auf dem Einheitskreis.

6 Variablenwerte ändern

Lernziele In diesem Abschnitt lernst du:

- Den Wert einer Variable zu verändern.
- Variablen in Schleifen zu brauchen.

Einführung Eine Variable steht für einen Wert bzw. eine Zahl. Mit der Zuweisung `x = 3` sagst du dem Computer, dass die Variable `x` für den Wert 3 steht. Aber: Variablen können ihren Wert im Laufe des Programms ändern! Damit kannst du die gleiche Rechnung für verschiedene Zahlen durchführen.

Erinnerst du dich an die Schleifen aus dem Kapitel über Turtle-Grafik? Bei einer Schleife wird ein Programmteil mehrmals hintereinander ausgeführt. Mit der Technik aus diesem Abschnitt kannst du bei jedem Durchgang (Wiederholung) der Schleife den Wert deiner Variablen ändern. Darin liegt die Stärke dieser Technik.

Das Programm (I) Dieses erste kurze Programm gibt untereinander die Quadratzahlen von 1 bis 100 aus. Bei jedem Durchgang der Schleife wird in Zeile 3 zuerst der Wert von `x` um 1 erhöht. In Zeile 4 wird dann das Quadrat von `x` ausgegeben.

```
1 x = 0
2 repeat 10:
3     x += 1
4     print x*x
```

Es lohnt sich, den Ablauf dieses Programms mit dem Debugger zu beobachten. Klicke dazu auf den Käfer oben im Editorfenster. Wenn du jetzt das Programm startest, dann kannst du im Debugfenster beobachten, wie sich der Wert von `x` ändert.

Das Programm (II) Das zweite Programm schreibt die Quadratzahlen nicht einfach auf den Bildschirm, sondern zählt sie zusammen. Auch das geschieht wieder mit einer Variable: `summe`.

```
1 x = 0
2 summe = 0
3 repeat 10:
4     x += 1
5     summe += x*x
6 print summe
```

Verwende auch hier den Debugger, um den Programmablauf genau zu verfolgen und sicherzustellen, dass du die einzelnen Anweisungen und Schritte verstehst.

Die wichtigsten Punkte Du kannst den Wert einer Variable während des Programms ändern, indem du etwas hinzuzählst, abziehst oder mit einer Zahl multiplizierst bzw. dividierst. Dazu verwendest du die Operatoren `+=`, `-=`, `*=`, `/=` sowie `//=` und `%=`. Die folgende Codezeile verdreifacht den Wert der Variablen `a`:

```
a *= 3
```

Diese Technik brauchst du vor allem im Zusammenhang mit Schleifen, etwa um alle Zahlen in einem bestimmten Bereich durchzugehen.

AUFGABEN

20. Verwende `*=`, um den Wert einer Variablen bei jedem Schritt zu verdoppeln und lass dir damit alle Zweierpotenzen $(2, 4, 8, \ldots)$ bis $2^{10} = 1024$ ausgeben.

21. Lass dir vom Computer die ersten 20 (a) geraden, (b) ungeraden Zahlen ausgeben.

22. Lass dir vom Computer alle natürlichen Zahlen von 1 bis 100 zusammenzählen und bestätige, dass das Resultat 5050 ist.

23. Schreibe ein Programm, das alle natürlichen Zahlen von 1 bis 10 multipliziert und das Resultat $3\,628\,800$ auf den Bildschirm schreibt.

24. Bilde mit dem Computer die Summe der ersten n Stammbrüche:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots + \frac{1}{n}$$

Probiere aus: Wie gross muss das n sein, damit die Summe grösser ist als 5?

25.* Berechne die Summe der ersten $10\,000$ Stammbrüche einmal von «vorne» und einmal von «hinten» (also $1 + \frac{1}{2} + \ldots$ bzw. $\frac{1}{10000} + \frac{1}{9999} + \ldots$). Wie gross ist der Unterschied zwischen den beiden Summen?

7 Fallunterscheidung

Lernziele In diesem Abschnitt lernst du:

- Mit dem Computer zwischen verschiedenen Fällen zu unterscheiden und dadurch auch Spezialfälle zu berücksichtigen.
- Programmcode nur in bestimmten Situationen ausführen zu lassen.

Einführung Ein Programm soll nicht immer alle Befehle der Reihe nach durcharbeiten, sondern manchmal eine Auswahl treffen und gewisse Befehle nur ausführen, wenn auch die Voraussetzungen dafür gegeben sind. In anderen Worten: Das Programm muss verschiedene Fälle unterscheiden können und dabei auch Spezialfälle berücksichtigen.

Stell dir z. B. vor, dein Programm soll die Wurzel einer Zahl x ziehen. Das geht nur, wenn die Zahl x nicht negativ ist! Es hat also Sinn, vor dem Wurzelziehen den Wert von x mit `if` zu überprüfen:

```
if x >= 0:
    Wurzel ziehen
```

Mit `if` hast du also beim Programmieren die Möglichkeit, auf spezielle Situationen gezielt zu reagieren. Dazu braucht `if` immer eine Bedingung, um entscheiden zu können, ob diese Situation wirklich eintrifft.

Das Programm Woran erkennst du, ob eine Zahl eine Quadratzahl ist? Und wie programmierst du den Computer so, dass er Quadratzahlen erkennt? In diesem Programm hier haben wir folgende Idee verwendet: Wenn du die Wurzel einer Quadratzahl ziehst, dann sind die Nachkommastellen alle Null.

Der ganze Trick funktioniert aber nur, wenn die Zahl nicht-negativ ist. Von negativen Zahlen können wir keine Wurzeln ziehen und das Programm würde abstürzen.

```
1 from math import *
2 zahl = 74
3 if zahl >= 0:
4     wurzel = sqrt(zahl)
5     kommateil = wurzel % 1
6     if kommateil == 0.0:
7         print "Zahl ist eine Quadratzahl."
8     if kommateil != 0.0:
```

```
9           print "Zahl ist keine Quadratzahl."
10  if zahl < 0:
11      print "Negative Zahlen sind nie Quadrate."
```

Ändere den Anfangswert `zahl = 74` in Zeile 2 ab und probiere verschiedene Werte aus. Mache dich so soweit mit dem Programm vertraut, dass du die `if`-Struktur wirklich verstehst!

Die wichtigsten Punkte Die `if`-Struktur hat immer eine Bedingung und darunter Programmcode, der eingerückt ist. Dieser eingerückte Code wird nur dann ausgeführt, wenn die Bedingung bei `if` erfüllt ist. Ansonsten überspringt Python den eingerückten Code.

```
if Bedingung:
    Code, der nur ausgeführt wird,
    wenn die Bedingung wahr ist.
```

Warum braucht es bei Vergleichen ein doppeltes Gleichheitszeichen? Weil das einfache Gleichheitszeichen für Python immer *eine Zuweisung ist.* `x = 3` *heisst also* in jedem Fall, *die Variable* x *soll den Wert 3 haben.*
`if x = 3` *bedeutet für Python übersetzt: «Die Variable* x *hat jetzt den Wert 3 und falls ja, dann...». Das hat nicht wirklich Sinn!*

Die Bedingung ist meistens ein Vergleich. Dabei ist speziell, dass du zwei Gleichheitszeichen brauchst, um zu prüfen, ob zwei Werte gleich sind!

`x == y`	gleich	`x != y`	ungleich
`x < y`	kleiner als	`x >= y`	grösser oder gleich
`x > y`	grösser als	`x <= y`	kleiner oder gleich

AUFGABEN

26. Schreibe ein Programm, das überprüft, ob eine Zahl gerade ist und entsprechend «gerade» oder «ungerade» auf den Bildschirm schreibt.

27. Schreibe ein Programm, das überprüft, ob ein gegebenes Jahr ein Schaltjahr ist. Achte darauf, dass dein Programm auch mit vollen Jahrhunderten (1600, 1700, etc.) richtig umgehen kann. Mit der Einführung des gregorianischen Kalenders 1582 wurde die Schaltjahrregelung nämlich so ergänzt, dass von den vollen Jahrhunderten nur diejenigen Schaltjahre sind, deren erste zwei Ziffern durch 4 teilbar sind.

28. Schreibe ein Programm, das zu einer Zahl alle Teiler sucht und ausgibt. Verwende dazu eine Schleife. In dieser Schleife prüfst du mit der «Division mit Rest» alle möglichen Teiler durch und schreibst diese möglichen Teiler auf den Bildschirm, wenn der Rest Null ist.

8 Ein- und Ausgaben

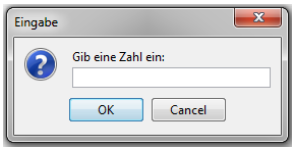
Lernziele In diesem Abschnitt lernst du:

- Den Wert einer Variable erst dann einzugeben, wenn das Programm bereits läuft.
- Resultate und Mitteilungen in einem eigenen Fensterchen auszugeben.

Einführung Bis jetzt hast du alle Werte für das Programm direkt im Programmcode angegeben. In diesem Abschnitt lernst du eine Alternative dazu kennen: Du kannst dem Computer beim Programmieren auch sagen, dass du die Werte erst dann eingibst, wenn das Programm läuft. Wie machst du das?

Anstatt der Variablen `meine_zahl` direkt einen Wert zuzuweisen, z. B. `meine_zahl = 3`, schreibst du:

```
meine_zahl = input("Gib eine Zahl ein:")
```



`input` steht für «Eingabe» und heisst, das du den Wert für `zahl` erst später eingibst. Der Computer zeigt dir dazu jeweils ein kleines Fensterchen an und schreibt den Text «Gib eine Zahl ein:» hinein.

Wenn du willst, kannst du auch deine Ausgaben mit solchen kleinen Fensterchen machen: Ersetze einfach **`print`** durch `msgDlg` und vergiss die Klammern nicht (die du bei **`print`** nicht brauchst):

```
msgDlg("Hallo Welt!")
```

Das Programm Das folgende Programm erfüllt eine ziemlich einfache Aufgabe. Wenn es läuft, dann fordert es dich auf, nacheinander zwei Zahlen einzugeben. Die beiden Zahlenwerte, die du eingibst, werden Variablen zugewiesen, damit das Programm nachher damit rechnen kann.

In der vierten Zeile berechnet das Programm den Durchschnitt der beiden Zahlen und gibt diesen am Schluss aus. `msgDlg` funktioniert wie **`print`**: Du kannst verschiedene Werte und Mitteilungen durch Komma getrennt auf einmal ausgeben.

```
1 zahl1 = input("Gib eine erste Zahl ein:")
2 zahl2 = input("Gib eine zweite Zahl ein:")
3
4 durchschnitt = (zahl1 + zahl2) / 2
```

```
5
6 msgDlg("Der Durschschnitt von", zahl1, "und",
7        zahl2, "ist", durchschnitt)
```

Die wichtigsten Punkte Wenn du möchtest, dass die Werte von Variablen erst dann festgelegt werden, wenn das Programm läuft, dann verwendest du dafür `input`. `input` zeigt ein kleines Eingabefenster auf dem Bildschirm an, in dem du den gewünschten Wert eingibst. Als Argument gibst du bei `input` an, was im Eingabefenster stehen soll:

```
variable = input("Text im Fenster")
```

`input` kennt noch zwei Varianten für ganze bzw. wissenschaftliche Zahlen (in der Fachsprache `int` bzw. `float`, vergleiche p. 32): `inputInt` erwartet eine ganze Zahl und `inputFloat` eine beliebige Zahl (mit oder ohne Nachkommastellen).

Als Gegenstück zu `input` kannst du mit `msgDlg` Mitteilungen und Resultate in einem eigenen Fenster ausgeben. Du kannst beliebig viele Werte angeben, die du wie bei **`print`** durch Komma trennst. Achte auch darauf, Text in Gänsefüsschen zu setzen, Variablen aber nicht.

```
msgDlg("Das Ergebnis ist:", zahl)
```

Text eingeben Du kannst mit **`if`** auch prüfen, ob die Eingabe ein bestimmter Text ist. Dafür musst du den Text, der keine Variable ist, auch hier wieder in Gänsefüsschen setzen.

```
Eingabe = input("Wie heisst du?")
if Eingabe == "Daisy":
    print "Hallo Daisy!"
```

Ein weiteres Beispiel dazu hast du ganz am Anfang in der Einleitung dieses Scripts gesehen!

AUFGABEN

29. Schreibe das Programm aus dem Abschnitt über Fallunterscheidung (Seite 44) so um, dass du über `input` eine Zahl eingeben kannst, die dann daraufhin überprüft wird, ob sie eine Quadratzahl ist.

30.* Schreibe ein Programm, das dein Sternzeichen herausfindet. Dazu muss man zuerst den Tag und den Monat seiner Geburt eingeben. Das Programm antwortet dann z. B. mit «Du bist Schütze.» Tipp: Dafür brauchst du eine ganze Reihe von Fallunterscheidungen mit **`if`**.

9 Alternativen

Lernziele In diesem Abschnitt lernst du:

- Bei einer `if`-Bedingung auch eine Alternative anzugeben.

Einführung Du kannst mit `if` kontrollieren, ob ein bestimmtes Stück in deinem Programmcode ausgeführt werden soll oder nicht. Das haben wir im letzten Abschnitt verwendet, um zu unterscheiden, ob eine Zahl positiv oder negativ ist – weil wir die Wurzel einer negativen Zahl nicht berechnen können. Für eine solche Unterscheidung von *zwei Alternativen* gibt es in Python eine Kurform: Das `else`.

`else` heisst übersetzt «andernfalls» und ersetzt das zweite `if`. Die beiden Codebeispiele hier sind absolut gleichwertig:

```
if zahl >= 0:
    print sqrt(zahl)
if zahl < 0:
    print "Zahl negativ"
```

```
if zahl >= 0:
    print sqrt(zahl)
else:
    print "Zahl negativ"
```

Allerdings hat das `else` natürlich nur deshalb Sinn, weil sich die beiden `if` links ergänzen: Entweder ist `zahl >= 0` erfüllt oder dann `zahl < 0`.

