Python

Eine Einführung in die Computer-Programmierung

Tobias Kohn

INHALTSVERZEICHNIS

K a p i t e l E i n s

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 eigenes 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://jython.tobiaskohn.ch/`

herunter und speichere es am besten in einem eigenen Ordner, dem du z. B. den Namen `Python` geben kannst.

Damit ist die Installation eigentlich bereits abgeschlossen und du kannst das Programm `tigerjython2.jar` starten (Voraussetzung ist allerdings eine aktuelle Version von *Java/JRE*). Die Abbildung 1.1 zeigt, wie das etwa aussehen sollte (allerdings wird das Editorfeld in der Mitte bei dir leer sein).

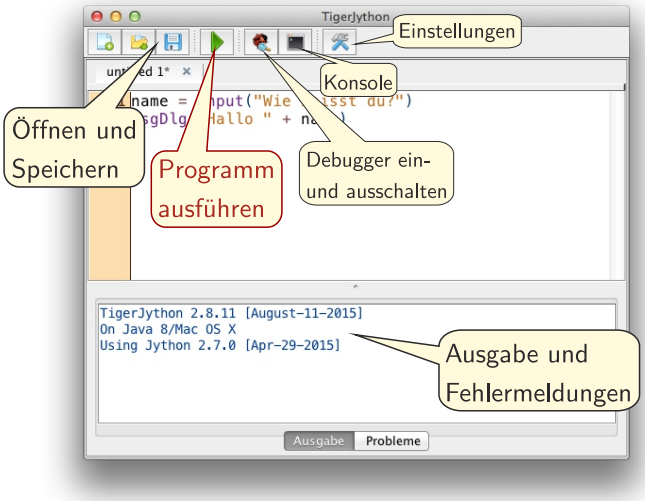


Abbildung 1.1: So präsentiert sich das Programm *TigerJython*. Den Umgang mit der Konsole und dem Debugger wirst du später im Kurs lernen. Am wichtigsten ist die Schaltfläche zum Ausführen des Programms.

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

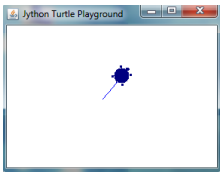
Die *Turtle* ist eine kleine Schildkröte, die eine Spur zeichnet, wenn sie sich bewegt. Du kannst ihr sagen, sie soll vorwärts gehen oder sich nach links oder rechts abdrehen. Indem du diese Anweisungen geschickt kombinierst, entstehen Zeichnungen und Bilder.

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(120 * 1.414216)
6 left(135)
7 forward(120)
8 left(90)
9 forward(120)
```

Achte beim Programmieren auf die exakte Schreibweise! Die Gross- und Kleinschreibung ist wichtig. Die Befehle müssen alle ganz links in der Zeile beginnen. Vergiss auch das Sternchen nach `import` *oder die Klammern nach* `makeTurtle` *nicht!*

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.

Im gleichschenkligen rechtwinkligen Dreieck ist die Hypotenuse $\sqrt{2}$ (ca. 1.414216) Mal so lang wie die beiden Katheten. Deshalb multiplizieren wir in der Zeile 5 die Länge der kürzeren Seiten mit diesem Wert, um die Länge der längsten Seite zu erhalten.

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. Jede Anweisung steht auf einer eigenen Zeile. Die Anweisungen, die die Turtle sicher versteht sind:

`forward(s)`	s Pixel vorwärts bewegen.
`back(s)`	s Pixel rückwärts bewegen.
`left(w)`	Um den Winkel w nach links drehen.
`right(w)`	Um den Winkel w nach rechts drehen.
`dot(d)`	Einen Punkt mit Durchmesser d zeichnen.
`speed(-1)`	Turtle auf volle Geschwindigkeit setzen.

AUFGABEN

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

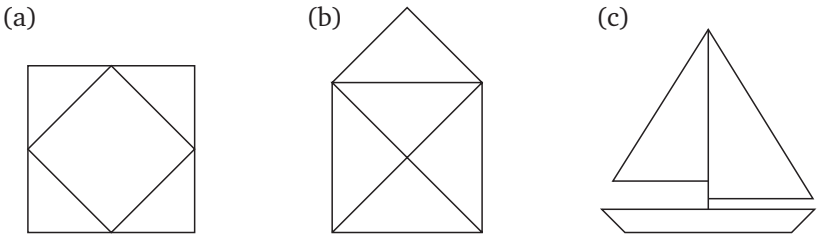


Abbildung 2.1: In der Mitte das «Haus von Nikolaus» und rechts ein einfaches Segelboot.

2. 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.

3.* Lass die Turtle ein einfaches Segelschiff zeichnen wie in der Abbildung 2.1(c).

2 Den Farbstift steuern

Lernziele In diesem Abschnitt lernst du:

- Die Farbe und 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*). 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.

Auch die Befehle `penUp()` und `penDown()` funktionieren nur, wenn du die Gross- und Kleinschreibung beachtest und die leeren Klammern hinschreibst.

Über die Anweisung `setPenColor(Farbe)` wählst du die Farbe des Stifts aus. Wichtig ist, dass du den Farbnamen in Gänsefüsschen setzt. Wie immer beim Programmieren kennt die Turtle nur *englische* Farbnamen. Für eine rote Stiftfarbe schreibst du zum Beispiel:

