Python

Eine Einführung in die Computer-Programmierung für Fortgeschrittene

Tobias Kohn

http://jython.tobiaskohn.ch/
Version vom 14. Mai 2017

INHALTSVERZEICHNIS

Kapitel Eins

TURTLEGRAFIK

Das Prinzip der Turtlegrafik ist relativ einfach: Du steuerst eine kleine Figur (die «Turtle») auf dem Bildschirm, indem du ihr Befehle wie `left`, `right` oder `forward` gibst und zeichnest damit Bilder. Auch wenn die Turtlegrafik zunächst sehr einfach erscheint, lassen sich damit doch auch komplexere Grafiken erstellen (ursprünglich bis hin zu Simulationen der allgemeinen Relativitätstheorie).

In diesem Kapitel lernst du die Turtle kennen und steuern. Wir gehen davon aus, dass du bereits Programmiererfahrung mitbringst und einige Grundkenntnisse in Python hast. Die Begriffe «Variable», «Funktion», «Parameter» und «Schleife» sind dir also vertraut und du kannst damit umgehen.

1 Den PacMan zeichnen

Lernziele In diesem Abschnitt lernst du:

- ▷ Mit der Turtle Bilder zu zeichnen.
- ▷ Die PacMan-Figur korrekt zu zeichnen.

Einführung «PacMan» ist ein beliebtes Spiel aus den 1980er Jahren und du kennst bestimmt die gelbe PacMan-Figur. In diesem ersten Abschnitt geht es darum, diese Figur korrekt zu zeichnen und dabei das Zeichnen mit der Turtle zu repetieren.

Beim Programmieren ist es wichtig, dass du deine Programme in sinnvolle Einheiten aufteilst und gute Namen vergibst. In unserem Beispiel schreiben wir eine Funktion `draw_pacman`, die mit dem Namen bereits sagt, was sie tut. Auch den Parameter `radius` haben wir so benannt, dass sofort klar ist, was er angibt (und nicht etwa `x` oder `r`, wie man das so oft sieht).

Das Programm Das Programm zeichnet einen «PacMan» mit einer $90°$-Öffnung rechts. In den ersten zwei Zeilen laden wir zunächst das vollständige Turtle-Modul und aus dem Mathematik-Modul die Konstante π. Das `makeTurtle()` in der dritten Zeile erstellt das Turtlefenster und mit `speed(-1)` setzen wir die Turtle auf volle Geschwindigkeit.

In den Zeilen 6 bis 15 definieren wir einen neuen Befehl `draw_pacman`, der zu einem vorgegebenen Radius den PacMan zeichnet. `heading(45)` richtet die Turtle auf einen $45°$-Winkel absolut zum Fenster aus. Im Gegensatz dazu dreht sich die Turtle mit `left()` und `right()` relativ zu ihrer aktuellen Ausrichtung.

Den Kreis müssen wir bei der Turtle als Polygon annähern. In diesem Fall verwenden wir ein 36-eck. Entsprechend wird der Umfang in Zeile 10 in 36 Stücke geteilt und nach jedem Stück dreht sich die Turtle um $10°$.

```
1 from gturtle import *
2 from math import pi
3 makeTurtle()
4 speed(-1)
5
6 def draw_pacman(radius):
7     heading(45)
8     forward(radius)
```

```
9       left(90 + 5)
10      s = 2 * radius * pi / 36
11      repeat 27:
12          forward(s)
13          left(10)
14      left(90 - 5)
15      forward(radius)
16
17  setPenColor("red")
18  draw_pacman(100)
19  hideTurtle()
```

Beachte in den Zeilen 9 und 14, wie wir den Winkel um jeweils $5°$ korrigieren, anstatt die Turtle einfach um $90°$ zu drehen. Diese Korrektur ist nötig, weil die Turtle eben keinen Kreis, sondern ein Polygon zeichnet; dann steht der Radius nicht mehr zwingend rechtwinklig zum Umfang wie beim Kreis. Die $5°$ entsprechen auch gerade der Hälfte der $10°$, um die sich die Turtle jeweils dreht.

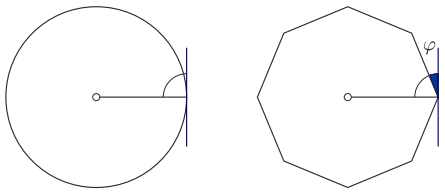


Abbildung 1.1: Beim Kreis steht der Radius senkrecht zur Kreislinie (links). Bei einem Polygon ist der Winkel zwischen Radius und Umfangslinie nicht ganz rechtwinklig (rechts).

Die wichtigsten Punkte Die Turtle lässt sich mit `left`, `right` und `forward` steuern, um Bilder zu zeichnen. Mit der **`repeat`**-Schleife kannst du zudem einen Block eine bestimmte Anzahl Male wiederholen und mit **`def`** definierst du eine neue Funktion bzw. Prozedur.

AUFGABEN

1. Ändere das Programm oben so ab, dass die Turtle anstelle des 36-Ecks ein 72-Eck als «Kreis» verwendet.

2. Ändere das Programm so ab, dass die Mundöffnung des PacMan nach links bzw. nach unten zeigt.

3. Ergänze die Funktion `draw_pacman` mit einem zweiten Parameter `direction`, der angibt, in welche Richtung der Mund zeigen soll.

2 Den PacMan animieren

Lernziele In diesem Abschnitt lernst du:

- Einfache Animationen zu programmieren.
- Variablenwerte zu ändern.

Einführung Animationen basieren darauf, dass du ein Bild in leicht veränderter Form mehrmals neu zeichnest. Gute Ergebnisse erreichst du bereits mit etwa 25 Bildern pro Sekunde. Wenn die Animation zu schnell abläuft, dann sehen wir sie gar nicht mehr, deshalb brauchen wir `sleep(0.1)`, um den Computer zwischen den Bildern eine Zehntelsekunde warten zu lassen.

Wichtiger noch ist aber die Idee, dass du den Wert einer Variable in der Schleife jedes Mal leicht änderst. Um zum Beispiel den Wert der Variable x bei jedem Schleifen-Durchgang (Iteration) um 1 zu erhöhen, verwendest du das folgende Programmmuster:

```
x = 1
repeat 20:
    print x**2
    x += 1
```

Dieses kleine Prgrämmchen gibt dir zum Beispiel die Quadratzahlen von 1^2 bis 20^2 aus.

Das Programm Die meisten Teile des Programms kennst du bereits aus dem letzten Abschnitt. Neu kommt hinzu, dass die Funktion `draw_pacman` jetzt einen zusätzlichen Parameter `mouth` für die Mundöffnung hat. Damit können wir steuern, wie weit der Mund des PacMan geöffnet sein soll (sinnvolle Werte gehen von 1 bis 9).

Wenn wir die Turtle mit `hideTurtle()` gleich von Beginn ausschalten, dann gibt es auch keine Animation beim Zeichnen mehr und die Bilder entstehen genug schnell für unsere Animationen.