Neben der einfachen `if`-Bedingung gibt es also noch eine `if-else`-Unterscheidung zwischen zwei Alternativen. Das `else` selber hat nie eine eigene Bedingung, sondern tritt immer dann in Kraft, wenn die Bedingung im `if` zuvor *nicht* erfüllt war.

Das Programm Das Osterdatum ändert sich jedes Jahr und muss daher immer neu berechnet werden. Carl Fiedrich Gauss hat für diese Berechnung ein Verfahren (Algorithmus) vorgestellt. Weil Ostern immer im März oder April sind, gibt seine Formel den Tag ab dem 1. März an. Der Tag «32» entspricht dann einfach dem 1. April.

In unserem Programm berechnet `easterday` aus dem Modul `tjaddons` den Ostertag. Danach müssen wir aber selber noch prüfen, ob es im März oder April liegt. Weil wir nur zwei Alternativen haben, können wir das mit `if-else` machen (Zeilen 5 und 11). In der Zeile 6 berücksichtigen wir noch eine Spezialregelung: Wenn das Datum auf den 26. April fällt, dann wird Ostern auf den 19. April vorverschoben.

```
1 from tjaddons import *
2
3 Jahr = inputInt("Gib ein Jahr ein:")
```

```
4  Tag = easterday(Jahr)
5  if Tag > 31:
6      if Tag == 57:
7          Tag = 19
8      else:
9          Tag -= 31
10     msgDlg(Tag, "April", Jahr)
11 else:
12     msgDlg(Tag, "März", Jahr)
```

In diesem Programm kommen zwei `else` vor. Dasjenige in der Zeile 8 gehört zum `if` in der Zeile 6 und das in der Zeile 11 gehört zum `if` in der Zeile 5. Woher weiss der Computer, welches `else` zu welchem `if` gehört? An der Einrückungstiefe! Jedes `else` muss genau gleich eingerückt sein wie das `if`, zu dem es gehört.

Die wichtigsten Punkte Bedingungen mit `if` können entweder alleine stehen oder zusammen mit einer Alternative. Diese wird über `else` eingeleitet und wird immer dann ausgeführt, wenn die Bedingung bei `if` nicht erfüllt ist und der Computer den Code von `if` überspringt. Weil `else` zu einem `if` gehört, hat es nie eine eigene Bedingung.

```
if Bedingung:
    Code ausführen, wenn die
    Bedingung erfüllt ist.
else:
    Code ausführen, wenn die
    Bedingung nicht erfüllt ist.
```

AUFGABEN

31. Schreibe ein einfaches Quiz: Dein Programm stellt also eine Frage und prüft dann, ob die Antwort richtig ist. Wenn ja, gibt es «Richtig!» aus, ansonsten «Falsch!».

32.* Du kannst die Osterformel von Gauss auch selbst programmieren. Hier sind die Formeln dafür (natürlich kannst du in Python nicht alles so kompakt schreiben wie hier):

```
k = Jahr // 100,   p = k // 3,   q = k // 4,
M = (15 + k - p - q) % 30,   N = (4 + k - q) % 7
a = Jahr % 19,   b = Jahr % 4,   c = Jahr % 7,
d = (19a + M) % 30,   e = (2b + 4c + 6d + N) % 7
Ostertag = 22 + d + e
```

10 Schleifen abbrechen

Lernziele In diesem Abschnitt lernst du:

- Eine Schleife abzubrechen, bevor sie fertig ist.

Einführung Schleifen verwendest du beim Programmieren oft dazu, um etwas zu suchen oder zu berechnen. Dabei kannst du nicht immer genau wissen, wie oft sich die Schleife wiederholen muss, bis der Computer das Ergebnis gefunden hat. Deshalb ist es manchmal sinnvoll, eine Schleife mit `break` abzubrechen.

Nehmen wir als Beispiel an, der Computer soll überprüfen, ob 91 eine Primzahl ist. Dazu muss er grundsätzlich alle Zahlen von 2 bis 90 durchgehen und ausrechnen, ob sich 91 ohne Rest durch eine kleinere Zahl teilen lässt. Nachdem der Computer aber festgestellt hat, dass 91 durch 7 teilbar ist, muss er die anderen Zahlen nicht mehr überprüfen. Er hat die Antwort auf unsere Frage und kann daher mit der Berechnung aufhören.

Das Programm Das Programm fordert den Anwender in der ersten Zeile dazu auf, eine ganze Zahl einzugeben (erinnerst du dich daran, dass `int` für eine ganze Zahl steht?).

In den Zeilen 2 bis 8 prüft das Programm der Reihe nach alle möglichen Teiler durch. Wenn die eingegebene Zahl durch einen Teiler wirklich teilbar ist, dann wird die Schleife in Zeile 7 abgebrochen. Die Variable `teiler` enthält jetzt den kleinsten Teiler der eingegebenen Zahl (ausser 1 natürlich).

Am Schluss prüfen wir, ob der gefundene Teiler kleiner ist als die Zahl und geben entsprechend aus, dass es eine Primzahl ist oder nicht.

```
1  zahl = inputInt("Bitte gib eine ganze Zahl ein:")
2  teiler = 2
3  repeat zahl-1:
4      rest = zahl % teiler
5      if rest == 0:
6          break
7      teiler += 1
8
9  if teiler < zahl:
10     msgDlg(zahl, "ist durch", teiler, "teilbar!")
11 if teiler == zahl:
12     msgDlg(zahl, "ist eine Primzahl!")
```

Probiere das Programm mit verschiedenen Zahlen aus und beobachte mit dem Debugger, wann was passiert. Achte dabei vor allem darauf, was `break` bewirkt und wo das Programm nach `break` weiterfährt.

Die wichtigsten Punkte Schleifen wiederholen einen Programmteil für eine feste Anzahl von Durchläufen. Mit `break` kannst du die Schleife aber auch vorzeitig abbrechen. Bei `break` fährt der Computer also nach dem Schleifencode weiter: Er überspringt sowohl den Rest des Schleifencodes als auch alle Wiederholungen, die noch ausstehen.

Die `break`-Anweisung hat nur innerhalb einer `if`-Bedingung Sinn, weil die Schleife sonst sofort abgebrochen und gar nicht wiederholt würde.

```
repeat n:
    Schleifencode
    if Abbruch-Bedingung:
        break
    Schleifencode
```

AUFGABEN

33. Wie oft musst du 1.5 mit sich selbst multiplizieren, bis das Ergebnis grösser ist als 100? Wie sieht es auf mit 1.05 ...?

Schreibe ein Programm, das die Antwort mit einer Schleife sucht. Sobald 1.5^n grösser ist als 100, bricht die Schleife ab und das Programm gibt das Ergebnis aus. Mit `anzahl += 1` kannst du z. B. zählen, wie oft die Schleife tatsächlich wiederholt wird.

34. Zerlege eine Zahl in ihre Primfaktoren. Das Grundprinzip funktioniert ähnlich wie beim Primzahltest oben. Wenn allerdings `rest == 0` ist, dann teilst du die Zahl `zahl` mit `/=` durch den Teiler. Und nur wenn der Rest nicht Null ist, erhöhst du den Teiler um Eins (eine Zahl kann ja auch mehrfach durch 3 teilbar sein).

Woran erkennst du, dass die Zahl fertig zerlegt ist und du die Schleife abbrechen kannst?

35. Schreibe eine «Passwort-Schleife», die sich so lange wiederholt, bis jemand das richtige Passwort (oder die richtige Antwort auf eine Quizfrage oder Rechnung) eingegeben hat.

11 Korrekte Programme

Lernziele In diesem Abschnitt lernst du:

- Dass du ein Programm immer auch daraufhin testen musst, ob es die korrekten Ergebnisse liefert.

Einführung Zu einem fehlerfreien Programm gehören zwei Aspekte. Erstens musst du das Programm so schreiben, dass es der Computer auch versteht und ausführen kann. Zweitens muss dein Programm aber auch das richtige tun bzw. bei einer Berechnung das korrekte Ergebnis liefern. Um diese zweite Art von Korrektheit geht es in diesem Abschnitt.

Wie stellst du sicher, dass dein Programm korrekt ist und wirklich immer das richtige Resultat liefert? Diese Frage ist gar nicht so einfach zu beantworten und wird immernoch erforscht. Was du aber sicher tun kannst: Teste deine Programme und zwar in möglichst verschiedenen Situationen bzw. mit verschiedenen Eingabewerten.

Das Programm Wenn du unendlich viele Zahlen zusammenzählst, dann wird auch das Ergebnis unendlich gross, oder? Erstaunlicherweise nicht immer. Ein Beispiel für eine solche Summe mit endlichen Ergebnis ist:

$$1 - \frac{1}{2} + \frac{1}{4} - \frac{1}{8} + \frac{1}{16} - \frac{1}{32} + \cdots = \frac{2}{3}$$

Du siehst sicher sofort die Struktur hinter dieser Summe:
$\left(-\frac{1}{2}\right)^0 + \left(-\frac{1}{2}\right)^1 + \left(-\frac{1}{2}\right)^2 + \left(-\frac{1}{2}\right)^3 + \left(-\frac{1}{2}\right)^4 + \left(-\frac{1}{2}\right)^5 + \left(-\frac{1}{2}\right)^6 + \cdots$
Weil diese Summe unendlich lang ist, kannst du zwar nicht alles zusammenzählen. Aber es reicht nur schon, z. B. die ersten hundert Summanden zu nehmen und das Ergebnis auszurechnen. Genau das macht das Programm hier.

```
1 i = 0
2 summe = 0
3 repeat 100:
4     summe += -1/2 ** i
5     i += 1
6 print summe
```

Das Programm läuft soweit einwandfrei, nur: Das Ergebnis stimmt nicht! Es sollte $\frac{2}{3}$ sein und nicht -2. Findest du heraus, wo der Fehler liegt? Nimm vielleicht auch den Debugger zu Hilfe oder spiele mit der Anzahl der Wiederholungen, um ein besseres Gefühl zu bekommen (die Lösung erfährst du auf der nächsten Seite).

Und hier die Auflösung: Der Fehler liegt in der Zeile 4. Der Computer berechnet bei `-1/2 ** i` *zuerst* `2**i` *aus und dann den Rest. Um also wirklich* $-\frac{1}{2}$ *zu potenzieren, braucht es Klammern:* `(-1/2) ** i`.

Die wichtigsten Punkte Teste deine Programme immer auch darauf, ob sie die korrekten Ergebnisse liefern. Beginne zuerst mit einfachen Fällen, überprüfe aber immer auch möglichst schwierige, spezielle oder sogar unmögliche Fälle. Erst dann siehst du, ob Dein Programm wirklich funktioniert!

AUFGABEN

36. Die Lösungen einer quadratischen Gleichung $ax^2+bx+c=0$ kannst du mithilfe der Lösungsformel berechnen:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

An der Diskriminante $D = b^2 - 4ac$ siehst du jeweils, ob die Gleichung keine ($D < 0$), eine ($D = 0$) oder zwei ($D > 0$) Lösungen hat.

Schreibe ein Programm, das aus den Koeffizienten a, b und c die Lösungen der Gleichung berechnet. Das Programm soll für jeden Fall funktionieren! Wenn die Gleichung z. B. keine Lösungen hat, dann zeigt dein Programm auch eine entsprechende Meldung an!

Teste das Programm an folgenden Gleichungen (in Klammern sind jeweils die Lösungen angegeben):
$x^2 - 10x + 21 = 0$ $(3, 7)$; $6x^2 - 18x - 60 = 0$ $(-2, 5)$; $3x^2 - 24x + 48 = 0$ (4); $2x^2 - 3x - 5 = 0$ $(-1, 2.5)$; $2x^2 + 3x + 5 = 0$ (–); $x^2 + 5x + 6.5 = 0$ (–).

37. Was passiert, wenn du bei deinem Programm für den Wert von a Null eingibst? Stelle sicher, dass dein Programm auch dann korrekt funktioniert!

Quiz

5. Du möchtest die Rechnung `1 + 2 = 3` auf den Bildschirm ausgeben. Welche Anweisung ist dazu die richtige?

   ☐ a. `print 1 + 2 = 3`

   ☐ b. `print 1 + 2 == 3`

   ☐ c. `print "1 + 2 = 3"`

   ☐ d. `print "1"+"2"="3"`

6. Wie prüfst du, ob die Zahl `x` gerade ist?

   ☐ a. `if x // 2:`

   ☐ b. `if x // 2 == 0:`

   ☐ c. `if x % 2:`

   ☐ d. `if x % 2 == 0:`

7. Welches Codestück hier wird sicher nie ausgeführt, sondern immer übersprungen?

```
if a > 5:
    if a >= 5:
        #A
    else:
        #B
if a <= 5:
    #C
else:
    #D
```

   ☐ a. `#A` ☐ b. `#B` ☐ c. `#C` ☐ d. `#D`

8. Welchen Wert liefert das folgende kurze Programm?

```
resultat = 1
x = 1
repeat 3:
    x //= 2
    resultat += x
print resultat
```

   ☐ a. 1.0 ☐ b. 1.75 ☐ c. 2.0 ☐ d. 7.0

KAPITELTEST 1

Der Name «foo» hat keine eigene Bedeutung und lässt sich wohl am Besten mit «Dings» übersetzen. In Programmieranleitungen werden «foo», «bar» und «baz» gerne als Beispielnamen verwendet, die keinen eigenen Sinn haben. In Python findest auch oft «spam» und «egg».

1. Im folgenden Programm zeichnet die Turtle ein Bild. Zeichne das Bild möglichst genau, ohne das Programm mit dem Computer auszuführen.

```
from gturtle import *
makeTurtle()
def foo(r):
    s = 3.1415926 / 2 * r / 30
    forward(s / 2)
    repeat 29:
        left(3)
        forward(s)
    left(3)
    forward(s / 2)
repeat 2:
    forward(200)
    left(90)
foo(200)
```

2. Im folgenden Programm soll die Turtle ein gleichseitiges Dreieck zeichnen. Dabei fehlt eine Codesequenz. Welche der untenstehenden Sequenzen kannst du dort einsetzen?

```
repeat 3:
    forward(100)
    #???#
```

(a)
```
left(60)
```

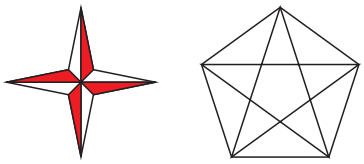
(b)
```
right(300)
```

(c)
```
left(120)
```

(d)
```
repeat 4:
    right(60)
```

(e)
```
repeat 3:
    left(20)
```

3. Schreibe jeweils ein Turtle-Programm, das die Windrose (links) bzw. das Pentagramm (rechts) zeichnet:



4. Mit welchen Codesequenzen wird die positive Zahl x auf 0.5 genau gerundet ausgegeben? Die Zahl 2.3 würde also auf 2.5 gerundet.