```
setPenColor("red")
```

Schliesslich kennt die Turtle noch den Befehl `setPenWidth(Breite)`. Damit stellst du die Breite der gezeichneten Linie ein. Die Breite gibst du 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  setPenWidth(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. Den Farbnamen setzst du immer in Anführungszeichen. 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.

`penUp()`	Stift hochheben, also keine Spur mehr zeichnen.
`penDown()`	Stift senken, also wieder eine Spur zeichnen.
`setPenColor(c)`	Die Farbe c des Stifts festlegen.
`setPenWidth(b)`	Die Breite b des Stifts festlegen.
`clean(c)`	Das ganze Fenster mit der Farbe c füllen.

Eine kleine Auswahl der Farbnamen, die die Turtle kennt:

yellow	cyan	green
gold	blue	lime green
orange	navy	dark green
red	purple	white
dark red	magenta	gray
brown	sienna	black

AUFGABEN

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

5. Zeichne mit der Turtle die «Regenbogen-Blume» in der Abbildung 2.2. Jeder Kreis hat eine andere Farbe: gelb, orange, rot, violet und blau.

Abbildung 2.2: Regenbogen-Blume.

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 Ein zentraler Aspekt beim Programmieren ist, dass du selbst neue Befehle definierst. Du erweiterst also den bisherigen Wortschatz der Turtle um eigene Befehle. Damit werden deine Programme nicht nur verständlicher, du kannst dir auch viel Arbeit ersparen.

Jeder Befehl braucht einen eindeutigen Namen, Klammern und den Programmcode, der den Befehl beschreibt. Den neuen Befehl definierst du mit `def NeuerName():`. Darunter schreibst du *eingerückt* alle Anweisungen und Befehle auf, die zum neuen Befehl gehören. Dieser beschreibende Programmcode heisst der *Körper* des Befehls und muss zwingend gleichmässig eingerückt sein.

Wichtig: Bei der Definition eines Befehl führt die Turtle den Code nicht aus! Damit der Code ausgeführt wird musst du den Befehl später explizit aufrufen (d. h. verwenden).

Der Name eines Befehls darf keine Kommata oder Leerschläge enthalten. Wenn du mehrere Wörter kombinieren willst, dann mach das so: `grosses_quadrat()`.

Das Programm In diesem Programm definieren wir in den Zeilen 3 bis 11 mit `def` den Befehl `quadrat()`. Beachte: Alle acht Anweisungen, die das Quadrat ausmachen, sind um vier Leerschläge eingerückt.

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

Hier definieren wir den Befehl

Hier rufen wir den Befehl auf

```
16 penUp()
17 forward(50)
18 left(60)
19 forward(50)
20 penDown()
21 setPenColor("blue")
22 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 zu kombinieren. Im Programm oben wird die Turtle in Zeile 12 deshalb auch nochmals um $90°$ gedreht (was ja nicht nötig wäre).

Die wichtigsten Punkte Mit der Anweisung `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 heissen *Körper* des Befehls und müssen *eingerückt* sein, damit der Computer weiss, was alles dazu gehört.

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

Vergiss die Klammern und den Doppelpunkt nach dem Namen nicht!

AUFGABEN

6. Definiere einen Befehl für ein gleichseitiges Dreieck und zeichne damit die Figur in der Abbildung 2.3. Die fünf Dreiecke berühren sich nicht, sondern haben einen kleinen Abstand und die Farben *blau, schwarz, rot* (oben) und *gelb, grün* (unten).

Abbildung 2.3: Olympische Dreiecke.

7. Das Muster in der Abbildung 2.4 besteht aus einem sich wiederholenden Element. Definiere zuerst einen Befehl, der das wiederholende Element zeichnet und verwende dann den Befehl, um das ganze Muster zu zeichnen.

Abbildung 2.4: Sich wiederholendes Muster.

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(s)` gibst du in Klammern an, *wieviel* die Turtle vorwärts gehen soll. Dieser Platzhalter s heisst *Parameter*. Beim Aufruf `forward(10)` gibst du an, dass der Parameter den Wert 10 haben soll.

Der Wert eines Parameters wird als «Argument» bezeichnet. Diese Unterscheidung wird nicht immer korrekt eingehalten, so dass Parameter und Argument oft als gleichbedeutend angesehen werden.

Im letzten Abschnitt hast du einen eigenen Befehl `quadrat()` definiert. Im Unterschied zu `forward(s)` 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 (I) Auch in diesem Programm definieren wir in Zeilen 3 bis 11 einen Befehl, der ein Quadrat zeichnet. Der Befehl hat in den Klammern einen Parameter (Platzhalter) `seite`. Wenn du den Befehl `quadrat(seite)` aufrufst, dann musst du für den Parameter `seite` einen konkreten Wert angeben (ein sogenanntes *Argument*).

In Zeile 15 rufen wir den Befehl `quadrat(80)` mit dem Argument 80 auf. Wenn Python jetzt den Code im Körper von `quadrat(seite)` ausführt, dann wird `seite` jedes Mal durch den Wert 80 ersetzt.

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

```
17 setPenColor("blue")
18 quadrat(50)
```

Das Programm (II) Der Befehl `color_dot(farbe)` hat den Parameter `farbe`. Als Argument geben wir in Zeile 8 allerdings keine Zahl an, sondern einen Farbnamen. Der Farbname muss auch hier in Anführungszeichen stehen.

```
1 from gturtle import *
2
3 def color_dot(farbe):
4     setPenColor(farbe)
5     dot(10)
6
7 makeTurtle()
8 color_dot("green")
```

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. Dieser Wert ist das Argument.

```
Befehlsname(123)
```

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. Tipp: Verwende `penUp()` und `penDown()`.

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 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, der Turtle zu sagen, sie soll diese zwei Befehle 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 5 aber die Anweisung **`repeat`** und sagen der Turtle damit, sie soll die zwei eingerückten Befehle in Zeilen 6 und 7 neunmal wiederholen.