In den Zeilen 18 bis 23 verwenden wir jetzt eine Schleife, um die PacMan-Figur achtmal zu zeichnen, jedes Mal mit einem stärker geschlossenen Mund. Dazu verwenden wir eine Variable `mouth_opening`, die wir zunächst auf den Wert 9 setzen (Zeile 18, geöffneter Mund) und dann bei jeder Wiederholung um 1 kleiner werden lassen (Zeile 22). Mit `clear()` wird übrigens das ganze Turtlefenster gelöscht.

```
1  from gturtle import *
2  from math import pi
3  from time import sleep
4  makeTurtle()
5  hideTurtle()
```

```
6
7  def draw_pacman(radius, mouth):
8      heading(90 - mouth * 5)
9      forward(radius)
10     left(90 + 5)
11     s = 2 * radius * pi / 36
12     repeat 36 - mouth:
13         forward(s)
14         left(10)
15     left(90 - 5)
16     forward(radius)
17
18 mouth_opening = 9
19 repeat 8:
20     clear()
21     draw_pacman(100, mouth_opening)
22     mouth_opening -= 1
23     sleep(0.1)
```

Im Moment flackert unsere Animation noch ein wenig. Das werden wir später dann noch beseitigen.

AUFGABEN

4. Der PacMan soll den Mund auch wieder öffnen. Ergänze dazu das Programm oben mit einer zweiten Schleife ab Zeile 24, so dass der Mund sich jedes Mal wieder ein Stück öffnet.

5. Schreibe das Programm wiederum so um, dass `draw_pacman` ein 72-Eck als Grundlage verwendet. Die Angabe für die Mundöffnung reicht dann natürlich von 1 bis 18.

6.* Die Funktion `abs(x)` gibt den Absolutbetrag $|x|$ einer Zahl x zurück. Zum Beispiel ist $|-7| = 7$ und $|9| = 9$. Verwende diese Funktion, um das schliessen und öffnen des Mundes in eine einzige Schleife zu verpacken.

7.* Der PacMan soll seinen Mund drei Mal schliessen und öffnen. Das lässt sich erreichen, indem du nur eine einzige Zeile mit **`repeat`** hinzufügst und das Programm dann minimal anpasst.

3 Von Farben und Flächen

Lernziele In diesem Abschnitt lernst du:

- Farben zu mischen.
- Den Zeichenstift der Turtle zu steuern.
- Flächen zu füllen.

Einführung Bildschirme stellen die verschiedenen Farben aus drei Grundfarben zusammen: Rot, Grün und Blau. Dieses Farbschema heisst entsprechend «RGB». So entspricht Gelb einer Mischung aus 100 % Rot, 100 % Grün und 0 % Blau. In der Turtlegrafik codieren wir das als $(1.0, 1.0, 0.0)$. Hier sind einige Beispiele:

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	$(1.0, 1.0, 1.0)$

Das Programm (I) Das Programm zeichnet einen Farbverlauf von Rot bis Gelb. Wir verwenden dazu die Funktion `makeColor(red, green, blue)` (Zeile 4), die aus den Farbanteilen für Rot, Grün und Blau eine Farbe mischt. Jeder Anteil muss im Bereich von 0.0 bis 1.0 liegen.

In den Zeilen 15 bis 21 findest du ein vertrautes Programmmuster: Wie erhöhen den Wert der Variable `green` in jedem Durchgang um einen Hundertstel, so dass die Werte von 0.0 bis schliesslich 1.0 reichen.

```
1  from gturtle import *
2
3  def draw_line(length, color_value):
4      color = makeColor(1.0, color_value, 0.0)
5      setPenColor(color)
6      right(90)
7      forward(length)
8      right(180)
9      forward(length)
10     right(90)
11
12 makeTurtle()
13 clear("black")
14 setPenWidth(2)
```

```
15 green = 0
16 repeat 100:
17     draw_line(100, green)
18     penUp()
19     forward(2)
20     penDown()
21     green += 1 / 100
```

Übrigens: Die Funktion `clear()` löscht nicht nur den Fensterinhalt und füllt alles mit der gewünschten Farbe, sondern beinhaltet auch ein implizites `hideTurtle()`.

Das Programm (II) Dieses zweite Programm ist dir im Prinzip bereits vertraut. Es ist die bekannte PacMan-Figur, die hier gezeichnet wird. Dieses Mal füllen wir die Figur allerdings mit `fillToPoint()` aus. Die Funktion `fillToPoint()` merkt sich den aktuellen Punkt der Turtle und verbindet alle weiteren Punkte der Turtle mit diesem ersten Punkt, so dass Flächen entstehen. Mit `fillOff()` wird dieses Verhalten wieder abgeschaltet.

```
1  from gturtle import *
2  makeTurtle()
3  fillToPoint()
4  right(45)
5  forward(100)
6  left(90 + 5)
7  repeat 27:
8      forward(100 * 2 * 3.1415 / 36)
9      left(10)
10 fillOff()
```

AUFGABEN

8. Ändere das erste Programm so ab, dass die Farbenwerte von blau bis grün reichen.

9. Die ausgefüllte Fläche soll kein Rechteck sein, sondern ein Dreieck. Schreibe das Programm also so um, dass die Fläche gegen oben schmaler wird bzw. spitz zuläuft. Verwende dazu eine zweite Variable `width` für die aktuelle Breite.

10. Lass die Farben von Rot nach Gelb und wieder zurück nach Rot gehen. Tipp: Verwende wiederum die `abs()`-Funktion für den Absolutbetrag einer Zahl (p. 9).

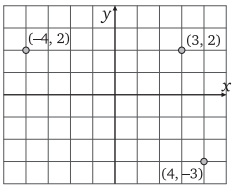
11. Zeichne den Farbverlauf als ausgefüllte Kreisscheibe, indem du die Techniken aus den beiden Beispielprogrammen kombinierst.

4 Der Ball in der Box

Lernziele In diesem Abschnitt lernst du:

- Die Turtle an vorgegebene Koordinaten zu setzen.
- Animationen ohne Flackern zu zeichnen.

Einführung Das Fenster der Turtle ist mit einem Koordinatensystem ausgestattet, dessen Nullpunkt $(0, 0)$ in der Fenstermitte liegt. Mit `setPos(x, y)` kannst du die Turtle in diesem Koordinatensystems beliebig positionieren. Wenn die Turtle eine Linie zur neuen Position zeichnen soll, verwendest du `moveTo(x, y)`.



Das Programm In diesem Programm verwenden wir ausschliesslich `setPos` und `moveTo`, um die Turtle zu positionieren und Linien zu zeichnen. Grundsätzlich lassen sich diese Befehle aber auch problemlos mit `forward`, `left` und `right` kombinieren.

Um die Richtung umzukehren müssen wir beim «Aufprall» des Balls das Vorzeichen der Geschwindigkeit ändern. Dazu multiplizieren wir v_x mit -1. In Python schreiben wir das hier als `v_x *= -1`. Genauso gut könnten wir aber auch `v_x = v_x * -1` oder `v_x = - v_x` verwenden.