(a)
```
print round(x, 0.5)
```

(b)
```
print round(2 * x) / 2
```

(c)
```
print round(2 * x, 0) // 2
```

(d)
```
print (x * 2 + 1) // 1 / 2
```

(e)
```
print (x + 0.5) * 2 // 1 / 2
```

(f)
```
print x + 0.25 - (x + 0.25) % 0.5
```

5. Schreibe ein Programm, das die ersten fünfzig Quadratzahlen zusammenzählt und das Ergebnis auf den Bildschirm schreibt.

6. Ein Programmierer möchte diese Summe ausrechnen:

$$1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\frac{1}{64}+\cdots+\frac{1}{1024}$$

Dazu hat er das folgende Programm geschrieben. In jeder Zeile hat es aber einen Fehler. Finde die Fehler und korrigiere sie, so dass das Programm richtig funktioniert.

```
summe == 1
1 = x
repeat 10
    x //= 2
    summe + x
print "summe"
```

K a p i t e l V i e r

KOORDINATENGRAFIK

Mit der Turtle kannst du im Prinzip beliebige Bilder zeichnen. Es gibt sogar Grafiken, für die die Turtle die beste Möglichkeit darstellt, um sie zu zeichnen. Auf der anderen Seite gibt es aber auch Fälle, in denen andere Verfahren besser und schneller sind.

Zuerst werden wir in diesem Kapitel die Turtlegrafik und das Rechnen mit Variablen zusammenführen. Das ist ein wichtiger Schritt, um komplexere Bilder zeichnen zu können. Der Schwerpunkt dieses Kapitels liegt dann aber auf der Grafik mit einem Koordinatensystem bis hin zu einfachen Spielen.

Es gibt beim Programmieren eine Reihe von Grundtechniken, die du sicher beherrschen musst. Dazu gehören Schleifen (Wiederholungen), Variablen, Fallunterscheidungen und das Definieren eigener Befehle. Ein wichtiges Ziel dieses Kapitels ist, dass du diese Grundtechniken soweit geübt hast, dass du sie vollständig verstehst und problemlos damit umgehen kannst.

Du wirst auch sehen, dass die Beispielprogramme in diesem Kapitel immer länger und grösser werden. Je grösser ein Programm ist, umso wichtiger ist, dass du es mit der Definition neuer Befehle in überschaubare Einzelteile zerlegst und damit sinnvoll strukturierst.

In den späteren Kapiteln werden wir dann diese Grundtechniken so ergänzen, dass du auch grössere und kompliziertere Grafiken und Spiele programmieren kannst. Du wirst aber sehen, dass sich dir mit den Grundtechniken hier schon viele Möglichkeiten bieten.

1 Wiederholungen der Turtle

Lernziele In diesem Abschnitt lernst du:

- Das Verhalten der Turtle in einer Schleife zu variieren, so dass sich die Turtle z. B. abwechselnd nach links und nach rechts bewegt.
- Dass du `else` nicht immer durch `if` ersetzen kannst.

Einführung In diesem Abschnitt kehren wir zur Turtle-Grafik zurück und wiederholen gleich die zwei wichtigsten Prinzipien: Mit `def` kannst du einen neuen Befehl für die Turtle definieren. Und mit `repeat n:` wird der eingerückte Programmteil n Mal wiederholt.

Neu kommt jetzt hinzu, dass du das Verhalten der Turtle auch mit Variablen steuern kannst. In unserem Beispiel lassen wir die Turtle abwechselnd nach links oder nach rechts gehen, indem wir den Wert einer Variablen bei jedem Schleifendurchgang wieder ändern. In den Übungen wirst du mit dieser Technik auch eine Spirale zeichnen.

Das Programm Das Programm beginnt damit, dass wir das Turtle-Modul laden (importieren) und eine neue Turtle mit Fenster erzeugen. Danach definieren wir in den Zeilen 4 bis 9 einen neuen Befehl `treppe` für die Turtle. Zur Erinnerung: Mit dieser Definition macht die Turtle noch nichts! Damit wird erst der Befehl definiert. Ganz am Schluss in Zeile 11 sagen wir dann der Turtle, sie soll eine Treppe zeichnen, und zwar mit einer Stufenlänge von 25 Pixeln.

```
1  from gturtle import *
2  makeTurtle()
3
4  def treppe(stufe):
5      winkel = 90
6      repeat 10:
7          forward(stufe)
8          right(winkel)
9          winkel *= -1
10
11 treppe(25)
```

An diesem Programm ist besonders, dass wir innerhalb der Schleife in den Zeilen 6 bis 9 den Wert der Variablen `winkel` bei jedem Durchgäng ändern. Durch die Multiplikation mit -1 in der Zeile 9 wechselt der Wert von `winkel` immer zwischen -90 und $+90$ ab. Damit dreht sich die Turtle abwechselnd nach links oder nach rechts.

Eine Alternative Es gibt eine zweite Möglichkeit, wie du die Variable `winkel` im Programm bei jedem Schleifendurchgang ändern kannst: Mit `if` und `else`.

```
def treppe(stufe):
    winkel = 90
    repeat 10:
        forward(stufe)
        right(winkel)
        if winkel == 90:
            winkel = -90
        else:
            winkel = 90
```

Das hier ist übrigens ein Fall, bei dem du das `else` nicht durch ein `if winkel != 90:` ersetzen kannst. Warum das nicht geht, ist eine Frage in den Übungenen.

Die wichtigsten Punkte Die Turtle selber kennt zwar keine Variablen. Aber du kannst beim Programmieren trotzdem geschickt Variablen einsetzen, um das Verhalten der Turtle zu steuern.

AUFGABEN

1. Lass die Turtle in einer Schleife eine Spirale zeichnen, die auf einem Rechteck beruht. Dazu dreht sich die Turtle jedes Mal um $90°$. Dafür geht sie bei jedem Mal etwas weiter geradeaus (z. B. 5 Pixel mehr), bevor sie sich wieder dreht.

2.* Vergrössere die Seitenlängen der Spirale jedes Mal um $5\,\%$.

3. Warum kannst du in der zweiten Variante des Programms oben das `if/else` *nicht* (!) durch folgenden Code ersetzen? Warum funktioniert das nicht wie gewünscht?

```
if winkel == 90:
    winkel = -90
if winkel == -90:
    winkel = 90
```

Hinweis: Es kann hilfreich sein, wenn du das Programm mit dem Debugger verfolgst.

4. Lass die Turtle mit der Technik aus diesem Abschnitt eine gestrichelte Linie zeichnen:

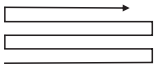
5.* Zeichne mit der Turtle eine drei- oder vierfarbige Linie nach dem Prinzip der gestrichelten Linie. Die Farben wechseln sich also immer ab: Gelb, rot, blau, gelb, rot, blau, ...

2 Farben mischen

Lernziele In diesem Abschnitt lernst du:

- Selber Farben zu mischen und damit Farbverläufe zu zeichnen.

Einführung Wie füllst du am besten ein Rechteck mit einer Farbe aus? Eine Möglichkeit zeigt dir das Bild nebenan: Die Turtle baut das Rechteck Zeilenweise auf. Zwischen den Zeilen muss die Turtle natürlich immer ein Pixel nach oben gehen.



Ein Rechteck ausfüllen, das konntest du aber auch schon vorher. Jetzt variieren wir das Programm aber leicht, indem die Turtle jede Zeile mit einer anderen Farbe zeichnen soll. Mit dieser Technik kannst du sehr einfach schöne Farbverläufe erzeugen. Allerdings reichen dafür wiederum die üblichen Farben nicht aus: Du musst deine Farben schon selber mischen. Das machst du mit der Funktion `makeColor(R, G, B)`. Hier stehen `R` für *rot*, `G` für *grün* und `B` für *blau*. Jeder dieser Anteile ist ein Wert zwischen 0.0 und 1.0, zum Beispiel:

blau	$(0.0, 0.0, 1.0)$	dunkelblau	$(0.0, 0.0, 0.5)$
gelb	$(1.0, 1.0, 0.0)$	grün	$(0.0, 0.5, 0.0)$
hellblau	$(0.4, 0.6, 1.0)$	rot	$(1.0, 0.0, 0.0)$
schwarz	$(0.0, 0.0, 0.0)$	türkis	$(0.0, 0.5, 0.6)$
violett	$(0.5, 0.0, 0.5)$	weiss	$(1.0, 1.0, 1.0)$

Der Computer kann bei jedem Farbanteil rund 250 verschiedene Werte unterscheiden, was dann etwa 16 Millionen mögliche Farben ergibt.

Im Programm schreibst du dann `setPenColor( makeColor(...) )` mit den entsprechenden Farbanteilen. Wichtig ist, dass du auch dann die Nachkommastellen angibst, wenn sie Null sind. Also 0.0 und nicht einfach 0.

Das Programm Dieses Programm zeichnet ein Rechteck mit einem blauen Farbverlauf. Dafür lassen wir die Variable `i` von 1 bis 100 laufen und setzen den Blauanteil jeweils auf: $\frac{i}{100} = \frac{1}{100}, \frac{2}{100}, \frac{3}{100}, \ldots, \frac{100}{100}$.

```
1 from gturtle import *
2 makeTurtle()
3 hideTurtle()
4
5 right(90)
6 i = 0
7 winkel = 90
8 repeat 100:
```

```
9     i += 1
10    setPenColor( makeColor(0.0, 0.0, i/100) )
11    forward(150)
12    left(winkel)
13    forward(1)
14    left(winkel)
15    winkel *= -1
```

Die wichtigsten Punkte Mit `makeColor(R, G, B)` kannst du dir selber alle Farben mischen, die der Computer überhaupt kennt. Die drei Parameter R, G und B stehen für *rot*, *grün* bzw. *blau* und müssen *gebrochene Zahlen* (floats) zwischen 0.0 und 1.0 sein.

AUFGABEN

6. Das Rechteck im Beispielprogramm ist 150 Pixel breit und 100 Pixel hoch. Ändere das so ab, dass es neu 240 Pixel breit und 50 Pixel hoch ist. Die oberste Linie sollte immer noch ein kräftiges Blau sein.

7. Ändere die Farben des Farbverlaufs z. B. von rot nach gelb.

8. Links oben hat das Rechteck einen «Zipfel»: Die Turtle zeichnet da einen einzelnen Pixel auf das Rechteck drauf. Warum macht sie das (welche Zeilen im Programm sind dafür verantwortlich)? Und wie kannst du das beheben bzw. verhindern?

9. Im Beispielprogramm steht in Zeile 5 ein `right(90)`. Welche Bedeutung hat diese Drehung? Was passiert, wenn du diese Winkelangabe änderst, etwa zu `right(30)`? Tipp: Ein `penWidth(2)` an der richtigen Stelle kann das Bild retten.

10. Definiere einen neuen Befehl `farbVerlauf(breite, hoehe)`, der ein Rechteck mit einem Farbverlauf zeichnet. Dabei lassen sich die Breite und die Höhe als Parameter angeben.

11.* Zeichne mit der Turtle einen Farbkreis mit den drei Grundfarben *rot*, *grün* und *blau*. Zwischen diesen Grundfarben gibt es einen fliessenden Übergang. Verwende `penWidth(5)`, damit man die Farben auch gut sieht.

12. Mit `dot(durchmesser)` zeichnet die Turtle an der aktuellen Position einen Punkt mit dem angegebenen Durchmesser.

Zeichne einen «runden Farbverlauf», indem du `dot()` verwendest und immer kleiner werdende Punkte an der gleichen Stelle zeichnest.

13.* Wenn du die Turtle bei einem runden Farbverlauf langsam nach links oben wandern lässt, dann entsteht der Eindruck einer Kugel. Zeichne eine solche Kugel.

3 Mit dem Zufall spielen

Lernziele In diesem Abschnitt lernst du:

- Eine ganzzahlige Zufallszahl in einem festen Bereich zu ziehen.

Einführung Der Zufall spielt in der Informatik eine erstaunlich grosse Rolle. Zwei Beispiele wären Simulationen und Computerspiele, bei denen sich eine Figur zufällig verhält. Wir verbinden sogar beides und präsentieren als Musterprogramm eine Simulation für die *brownsche Bewegung*, bei der die Turtle zufällig umherirrt.

Die einzelnen Teilchen in einem Gas oder in einer Flüssigkeit wandern immer ein wenig umher. Dabei stossen sie immer wieder mit anderen Teilchen zusammen und ändern daher plötzlich ihre Richtung. Diese zufällige Bewegung heisst «brownsche Bewegung». Und die können wir mit der Turtle hervorragend simulieren. Weil wir nur eine einzige Turtle als «Teilchen» haben, tun wir einfach so, als würde sie mit anderen zusammenstossen und ändern alle paar Pixel die Richtung.

Das Programm In diesem Programm lassen wir die Turtle ziemlich ziellos umherirren. Sie geht immer 35 Pixel gerauseaus und dreht sich dann um einen zufälligen Winkel. Für diesen zufälligen Winkel verwenden wir die Funktion `randint` (Zeile 7). Sie zieht eine zufällige ganze Zahl im angegebenen Bereich (bei uns also zwischen 1 und 360).