Der Befehl `right(90)` in Zeile 8 wird erst ausgeführt, wenn Python die Schleife neun Mal durchlaufen hat. Dass `right(90)` erst *nach* der Schleife ausgeführt wird erkennst du daran, dass es nicht eingerückt ist.

Für ein Neuneck muss sich die Turtle an jeder Ecke um $360°/9 = 40°$ drehen. Wir überlassen es aber dem Computer, diesen Wert selbst auszurechnen.

```
1 from gturtle import *
2 makeTurtle()
3
4 left(90)
5 repeat 9:
6     forward(50)
7     left(360 / 9)
8 right(90)
```

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

```
def neuneck(seite):
    repeat 9:
        forward(seite)
        left(360 / 9)
```

`repeat Anzahl` ist eine Anweisung, *aber kein* Befehl. *Das siehst du daran, dass* `repeat` *von Python automatisch fett und blau angezeigt wird. Für dich heisst das: Die Anzahl der Wiederholungen steht* ohne Klammern *da!*

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

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

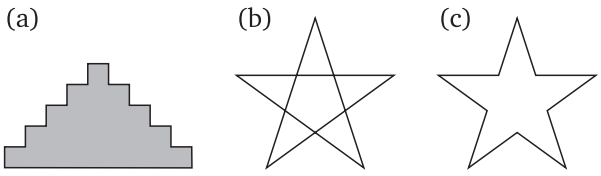


Abbildung 2.5: Treppen und Sterne

12. 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.

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

14.* Ein gewöhnliches n-Eck erhältst du immer dann, wenn sich die Turtle an jeder Ecke um den Winkel $\frac{360°}{n}$ dreht (insgesamt hat sie sich nach dem Zeichnen also um $360°$ gedreht). Für einen regelmässigen Stern muss sich die Turtle an jeder Ecke um einen doppelt so grossen Winkel drehen.

Schreibe damit einen Befehl `stern(n)`, der einen regelmässigen n-zackigen Stern zeichnet.

6 Modularität

Lernziele In diesem Abschnitt lernst du:

- Grössere Programme modular aufzubauen.

Einführung Um grössere Programme zu entwickeln ist es wichtig, dass du die Programme sinnvoll aus kleineren Stücken aufbaust. Du definierst also zunächst Befehle, die dir zum Beispiel ein Quadrat oder ein Dreieck zeichnen. Damit definierst du dann einen nächsten Befehl, der dir aus einem Quadrat und einem Dreieck ein Haus zeichnet.

Die Vorgehensweise, ein komplexes Programm aus kleineren Stücken zusammenzubauen heisst «modularer Entwurf». Versuche ab jetzt diesem Prinzip zu folgen und zunächst Befehle zu definieren, die eine klare Aufgabe erfüllen und einen verständlichen Namen haben.

Das Programm Das Programm zeichnet drei Häuser, die auf einer Wiese stehen. Weil wir den Befehlen sinnvolle Namen gegeben haben, siehst du für jeden Befehl sofort, was er macht.

```
1  from gturtle import *
2
3  def quadrat(seite):
4      repeat 4:
5          forward(seite)
6          right(90)
7
8  def quadrat_gefuellt(seite):
9      fillToPoint()
10     repeat 4:
11         forward(seite)
12         right(90)
13     fillOff()
14
15 def dreieck_gefuellt(seite):
16     right(30)
17     fillToPoint()
18     repeat 3:
19         forward(seite)
20         right(120)
21     fillOff()
22     left(30)
23
24 def hintergrund():
```

Mit `fillToPoint()` sagst du der Turtle, dass sie nicht nur eine Linie zeichnen, sondern eine ganze Fläche ausfüllen soll. Dabei dient die aktuelle Position der Turtle als «Ankerpunkt»: Von diesem Punkt aus werden die Flächen aufgespannt. Mit `fillOff()` schaltest du wieder auf das Zeichnen von Linien zurück. Du kannst die Wirkung im Programm gut beobachten, wenn die Turtle ein gefülltes Quadrat zeichnet.

```
25      clean("sky blue")
26      penUp()
27      back(20)
28      left(90)
29      back(200)
30      setPenColor("dark green")
31      setPenWidth(40)
32      penDown()
33      forward(400)
34      penUp()
35      right(90)
36      forward(20)
37      setPenWidth(1)
38
39  def haus():
40      penDown()
41      setPenColor("light yellow")
42      quadrat_gefuellt(100)
43      setPenColor("black")
44      quadrat(100)
45      forward(100)
46      left(90)
47      forward(5)
48      right(90)
49      setPenColor("dark red")
50      dreieck_gefuellt(110)
51      penUp()
52      back(100)
53
54  makeTurtle()
55  speed(-1)
56  hintergrund()
57  haus()
58  repeat 2:
59      right(90)
60      forward(150)
61      left(90)
62      haus()
63  back(50)
```

AUFGABEN

15. Experimentiere mit dem Programm aus dem Beispiel und halbiere z. B. die Grösse der einzelnen Häuser.

16. Schreibe ein modular aufgebautes Programm, das eine Szene mit drei einfachen Segelschiffen auf dem Meer zeichnet. Besonders gut ist es, wenn die Segelschiffe unterschiedlich gross sind.

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 $360° : 72 = 5°$ nach links dreht und wieder 6 Pixel vorwärts geht.

```
1 from gturtle import *
2
3 def kreis():
4     repeat 72:
5         forward(6)
6         left(360 / 72)
7
8 makeTurtle()
9 kreis()
```

In der Zeile 5 kannst du die 6 ändern, um einen grösseren oder kleineren Kreis zu erhalten.

Radius und Umfang Du weisst sicher, dass sich der Umfang u eines Kreises aus dem Radius r berechnen lässt: $u = 2\pi \cdot r$. Der Umfang ist demnach rund 6.2832 Mal so lang wie der Radius.