Bei Animationen kommt das Flackern daher, dass wir zuerst den Bildschirm löschen (Zeile 15) und danach wieder etwas zeichnen. Indem wir mit `enableRepaint()` in Zeile 5 das automatische Neuzeichnen abschalten, können wir das Bild gezielt dann mit `repaint()` aktualisieren, wenn die neue Zeichnung fertig ist (Zeile 19).

```
1  from gturtle import *
2  from time import sleep
3  makeTurtle()
4  setPenColor("yellow")
5  enableRepaint(False)
6
7  def rectangle(x1, y1, x2, y2):
8      setPos(x1, y1)
9      moveTo(x2, y1)
10     moveTo(x2, y2)
11     moveTo(x1, y2)
12     moveTo(x1, y1)
13
14 def paint_scene(ball_x):
15     clear("black")
```

```
16      rectangle(-310, -210, 310, 210)
17      setPos(ball_x, 0)
18      dot(20)
19      repaint()
20
21  x = -200    # Anfangsposition in x-Richtung
22  v_x = 4     # Geschwindigkeit in x-Richtung
23  repeat:
24      # Bei Tastendruck wird die Schleife beendet:
25      if getKeyCode() != 0:
26          break
27      # Den Ball "bewegen" und ggf. abprallen lassen
28      x += v_x
29      if (x < -300) or (x > 300):
30          v_x *= -1
31      paint_scene(x)
32      sleep(0.005)
33  # Das Fenster schliessen
34  dispose()
```

Die wichtigsten Punkte Mit `setPos(x, y)` setzst du die Turtle direkt an die Position (x, y), mit `moveTo(x, y)` lässt du die Turtle von der aktuellen Position aus eine Linie bis zum Punkt (x, y) zeichnen.

AUFGABEN

12. Baue das Programm so aus, dass der Ball auch eine Geschwindigkeit in y-Richtung hat. Achte darauf, dass der Ball dann auch von der Boden bzw. Deckplatte abprallt.

Hinweis: Die Bewegungen des Balls in x- und y-Richtung sind unabhängig voneinander.

13. Ändere das Programm so ab, dass der Rahmen weiss ist und der Ball seine Farbe ändert: Wenn er nach rechts fliegt soll er rot sein, auf dem Weg nach links hingegen gelb.

14. Lass den Ball bei jedem Aufprall einen Teil seiner «Energie» verlieren. Dazu multiplizierst du seine Geschwindigkeit beim Aufprall mit dem Faktor 0.95.

15.* Schreibe ein Programm, bei dem ein Schiffchen auf dem Meer von links nach rechts über den Bildschirm fährt.

5 Random Walk

Lernziele In diesem Abschnitt lernst du:

- Wie du Zufallszahlen erzeugst und anwendest.
- Die aktuelle Position der Turtle zu ermitteln.

Einführung Der Zufall spielt in der Informatik eine ausserordentlich wichtige Rolle. In Python können wir uns mit der Funktion `randint` eine Pseudo-Zufallszahl erzeugen lassen. «Echte» Zufallszahlen sind das nicht, dafür sind sie so konstruiert, dass die Zahlen wirklich schön gleichmässig vorkommen. Im Prinzip handelt es sich um eine sehr lange (aber festgelegte) Zahlenfolge. Eigentlich zufällig ist nur die Stelle, an der das Programm in dieser Zahlenfolge einsteigt.

Das Programm Die Turtle bewegt sich in diesem Programm nach einem zufälligen Muster (sog. «Random Walk»). Sie geht immer 20 Pixel vorwärts, bevor sie sich dank `randint(0, 360)` für eine beliebige neue Richtung entscheidet (Zeilen 14 und 15).

Bei dieser Simulation möchten wir wissen, wie lange es dauert, bis die Turtle aus ihrer «Arena» hinausläuft. Entsprechend zählen wir mit der Variable `counter` die Anzahl der Schritte. In den Zeilen 19 und 20 ermitteln wie jeweils die aktuelle Position der Turtle und rechnen dann mit dem Satz des Pythagoras in der Zeile 21 die Distanz der Turtle zum Zentrum aus. Sobald die Distanz grösser als 200 ist, ist die Turtle ausserhalb der «Arena» und wir beenden das Programm.

```
1  from gturtle import *
2  from random import randint, seed
3  from math import sqrt
4  from time import sleep
5  makeTurtle()
6  hideTurtle()
7  clear("black")
8  setPenColor("white")
9  dot(400)
10
11 counter = 0
12 setPenColor("red")
13 repeat 500:
14     angle = randint(0, 360)
15     left(angle)
16     forward(20)
```

```
17      dot(5)
18      counter += 1
19      x = getX()
20      y = getY()
21      dist = sqrt(x**2 + y**2)
22      if dist > 200:
23          dot(10)
24          print counter
25          break
26      sleep(0.1)
```

AUFGABEN

16. Ändere das Programm so ab, dass die Turtle die Arena nicht verlassen kann. Wenn sie aus der Arena hinausläuft, so soll wieder genau auf das Zentrum ausgerichtet werden. Das erreichst du mit der Anweisung `heading(towards(0, 0))`.

Lass dein Programm zählen, wie oft die Turtle die Arena verlassen wollte.

17. Schreibe das Programm so um, dass das Zentrum nicht mehr im Ursprung liegt, sondern im Punkt $(100, 20)$.

18. Füge vor der Schleife die Anweisung `seed("Zufall")` ein und führe das Programm einige Male aus. Welchen Effekt hat diese Anweisung auf der Verhalten der Turtle?

19.* Codiere die Entfernung der Turtle vom Zentrum mit Farben. Wenn die Turtle genau im Zentrum ist, sind die Punkte blau. Wenn sie weiter raus läuft, wechselt die Farbe schliesslich zu rot (auf dem Rand der «Arena»). Punkte ausserhalb der Arena sollen alle gelb sein.

20.* Ändere die Geschwindigkeit der Turtle in Abhängigkeit vom Winkel. Und zwar soll sie grundsätzlich 2 Pixel plus ein Zehntel des Winkels vorwärts gehen. Wenn der Winkel $0°$ ist (die Turtle ändert also ihre Richtung nicht), so geht sie nur 2 Pixel vorwärts. Bei einem Winkel von $180°$ geht sie 20 Pixel zurück $(2 + 18)$.

Allerdings soll auch ein Winkel von $270°$ als $90°$ zählen. Die Turtle unterscheidet also nicht zwischen links und rechts.

6 Die Turtle steuern

Lernziele In diesem Abschnitt lernst du:

- Die Turtle mit der Tastatur zu steuern.
- Die Farbe von Pixeln auszulesen.

Einführung Animationen werden dann interessant, wenn du die Figuren darin direkt steuern kannst. In diesem Abschnitt lernst du, die Turtle mit der Tastatur zu steuern. Das «Spiel» ist sehr einfach gehalten, aber lässt sich dafür beliebig ausbauen.