In den Zeilen 8 und 10 verwenden wir einen Trick, um die Grafik etwas schneller zu machen: Bevor wir die Turtle jeweils drehen, machen wir sie unsichtbar mit `hideTurtle` und nach der Drehung wird sie mit `showTurtle` wieder sichtbar. Damit dreht sich die Turtle ohne die übliche Animation und ist dadurch viel schneller.

```
1  from gturtle import *
2  from random import randint
3
4  makeTurtle()
5  repeat 100:
6      forward(35)
7      winkel = randint(1, 360)
8      hideTurtle()
9      left(winkel)
10     showTurtle()
```

Bevor wir die Funktion `randint` verwenden können, müssen wir sie aus dem Modul `random` laden (importieren).

Die wichtigsten Punkte Im Modul `random` ist die Funktion `randint` definiert. Diese Funktion zieht eine ganzzahlige Zufallszahl aus einem festen Bereich. Diese Zufallszahl wirst du fast immer zunächst in einer Variablen ablegen.

```
from random import randint
zufaellige_variable = randint(0, 10)
```

AUFGABEN

14. Ändere das Programm aus dem Text so ab, dass auch die Länge der Strecke, die die Turtle geht, zufällig ist. Die Turtle soll also nicht immer 35 Pixel, sondern einen zufälligen Wert zwischen 25 und 50 vorwärts gehen.

15. Mit dem Befehl `dot(durchmesser)` zeichnet die Turtle an der aktuellen Position einen (runden) Punkt mit dem angegebenen Durchmesser. Das funktioniert auch, wenn der Farbstift «oben» ist und die Turtle eigentlich keine Spur zeichnet.

Ergänze das Programm aus dem Text damit so, dass die Turtle bei jeder Drehung noch einen Punkt mit Radius 5 zeichnet.

16. Definiere einen Befehl `zufallsfarbe`, der eine zufällige Farbe setzt. Dazu ziehst du zuerst eine Zufallszahl zwischen z. B. 1 und 4. Danach schreibst du eine Reihe von `if`-Bedingungen, nach dem Muster:

```
if zahl == 1:
    setPenColor("red")
```

Zeichne dann damit ein Quadrat, bei dem jede Seite eine (andere) zufällige Farbe hat.

17. Schreibe mit `dot` ein Programm, das zufällige Punkte auf den Bildschirm zeichnet. Natürlich kannst du auch den Radius der Punkte zufällig ziehen lassen. Verwende deinen Befehl `zufallsfarbe` aus der letzten Aufgabe, damit die Punkte auch verschiedene Farben haben.

18.* Schreibe ein Programm, das einen Würfelwurf simuliert, indem es eine zufällige Zahl zwischen 1 und 6 zieht. Lass dein Programm dann 50 Mal den «Würfel werfen» und abbrechen, wenn es das erste Mal eine Sechs «gewürfelt» hat.

Dein Programm soll jede gewürfelte Zahl mit `print` auf den Bildschirm schreiben und am Schluss ausgeben, wie oft es würfeln musste, bis eine Sechs kam.

4 Turtle mit Koordinaten

Lernziele In diesem Abschnitt lernst du:

- Die Turtle an eine bestimmte Position zu setzen oder eine Linie zu einem vorgegebenen Punkt hin zu zeichnen.

Einführung Bis jetzt hast du die Turtle immer *relativ* zu ihrer aktuellen Position gesteuert, indem du Winkel und Längen angegeben hast. Je nach Grafik ist es aber praktischer, die Turtle direkt an eine bestimmte, *absolute* Stelle im Grafikfenster zu setzen, ohne dass du dafür zuerst ausrechnen musst, in welchem Winkel und wie weit sie gehen soll.

Für diese *absolute* Steuerung (d. h. unabhängig von der aktuellen Position und Richtung der Turtle) verwenden wir ein Koordinatensystem, wie du es auch aus der Mathematik kennst. Du gibst also die Positionen innerhalb des Fensters mit einer x- und einer y-Koordinate an (siehe Abb. 4.1). Die Fenstermitte hat dabei immer die Koordinaten $(0, 0)$.

Die zwei wichtigsten Befehle hier sind `setPos(x, y)` um die Turtle direkt an eine bestimmte Stelle zu setzen, und `moveTo(x, y)` um von der aktuellen Position aus eine Linie bis zum Punkt (x, y) zu zeichnen.

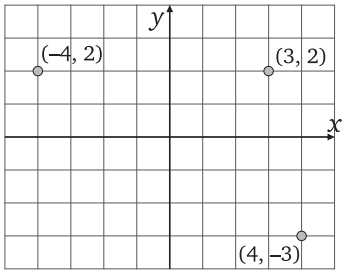


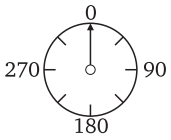
Abbildung 4.1: Das Koordinatensystem mit drei eingezeichneten Punkten.

Das Programm In diesem Programm zeichnet die Turtle zuerst die Koordinatenachsen zur Orientierung. Dabei nutzen wir aus, dass du die Turtle mit `setPos(0, 0)` in die Mitte zurücksetzen kannst. Anschliessend zeichnet die Turtle noch ein rotes Dreieck. Das spezielle an diesem roten Dreieck ist, dass die drei Eckpunkte mit Koordinaten vorgegeben sind: Wir müssen uns also dieses Mal nicht darum kümmern, wie gross die Winkel oder Seitenlängen des Dreiecks sind!

```
1 from gturtle import *
2 makeTurtle()
3
```

```
4  # Die Koordinatenachsen zeichnen
5  setPenColor("black")
6  repeat 4:
7      forward(250)
8      setPos(0, 0)
9      left(90)
10
11 # Das rote Dreieck zeichnen
12 setPenColor("red")
13 penWidth(3)
14 setPos(160, 130)
15 moveTo(-120, -10)
16 moveTo(50, -180)
17 moveTo(160, 130)
```

Die wichtigsten Punkte Die Turtle kann sich entweder relativ zu ihrer aktuellen Position bewegen oder zu einem festen Punkt (x, y) im Koordinatensystem gehen. Mit `setPos(x, y)` setzst du dabei die Turtle direkt an den entsprechenden Punkt und mit `moveTo(x, y)` zeichnet die Turtle eine Linie dahin.



Du kannst die relative und die absolute Steuerung nach Belieben mischen. Dazu gibt es noch einen weiteren Befehl, der nützlich sein dürfte: Mit `heading(winkel)` kannst du die Turtle ausrichten. Ein Winkel von 0 entspricht «gerade nach oben», 90 «direkt nach rechts», etc.

AUFGABEN

19. Definiere zwei neue Befehle `xline(y)` und `yline(x)`. Bei `yline` zeichnet die Turtle eine vertikale (senkrechte) Linie über die ganze Fensterhöhe an der angegebnen x-Position. Bei `xline(y)` zeichnet die eine horizontale (waagrechte) Linie über die ganze Fensterbreite.

20. Nutze die beiden Befehle `xline` und `yline` aus der letzten Aufgabe, um ein Gitter über das ganze Fenster zu zeichnen und zwar sollen die einzelnen Linien 50 Pixel voneinander entfernt sein.

21. Zeichne mit der Turtle noch einmal die Figuren aus der Abbildung 2.1 auf Seite 9. Verwende dieses Mal aber nur absolute Koordinaten, d. h. `setPos(x, y)` und `moveTo(x, y)`.

22. Definiere einen Befehl `rectangle(x1, y1, x2, y2)`, mit dem die Turtle ein Rechteck mit den Eckpunkten $A(x_1, y_1)$, $B(x_1, y_2)$, $C(x_2, y_2)$ und $D(x_2, y_1)$ zeichnet.

5 Koordinaten abfragen

Lernziele In diesem Abschnitt lernst du:

- Herauszufinden, wo sich die Turtle im Moment befindet.

Einführung In diesem Abschnitt geht es darum, die Turtle nicht nur auf einen speziellen Punkt zu setzen, sondern auch herauszufinden, wo sich die Turtle im Moment befindet. Dazu gibt es zwei Funktionen `getX()` und `getY()` für die x- und die y-Koordinaten.

Das Programm (I) Das Grundprogramm hier füllt den mittleren Teil des Bildschirms mit Punkten: Dazu zieht es für die x- und die y-Koordinate Zufallszahlen im Bereich zwischen -200 und 200.

```
1  from gturtle import *
2  from random import randint
3
4  def zufallsPunkt():
5      radius = randint(4, 7)
6      dot(radius * 2)
7
8  makeTurtle()
9  setColor("black")
10 setFillColor("black")
11 fill()
12 repeat 1000:
13     x = randint(-200, 200)
14     y = randint(-200, 200)
15     setPos(x, y)
16     zufallsPunkt()
```

Das Programm (II) Ausgehend vom Grundprogramm ersetzen wir den Befehl `zufallsPunkt` durch eine Alternative. Hier ist der Radius nicht zufällig, sondern hängt von der Höhe des Punktes ab. Je weiter oben der Punkt liegt, umso grösser wird der Radius.

```
def zufallsPunkt():
    y = getY()
    radius = (y + 200) // 40
    dot(radius * 2)
```

Die Koordinaten liegen alle im Bereich zwischen -200 und 200, weil wir die Zufallszahlen im Hauptprogramm oben aus diesem Bereich ziehen.Weil der Radius nicht negativ sein kann, addieren wir 200 und teilen das Ergebnis dann durch 40. Damit liegt der Radius immer im Bereich von 0 bis 10.

Das Programm (III) In dieser dritten Variante kehren wir wieder zu Punkten mit einem zufälligen Radius zurück. Dafür setzen wir die Farbe in Abhängigkeit von der x-Koordinate des Punkts.

```
from tjaddons import makeRainbowColor

def zufallsPunkt():
    x = getX() + 200
    clr = makeRainbowColor(x, 400)
    setPenColor(clr)
    radius = randint(4, 7)
    dot(radius * 2)
```

Die Funktion `makeRainbowColor(index, maxIndex)` gibt eine «Regenbogenfarbe» zurück. Der `index` muss dazu zwischen `0` und `maxIndex` liegen (kleiner als `maxIndex`, aber 0 ist erlaubt).

AUFGABEN

23. Ändere die Version (II) so ab, dass (a) die Radien der Punkte im Bereich zwischen 2 und 18 liegen, (b) die grossen Punkte unten liegen und die kleinen Punkte oben.

24. Schreibe den Befehl `zufallsPunkt()` so um, dass die Punkte in jedem Quadranten eine andere Farbe haben. Die Punkte rechts oben sind z. B. rot, die Punkte links oben blau, etc.

25.* Ändere die Version (III) so ab, dass die Regenbogenfarben nicht mehr horizontal (waagrecht) verteilt sind, sondern diagonal: Links unten ist violet und rechts oben rot.

26.* Schreibe den Befehl `zufallsPunkt()` so um, dass tatsächlich ein Regenbogen entsteht: Die Verteilung der Farben soll also kreisförmig oder sogar bogenförmig sein. Tipp: Du kannst mit dem Satz des Pythagoras die Distanz zwischen zwei Punkten berechnen.

27.* In dieser Aufgabe geht es darum, die relative und absolute Turtlesteuerung zu kombinieren, um ein Pentagramm zu zeichnen (vgl. p. 56). Zeichne mit der Turtle zunächst ein Fünfeck und lass dir bei jedem Punkt die aktuellen Koordinaten ausgeben. Verwende danach `moveTo`, um das Fünfeck mit dem Stern zu ergänzen.

28. Schreibe einen Befehl `dotXY(x, y, r)`, der an der Stelle (x, y) einen Punkt mit Radius r zeichnet und danach die Turtle dahin zurücksetzt, wo sie *vor* dem Befehl war.

29. Definiere einen Befehl `squareDot(s)`, der ein ausgefülltes Quadrat mit der Seitenlänge s zeichnet. Die aktuelle Turtleposition ist dabei der Mittelpunkt.

6 Schleifen in Schleifen

Lernziele In diesem Abschnitt lernst du:

- Schleifen (repeat) zu verschachteln.

Einführung Mit Schleifen kannst du sehr gut verschiedene Werte für eine Variable durchgehen (z. B. mit `i += 1`). Im Koordinatensystem sollst du nun *Paare* von Koordinaten (x, y) durchgehen, etwa so:

$$(1,1), (1,2), (1,3), (1,4), (2,1), (2,2), (2,3), (2,4), (3,1), \ldots$$

Das erreichst du am einfachsten, in dem du *Schleifen verschachtelst*, d. h. eine Schleife innerhalb einer anderen Schleife schreibst.

Das Programm (I) Mit zwei üblichen Spielwürfeln kannst du verschiedene Kombinationen der Zahlen von 1 bis 6 werfen. Dieses Programmchen erstellt eine Liste aller möglichen Zahlen-Kombinationen und schreibt diese auf den Bildschirm. Die äussere Schleife umfasst die Zeilen 3 bis 7. Bei jedem Durchgang wird zuerst die zweite Zahl auf 1 gesetzt, dann wird die innere Schleife 6 Mal wiederholt und schliesslich die erste Zahl um 1 erhöht.

Arbeite hier mit dem Debugger, um ganz sicher zu verstehen, in welcher Reihenfolge Python die Programmzeilen ausführt.

```
1 erste_zahl = 1
2 repeat 6:
3     zweite_zahl = 1
4     repeat 6:
5         print erste_zahl, zweite_zahl
6         zweite_zahl += 1
7     erste_zahl += 1
```

Warum wird die Variable `zweite_Zahl` innerhalb der äusseren Schleife auf 1 gesetzt und nicht wie die Variable `erste_Zahl` ganz am Anfang? Was passiert, wenn du diese Zuweisung ganz an den Anfang nimmst?

AUFGABEN

30. Füge in Zeile 5 eine `if`-Bedingung ein, so dass die Paare nur ausgegeben werden, wenn die zweite Zahl grösser ist als die erste.

31. Ergänze das Programm so, dass es für jede Zahlen-Kombination auch die Summe der beiden Zahlen ausgibt.

32. Schreibe ein Programm, das pythagoräische Zahlentrippel sucht und ausgibt, d. h. ganze Zahlen a, b, c mit $a^2 + b^2 = c^2$ (z. B. $(3, 4, 5)$).

Ein zweidimensionaler Farbverlauf mit rot und blau.