Bei einem Vieleck gilt das aber nicht! Das Verhältnis zwischen Radius und Umfang variiert je nach Anzahl der Ecken n. Die Tabelle rechts gibt für einige Beispiele das Verhältnis zwischen Umfang und Radius

n	u/r
6	6.0000
18	6.2513
36	6.2752
72	6.2812
360	6.2831

an. Selbst bei 360 Ecken stimmt der Wert erst auf vier Stellen mit 2π überein.

Es ist deshalb schwierig, einen Kreis mit einem vorgegebenen Mittelpunkt und Radius zu zeichnen. Wir werden das im nächsten Absatz noch einmal diskutieren.

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

17. Verwende `fillToPoint()` und `fillOff()` (siehe Seite 18), um einen ausgefüllten Kreis mit dem Umfang von 125 Pixeln zu zeichnen.

18. Definiere einen eigenen Befehl `circle(s)`, um einen Kreis zu zeichnen und einen Befehl `fillcircle(s)`, um einen ausgefüllten Kreis zu zeichnen. Der Kreis wird als 72-Eck gezeichnet und der Parameter s gibt dabei an, um wieviel die Turtle bei jedem Schritt vorwärts gehen soll.

19. Bei einem 36-Eck ist das Verhältnis zwischen Umfang und Radius 6.27521347783. Definiere damit einen Befehl `circle2(r)`, der einen Kreis mit dem angegebenen Radius zeichnet. Der Mittelpunkt des Kreises ist der Startpunkt der Turtle.

20. Lass deine Turtle den PacMan (a), das Yin-Yang-Symbol (b) und ein beliebiges Smiley (c) zeichnen wie in der Abbildung 2.6.

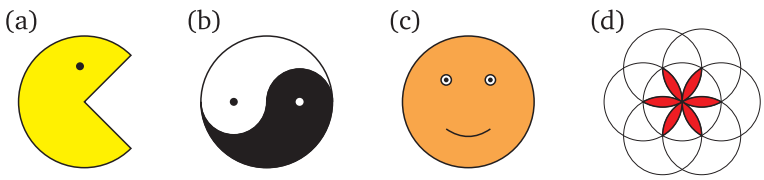


Abbildung 2.6: Kreisfiguren.

21.* 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.

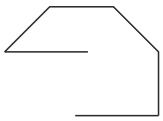
8 Kuchenstücke

Lernziele In diesem Abschnitt lernst du:

- Beim Zeichnen von Kreisen mit der Turtle Winkel zu korrigieren.
- Mit der Turtle exakte Kreissektoren und Kuchenstücke zu zeichnen.

Einführung Der Computer kann keine Kreise zeichnen. Aber ein Polygon (Vieleck) mit genügend Ecken sieht auf dem Bildschirm aus wie ein Kreis. Nur gibt es dabei ein Problem: Sobald du nur einen Teil des Kreisbogens zeichnest oder den Radius brauchst, wird das Bild ungenau und fehlerhaft (das hast du sicher selbst bei der Aufgabe (2.20) zum «PacMan» bemerkt): Der Mittelpunkt des Kreises ist am falschen Ort.

Das Programm (I) Um das Problem zu illustrieren, zeichnen wir mit der Turtle einen «Halbkreis», den wir mit einem Achteck annähern. Obwohl sich die Turtle dabei um $360°$ dreht ergibt sich kein geschlossener Halbkreis, sondern die Figur rechts. Warum? Wo liegt der Fehler?



```
from gturtle import *
makeTurtle()
right(90)
forward(52)
left(90)
repeat 4:
    forward(40)
    left(45)
left(90)
forward(52)
```

Das Programm (II) In diesem Programm verwenden wir den Befehl `circle2(r)` aus der Aufgabe 2.19. Tatsächlich sind die beiden gezeichneten Kreise gleich gross, haben aber nicht den selben Mittelpunkt.

```
from gturtle import *
makeTurtle()
setPenColor("yellow")
dot(300)
setPenColor("red")
circle2(150)
```

Winkel korrigieren Der Grund, warum sich oben keine geschlossene Figur ergibt liegt bei den Winkeln. Du siehst in der Abbildung 2.7, wo das Problem liegt: Bei einem Polygon steht der Radius nicht ganz senkrecht auf der Umfangslinie. Es gibt eine kleine Abweichung φ. Wie gross ist diese Abweichung φ? Was musst du im Programm (I) oben ändern, damit diese Abweichung korrekt berücksichtigt wird?

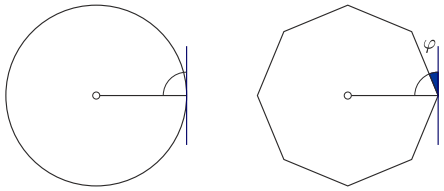


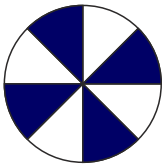
Abbildung 2.7: Beim Kreis steht der Radius senkrecht zur Kreislinie (links). Bei einem Polygon ist der Winkel zwischen Radius und Umfangslinie nicht ganz rechtwinklig (rechts).

Die Abweichung φ ist gerade der halbe Winkel, um den sich die Turtle bei jedem Schritt drehen muss. In unserem Fall also die Hälfte von $45°$ und damit ist $\varphi = 22.5°$. Wenn wir im Programm (I) die Zeilen 5 bis 9 entsprechend anpassen, dann ergibt sich auch ein viel besseres Bild.

```
left(90 + 22.5)
repeat 4:
    forward(40)
    left(45)
left(90 - 22.5)
```

Die wichtigsten Punkte Um einen Kreis mit festem Mittelpunkt oder einen Kreissektor zu zeichnen musst du einige Winkel korrigieren. Die Korrektur φ entspricht immer der Hälfte des Winkels, um den sich die Turtle in der Schleife dreht.