Das Programm Das Programm ist ein einfaches Spiel. Die Turtle hat ein weisses Spielfeld, das wir zuerst mit einer dicken Linie erzeugen (Zeilen 11 bis 13). Wenn die Turtle mit `getPixelColorStr()` feststellt, dass unter ihr kein weisser Pixel mehr ist, dann bricht das Spiel ab (Zeilen 26 und 27).

Die Steuerung der Turtle erreichen wir, indem wir mit `getKeyCode()` in Zeile 18 prüfen, ob eine Taste gedrückt wird. Abhängig von der gedrückten Taste drehen wir die Turtle dann nach links oder rechts bzw. beenden das Spiel. Solange keine Taste gedrückt wird ist der Rückgabewert von `getKeyCode()` übrigens immer Null.

```
1  from gturtle import *
2  from time import sleep
3
4  SPACE = 32
5  LEFT = 37
6  RIGHT = 39
7
8  makeTurtle()
9  clear("black")
10 setPenColor("white")
11 setPenWidth(150)
12 setPos(-20, 0)
13 moveTo(20, 0)
14 setPos(0, 0)
15 showTurtle()
16 penUp()
17 repeat:
18     key = getKeyCode()
19     if key == SPACE:
20         break
```

```
21      if key == LEFT:
22          left(5)
23      if key == RIGHT:
24          right(5)
25      forward(1)
26      if getPixelColorStr() != "white":
27          break
28      sleep(0.05)
29  dispose()
30  msgDlg("GAME OVER!")
```

AUFGABEN

21. Lass die Turtle mit der Zeit immer schneller werden. Im Moment geht sie jedes Mal nur einen Pixel vorwärts. Indem du diese Zahl langsam erhöhst (verwende gebrochene Zahlen und erhöhe den Wert zum Beispiel bei jedem Durchgang um 0.01) wird das Spiel immer schwieriger.

22. Ändere das Spielfeld ab und zeichne zuerst ein Labyrinth, durch das die Turtle sich bewegen kann.

23. Lass alle paar Sekunden zufällig einen roten Punkt im Spielfeld zeichnen, den die Turtle dann «fressen» soll. Wenn die Turtle feststellt, dass der Pixel unter ihr rot ist, dann erhöht sich die Punktzahl des Spielers und der rote Punkt wird wieder entfernt.

Um den roten Punkt zu zeichnen musst die Turtle kurzzeitig verstecken und ihre Position mit `getX()` und `getY()` merken, um sie wieder dorthin zurückzusetzen wo sie eigentlich steht.

24. Ersetze die Turtle durch einen Punkt, der sich im Spielfeld bewegt. Verwende Variablen, um die Position und Geschwindigkeit deiner Figur festzuhalten. Auch für die roten Nahrungs-Punkte wirst du jetzt Variablen brauchen.

25.* Modifiziere das Programm so, dass wenn die Figur links aus dem Spielfeld hinausläuft, so ist das Spiel nicht beendet, sondern die Spielfigur erscheint dann am rechten Rand wieder. Vielleicht gibt es auch nur eine Türe, durch die die Spielfigur auf die andere Seite gehen kann.

26.* Füge einen vom Computer gesteuerten Gegenspieler hinzu, der ebenfalls versucht, die Nahrungspunkte zu fressen. Der Gegenspieler kann sich zufällig oder nach einem vorgegebenen Muster bewegen, wird aber nicht aus dem Spielfeld hinausgehen.

7 Auf die Maus reagieren

Lernziele In diesem Abschnitt lernst du:

- Was *Ereignis* und *Callback* sind.
- Mit globalen Variablen zu arbeiten.

Einführung Wenn du im Programm mit der Maus klickst oder die Maus bewegst, dann ruft Python jeweils eine Funktion auf, die auf das *Ereignis* reagieren soll (typische Ereignisse sind «Maus bewegt», «Taste gedrückt», «Fenstergrösse geändert», «USB-Stick angeschlossen» etc). Um auf solche Ereignisse zu reagieren kann dein Programm einen Callback (Rückruf) angeben: Eine Funktion, die beim entsprechenden Ereignis ausgeführt werden soll.

Wenn dein Programm auf den Druck einer Taste reagieren soll, dann schreibst du eine Funktion mit einem Parameter für den Tastencode und stellst ein `@onKeyHit` davor:

```
@onKeyHit
def my_key_handler(key):
    print key
```

Wie die Funktion bzw. der Parameter heisst spielt keine Rolle. Sobald du dann im Turtle-Fenster eine Taste drückst, wird Python den Tastendruck an deine Funktion weiterleiten.

Wie aber kannst du aus einer solchen Callback-Funktion heraus Variablenwerte im Programm selbst ändern? Der folgende Code hier funktioniert zum Beispiel nicht. Der Grund dafür liegt darin, dass Python in Zeile 4 hier eine *neue* Variable `last_key` anlegt und nicht die *globale* Variable ändert.

```
last_key = 0
@onKeyHit
def my_key_handler(key):
    last_key = key
```

Um aus einer Funktion heraus eine globale Variable zu ändern, musst du *in der Funktion* als erste Anweisung ein `global last_key` einfügen. Damit erhält die Funktion Zugriff auf die globale Variable.

Das Programm Für dieses einfache Zeichnungsprogramm installieren wir drei *Callback*-Funktionen für die Ereignisse «Maustaste gedrückt», «Maus gezogen» (bei gedrückter Taste) und «Mit Maus geklickt».

Beachte, dass die einzelnen Ereignisse jeweils die Koordinaten der Maus an die Callback-Funktionen übergeben. Mit `isRightMouseButton()` in Zeile 14 prüfen wir dort, ob die rechte Maustaste gedrückt wurde.

Damit die `color_index`-Variable nicht bei jedem Mausklick neu angelegt und danach entfernt wird, definieren wir diese in Zeile 13 als «global» (innerhalb der Funktion!). Dazu muss es dann aber auch eine globale Variable dazu geben (Zeile 21).

```
1  from gturtle import *
2
3  @onMousePressed
4  def mousePressed(x, y):
5      setPos(x, y)
6
7  @onMouseDragged
8  def mouseDragged(x, y):
9      moveTo(x, y)
10
11 @onMouseClicked
12 def mouseClicked(x, y):
13     global color_index
14     if isRightMouseButton():
15         color_index += 1
16         if color_index == 1:
17             setPenColor("red")
18         if color_index == 2:
19             setPenColor("green")
20
21 color_index = 0
22 makeTurtle()
23 hideTurtle()
```

AUFGABEN

27. Erweitere das Programm um weitere Farben und mach die Farbwahl zyklisch, d. h. nach z. B. 4 Farben soll es wieder bei Schwarz beginnen.

28. Ändere das Programm so ab, dass es bei einem Rechtsklick mit der Maus die aktuelle Farbe unter der Turtle aufnimmt. Dazu verwendest du die Funktion `getPixelColorStr()`, die die Farbe unter der Turtle zurückgibt. Vergiss nicht, die Turtle zuerst an die richtige Stelle zu setzen! Für ein Zeichenprogramm braucht es dann am Rand Farbtupfer oder ein Farbspektrum, um die Farbe auch wirklich ändern zu können.