Das Programm (II) Die Technik der verschachtelten Schleifen wenden wir jetzt an, um einen «zweidimensionalen Farbverlauf» (siehe Bild nebenan) zu zeichnen. Die Grundtechnik des Farbverlaufs kennst du bereits von der Seite 60. In diesem Programm ändern wir die *rot*- und die *blau*-Komponente unabhängig voneinander und zeichnen damit ein Quadrat.

```
1  from gturtle import *
2  makeTurtle()
3  hideTurtle()
4  y = 0
5  repeat 20:
6      x = 0
7      repeat 20:
8          setPenColor( makeColor(x / 20, 0.0, y / 20) )
9          setPos(x * 10, y * 10)
10         dot(10)
11         x += 1
12     y += 1
```

Die wichtigsten Punkte Du kannst Schleifen nach belieben ineinander verschachteln. Das ist besonders praktisch, wenn du verschiedene Schleifen-Variablen miteinander zu Zahlenpaaren kombinieren willst.

```
i = 0
repeat n:
    j = 0
    repeat m:
        Schleifencode
        j += 1
    i += 1
```

AUFGABEN

33. Ändere die Farben und die Grösse der Punkte im Programm (II).

34. Schreibe ein Programm, das ein Quadrat aus 12×12 Punkten zeichnet, die sich alle berühren und alle eine zufällige Farbe haben.

35. Schreibe das Programm so, dass die Punkte von links nach rechts immer kleiner und von unten nach oben immer heller (d. h. farbiger) werden.

36.* Definiere einen eigenen Befehl `squareDot(s)`, um einen quadratischen Punkt zu zeichnen und zeichne damit einen durchgehenden zweidimensionalen Farbverlauf (ohne die Lücken, die bei den runden Punkten entstehen).

7 Die Maus jagen

Lernziele In diesem Abschnitt lernst du:

- Mit der Turtle auf Mausklicks zu reagieren.
- Mit `if` und `and` gleich mehrere Bedingungen auf einmal zu prüfen.

Einführung Es ist an der Zeit, unsere Programme etwas interaktiver zu gestalten. Du kannst die Turtle sehr einfach so programmieren, dass sie auf Mausklicks reagiert. Dazu musst du aber zwingend einen Befehl definieren, der bei einem Mausklick ausgeführt werden soll. Wie dieser Befehl heisst, spielt keine grosse Rolle, aber er muss genau zwei Parameter haben: `x` und `y`. Damit gibt dir die Maus an, an welchen Koordinaten du geklickt hast. Das sieht dann z. B. so aus:

```
def onMouseClicked(x, y):
    print x, y
makeTurtle(mouseHit = onMouseClicked)
```

Das Programm In diesem Programm kombinieren wir zwei wichtige Ideen. Zum einen siehst du, dass wir wieder einen Befehl `onClick` definieren, der jedes Mal ausgeführt wird, wenn du mit der Maus klickst. Dazu setzst du bei `makeTurtle` den Parameter `mouseHit` entsprechend.

Wenn die Maus geklickt wird, soll die Turtle eine Linie an die entsprechende Stelle zeichnen. Aber nur innerhalb des Quadrats in der Mitte des Fensters. Dazu prüfen wir, ob die Koordinaten `x` und `y` im entsprechenden Bereich liegen.

```
1  from gturtle import *
2
3  def onClick(x, y):
4      if (-200 < x < 200) and (-200 < y < 200):
5          moveTo(x, y)
6      else:
7          setPenColor("red")
8
9  makeTurtle(mouseHit = onClick)
10 hideTurtle()
11 setPos(-200, -200)
12 repeat 4:
13     forward(400)
14     right(90)
15 setPos(0, 0)
16 showTurtle()
```

In Zeile 4 möchten wir prüfen, ob die x- und die y-Koordinaten *beide gleichzeitig* im Bereich zwischen -200 und 200 liegen. Das erreichen wir mit **and**. Dadurch müssen *beide* Bedingungen erfüllt sein. Ansonsten wird das `moveTo` nicht ausgeführt, sondern das `setPenColor` im **else**-Teil. Für die Bereichsüberprüfung kannst du bei Bedarf z. B. auch schreiben:

```
    if (-100 <= x <= 200) and (y < 180):
```

Die wichtigsten Punkte *Auf Mausklicks reagieren:* Definiere zuerst einen eigenen Befehl, der die beiden Parameter x und y für die Mauskoordinaten hat. Bei `makeTurtle` gibst du dann an, dass dein Befehl bei jedem Mausklick ausgeführt werden soll:

```
makeTurtle(mouseHit = meinKlickBefehl)
```

Wichtig: Bei dieser Definition gibst du *nur den Namen* deines Befehl an! Hier darfst du nie Klammern schreiben!

Bedingungen verknüpfen: Wenn du auf Mausklicks reagieren möchtest, dann wirst du oft prüfen, ob die Maus in einem bestimmten Bereich liegt. Um die x- und die y-Koordinaten gleichzeitig zu überprüfen, kannst du die Bedingungen mit **and** zusammenhängen.

```
if Bedingung-1 and Bedingung-2:
```

AUFGABEN

37. Ändere das Programm oben so ab:

(a) Dass es bei einem Klick ausserhalb des Quadrats eine Zufallsfarbe für die Stiftfarbe wählt.

(b) Dass der Zeichenbereich in der Mitte kein Quadrat, sondern ein Rechteck mit unterschiedlicher Höhe und Breite ist.

(c) Dass die Maus im Zeichenbereich keine Linie zeichnet, sondern nur Punkte an der Stelle, an der du geklickt hast.

(d) Der Zeichenbereich in der Mitte ein Kreis ist.

38. Im Programm oben hast du in der Mitte einen «Zeichenbereich». Ergänze das Programm nun mit farbigen Rechteck so, dass man gezielt die Stiftfarbe wählen kann. Wenn man z. B. auf ein rotes Rechteck aussenrum klickt, dann ist die Stiftfarbe nachher rot.

39.* Baue dein Programm zu einem Zeichenprogramm aus, bei dem du auch die Linienbreite auswählen kannst und die Möglichkeit hast, die Maus zu bewegen, ohne dass eine Linie gezeichnet wird.

8 Globale Variablen

Lernziele In diesem Abschnitt lernst du:

- Innerhalb einer Befehlsdefinition auf globale Variablen zuzugreifen und diese zu verändern.

Einführung Quizfrage: Welchen Wert hat die Variable `x` am Ende dieses Programms? Welcher Zahlenwert wird am Schluss ausgegeben?

```
def twelve():
    x = 12

x = 5
twelve()
print x
```

Der Befehl `twelve` setzt den Wert von `x` auf 12 und weil Python diesen Code nach dem `x = 5` ausführt, sollte die Variable `x` den Wert 12 haben. Tut sie aber nicht. Warum?

```
=== Lokale Variablen ===
x = 12 [int]
=== Globale Variablen ===
x = 5 [int]
```

Um dem auf den Grund zu gehen ist es sinnvoll, dass du den Debugger verwendest und das Programm langsam Schritt für Schritt ausführst. Sieh dir dabei im Debuggerfester an, welchen Wert die Variable `x` hat. Während das Programm den Befehl `twelve` ausführt, wirst du im Debuggerfenster zwei verschiedene `x` sehen (so wie nebenan).

In Python hat *jeder Befehl seine eigenen lokalen Variablen*! Aus Sicht von Python hat das `x` im Befehl `twelve` also grundsätzlich nichts mit dem `x` unten zu tun: Das sind zwei verschiedene Variablen, die nur zufällig gleich heissen. Falls du möchtest, dass der Befehl `twelve` mit der *globalen Variable* arbeitet, dann musst du das mit `global x` angeben:

*Eine Variable, die innerhalb einer Befehlsdefinition verwendet wird, heisst **lokale Variable**. Eine Variable, die ausserhalb einer Definition steht heisst hingegen **globale Variable**.*

```
def twelve():
    global x
    x = 12
```

Schau dir auch dieses Programm schrittweise an. Du wirst feststellen, dass Python jetzt keine lokale Variable `x` mehr verwendet: Beide `x` sind jetzt ein und dieselbe Variable!

Also: Python unterscheidet zwischen *globalen Variablen* im Hauptprogramm selber und *lokalen Variablen* innerhalb einer Befehlsdefinition. Lokale Variablen werden immer dann erzeugt, wenn ein Befehl ausgeführt wird und am Ende wieder gelöscht. Wenn du also den gleichen Befehl ein zweites Mal ausführst, dann erzeugt Python dafür wieder eine *neue* lokale Variable `x`.

Das Programm In diesem Programm repräsentiert ein Punkt im Fenster eine kleine Lampe. Wenn du mit der Maus klickst, soll sich diese Lampe ein- oder ausschalten. Das Programm zeichnet also den Punkt in der Fenstermitte gelb oder schwarz.

Die Variable `lampe` enthält den aktuellen Farbwert der Lampe, am Anfang also schwarz. In der Definition des Befehls von `onClick` müssen wir angeben, dass wir diese globale Variable innerhalb der Definition verwenden (und verändern) wollen (Zeile 5).

```
1  from gturtle import *
2  lampe = "black"
3
4  def onClick(x, y):
5      global lampe
6      if lampe == "black":
7          lampe = "yellow"
8      else:
9          lampe = "black"
10     setPenColor(lampe)
11     dot(20)
12
13 makeTurtle(mouseHit = onClick)
14 clear("black")
```

Mit dem `clear("black")` in Zeile 14 löschen wir am Anfang nicht nur den ganzen Fensterinhalt, sondern setzen die Hintergrundfarbe auch gleich noch auf «schwarz».

Die wichtigsten Punkte Variablen, die innerhalb eines Befehls verwendet werden, sind *lokal*. Das heisst, dass diese Variablen erst dann definiert werden, wenn der Befehl ausgeführt wird. Sobald der Befehl fertig ist, werden diese Variablen wieder gelöscht. Wenn du eine *globale* Variable innerhalb eines Befehls verwenden möchtest, dann gib das mit «`global variable1, variable2, variable3`» an:

```
def mein_befehl():
    global globale_variable
    code
```

AUFGABEN

40. Schreibe ein Programm, das die Anzahl der Mausklicks zählt und ausgibt.

41. Schreibe ein Programm, das bei jedem Mausklick die Hintergrundfarbe wechselt, z. B. mit `makeRainbowColor()`.

9 Mehrere Alternativen

Lernziele In diesem Abschnitt lernst du:

- Verschiedene `if`-Bedingungen mit `elif` zu verketten, so dass nur eine von vielen Alternativen ausgewählt wird.

Einführung Mit `if` und `else` kannst du zwischen genau zwei Alternativen unterscheiden. Wenn du mehr als zwei Alternativen hast, wird das vor allem in Bezug auf das `else` problematisch:

`getPixelColorStr()` *gibt die Farbe des Pixels an, auf dem die Turtle sitzt. Das* `Str` *am Schluss bedeutet, dass du die Farbe als englischen Namen möchtest.*

```
if getPixelColorStr() == "red":
   print "Der Pixel ist rot."
if getPixelColorStr() == "blue":
   print "Der Pixel ist blau."
else:
   print "Der Pixel ist weder rot noch blau."
```

Das Problem: Bei einem roten Pixel schreibt das Programm nicht nur, dass der Pixel rot ist, sondern auch «`Der Pixel ist weder rot noch blau.`» Warum? Weil sich das `else` nur auf den Test mit blau bezieht und nicht auch auf das rot!

Wir können die beiden `if`-Bedingungen aber mit einem `elif` zusammenhängen, so dass das Programm richtig funktioniert, sogar wenn wir noch weitere Tests hinzufügen:

```
if getPixelColorStr() == "red":
   print "Der Pixel ist rot."
elif getPixelColorStr() == "blue":
   print "Der Pixel ist blau."
elif getPixelColorStr() == "green":
   print "Der Pixel ist grün."
else:
   print "Der Pixel ist weder rot, blau noch grün."
```

`elif` steht für `else if`. Wenn du eine Bedingung mit `elif` an die vorhergehende anhängst, dann wird sie nur dann ausgeführt, wenn *keine* der vorausgehenden Bedingungen erfüllt war. Python wählt aus einer solchen Kette von `if`, `elif` und `else` also nur genau eine Alternative aus!

Das Programm Das Programm ist ein kleines Farbenspiel. Die Turtle springt von Punkt zu Punkt. Je nach Farbe des Punkts ändert die Turtle ihre Richtung und ändert die Farbe des Punkts, auf dem sie sitzt. Wenn du die «Farbregeln» änderst, entstehen je nachdem andere Muster.

In unserem Fall macht die Turtle 100 Schritte (Zeile 22) und zeichnet dabei gelbe, rote und blaue Punkte.

```
1  from gturtle import *
2  from time import sleep
3
4  def doStep():
5      hideTurtle()
6      forward(24)
7      if getPixelColorStr() == "white":
8          setPenColor("yellow")
9          dot(24)
10         right(60)
11     elif getPixelColorStr() == "yellow":
12         setPenColor("red")
13         dot(24)
14         left(60)
15     else:
16         setPenColor("blue")
17         dot(24)
18     showTurtle()
19
20 makeTurtle()
21 penUp()
22 repeat 100:
23     doStep()
24     sleep(0.1)
```

Übrigens: `sleep(0.1)` aus dem `time`-Modul wartet 0.1 Sekunden, bevor das Programm weiterläuft. Damit sorgen wir dafür, dass die Turtle nicht zu schnell ist.

AUFGABEN

42. Ergänze das Programm durch weitere Regeln und Farben. Füge auch eine Regel hinzu, bei der die Turtle ein Feld überspringt (verwende `forward`) oder sich um 180° dreht.

43. In diesem Programm könntest du die **`elif`** auch dann *nicht* durch **`if`** ersetzen, wenn du das **`else`** weglässt. Warum nicht? Finde heraus, was passieren würde und erkläre, warum die **`elif`** hier wichtig sind.

44.* Das Programm basiert auf einer sechseckigen Grundstruktur. Ändere das Programm zu einer quadratischen Grundstruktur, indem du nur die Winkel 90° und 180° verwendest. Passe die Regeln so an, dass wiederum interessante Muster entstehen.