AUFGABEN



22. Korrigiere beim Befehl `circle2(r)` im Programm (II) oben die Winkel, so dass die beiden gezeichneten Kreise konzentrisch werden.

23. Zeichne einen acht-teiligen Kreis wie in der Abbildung nebenan. Verwende dazu ein 72-eck und färbe die Stücke abwechselnd mit zwei verschiedenen Farben ein.

24. Zeichne ein Kuchendiagramm mit drei Stücken, die $\frac{5}{12}$, $\frac{1}{3}$ bzw. $\frac{1}{4}$ des Kreises einnehmen. Verwende für jedes Stück eine andere Farbe und achte darauf, dass die einzelnen Kuchenstücke genau sind.

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 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 jeder Parameter einen eigenen Namen.

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

Befehl(123, 456, 789)
```

Beim Aufruf des Befehls haben die Argumente (die Werte der Parameter) immer die selbe Reihenfolge wie die Parameter.

AUFGABEN

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

26. Der Rhombus (Abb. 2.8(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 du die Seitenlänge und den Winkel angeben kannst, und der dann einen Rhombus zeichnet.

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

28. Definiere einen Befehl `gitter(breite, hoehe)`, der ein Gitter wie in Abbildung 2.8(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`).

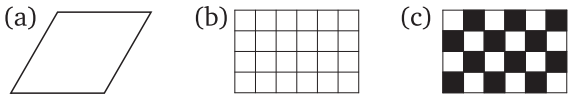


Abbildung 2.8: Rhombus, Gitter und Schachbrettmuster.

29.* Definiere einen Befehl `schachbrett(breite, hoehe)`, um ein Schachbrett wie in Abbildung 2.8(c) zu zeichnen. Gehe davon aus, dass `breite` und `hoehe` gerade Zahlen sind (wie auch in der Abbildung).

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:

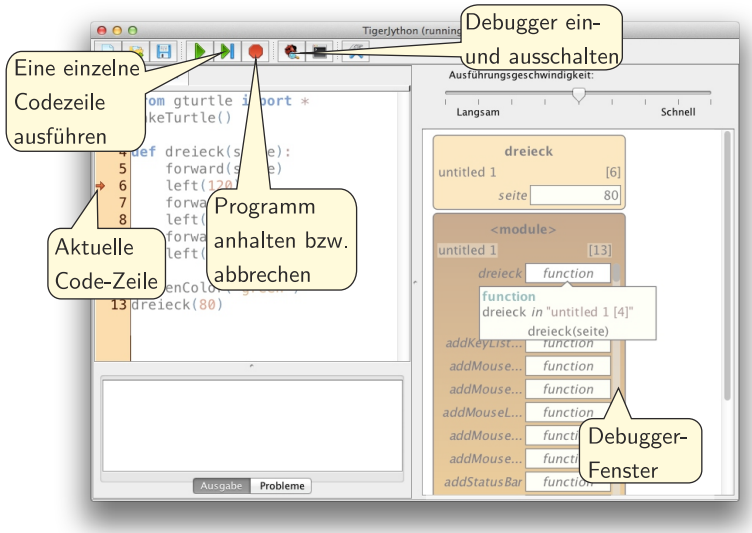


Abbildung 2.9: Der Debugger zeigt sich rechts im Fenster, sobald du auf den Käfer klickst. Solange das Debuggerfenster offen ist, kannst du den Programmablauf beobachten.

Das Programm Das Programm zeichnet ein Dreieck, so wie du es selbst schon gemacht hast. Tippe es ein und starte dann den Debugger (Abb. 2.9). 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

30. 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()
```

31. 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?

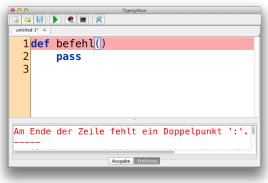
```
def quadrat(seite):
    repeat 4:
        forward(seite)
        left(90)
```

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. Die fehlerhafte Zeile wird rot markiert und unten siehst du den Hinweis, was falsch ist.

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 GTurtle import *
2  makeTurtle
3
4  def dreieck(seite)
5      repet 3:
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 gt bei gturtle muss klein sein
makeTurtle()              # Klammern vergessen

def dreieck(seite):       # Doppelpunkt vergessen
    repeat 3:             # Ein 'a' 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

32. 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.

33. 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.

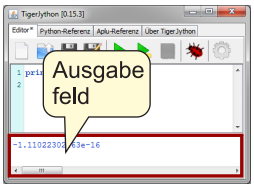
Computer arbeiten auch mit dieser wissenschaftlichen Schreibweise, konnten früher aber das $\cdot 10^{\square}$ nicht schreiben. Deshalb hat man sich vor langer Zeit auf die Darstellung mit einem `e` geeinigt: `5.98e+24` oder einfach `5.98e24` (`e` steht für «Exponent»).

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 für unbekannte Werte

Lernziele In diesem Abschnitt lernst du:

- Mit Variablen zu arbeiten, damit du die konkrten Werte für dein Programm erst zur Laufzeit eingeben musst.

Einführung Beim Programmieren kennst du nicht immer alle Werte schon im Voraus. Bei fast allen Programmen kannst du vielmehr zur Laufzeit (d. h. während das Programm läuft) einige Werte festlegen. Dafür brauchst du beim Programmieren *Variablen* für die noch unbekannten Werte. Das Konzept kennst du im Prinzip bereits von den Parametern her.