29.* Verwende die Funktionen `setFillColor(farbe)` und `fill()`, um bei einem Linksklick mit der Maus die aktuelle Fläche mit Farbe zu füllen.

8 Mit den Augen rollen

Lernziele In diesem Abschnitt lernst du:

- Punkte mit Tupeln darzustellen.
- Die Werte zweier Variablen zu vertauschen.

Einführung In der Mathematik schreibst du Punkte als *Tupel*, z. B. $A = (3, 4)$. Python beherrscht diese Schreibweise für Tupel auch und kann damit in einer Variablen nicht nur eine einzelne Zahl speichern, sondern einen ganzen Punkt mit beiden Koordinaten.

Um sinnvoll mit Tupeln arbeiten zu können musst du die Zahlen eines Tupels auch wieder extrahieren können. Du kannst also die einzelnen Werte des Tupels in andere Variablen kopieren:

```
A = (3, 4)
(x, y) = A
```

Im Prinzip geht das natürlich auch direkt:

```
(x, y) = (3, 4)
```

Besonders schön ist diese Schreibweise, wenn du die Werte zweier Variablen *vertauschen* möchtest (ein sogenannter «Swap»):

```
(x, y) = (y, x)
```

Das Programm Es ist eine beliebte Übung, Augen zu programmieren, die der Maus folgen. Mit der Turtle-Grafik geht das besonders einfach. Dazu schreiben wir eine Callback-Funktion für das Ereignis «Maus bewegt» und zeichnen dann jeweils die Augen neu.

Beachte, wie wir in diesem Programm praktisch nicht mehr mit einzelnen Koordinaten arbeiten, sondern mit Punkten. Anstelle von zwei Variablen für die x- und y-Koordinate eines Auges fassen wir beide Zahlen zu einem Punkt zusammen. In Zeile 8 extrahieren bzw. entpacken wir dann wieder die beiden einzelnen Koordinaten des Auges.

```
1 from gturtle import *
2
3 LEFT_EYE = (-30, 0)
4 RIGHT_EYE = (30, 0)
5
6 def drawEye(eye_pos, mouse_pos):
7     setPenColor("white")
```

```
8       (x, y) = eye_pos
9       setPos(x, y)
10      dot(32)
11      heading(towards(mouse_pos))
12      forward(8)
13      setPenColor("black")
14      dot(15)
15
16  @onMouseMoved
17  def mouseMoved(x, y):
18      pos = (x, y)
19      drawEye(LEFT_EYE, pos)
20      drawEye(RIGHT_EYE, pos)
21
22  makeTurtle()
23  clear("navy")
24  drawEye(LEFT_EYE, (0, 0))
25  drawEye(RIGHT_EYE, (0, 0))
```

Was ist eigentlich der Unterschied zwischen den beiden Ereignissen «`onMouseMoved`» und «`onMouseDragged`»? Bei «`onMouseDragged`» wird die Maus *mit gedrückter Maustaste* bewegt.

AUFGABEN

30. Programmiere einen «Zeiger», der in der Mitte des Fensters angemacht ist und dessen Ausrichtung sich mit der Maus einstellen lässt. Verwende hier das `onMouseDragged`-Ereignis, so dass sich der Zeiger nur bewegt, wenn du die Maustaste drückst.

31. Animiere den Zeiger aus der vorherigen Aufgabe und mach ihn zu einem Sekundenzeiger. Jede Sekunde springt also etwas weiter, ausser wenn die Maustaste gedrückt ist, um ihn zu setzen. Dazu brauchst du auch die Ereignisse «`onMousePressed`» und «`onMouseReleased`».

32.* Programmiere eine Uhr mit mindestens zwei Zeigern. Dabei kannst du jeden Zeiger einzeln stellen und anpassen. Dazu musst du die Zeiger-Position in eigenen Variablen (Winkeln) festhalten.

Falls Du beim Programmstart mit der aktuellen Zeit beginnen möchtest, so zeigt dir der folgende Programmausschnitt, wie du die aktuelle Zeit ermitteln kannst:

```
from datetime import datetime
zeit = datetime.now().time()
print zeit.hour, zeit.minute, zeit.second
```

9 Gravitation

Lernziele In diesem Abschnitt lernst du:

- Animationen mit Gravitation zu programmieren.
- Objekteigenschaften mit Tupeln darzustellen.

Einführung Erinnerst du dich an den Ball in der Box? Der Ball hatte dort im Wesentlichen zwei Eigenschaften: Die Position (x-Koordinate) und die Geschwindigkeit in x-Richtung. Durch die Anweisung «`x+=v_x`» hat sich der Ball dann mit der gegebenen Geschwindigkeit fortbewegt.

Jetzt bringen wir die Beschleunigung a_x dazu. So ändert sich neben der Position auch die Geschwindigkeit nach dem selben Muster:

```
x += v_x
v_x += a_x
```

Allerdings beschleunigen wir unseren Ball nicht in x-Richtung, sondern nach unten. So können wir hervorragend die Gravitation (Erdanziehung) simulieren: Die Gravitation ist eine konstante Beschleunigung nach unten.

Das Programm In diesem Programm lassen wir zwei Bälle nach unten fallen und zeichnen ihre Spur auf. Für jeden Ball verwenden wir ein *Tupel*, das die Eigenschaften (Position, Geschwindigkeit, Farbe) des Balles enthält (Zeilen 19 und 20). Wir packen also alle wesentlichen Eigenschaften des Balls in eine einzige Variable und können daher viel besser die Übersicht über verschiedene Bälle bewahren. Allerdings lassen sich einzelne Werte in einem Tupel nicht ändern; dazu müssen wir das Tupel zuerst entpacken, die Werte ändern und dann die neuen Werte wieder verpacken.

Die Funktion `moveObject` ist dafür verantwortlich, einen Ball effektiv zu bewegen und zu beschleunigen. Dazu werden die fünf Werte im Tupel `obj` in Zeile 7 zunächst einmal «entpackt» und in fünf eigenen Variablen gespeichert. In Zeile 11 verpackt die Funktion dann alle fünf Werte wieder in ein Tupel und gibt dieses zurück. Damit können wir in den Zeilen 24 und 25 dann sehr komfortabel die Eigenschaften für alle Bälle aktualisieren bevor wir sie zeichnen.

```
1 from gturtle import *
2 from time import sleep
3
```

```
4  GRAVITY = -0.5
5
6  def moveObject(obj):
7      (x, y, v_x, v_y, color) = obj
8      x += v_x
9      y += v_y
10     v_y += GRAVITY
11     return (x, y, v_x, v_y, color)
12
13 def drawObject(obj):
14     (x, y, v_x, v_y, color) = obj
15     setPos(x, y)
16     setPenColor(color)
17     dot(10)
18
19 ball1 = (-200, 300, 3, 0, "red")
20 ball2 = (310,  260, -9, 8, "green")
21 makeTurtle()
22 clear("black")
23 repeat 50:
24     ball1 = moveObject(ball1)
25     ball2 = moveObject(ball2)
26     drawObject(ball1)
27     drawObject(ball2)
28     sleep(0.1)
```

AUFGABEN

33. Passe das Programm so an, dass wiederum eine echte Animation entsteht, in der sich die Bälle bewegen und ihre Spur verschwindet. Zudem soll das Programm bei einem Tastendruck beendet werden.