45.* Lass die Turtle z. B. bei grünen Punkten zufällig eine Farbe auswählen oder entscheiden, ob sie nach links oder rechts gehen soll.

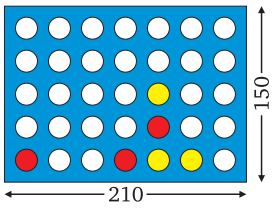
10 Vier Gewinnt

Lernziele In diesem Abschnitt lernst du:

- Fallende Punkte zu simulieren.

Einführung Kennst du das Spiel «Vier Gewinnt»? Du und deine Mitspielerin lassen abwechseln runde Chips (gelbe und rote) in ein Gerüst fallen und wenn du vier gleichfarbige Chips in einer Reihe hast (horizontal, vertikal oder diagonal), gewinnst du das Spiel.

Um das zu programmieren brauchen wir zwei neue Techniken. Zum einen müssen wir die Chips so programmieren, dass sie nach unten fallen (siehe unten). Zum anderen müssen wir aus den Koordinaten des Mausklicks herausfinden, welche Spalte das ist: Weil der linke Rand des Spielfeld bei $x = -105$ liegt, rechnen wir zur x-Mauskoordinate zuerst 150 dazu. Dadurch erhalten wir eine Zahl zwischen 0 und 210 (falls die Maus im Spielfeld ist). Teilen durch 30 (Spaltenbreite) ergibt dann einen Wert zwischen 0 und 6 für die Spalte.



Unser Spielfeld hat 5×7 Punkte, jeder Punkt hat einen Radius von 10 Pixeln und die einzelnen Punkte haben einen Abstand von 10 Pixeln.

Das Programm In diesem Programm definieren wir zuerst einmal vier Befehle. `drawGrid` zeichnet die weissen Punkte als Grundgerüst. `dotXY` zeichnet einen Punkt mit der angegebenen Farbe an die Stelle (x, y). `putChip` lässt einen «Chip» in die angegebene Spalte fallen. `onClick` schliesslich reagiert auf die Mausklicks. Beachte, dass wir hier `mouseHitX` und nicht `mouseHit` verwenden. Das `X` am Ende sorgt dafür, dass wir erst dann einen neuen Chip einwerfen können, wenn der alte bis nach unten gefallen ist (mit dem `X` kannst du also erst dann wieder klicken, wenn die Turtle mit allem fertig ist).

Beachte, dass dieses Programm nicht «fertig» ist. Hier sind nur gelbe Chips im Spiel. Um wirklich spielen zu können, musst du es im Zuge der Aufgaben noch ausbauen.

```
1  from gturtle import *
2  from time import sleep
3
4  def drawGrid():
5      setPenColor("white")
6      x = -90
7      repeat 7:
8          setPos(x, -60)
9          repeat 5:
10             dot(20)
11             forward(30)
12         x += 30
13
14 def dotXY(x, y, color):
15     setPenColor(color)
16     setPos(x, y)
```

```
17      dot(20)
18
19  def putChip(spalte):
20      x = -90 + spalte * 30
21      y = 60
22      repeat 5:
23          dotXY(x, y, "yellow")
24          sleep(0.25)
25          setPos(x, y - 30)
26          if getPixelColorStr() != "white":
27              break
28          dotXY(x, y, "white")
29          y -= 30
30
31  def onClick(x, y):
32      spalte = (x + 105) // 30
33      if 0 <= spalte <= 6:
34          putChip(spalte)
35
36  makeTurtle(mouseHitX = onClick)
37  hideTurtle()
38  penUp()             # Keine Spur zeichnen
39  clear("blue")       # Blauer Hintergrund
40  drawGrid()
```

Wie die Chips fallen Wie machen wir das, dass ein Chip nach unten fällt? Zuerst einmal rechnen wir die Koordinaten (x, y) des obersten Punkts aus. Danach gehen wir der Reihe nach von oben nach unten und versuchen, den Chip «fallen zu lassen»:

Zuerst setzen wir einen gelben Punkt (den Chip). Danach schauen wir, ob der Punkt darunter frei (d. h. weiss) ist. Wenn nicht, dann sind wir fertig und wir brechen die Schleife ab (Zeile 27). Ansonsten löschen wir den gelben Punkt von vorhin wieder, weil er ja nach unten fallen kann. Zwischen dem Zeichnen und dem Löschen des gelben Punkts fügen wir eine Pause von einer Viertelsekunde ein, damit man den Chip auch fallen sieht.

AUFGABEN

46. Definiere eine globale Variable `farbe` und wechsele bei jedem Mausklick (d. h. im Befehl `onClick`) den Wert zwischen `"yellow"` und `"red"` ab. Baue das anschliessend so in das Programm ein, dass die Chips abwechselnd rot und gelb sind und du mit einer Freundin spielen kannst.

47.* Spiele gegen den Computer. Ergänze dazu den Befehl `onClick` so, dass nach jedem Chip, den du hineinwirfst der Computer eine Zufallszahl zieht und in die entsprechende Spalte einen Chip einwirft.

11 Mausbewegungen*

Lernziele In diesem Abschnitt lernst du:

- Die Bewegungen der Maus abzufangen und darauf zu reagieren.

Einführung Im Abschnitt 4.7 hast du ein erstes Beispiel für einen *Callback* gesehen. Nachdem du einen eigenen Befehl «`onClick(x, y)`» definiert hast, kannst du der Turtle angeben, dass dieser Befehl immer dann ausgeführt werden soll, wenn du mit der Maus klickst. Der Name «Callback» kommt daher, dass dich die Maus über diesen Befehl «zurückruft» sobald etwas passiert.

Neben dem Mausklick (`mouseHit`) gibt es noch weitere Maus-Callbacks, die für dich interessant sein dürften: Mit `mouseMoved` und `mouseDragged` reagierst du z. B. auf Bewegungen der Maus.

`mouseMoved`	Maus ohne Tastendruck bewegt.
`mouseDragged`	Maus mit gedrückter Taste bewegt.
`mousePressed`	Maustaste gedrückt.
`mouseReleased`	Maustaste wieder losgelassen.
`mouseClicked`	Mit der Maus geklickt.

Im Unterschied zum Klick haben diese Callbacks hier aber andere Parameter. An die Stelle von `x` und `y` tritt eine «MouseEvent»-Struktur `e`. Diese Struktur hat zwei Funktionen `e.getX()` und `e.getY()`, die dir die *Bildschirm-Koordinaten* der Maus angeben. Diese musst du dann zuerst mit `toTurtleX()` bzw. `toTurtleY()` so umrechnen, dass sie die Turtle versteht.

Das Programm In diesem Programm kannst du nun «freihändig» zeichnen. Dazu nutzen wir den `mouseDragged`-Callback. Dieser wird aufgerufen (zurückgerufen), wenn du die Maus mit gedrückter Taste bewegst. Wie du siehst, können wir auch beliebig viele Callbacks kombinieren.

Mit `setCursor()` ändern wir zudem die Form des Mauszeigers während des Zeichnens. Dazu fangen wir auch die Callbacks für das Drücken und Loslassen der Maustaste ab. Sobald die Maus gedrückt wird, setzen wir die Turtle an die entsprechende Stelle, noch ohne etwas zu zeichnen.

```
from gturtle import *

def onDragged(e):
    x = toTurtleX(e.getX())
    y = toTurtleY(e.getY())
    moveTo(x, y)

def onMouseDown(e):
    setCursor(Cursor.CROSSHAIR_CURSOR)
    x = toTurtleX(e.getX())
    y = toTurtleY(e.getY())
    setPos(x, y)

def onMouseUp(e):
    setCursor(Cursor.DEFAULT_CURSOR)

def onClick(e):
    x = toTurtleX(e.getX())
    y = toTurtleY(e.getY())
    setPos(x, y)
    dot(10)

makeTurtle(mouseDragged = onDragged,
           mousePressed = onMouseDown,
           mouseReleased = onMouseUp,
           mouseClicked = onClick)
hideTurtle()
```

Für den Cursor hast du unter anderem folgende Möglichkeiten: `DEFAULT_CURSOR`, `CROSSHAIR_CURSOR`, `HAND_CURSOR`, `TEXT_CURSOR`.

AUFGABEN

48. Mit `setCursor(Cursor.HAND_CURSOR)` ändert sich der Mauszeiger zu einer kleinen Hand, die du von Links im Internetbrowser her kennst. Schreibe ein Programm, in dem du `mouseMoved` abfängst und den Mauszeiger immer dann zu einer Hand änderst, wenn sich die Maus über einem blauen Rechteck befindet (natürlich musst du das Rechteck zuerst noch zeichnen).

49. Schreibe ein vollständiges Zeichenprogramm, bei dem man gerade Linien ziehen oder freihändig zeichnen kann. Zudem kann man die Stiftbreite und -farbe wählen oder verschieden grosse Punkte setzen. Dazu gibt es in der Mitte einen Zeichenbereich und rundum verschiedene Rechtecke für die Farben und Werkzeuge.

Übrigens: Mit `label("Text")` schreibt dir die Turtle einen Text an der aktuellen Position.

12 Die Turtle im Labyrinth

Lernziele In diesem Abschnitt lernst du:

- Ein grösseres Programm zusammenzusetzen.

Einführung Lass uns ein kleines Spiel programmieren! Wir setzen die Turtle in ein «Labyrinth», in dem sie ihren Weg zum Ziel suchen muss. Dabei unterstützt du die Turtle, indem du ihren Weg mit schwarzen Feldern blockierst, so dass sie sich abdreht (ein schwarzes Feld erzeugst du über einen Mausklick).

Das Programm Wir geben dir hier erst einmal nur die Grundstruktur für das Spiel an. Die Turtle bewegt sich hier auf einer Art von «Schachbrett». Mit Klicken kannst du ein Feld schwarz färben.

Bei einem grösseren Programm ist es wichtig, dass du mit *Kommentaren* erklärst, was die einzelnen Teile machen. Kommentare beginnen jeweils mit «#». Python ignoriert alle solchen Kommentare – sie dienen nur dem Leser zum besseren Verständnis!

Nachdem wir zuerst alle Befehl definiert haben, beginnt das Hauptprogramm erst in Zeile 51. Das machen wir deutlich mit `### MAIN ###` (eine Kurzform für *main program*, also *Hauptprogramm*). Auch das hat für Python keine Bedeutung, hilft uns aber, die Übersicht zu behalten.

```
1  from gturtle import *
2  from time import sleep
3
4  CELLSIZE = 40    # Wähle zwischen: 10, 20, 40, 50
5
6  # Zeichnet das Grundgitter:
7  def drawGrid():
8      global CELLSIZE
9      hideTurtle()
10     setPenColor("gray")
11     x = -400
12     repeat (800 // CELLSIZE) + 1:
13         setPos(x, -300)
14         moveTo(x, +300)
15         x += CELLSIZE
16     y = -300
17     repeat (600 // CELLSIZE) + 1:
18         setPos(-400, y)
19         moveTo(+400, y)
```

```
20          y += CELLSIZE
21      setPos(0, 0)
22      showTurtle()
23
24  # Bei Mausklick eine Zelle schwarz färben.
25  def onClick(x, y):
26      # Die Position der Turtle speichern
27      turtle_x = getX()
28      turtle_y = getY()
29      # Zelle schwarz färben
30      hideTurtle()
31      setPos(x, y)
32      if getPixelColorStr() == "white":
33          setFillColor("black")
34          fill()
35      # Die Turtle wieder dahin zurücksetzen,
36      # wo sie am Anfang war.
37      setPos(turtle_x, turtle_y)
38      showTurtle()
39
40  def doStep():
41      hideTurtle()
42      # Einen Schritt nach vorne machen.
43      forward(CELLSIZE)
44      # Falls die Turtle auf einem schwarzen Feld landet,
45      # setzen wir sie wieder zurück und drehen sie dafür.
46      if getPixelColorStr() == "black":
47          back(CELLSIZE)
48          right(90)
49      showTurtle()
50
51  ### MAIN ###
52  makeTurtle(mouseHit = onClick)
53  drawGrid()
54  # An dieser Stelle könntest du ein Feld als Ziel färben.
55  # Die Turtle auf ein Anfangsfeld setzen:
56  setPos(-400 + 5*CELLSIZE // 2, -300 + 5*CELLSIZE // 2)
57  penUp()
58
59  repeat 1000:
60      doStep()
61      sleep(0.5)
```

Bevor wir die Turtle an eine bestimmte Stelle bewegen, machen wir sie mit `hideTurtle()` unsichtbar und zeigen sich danach wieder mit `showTurtle()`. Das hat zwei Gründe. Zum einen wäre es z. B. merkwürdig, wenn die Turtle bei einem Mausklick kurz zum Mauscursor springt und danach wieder zurückgeht. Zum anderen wäre das Programm zu langsam, wenn die Turtle bei all ihren Bewegungen sichtbar wäre.

AUFGABEN

50. Ergänze das Programm so, dass du mit der Maus schwarze Felder auch wieder wegklicken kannst. Wenn du also auf ein schwarzes Feld klickst, dann wird es wieder weiss.

51. Ergänze das Programm zu einem Spiel, indem du ein Feld rot einfärbst (Zeile 54). Wenn die Turtle dieses rote Feld erreicht hat, ist das Spiel fertig und man hat «gewonnen». Im Modul *sys* gibt es übrigens einen Befehl `exit()`, um das Programm sofort zu beenden:

```
from sys import exit

exit()    # Programm beenden
```

52. Im Moment kann es passieren, dass die Turtle aus dem Bild herausfällt. Ergänze das Programm also so, dass du alle Felder am Rand zuerst schwarz färbst.

53. Das Spiel wird erst dann interessant, wenn du gewisse Hindernisse oder Punkte einbaust. Du könntest z. B. die Anzahl der schwarzen Blöcke zählen, die man braucht, um das Spiel zu lösen. Je weniger Blöcke, umso höher die Punktzahl. Oder du zählst die Schritte, die die Turtle braucht. Überlege dir selber, wie du die Punktzahl berechnen möchtest und gib am Ende diese Punktzahl mit `msgDlg` aus!