Das Programm Das Programm zeichnet ein Polygon, wie du es schon lange kennst. Wenn du das Programm startest, dann ist allerdings noch nicht festgelegt, wie viele Ecken und welche Grösse das Polygon haben soll. Die Werte der Variablen `ecken` und `umfang` werden also erst festgelegt, wenn das Programm bereits läuft.

Wichtig sind die beiden Funktionen `inputInt()` und `inputFloat()`. Bei diesen Funktionen erscheint auf dem Bildschirm ein kleines Fenster, in dem du eine Zahl eingeben kannst. Bei `inputInt` muss es eine ganze Zahl sein, bei `inputFloat` eine gebrochene Zahl.

```
1  from gturtle import *
2  makeTurtle()
3
4  ecken = inputInt("Anzahl Ecken?")
5  umfang = inputFloat("Umfang?")
6  winkel = 360 / ecken
7  seite = umfang / ecken
8  repeat ecken:
9      forward(seite)
10     left(winkel)
```

Die wichtigsten Punkte Variablen stehen in einem Programm für unbekannte oder veränderliche Werte. Häufig gibst du den Wert für eine Variable nicht beim Programmieren ein, sondern erst wenn das Programm schon läuft. Dazu verwendest du die Eingabefunktionen `inputInt` und `inputFloat`.

```
VARIABLE = inputInt("Text")
VARIABLE = inputFloat("Text")
```

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: $x \neq X$*!*

Natürlich kannst du den Wert einer Variablen auch berechnen oder direkt setzen:

```
VARIABLE = RECHNUNG
```

Achtung: Bei der Zuweisung muss die Variable immer *links* stehen!

AUFGABEN

3. Verwende die Turtle, um eine Gitter zu zeichnen mit $n \times n$ Feldern zu zeichnen. Den konkreten Wert für n kann man dabei über `inputInt` eingeben.

4. Schreibe ein Programm, bei dem du den Radius r über `inputFloat` eingeben kannst. Das Programm rechnet dann den Umfang $u = 2\pi \cdot r$ und die Fläche $A = \pi \cdot r^2$ des Kreises mit Radius r aus und schreibt diese Werte mit **`print`** ins Ausgabefeld.

5. Schreibe ein Programm, bei dem du zwei Zahlen eingeben kannst. Der Computer rechnet dann den Durchschnitt dieser beiden Zahlen aus und schreibt ihn mit **`print`** ins Ausgabefeld.

6. 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 so, dass man die Temperatur in Fahrenheit eingeben kann und der Computer mit **`print`** den entsprechenden Wert in ° Celsius ausgibt.

Wenn dein Programm richtig arbeitet, dann gilt: $86°\,\mathrm{F} = 30°\,\mathrm{C}$. Teste damit dein Programm.

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

8.* Schreibe ein Programm, bei dem man eine Prozentzahl eingeben kann. Die Turtle zeichnet dann ein «Kuchenstück», das der eingegebenen Prozentzahl entspricht. Bei 100 % wird der ganze Kreis eingefärbt, bei 45 % nicht ganz die Hälfte.

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.[1]

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
```

[1] 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

9. 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.

10. 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. $480 \to 2 \cdot 200 + 4 \cdot 20$.

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

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

13.* 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 Für ein gutes Programm ist eine einfache und verständliche Ausgabe zentral. Das Programm gibt nicht nur einfach die Zahlenwerte aus, sondern schreibt auch hin, was sie bedeuten. Damit der Computer zwischen der Variablen `minuten` und dem Wort «Minuten» unterscheiden kann, musst du *alles, was keine Variable ist* in Gänsefüsschen setzen. Probiere es selbst aus:

```
minuten = 10
print minuten, "Minuten"
```

Der Text in den Gänsefüsschen darf auch länger sein, Leerschläge und Umlaute (äöü) enthalten.

*Text, der in Gänsefüsschen eingeschlossen ist heisst in der Fachsprache «**string**» oder «Zeichenkette».*

```
1 print "Wenn einer, der mit Mühe kaum,"
2 print "Gekrochen ist auf einen Baum,"
3 print "Schon meint, dass er ein",
4 print "Vogel wär,\nSo irrt sich der."
5 print "\tWilhelm Busch"
```

In diesem kurzen Gedicht von Wilhelm Busch haben wir auch gleich noch zwei Sonderzeichen verwendet. In einem Text zwischen den Gänsefüsschen steht die Kombination «`\n`» für einen Zeilenumbruch (new line) und «`\t`» für den Tabulator (Einrückung). Dafür verhindet ein Komma am Ende der Zeile bei **`print`** den Zeilenumbruch (Zeile 3).

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.

*Wenn du ganz am Ende der Zeile bei **`print`** noch ein Komma setzst, dann wird die Zeile nicht umgebrochen.*

```
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.

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

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:

```
print "Textstück"
```

Im Gegensatz zu Variablen darf ein solches Textstück auch Leerschläge und Umlaute (äöü) enthalten. Wenn du einmal Gänsefüsschen selber brauchst, dann verwende dafür die Kombination `\"`.

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

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

AUFGABEN

14. 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`

15. 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`

16. 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.`

17.* *Ascii-Art* ist die Kunst, nur mit den Buchstaben und Zeichen des Computers Bilder darzustellen[2]. Verwende **`print`**, um solche Ascii-Art-Bilder zu «zeichnen», z. B. die Eule:

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

[2] 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 du 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

18. 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?

19. 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.

20. 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.

21.* 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

22. 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.

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

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

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

26. 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?

27.* 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 Wiederholung mit Variation

Lernziele In diesem Abschnitt lernst du:

- Das Verhalten der Turtle in einer Schleife zu variieren.