34. Erweitere das Tupel mit den Ball-Eigenschaften um eine weitere Eigenschaft «diameter», die den Durchmesser des Balls enthält und gib deinen Bällen verschiedene Grössen.

35. Baue dein vorhandenes Wissen aus dem Abschnitt «Ball in der Box» in der Funktion `moveObject` ein, so dass die Bälle nun an den Wänden einer Box abprallen, aber nach wie vor der Gravitation unterliegen.

36.* Eine Farbe hat neben den drei Grundfarben Rot, Grün und Blau noch einen *Alpha*-Kanal, der die Transparenz angibt: $\alpha = 1.0$ ist die volle Farbe und $\alpha = 0.0$ ist vollständig transparent. Du kannst den Alpha-Kanal (die Transparenz) einer Farbe mit `makeColor` ändern:

```
color = makeColor(color, new_alpha)
```

Verwende diese Technik, um die Bälle mit der Zeit verschwinden zu lassen. Gib ihnen als weitere Eigenschaft einen Alpha-Wert und multipliziere diesen in jedem Schritt mit 0.95.

10 Formen in Listen angeben

Lernziele In diesem Abschnitt lernst du:

- Mit Listen zu arbeiten.
- Die Elemente einer Liste mit einer Schleife durchzugehen.

Einführung Während sich Tupel hervorragend eignen, um verschiedene Eigenschaften eines Objekts zusammenzufassen, kannst du mit einer *Liste* eine Sammlung von Objekten verwalten. Wichtig ist dabei, dass du mit einer `for`-Schleife alle Elemente einer Liste einzeln durchlaufen kannst.

Hier ist ein einfaches Beispiel, das für einige Primzahlen die Primzahl und dessen Quadrat ausgibt:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19]
for p in primes:
    print p, p**2
```

Das Programm Das Haus, das in diesem Programm gezeichnet wird ist sehr einfach. Die Funktion `draw_shape` erwartet als Parameter eine Liste `shape` mit Punkten. In Zeile 4 werden die Koordinaten des ersten Punkts in der Liste mit der Funktion `head` ausgelesen und als `x1` und `y1` gespeichert. Die Turtle wird dann auf diesen Punkt gesetzt und anschliessend geht die `for`-Schleife die Koordinaten jedes Punktes durch und zeichnet eine Linie zum Punkt. Am Ende wird die Figur geschlossen, indem die Turtle wieder an den Anfangspunkt (x_1, y_1) zurückkehrt.

```
1  from gturtle import *
2
3  def draw_shape(shape):
4      (x1, y1) = head(shape)
5      setPos(x1, y1)
6      for (x, y) in shape:
7          moveTo(x, y)
8      moveTo(x1, y1)
9
10 house = [(0, 0), (50, 0), (50, 45), (25, 70), (0, 45)]
11 makeTurtle()
12 speed(-1)
13 draw_shape(house)
```

Übrigens: Anstatt die Punkte für eine Form oder Figur aus dem Gedächtnis hinzuschreiben kannst du dafür auch ein eigenes Programm schreiben. Dafür verwendest du den Maus-Callback:

```
@onMouseClicked
def mouseClicked(x, y):
    print (x, y)
```

Hinweis: In Python geht eine **for**-Schleife *immer* die Elemente einer Liste durch.

AUFGABEN

37. Verwende `fillToPoint()` und `fillOff()`, um das gegebene Polygon mit einer Farbe auszufüllen.

38. Schreibe ein Programm, das zu einer Liste von Punkten den «Stern» zeichnet: Es werden also alle Punkte in der Liste mit dem ersten Punkt in der Liste verbunden.

39. Im Beispiel oben wird das Haus so gezeichnet, das die linke untere Ecke in der Fenstermitte $(0, 0)$ ist. Wenn du nun daneben ein zweites Haus zeichnen möchtest, dann wäre es kaum sinnvoll, dafür eine eigene Liste mit Punkten zu definieren. Vielmehr wollen wir das erste Haus einfach verschieben.

Erweitere die Funktion `draw_shape` um zwei Parameter `t_x` und `t_y` («Translation» ist das Fachwort für eine Verschiebung, daher das t). Die gegebene Form wird dann um (t_x, t_y) verschoben gezeichnet.

40.* Die Liste `house` oben enthält die absoluten Koordinaten der Punkte. Du kannst die Koordinaten der Punkte aber auch jeweils relativ zum vorangehenden Punkt angeben. `house` wird dann zu:

```
house = [(0, 0), (50, 0), (0, 45), (-25, 25), (-25, 25)]
```

Schreibe eine Funktion `draw_shape_relative`, die mit solchen «relativen» Listen arbeitet.

Tipp: Mit `x = getX()` und `y = getY()` kannst du auch die aktuellen Koordinaten der Turtle bestimmen. Natürlich gibt es aber auch ganz andere Möglichkeiten, um diese Aufgabe zu lösen.

11 Space Invaders

Lernziele In diesem Abschnitt lernst du:

- Neue Werte zu einer Liste hinzuzufügen.
- Die Werte einer Liste zu bearbeiten.

Einführung Die grosse Stärke von Listen besteht darin, dass du während der Ausführung des Programms neue Elemente hinzufügen oder alte entfernen kannst. Das verwenden wir in unserem Beispielprogramm, um beliebig viele Elemente zu haben, die sich über den Bildschirm bewegen.

Das Programm In diesem Programm kannst du mit der Leertaste gelbe Kugeln (engl. *bullets*) schiessen. Die Liste «`bullets`» enthält die Koordinaten der Kugeln, die im Spiel sind. In Zeilen 26 und 27 verwenden wir `append`, um eine neue Kugel zur Liste hinzuzufügen. Die Funktion `moveBullets` ab Zeile 7 ist dafür verantwortlich, die einzelnen Kugeln nach oben zu bewegen, indem bei jeder Kugel der y-Wert jeweils um 10 erhöht wird.

Wenn du eine globale Variable innerhalb einer Funktion *verändern* willst, dann musst du das mit **`global`** am Anfang der Funktion angeben (Zeilen 8 und 25).