54.* Baue ein Level, in dem du bereits gewisse Wände vorgibst, die der Turtle im Weg stehen. Damit wird das Spiel etwas schwieriger.

55.* Neben den schwarzen Blöcken könntest du noch graue Blöcke einführen, die der Turtle ebenfalls im Weg stehen. Im Unterschied zu den schwarzen Blöcken lassen sich die grauen Blöcke aber nicht mehr entfernen.

56.* Die Turtle könnte beim Gehen eine gelbe Spur hinterlassen und sich so weigern, ein zweites Mal auf ein Feld zu gehen.

57.* Mach das Spiel schwieriger, indem sich die Turtle zufällig nach links oder rechts abdreht. Du kannst die Turtle auch schneller machen, indem du den Wert in `sleep()` veränderst. Sie sollte aber nicht zu schnell sein, weil sonst das Erzeugen und Entfernen der Blocks nicht mehr richtig funktioniert.

58.* Für Profis: Programmiere das Spielfeld so, dass wenn die Turtle rechts hinausläuft, dass sie dann von links wieder hineinkommt. Natürlich funktioniert das dann auch in die entgegengesetzte Richtung und genauso für oben/unten.

Quiz

9. Wie kannst du eine Turtlegrafik mit absoluten Koordinaten so an der Mitte spiegeln, dass links und rechts danach vertauscht sind?

   ☐ a. Wechsle bei allen x-Koordinaten das Vorzeichen.

   ☐ b. Wechsle bei allen y-Koordinaten das Vorzeichen.

   ☐ c. Vertausche überall die x- und die y-Koordinaten.

   ☐ d. Das erfordert aufwändige Berechnungen.

10. Wie viele «x» scheibt das folgende Programm auf den Bildschirm?

```
k = 1
repeat 20:
    repeat k:
        print "x",
    k += 1
```

   ☐ a. 20 ☐ b. 21 ☐ c. 210 ☐ d. 400

11. Welchen Wert schreibt das folgende Programm auf den Bildschirm?

```
x = 2
def egg():
    global x
    x = 3
def spam(x):
    x = 5
x = 4
egg()
spam(x)
print x
```

   ☐ a. 2 ☐ b. 3 ☐ c. 4 ☐ d. 5

K a p i t e l F ü n f

FUNKTIONEN

Nachdem du mit den Grundtechniken des Programmierens vertraut bist, führen wir in diesem Kapitel eine der wichtigsten Strukturen überhaupt ein: Die *Funktion*. Funktionen sind eine Erweiterung der selber definierten Befehle und helfen dir, deine Programme so zu strukturieren, dass du die Übersicht behältst und auch grössere Programme schreiben kannst.

Im Zuge dieses Kapitels werfen wir aber auch einen Blick «unter die Haube». Du lernst also, wie der Computer Wurzeln berechnet und Zufallszahlen zieht. Dieses Wissen hilft dir später, dem Computer «intelligentes Verhalten» beizubringen, so dass du z. B. interessantere Spiele programmieren kannst.

1 Wahr und Falsch

Lernziele In diesem Abschnitt lernst du:

- Mit den boolschen Werten «True» und «False» zu arbeiten.

Einführung Beim Programmieren musst Du sehr oft zwischen «wahr» und «falsch» oder «ja» und «nein» unterscheiden. Dazu gibt es in Python zwei spezielle Werte: `True` für «wahr» oder «ja» und `False` für «falsch» oder «nein» (beachte, dass du `True` und `False` in Python immer gross schreiben musst). Diese beiden Werte heissen *boolsche Werte* (benannt nach dem Logiker George Boole).

Natürlich kannst du boolsche Werte wie üblich einer Variablen zuweisen:

```
isPythonCool = True
```

Besonders praktisch ist aber, dass `if` bei einem boolschen Wert automatisch prüft, ob er `True` ist (also ohne Vergleich mit `=`):

```
if isPythonCool:
    print "Python macht Spass!"
```

Und wie prüfst du, ob der Wert `False` ist? Mit `not` (nicht)!

```
if not isPythonCool:
    print "Falsche Antwort ;-)"
```

Das Programm Dieses Programm sucht alle Primzahlen, die kleiner als 100 sind und schreibt sie auf den Bildschirm. Dazu verwenden wir einen Befehl `testPrime(zahl)`, der prüft, ob `zahl` eine Primzahl ist. Du kennst das bereits: Wir probieren alle möglichen Teiler durch und sehen, ob Zahl durch einen dieser Teiler teilbar ist.

Das Resultat der Überprüfung in `testPrime` schreiben wir in die globale Variable `isPrime`. Hier kommen nun die boolschen Werte ins Spiel: `True` heisst «wahr» (in diesem Zusammenhang also «es ist eine Primzahl»), `False` heisst «falsch» (hier also «es ist *keine* Primzahl»). Grundsätzlich gehen wir bei jeder Zahl davon aus, dass es eine Primzahl ist, bis wir einen Teiler gefunden haben.

```
1 def testPrime(zahl):
2     global isPrime
3     isPrime = True
4     teiler = 2
5     repeat zahl:
```

```
6           if teiler * teiler > zahl:
7               break
8           if zahl % teiler == 0:
9               isPrime = False
10              break
11          teiler += 1
12
13  isPrime = True
14  x = 2
15  repeat 99:
16      testPrime(x)
17      if isPrime:
18          print x
19      x += 1
```

Die wichtigsten Punkte Die boolschen Werte `True` und `False` stehen für «wahr» und «falsch». Sie sind besonders praktisch im Zusammenhang mit `if`, weil `if` automatisch (d. h. ohne Vergleich) prüft, ob ein Wert «wahr» ist. Mit `not` («nicht») wird ein boolscher Wert umgedreht: Aus `True` wird `False` und aus `False` wird `True`.

```
variable = True oder False
if variable:
    print "Die Variable ist True"
if not variable:
    print "Die Variable ist False"
```

AUFGABEN

1. Schreibe ein analoges Programm, das alle Zahlen bis 100 durchgeht und prüft, ob es sich jeweils um eine Quadratzahl handelt oder nicht.

2. Suche die Zahlen bis 100 heraus, die Kubikzahlen sind (d. h. von der Form $x = n^3$).

3. Eine perfekte Zahl ist eine Zahl x, deren Teiler (ausser x) zusammengezählt gerade x ergeben. Bsp: Die Teiler von 6 sind: $1, 2, 3, 6$ und $1 + 2 + 3 = 6$. 6 ist also eine perfekte Zahl.

Neben 6 gibt es eine weitere perfekte Zahl, die kleiner ist als 100 und eine dritte, die kleiner ist als 1000. Finde sie!

2 Werte zurückgeben: Die Funktion

Lernziele In diesem Abschnitt lernst du:

- Was der Unterschied zwischen einem Befehl, einer Funktion und einer Variable ist.
- Mit `return` aus einer Funktion True oder False zurückzugeben.

Einführung Über Parameter gibt es eine sehr elegante Methode, um die Werte in einem Befehl festzulegen. Oft wäre es aber auch sehr praktisch, wenn der Befehl seinerseits Werte *zurückgeben* könnte. Im vorhergehenden Abschnitt haben wir dieses Problem über die globale Variable `isPrime` gelöst. Besonders elegant ist das aber nicht.

Viel besser ist es, aus dem Befehl eine *Funktion* zu machen. Im Gegensatz zum Befehl hat eine Funktion am Schluss immer einen bestimmten Wert, ein Ergebnis bzw. Resultat. Diesen Wert gibst du mit `return` an: `return` macht also aus einem Befehl eine Funktion.

Du kennst auch bereits eine Reihe von Funktionen (z. B. `sqrt()` oder `randint()`) und weisst daher schon, wie du Funktionen anwendest. Neu ist jetzt, dass du auch lernst, Funktionen selber zu definieren.

Der Ausdruck `sqrt(2)` ist im Prinzip ein fester Wert (ca. 1.414). Dieser Wert ist aber nicht wie bei einer Variablen einfach gespeichert, sondern wird jedes Mal neu berechnet, wenn du ihn brauchst. Das ist die Idee einer Funktion.

Das Programm Das Programm ist das selbe wie im letzten Abschnitt. Es geht alle Zahlen von 2 bis 100 durch und schreibt die Primzahlen auf den Bildschirm.

Im Unterschied zur ersten Version vorhin ist `isPrime` jetzt eine *Funktion* und verbindet damit die Variable `isPrime` und den Befehl `testPrime` in einem! In Zeile 13 prüft ein `if`, welchen Wert `isPrime()` für die Zahl x hat. `isPrime` ist kein fester Wert, sondern wird in diesem Moment neu berechnet: Das Programm springt also wie bei Befehlen auch zur Zeile 1.

Wenn die Funktion `isPrime` einen Teiler findet, dann wird ihr Wert in Zeile 7 auf `False` gesetzt (es ist *keine* Primzahl). Ansonsten wird der Wert in Zeile 9 auf `True` gesetzt.

```
1  def isPrime(zahl):
2      teiler = 2
3      repeat zahl:
4          if teiler * teiler > zahl:
5              break
6          if zahl % teiler == 0:
```

```
7                return False
8            teiler += 1
9        return True
10
11   x = 2
12   repeat 99:
13       if isPrime(x):
14           print x
15       x += 1
```

Verwende den Debugger, um den Programmablauf zu beobachten und zu verstehen, wie die Ausführung von `isPrime` funktioniert.

Die wichtigsten Punkte Funktionen stehen zwischen Befehlen und Variablen: Im Gegensatz zu Variablen wird der Wert einer Funktion erst dann «berechnet», wenn er gebraucht bzw. verwendet wird. Zudem kann sich der Wert einer Funktion bei jeder Verwendung ändern.

Du definierst Funktionen genau gleich wie Befehle mit **`def`**. Im Unterschied zu Befehlen haben aber Funktionen eine **`return`**-Anweisung, die den Wert festlegt.

```
def Funktionsname(Parameter):
    Code
    return Wert
```

Eine Funktion darf beliebig viele **`return`**-Anweisungen haben. Sobald Python ein **`return`** ausführt, wird die Funktion nicht mehr weiter ausgeführt. Mehere **`return`** haben also nur im Zusammenhang mit **`if`** Sinn.

Besonders praktisch sind Funktionen, deren Wert entweder `True` oder `False` ist. Damit kannst du eine komplexe Überprüfung im Prinzip direkt in eine **`if`**-Bedingung einbetten. Im Verlaufe dieses Kapitels wirst aber noch weitere Funktionen sehen und programmieren.

AUFGABEN

4. Schreibe eine Funktion, die prüft, ob ...

(a) `isSquare(x)` prüft, ob x eine Quadratzahl ist.

(b) `isPerfect(x)` prüft, ob x eine perfekte Zahl ist.

5. Schreibe eine Funktion `isPythagorean(a, b, c)`, die prüft, ob a, b und c ein pythagoräisches Trippel ist, d. h. $a^2 + b^2 = c^2$. Achtung: Die Funktion sollte auch dann `True` angeben, wenn die Zahlen in der «falschen» Reihenfolge sind, also z. B. $a^2 = b^2 + c^2$.

3 Werte mit Funktionen berechnen

Lernziele In diesem Abschnitt lernst du:

- Mit einer Funktion Werte zu berechnen und zurückzugeben.

Einführung Im letzten Abschnitt ging es um Funktionen, die entweder `True` oder `False` zurückgeben. Eine Funktion kann aber auch beliebige (Zahlen-)Werte berechnen und mit `return` zurückgeben. In diesem Abschnitt siehst du zwei Beispiele dafür.

Übrigens macht man in Python oft keinen Unterschied zwischen Befehlen, die etwas ausführen und Funktionen, die etwas berechnen: Python-Programmierer nennen kurzerhand alles eine Funktion, was mit `def` definiert wurde (was nicht ganz korrekt ist).

Das Programm (I) Die Funktion `sumSquares(n)` berechnet die Summe der ersten n Quadratzahlen: $1^2 + 2^2 + 3^2 + \cdots + n^2$. Darunter (ab Zeile 9) suchen wir heraus, wie viele Quadratzahlen es braucht, damit die Summe grösser ist als 1000.

```
1  def sumSquares(n):
2      summe = 0
3      k = 0
4      repeat n:
5          k += 1
6          summe += k**2
7      return summe
8
9  i = 1
10 repeat 50:
11     if sumSquares(i) > 1000:
12         print "Es braucht", i, "Quadratzahlen, damit",
13         print "die Summe grösser ist als 1000."
14         break
15     i += 1
```

Das Programm (II) Beim Programmieren musst du öfters bestimmen, welches die grössere (oder kleinere) von zwei Zahlen a und b ist. Die Funktion `maximum(a, b)` hier erledigt genau diese Aufgabe und gibt die grössere die beiden Zahlen zurück. Die entsprechende Funktion `minimum(a, b)` hätten wir natürlich genau gleich programmieren können wir `maximum`. In diesem Fall haben wir uns aber absichtlich dafür entschieden, eine Alternative aufzuzeigen.

```
1  def maximum(a, b):
2      if a >= b:
3          return a
4      else:
5          return b
6
7  def minimum(a, b):
8      result = a + b - maximum(a, b)
9      return result
10
11 zahl1 = 17
12 zahl2 = 3
13 print "Die grössere Zahl ist:", maximum(zahl1, zahl2)
14 print "Die kleinere Zahl ist:", minimum(zahl1, zahl2)
```

Die wichtigsten Punkte Innerhalb einer Funktion kannst du auch kompliziertere Berechnungsverfahren verwenden, um das Resultat zu berechnen. Sobald das Resultat berechnet (oder bestimmt) ist, gibst du es mit `return` zurück. Es spielt keine Rolle, ob du nur ein `return` am Ende oder mehrere `return` in der Funktion hast. Aber bei `return` verlässt Python *immer* die Funktion sofort – das Resultat steht ja schliesslich fest!