Einführung Eine Schleife, in der du den Wert einer Variable änderst ist ein sehr wichtiges und mächtiges Werkzeug. In diesem Abschnitt verwendest du das Konzept, um das Verhalten der Turtle bei jedem Durchgang einer Schleife leicht zu variieren. Du kannst zum Beispiel bei jedem Durchgang der Schleife die Farbe oder die Länge der Strecke ändern.

Das Programm In diesem Programm zeichnet die Turtle eine Kette von 10 farbigen Punkten. Dazu verwenden wir die Funktion `makeColor` zum Erzeugen der Farben. Der erste Parameter `"rainbow"` gibt an, dass wir eine Regenbogenfarben erzeugen möchten. Als zweiten Parameter gibst du dann einen Wert zwischen `0.0` und `1.0` an, der die Farbe im Spektrum angibt, z. B. `makeColor("rainbow", 0.69)` für «gelb».

```
1  from gturtle import *
2  makeTurtle()
3
4  penUp()
5  i = 0.05
6  repeat 10:
7      farbe = makeColor("rainbow", i)
8      setPenColor(farbe)
9      dot(20)
10     forward(20)
11     i += 0.1
12 hideTurtle()
```

Beim Befehl `dot(durchmesser)` zeichnet die Turtle übrigens einen Punkt mit dem angegebenen Durchmesser an der aktuellen Position. Auch dann, wenn der Stift gerade «oben» ist wie in diesem Beispiel.

Die wichtigsten Punkte In sehr vielen Problemstellungen musst du in einer Schleife Werte durchzählen. Das machst du immer mit dem Schema:

```
VARIABLE = STARTWERT
repeat ANZAHL:
    CODE
    VARIABLE += AENDERUNG
```

Je nach Situation verwendest du anstelle von `+=` auch z. B. `-=` oder `*=`.

AUFGABEN

28. Ändere das Programm so ab, dass die Turtle 20 Punkte horizontal (waagrecht) zeichnet. Die Variable `i` soll also nacheinander 20 verschiedene Werte zwischen `0.0` und `1.0` annehmen.

29. (a) Ersetze in Zeile 7 im Programm das `"rainbow"` durch ein `"gray"` und beobachte, was dabei passiert. Welche Werte muss die Variable `i` haben, damit die Farbe «schwarz» bzw. «weiss» entsteht?

(b) Ersetze die Zeile 5 durch `i = 0` und Zeile 11 durch `i += 1`. Erzeuge dann die Farbe in Zeile 7 mit:

```
farbe = makeColor("gray", i % 2)
```

Was bewirkt der Ausdruck `i % 2` in den einzelnen Schleifendurchgängen? Was für ein Muster zeichnet die Turtle hier?

30. Lass die Turtle eine Kette von immer kleiner werdenden Punkten zeichnen, die sich alle berühren.

31. Zeichne mit der Turtle eine «sechseckige» Spirale. Dazu zeichnest du ein Sechseck und vergrösserst bei jedem Durchgang der Schleife die Länge der Seite um 2 Pixel (Abbildung 3.1(a)).

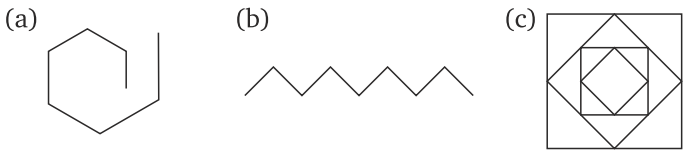


Abbildung 3.1: Verschiedene Turtle-Figuren.

32.* Lass die Turtle eine Zick-Zack-Linie bzw. Treppe zeichnen. Dazu dreht sich die Turtle in jedem Schleifendurchgang entweder nach links oder rechts. Hinweis: Die Turtle dreht sich auch dann nach rechts, wenn der Winkel in `left` negativ ist (Abbildung 3.1(b)).

33.* Eine kleine Herausforderung: Lass die Turtle die verschachtelten Quadrate in der Abbildung 3.1(c) zeichnen.

8 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

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

35. 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.

36. 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.

9 Alternativen

Lernziele In diesem Abschnitt lernst du:

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

Einführung Mit `if` kontrollierst du, 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. 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.

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

AUFGABEN

37. Bei der Collatz-Vermutung startest du mit einer Zahl x und erzeugst dann eine Zahlenfolge noch dem folgenden Prinzip: Wenn x gerade ist, dann teile durch 2. Andernfalls multipliziere x mit 3 und zähle 1 dazu:

$$17 \quad 52 \quad 26 \quad 13 \quad 40 \quad 20 \quad 10 \quad 5 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1$$

Schreibe ein Programm, das zu einer beliebigen Startzahl 10 Folgezahlen nach diesem Prinzip berechnet und ausgibt.

38.* 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 Ein- und Ausgaben

Lernziele In diesem Abschnitt lernst du:

- ▷ Mit beliebigen Texteingaben zu arbeiten.
- ▷ Resultate und Mitteilungen in einem Fensterchen auszugeben.

Einführung Du kennst bereits `inputInt` und `inputFloat`, um Zahlen einzugeben und kannst mit **`print`** Resultate auf den Bildschirm ausgeben. Neu kommen `inputString` und `msgDlg` hinzu, um auch Text eingeben und mit einem Fensterchen ausgeben zu können.