```
1  from gturtle import *
2  from time import *
3
4  bullets = []
5  shuttle_pos = 0
6
7  def moveBullets():
8      global bullets
9      new_bullets = []
10     for (x, y) in bullets:
11         new_bullets.append((x, y+10))
12     bullets = new_bullets
13
14 def drawScene():
15     clear("black")
16     setPenColor("yellow")
17     for (x, y) in bullets:
18         setPos(x, y)
19         dot(5)
```

```
20      setPenColor("white")
21      setPos(shuttle_pos-10, -100)
22      moveTo(shuttle_pos+10, -100)
23
24  def shoot():
25      global bullets
26      bullet = (shuttle_pos, -100)
27      bullets.append(bullet)
28
29  makeTurtle()
30  repeat:
31      key = getKeyCode()
32      if key == 32:   # Space
33          shoot()
34      if key == 27:   # Escape
35          break
36      moveBullets()
37      drawScene()
38      sleep(0.1)
```

AUFGABEN

41. Ergänze das Programm um folgende Punkte:

- Mit den Pfeiltasten soll man das Shuttle nach links und rechts bewegen können. Natürlich erscheinen dann auch die Kugeln an der entsprechenden Stelle.
- Achte darauf, dass das Shuttle nicht aus dem Fenster herausfällt.
- Füge in der Funktion `moveBullets` die Kugeln nur dann zur Liste `new_bullets` hinzu, wenn sie überhaupt noch auf dem Bildschirm zu sehen sind.

42.* Verwende die Maus-Callbacks `onMouseMoved` und `onMouseHit`, um das Shuttle mit der Maus zu steuern.

43. Baue das Programm so aus, dass es auch ein Ziel gibt, das man treffen kann. Im Original «Space Invaders» gab es dutzende von «Raumschiffen» oder «Aliens», die man abschiessen konnte.

44.* Schreibe ein Programm, das auf Mausklick ein «Feuerwerk» erzeugt. Dabei brauchen die einzelnen Punkte neben den Koordinaten (x, y) für ihre Position auch Werte (v_x, v_y) für ihre Geschwindigkeit. Bei einem Mausklick erzeugst du dann z. B. 20 neue Punkte an der aktuellen Position. Die Geschwindigkeiten wählst du zufällig mit:

```
v_x = randint(-50, 50) / 50
v_y = randint(-50, 50) / 50
```

K a p i t e l Z w e i

SPRACHEN ERKENNEN

Für den Computer spielt es eine grosse Rolle, *in welcher Form* Daten dargestellt werden. In Python gibt es zum Beispiel einen Unterschied zwischen `"12"* 3` und `12 * 3`. Im ersten Fall multiplizierst du einen Text mit drei und erhältst `"121212"`. Im zweiten Fall multiplizierst du eine Zahl mit drei und bekommst `36`.

Oft sind die Eingaben in Form eines Textes (String) gegeben. Zum Beispiel gibst du die Rechnung $12 + 3 \cdot 45$ ein. Dann muss der Computer aus diesem Text die relevanten Informationen heraussuchen. Er muss erkennen, dass der Text drei Zahlen und zwei Operationen enthält. Er muss aber auch wissen, dass die Multiplikation vor der Addition berechnet werden muss, entgegen der Regel, dass die Rechnungen von links nach rechts ausgewertet werden.

Eine Rechnung aus dem Text heraus erkennen ist ein Beispiel für «Spracherkennung». Es geht dabei nicht darum, *gesprochene* Sprache zu erkennen, sondern eine *formale* Sprache wie die der Mathematik oder eine Programmiersprache.

1 Zahlen lesen

Lernziele In diesem Abschnitt lernst du:

- Aus einem String heraus eine Zahl zu erkennen.

Einführung Wie kannst du aus einem Text heraus eine Zahl einlesen? Das Problem besteht im Prinzip aus zwei Schritten: Zum einen muss der Computer erkennen, wo die Zahlen beginnt und wo sie aufhört. Zum anderen muss er den entsprechenden Textteil dann in eine Zahl umwandeln.

Erkennen, ob ein Buchstabe noch zu einer Zahl gehört lösen wir in Python sehr einfach: Wir prüfen das mit `if '0'<= char <= '9'`.

Die Zahl auch tatsächlich erkennen erfolgt nach folgender Idee: Bei jeder neuen Ziffer, die dazukommt, multiplizierst du die biherige Zahl mit 10 und addierst dann die neue Ziffer. Wenn du also bereits 613 eingelesen hast und die nächste Ziffer eine Vier ist, dann rechnest du: $613 \cdot 10 + 4 = 6130 + 4 = 6134$.

Mit `ord(C)` kannst du den Ascii-Code des Zeichens C bestimmen. Allerdings beginnt der Ascii-Code aus historischen Gründen mit 32 Zeichen, die für «Zeilenumbruch», «Klingeln», «Übertragungsende» etc. stehen. Die Null hat tatsächlich den Code 48. Um daraus den *Wert* der Ziffer zu erhalten rechnen wir `ord(C) - ord('0')`, was `ord(C) - 48` entspricht.

Das Programm Kernstück des Programms ist die Funktion `str_to_num`, die aus dem gegebenen String heraus eine Zahl einliest und diese als Python-Zahl zurückgibt.

In den Zeilen 4 bis 7 wird der Ganzzahlteil der Zahl eingelesen. Falls danach ein Punkt kommt, so werden in den Zeilen 11 bis 14 auch noch die Nachkommastellen eingelesen. In der Zeile 15 berechnen wir die Anzahl der Nachkommastellen n und verwenden das in der Zeile 16, um die eingelesene Zahl durch 10^n zu teilen.

```
1 def str_to_num(text, start = 0):
2     result = 0
3     i = start
4     while i < len(text) and '0' <= text[i] <= '9':
5         result *= 10
6         result += ord(text[i]) - ord('0')
7         i += 1
```