AUFGABEN

6. Schreibe eine Funktion `average(a, b)`, die den Durchschnitt zweier Zahlen berechnet und zurückgibt.

7. Erweitere die Funktion `maximum` so, dass sie die grösste von *drei* Zahlen a, b und c zurückgibt.

8. Wenn du alle natürlichen Zahlen der Reihe nach multiplizierst, erhältst du die *Fakultät*:

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdots n, \qquad 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24, \quad 9! = 362\,880$$

Programmiere die Funktion `fact(n)`, die die Fakultät für eine natürliche Zahl n berechnet und zurückgibt.

9. Wie berechnest du den Abstand zweier Punkte $P(x_1, y_1)$ und $Q(x_2, y_2)$? Du verwendest dazu den Satz des Pythagoras:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Programmiere eine Funktion `distance(x1, y1, x2, y2)`, die diesen Abstand berechnet und zurückgibt. Teste deine Funktion mit $P(7, -2)$, $Q(3, 1)$ ($d = 5$).

4 Bedingungen verknüpfen

Lernziele In diesem Abschnitt lernst du:

- Innerhalb von `if` mehrere Bedingungen miteinander zu verknüpfen.

Einführung Wie prüfst du mit `if`, ob zwei Bedingungen gleichzeitig zutreffen? Mit `and` (und)! Das kennst du bereits aus dem Abschnitt 4.7. Dort haben wir mit `and` geprüft, ob die Variablen x und y beide in einem bestimmten Bereich liegen:

```
if 0 < x < 10 and 0 < y < 20:
```

Und wie prüfst du, ob von mehreren Bedingungen mindestens eine zutrifft? Mit `or` (oder)!

```
if x == 2 or x == 3 or x == 5 or x == 7:
    print "x ist eine einstellige Primzahl"
```

Schliesslich kannst du zusammen mit `not` auch noch komplexere Bedingungen zusammenbauen. Die folgenden drei Varianten prüfen alle genau das gleiche auf verschiedene Arten:

```
if (0 < x < 10) and not (x == 5):
```

```
if (0 < x < 10) and (x != 5):
```

```
if (0 < x < 5) or (5 < x < 10):
```

Es ist übrigens eine gute Idee, bei komplexeren Bedingungen Klammern zu setzen, auch wenn sie nicht immer nötig sind.

Das Programm Das Programm basiert auf einem Gesellschaftsspiel, bei dem es darum geht, der Reihe nach alle Zahlen durchzugehen. Sobald die Zahl durch 7 teilbar ist oder eine 7 enthält, wird die Zahl allerdings durch das Wort «Bingo» ersetzt. Das soll jetzt der Computer für dich machen.

Weil nicht alle Vergleiche auf eine Zeile passen, schreiben wir sie auf zwei Zeilen (3 und 4). Damit Python aber weiss, dass es auf der nächsten Zeile weitergeht, musst du das mit einem *Backslash* «\» ganz am Ende der ersten Zeile anzeigen.

```
1 zahl = 1
2 repeat 50:
3     if (zahl % 7 == 0) or (zahl == 17) or \
4           (zahl == 27) or (zahl == 37) or (Zahl == 47):
5         print "Bingo"
6     else:
7         print zahl
8     zahl += 1
```

Die wichtigsten Punkte Mit `and` (und) bzw. `or` (oder) kannst du mehrere Bedingungen zu einer verknüpfen. Wenn du `and` verwendest, dann müssen *alle Teilbedingungen gleichzeitig* erfüllt sein. Bei `or` muss *mindestens eine Teilbedingung* erfüllt sein.

Vorsicht: Umgangssprachlich bedeutet *oder* (`or`) meistens, dass nur eine der beiden Teilbedinungen erfüllt ist. In der Logik und hier beim Programmieren dürfen aber auch beide Teilbedingungen erfüllt sein.

AUFGABEN

10. Formuliere die Bedingungen jeweils mit einem einzigen `if`.

(a) Die Zahl x ist durch 5 teilbar aber nicht durch 10.

(b) Die Zahl x ist durch 11, durch 13 oder durch beide teilbar.

(c) Die Zahlen x und y sind entweder beide positiv oder beide negativ.

(d) Die Zahl z ist durch 2 oder durch 3 teilbar, aber nicht beides.

11. Prüfe mit einem einzigen `if`, ob von den drei Zahlen a, b und c eine die Summe der beiden anderen Zahlen ist (z. B. $3 + 4 = 7$).

12. Schreibe eine Funktion, die prüft, ob von drei Zahlen a, b und c genau zwei Zahlen gleich sind, aber nicht alle drei. Der Rückgabewert für $(1, 2, 2)$ wäre dann `True` und für $(1, 2, 3)$ oder $(4, 4, 4)$ `False`.

13. Schreibe eine Funktion `maximum(x, y, z)`, die die grösste der drei Zahlen x, y und z zurückgibt. Verwende dazu möglichst wenige `if`, dafür aber `and` bzw. `or`.

14.* Schreibe eine Funktion, die für eine zweistellige ganze Zahl x prüft, ob sie durch genau drei voneinander verschiedene Primfaktoren teilbar ist. Die kleinste solche Zahl wäre z. B. $30 = 2 \cdot 3 \cdot 5$.

5 Variablen: Aus alt mach neu

Lernziele In diesem Abschnitt lernst du:

- Aus dem aktuellen Wert einer Variable den neuen Wert zu berechnen.

Einführung Der griechische Mathematiker Heron fand einen einfachen *Algorithmus* (d. h. ein *Berechnungsverfahren*), um die Wurzel x einer Zahl n zu berechnen: $x = \sqrt{n}$. Am Beispiel von $\sqrt{2} \approx 1.414214$ ($n = 2$) sieht Herons Algorithmus so aus:

$$
\begin{array}{ll}
x_0 = \frac{n}{2} & x_0 = 1 \\
x_1 = \frac{x_0 + \frac{n}{x_0}}{2} & x_1 = \frac{1 + \frac{2}{1}}{2} = 1.5 \\
x_2 = \frac{x_1 + \frac{n}{x_1}}{2} & x_2 = \frac{1.5 + \frac{2}{1.5}}{2} \approx 1.4167 \\
x_3 = \frac{x_2 + \frac{n}{x_2}}{2} & x_3 = \frac{1.4167 + \frac{2}{1.4167}}{2} \approx 1.414216 \\
x_4 = \cdots &
\end{array}
$$

Dieses Verfahren bzw. Algorithmus kommt sehr schnell zu einem guten (Näherungs-)Ergebnis (in der Fachsprache sagt man «der Algorithmus konvergiert sehr schnell»). Weil in jedem Schritt genau das gleiche passiert, eignet sich der Algorithmus von Heron besonders gut zur Berechnung mit dem Computer. In jedem Schritt wird das x nach dem Muster

$$x_{neu} = \frac{x_{alt} + \frac{n}{x_{alt}}}{2}$$

neu berechnet. Im nächsten Schritt wird dann das x_{neu} natürlich zum x_{alt} – im Wesentlichen haben wir also nur ein x, das immer neu berechnet wird.

Beim Programmieren lassen wir das «alt» und «neu» gleich ganz weg und schreiben:

```
x = (x + n / x) / 2
```

Wichtig: Obwohl diese Zeile ein Gleichheitszeichen enthält, ist es *keine* Gleichung! Python versteht Gleichungen nicht: Das Gleichheitszeichen bedeutet für Python eine *Zuweisung*: Hier wird der Wert der Variable x festgelegt.

Wenn eine Variable x links und rechts vom Gleichheitszeichen (eigentlich also «Zuweisungszeichen») vorkommt, dann nimmt Python *immer* an, dass links ein x_{neu} und rechts ein x_{alt} gemeint ist. Das lässt sich nicht ändern, sondern ist fest eingebaut.

Das Programm Hier kommt nun der Algorithmus von Euklid als ganzes Programm, das die Wurzel einer Zahl n berechnet.

```
1 n = inputFloat("Wurzel berechnen aus:")
2 x = n / 2
3 repeat 10:
4     x = (x + n / x) / 2
5 print "Die Wurzel aus", n, "ist", x
```

Die wichtigsten Punkte Das Gleichheitszeichen bedeutet für Python immer, dass der Variablen links ein neuer Wert zugewiesen wird. Den neuen Wert kannst du dabei auch aus dem vorherigen Wert der Variable berechnen. So bedeutet die Zeile

```
a = a * a - 3 * a
```

ausformuliert: $a_{neu} = a_{alt} \cdot a_{alt} - 3 \cdot a_{alt}$.

AUFGABEN

15. Schreibe die folgenden Programmzeilen nach Möglichkeit mit einer der vier Kurzformen `x += 1`, `x -= 1`, `x *= 2`, `x /= 2`.

(a) `x = x - 1`
(b) `x = 1 - x`
(c) `x = 2 * x`
(d) `x = x / 2`
(e) `x = 2 / x`
(f) `x = 1 + x`
(g) `x = x // 2`
(h) `x = x * 2`
(i) `x = x + 1`

16. Ersetze im Programm oben die Zeile 4 durch folgende zwei (gleichwertige) Zeilen:

```
    x_neu = (x + n / x) / 2
    x = x_neu
```

Baue das Programm nun so aus, dass es die Schleife abbricht, wenn die Änderung zwischen x und x_{neu} kleiner ist als 0.001.

17. Programmiere folgenden Algorithmus: Starte mit einer beliebigen natürlichen Zahl x und prüfe bei jedem Schritt: Wenn x gerade ist, dann teile x durch 2, ansonsten rechne $x_{neu} = 3 \cdot x_{alt} + 1$. Lass alle diese Zahlen ausgeben.

Die Frage dazu lautet dann: Wie lange dauert es jeweils, bis der Algorithmus zu $x = 1$ kommt? Es ist auch noch unbekannt, ob der Algorithmus tatsächlich für jeden Startwert zu 1 kommt.

Für den Startwert $x = 7$ ergibt sich z. B. folgende Ausgabe:
`7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 -> 17`
Was ist die längste solche Zahlenfolge, die du findest?

6 Algorithmen mit mehreren Variablen

Lernziele In diesem Abschnitt lernst du:

- Mehrere Variablen gleichzeitig zu ändern.

Einführung Algorithmen (Berechnungsverfahren) spielen eine zentrale Rolle in der Informatik. Wir bauen daher den letzten Abschnitt aus und schauen uns auch hier wieder einen solchen Algorithmus an. Im Unterschied zum letzten Abschnitt funktioniert der Algorithmus hier mit mehreren Variablen gleichzeitig: Aus zwei alten Werten x_{alt} und y_{alt} werden zwei neue Werte x_{neu} und y_{neu} berechnet.

Kennst du die *Fibonacci*-Folge: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$ Du siehst sicher sofort, wie diese Zahlen gebildet werden. Die nächste Zahl ist immer die Summe der beiden letzten: $13 + 21 = 34$. Weil wir aber immer die *zwei* letzten Zahlen brauchen, reicht es nicht mehr, mit nur einer Variablen zu arbeiten.

x	y	$x+y$
1	1	2
↙	↙	
1	2	3
↙	↙	
2	3	5
↙	↙	
…	…	…

Wenn x und y jeweils die zwei letzten Zahlen der Finbonacci-Folge sind, dann können wir den Algorithmus so darstellen:

$$x_{neu} = y_{alt}$$
$$y_{neu} = x_{alt} + y_{alt}$$

Was *nicht* funktioniert ist eine der folgenden Lösungen. Warum funktionieren sie nicht? Finde selber heraus, wo das Problem liegt!

```
x = y
y = x + y
```

```
y = x + y
x = y
```

Das Problem wird sofort ersichtlich, wenn du ein «alt» bzw. «neu» richtig hinzusetzst.

$$x_{neu} = y_{alt}$$
$$y_{neu} = x_{neu} + y_{alt}$$

$$y_{neu} = x_{alt} + y_{alt}$$
$$x_{neu} = y_{neu}$$

Eine Lösung besteht darin, mit einer dritten Hilfsvariablen zu arbeiten:

```
summe = x + y
x = y
y = summe
```

In Python gibt es aber eine viel elegantere Variante. Du kannst zwei Variablen gleichzeitig neue Werte zuweisen:

```
x, y = y, x+y
```

Erinnerst du dich, dass die Variablen links immer die «neuen» Werte enthalten und rechts immer die «alten»? In der Schreibweise mit «alt» und «neu» ist das also: $x_{neu}, y_{neu} = y_{alt}, x_{alt} + y_{alt}$.

Das Programm In diesem Programm berechnen wir die ersten 20 Fibonacci-Zahlen nach dem Muster, das oben beschrieben wurde. In Zeile 3 hat das `print` ein Komma am Schluss, damit die Zahlen alle auf die gleiche Zeile geschrieben werden.

```
1 x, y = 1, 1
2 repeat 20:
3     print x,
4     x, y = y, x+y
```

Die wichtigsten Punkte Python ist in der Lage, Berechnungen nebeneinander (im Prinzip also unabhängig und gleichzeitig) auszuführen. Dazu fasst du die verschiedenen Zuweisungen mit Komma zu einer einzigen Zusammen – rechts vom Gleichheitszeichen werden immer die aktuellen bzw. «alten» Werte der Variablen eingesetzt.

Du kannst also gleichzeitig die Differenz und die Summe zweier Zahlen a und b berechnen mit:

```
a, b = a+b, a-b
```

AUFGABEN

18. Was bewirkt die Anweisung: «`a, b = b, a`»?

19. Der Algorithmus von Euklid ist ein Verfahren, um den ggT (*grösster gemeinsamer Teiler*) zweier Zahlen a und b zu berechnen. Programmiere diesen Algorithmus in Python.

- Zu Beginn muss $a \geq b$ gelten. Ansonsten vertausche a und b.
- Wenn a durch b teilbar ist (`a % b == 0`), dann ist b der ggT.
- Berechne a und b neu nach dem Muster ($a \,\%\, b$ ist hier der Rest der Division $a : b$, z. B. $7 \,\%\, 3 = 1$):

$$a_{neu} = b_{alt}$$
$$b_{neu} = a_{alt} \,\%\, b_{alt}$$

a	b	$a \,\%\, b$
480	252	238
↙	↙	
252	238	14
↙	↙	
238	14	0

Prüfe dein Programm mit $ggT(252, 480) = 14$ und $ggT(342, 408) = 6$.

INDEX