Magst du dich erinnern, dass ein Text (in Gänsefüsschen) im Programmieren «string» heisst? Ist es da nicht logisch, dass die Eingabefunktion für solchen Text `inputString()` genannt wird? Während du also bei `inputInt` und `inputFloat` nur Zahlen eingeben kannst, kannst du bei `inputString` jeden möglichen Text eingeben, z. B. deinen Namen:

```
name = inputString("Wie heisst du?")
print "Hallo", name
```

Wäre dieses Programm aber nicht noch besser, wenn auch die Ausgabe in einem kleinen Fensterchen erfolgen würde? Dafür gibt es `msgDlg()`. Es funktioniert (fast) genau gleich wie **`print`**, allerdings mit einem wichtigen Unterschied: Bei **`print`** setzst du *nie* Klammern, bei `msgDlg` dafür *immer*.

```
name = inputString("Wie heisst du?")
msgDlg("Hallo", name)
```

Das Programm Mit den neuen Ein- und Ausgaben können wir ein kleines Quiz schreiben, das auch gleich zählt, wie viele richtige und falsche Antworten gegeben wurden. Beachte, wie wir in Zeile 11 auch die Zahl 169 in Gänsefüsschen setzen müssen. `inputString` gibt uns nämlich auch dann einen String/Text zurück, wenn die Eingabe eigentlich eine Zahl wäre.

```
1 richtig = 0    # Anzahl richtige Antworten
2 falsch = 0     # Anzahl falsche Antworten
3
4 eingabe1 = inputString("Hauptstadt von Frankreich?")
5 if eingabe1 == "Paris":
6     richtig += 1
7 else:
8     falsch += 1
```

```
9
10 eingabe2 = inputString("Quadrat von 13?")
11 if eingabe2 == "169":
12     richtig += 1
13 else:
14     falsch += 1
15
16 msgDlg("Du hattest", richtig, "richtige und",
17         falsch, "falsche Antworten.")
```

Die wichtigsten Punkte Je nachdem, ob du als Eingabe eine ganze Zahl, eine beliebige Zahl oder einen String/Text erwartest, verwendest du eine andere Funktion: `inputInt()`, `inputFloat()`, `inputString()`.

Mit `msgDlg()` kannst du beliebigen Text, Variablen und Zahlen in einem eigenen kleinen Fensterchen ausgeben. Auch hier kannst du mit der Kombination `\n` in einem String einen Zeilenumbruch erzeugen.

AUFGABEN

39. Ändere die Zeile 10 im Beispielprogramm oben ab und verwende `inputInt` anstelle von `inputString`. Was musst du dann noch ändern, damit das Programm korrekt funktioniert?

40. Wenn jemand eine falsche Antwort gibt, dann lass dein Programm die korrekte Antwort ausgeben: `Falsch. Richtig wäre «Paris».`

41. Erweitere dein Quiz um eigene Fragen. In den Einstellungen von TigerJython kannst du auch das Editorfenster während der Programmausführung ausblenden lassen. Dann sieht man während des Quizes nicht gleich die Antworten im Code.

42. Schreibe das Programm aus dem Abschnitt über Fallunterscheidung (Seite 46) so um, dass du über `inputInt` eine Zahl eingeben kannst, die dann daraufhin überprüft wird, ob sie eine Quadratzahl ist.

43.* 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: Berechne aus dem Datum zuerst den Tag im Jahr. Der «3. Februar» wäre demnach der 34. Tag.

11 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

44. 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.

45. 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?

46. 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.

12 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

47. 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); $2x^2 - 3x - 5 = 0$ $(-1, 2.5)$; $2x^2 + 3x + 5 = 0$ (–); $x^2 + 5x + 6.5 = 0$ (–).

48. 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

LÖSUNGEN

Vorbemerkung Die Programme in den Lösungen hier zeigen jeweils *einen möglichen* Weg auf. Es gibt aber bei fast allen Aufgaben eine ganze Reihe von Programmen, die die Aufgabe korrekt lösen.

Lösungen zu den Quizfragen

1. d
2. b, c
3. b
4. a
5. c
6. d
7. b
8. a

13 Kapiteltest 2

7.
```
def handshake(n):
    h = n * (n-1) // 2
    return h
```

8.
```
from gturtle import *
from math import *

def myCircle(mx, my, r):
    # Die Kreisfunktion:
    def f(x):
        return sqrt(r**2 - x**2)

    x = -r
    setPos(mx-r, my)
    # Oberen Kreisbogen zeichnen:
    repeat 2*r:
        x += 1
        y = f(x)
        moveTo(mx + x, my + y)
    # Unteren Kreisbogen zeichnen:
    repeat 2*r:
        x -= 1
        y = -f(x)
        moveTo(mx + x, my + y)
```

9.
```
from gturtle import *

def f(x):
    return -x**2/200

makeTurtle()
setPenColor("black")
# Das gerade Stück zeichnen:
setPos(-125, 10)
moveTo(125, 10)
# Den Bogen zeichnen:
x = -120
setPos(x, f(x))
repeat 241:
    moveTo(x, f(x))
    x += 1
# Die vertikalen Verstrebungen zeichnen:
x = -120
repeat 13:
    setPos(x, 10)
    moveTo(x, f(x))
    x += 20
```

10.
```
def cbrt(a):
    x = a / 3
    repeat 50:
        x = (x + a / (x**2)) / 2
    return x
```

11. (a)
```
def intDiv(a, b):
    result = 0
    repeat:
        if a >= b:
            a -= b
            result += 1
        else:
            break
    return result
```

(b)
```
def intRest(a, b):
    repeat:
        if a >= b:
            a -= b
        else:
            break
    return a
```

(c)
```
def numDiv(a, b):
    result = 0
    x = 1
    repeat:
        if a >= b:
            a -= b
            result += x
        else:
            b /= 10
            x /= 10
            if x < 0.0000001:
                break
    return result
```

12.
```
zaehler = 0

def nextInt():
    global zaehler
    zaehler += 1
    return zaehler
```

INDEX