```
8       if i < len(text) and text[i] == '.':
9           i += 1
10          j = i
11          while i < len(text) and '0' <= text[i] <= '9':
12              result *= 10
13              result += ord(text[i]) - ord('0')
14              i += 1
15          number_digits = i - j
16          result = result / (10 ** number_digits)
17      return result
18
19  print str_to_num("123.45")
```

AUFGABEN

1. Die Funktion `str_to_int` funktioniert nur mit positiven Zahlen ohne Vorzeichen. Bei einem Plus- oder Minuszeichen gibt sie einfach Null zurück. Ergänze das Programm so, dass auch Zahlen mit einem Vorzeichen korrekt eingelesen werden.

Um das Vorzeichen richtig zu berücksichtigen prüfst du ganz am Anfang der Funktion, ob ein Vorzeichen vorliegt (und überspringst das wenn nötig). Wirklich berücksichtigt wird das Vorzeichen aber erst ganz am Ende der Funktion. Erst dann kannst du das Resultat, falls nötig, negativ machen.

2. Vor allem sehr grosse oder kleine Zahlen werden oft auch in der wissenschaftlichen Notation geschrieben. Zur Erinnerung: Die Masse der Erde ist z. B.:

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

Weil es schwierig ist, das 10^x korrekt darzustellen, verwendet man bei Computern in der Regel die Notation `5.98e+24` anstelle von $5.98 \cdot 10^{24}$.

Erweitere die Funktion `str_to_num` so, dass die Funktion auch Zahlen mit Exponenten lesen kann. Nach der Zahl steht dann also ein (grosses oder kleines) «E», gefolgt von einem Vorzeichen und einer ganzen Zahl für den Exponenten.

3.* Schreibe eine zweite Funktion `hex_to_int`, die Zahlen im *hexadezimalen* Format einliest und als ganze Zahl zurückgibt (es ist nicht üblich, gebrochene Zahlen hexadezimal zu schreiben). Hexadezimale Zahlen erkennst du am *Präfix* «`0x`».

2 Eingaben zerlegen

Lernziele In diesem Abschnitt lernst du:

- Einen Text in eine Liste mit einzelnen Grundelementen zu zerlegen.

Einführung Die einzelnen Grundelemente der Sprache werden *Token* genannt. Die Eingabe «$12 + 3 - 4.5$» besteht also aus den fünf Token 12, $+$, 3, $-$ und 4.5. Das Zerlegen der Eingabe in diese fünf Token wird im Englischen als *to tokenize* bezeichnet (im Deutschen müssen wir das als «in Zeichen/Symbole zerlegen» umschreiben). Diesen Begriffen wirst du im Folgenden immer wieder begegenen.

Das Programm Die Funktion `tokenize_input` zerlegt in diesem Programm eine Eingabe in eine Liste mit einzelnen *Token*. Wenn das aktuelle Zeichen eine Ziffer ist, dann wird eine Zahl eingelesen. Bei den vier Grundoperationen wird das Operationszeichen zum Resultat hinzugefügt. Alle anderen Zeichen werden im Moment einfach ignoriert.

Die Funktion `str_to_num` aus dem letzten Abschnitt haben wir hier gekürzt wiedergegeben. Beachte die geänderte Zeile 8, die nun nicht nur den gelesenen Wert zurückgibt, sondern auch die Position nach der Zahl. In Zeile 15 empfangen wir dann beide Rückgabewerte.

Übrigens: Negative Zahlen werden hier in ein Vorzeichen und die eigentliche Zahl zerlegt. Das ist durchaus üblich und richtig so. Die Zahl -8.9 im Beispiel unten werden wir dann in einem späteren Schritt wieder aus den zwei Token $-$ und 8.9 zusammensetzen.

```
1  def str_to_num(text, start = 0):
2      result = 0
3      i = start
4      while i < len(text) and '0' <= text[i] <= '9':
5          result *= 10
6          result += ord(text[i]) - ord('0')
7          i += 1
8      return result, i
9
10 def tokenize_input(text):
11     i = 0
12     result = []
13     while i < len(text):
14         if '0' <= text[i] <= '9':
15             number, i = str_to_num(text, i)
```

```
16              result.append(number)
17          elif text[i] in ['+', '-', '*', '/']:
18              result.append(text[i])
19              i += 1
20          else:
21              i += 1
22      return result
23
24  print tokenize_input("123.45 + 67 * -8.9")
```

Die wichtigsten Punkte In Python können Funktionen beliebig viele Werte zurückgeben, die durch Kommata getrennt werden. Im Prinzip müssten noch Klammern gesetzt werden, weil Python hier tatsächlich mit Tupeln arbeitet. In der Praxis werden die Klammern hier aber in der Regel weggelassen.

In beiden Fällen erhalten die Variablen x, y und z die Werte $x = 11$, $y = 13$ und $z = 144$.

```
def foo(x):
    return (x-1, x+1, x**2)
(x, y, z) = foo(12)
```

```
def foo(x):
    return x-1, x+1, x**2
x, y, z = foo(12)
```

AUFGABEN

4. Füge das Beispiel mit Deiner Funktion `str_to_num` aus dem letzten Abschnitt zusammen, so dass sie die gebrochenen Zahlen 123.45 und 8.9 korrekt lesen kann.

5. Ergänze das Programm so, dass auch Klammern sowie das Zeichen `^` für Potenzen eingelesen werden.

Bei allen anderen (ungültigen) Zeichen, ausser dem Leerzeichen, soll das Programm dafür eine Fehlermeldung ausgeben und darauf hinweisen, dass das Programm ein ungültiges Zeichen enthält.

6. Erweitere das Programm so, dass auch Variablennamen korrekt eingelesen und zur Liste hinzugefügt werden. So soll man als Eingabe zum Beispiel auch die Formel «`2*pi`» eingeben können.

3 Rechnungen darstellen und auswerten

Lernziele In diesem Abschnitt lernst du:

- Rechnungen mit verschachtelten Tupeln darzustellen.
- Rechnungen in der Form von Tupeln auszuwerten.

Einführung Um Rechnungen auch tatsächlich mit dem Computer auswerten zu können, müssen wir sie sinnvoll darstellen können. Das heisst so, dass der Computer gut damit arbeiten kann. Zudem müssen wir darauf achten, dass wir die Punkt-vor-Strich-Regel eingehalten wird.

Die vier Grundrechenarten basieren alle auf dem gleichen Schema: «Wert links», «Operator», «Wert rechts». Solche Operationen werden auch *binär* genannt, weil sie *zwei* Werte miteinander verrechnen. Allerdings werden wir hier die *polnische Notation* verwenden, in der der Operator immer zuerst geschrieben wird (die Idee stammt von einem polnischen Logiker, daher der Name «polnische» Notation).

Die Rechnung $12 + 34$ wird damit zu $(+, 12, 34)$ und $4.5 \cdot 18$ wird zu $(\cdot, 4.5, 18)$. Komplexere Rechnungen können wir verschachtelt darstellen: $7 - 3 \cdot 14$ wird dann zu $(-, 7, (\cdot, 3, 14))$.

Das Programm Die Funktion `eval_tuple` ist dafür Zuständig ein Tupel auszuwerten (engl. *evaluate*). Das Tupel t muss dabei immer die Form *(operator, wert links, wert rechts)* haben. So kannst du diese Funktion zum Beispiel mit dem Tupel `('-', 50, 14)` aufrufen und erhältst als Antwort die Differenz $50 - 14 = 36$.

Weil die Rechnungen auch verschachtelt sein können, müssen wir in den Zeilen 4 bis 7 prüfen, ob die linke oder rechte Seite selbst wieder ein Tupel ist. Falls ja, so müssen wir zuerst dieses auswerten.

In diesem Fall hier soll die Funktion die Rechnung $12+3\cdot15$ auswerten, die du in Zeile 17 codiert findest.

```
1  def eval_tuple(t):
2      if len(t) == 3:
3          op, left, right = t
4          if type(left) is tuple:
5              left = eval_tuple(left)
6          if type(right) is tuple:
7              right = eval_tuple(right)
```

```
8           if op == '+':
9               return left + right
10          elif op == '-':
11              return left - right
12          elif op == '*':
13              return left * right
14          elif op == '/':
15              return left / right
16
17  calc = ( '+', 12, ( '*', 3, 15 ) )
18  print eval_tuple(calc)
```

Übrigens: Die Rechnung $1+2+3$ wird korrekt als $(+,(+,1,2),3)$ dargestellt (und nicht als $(+,1,(+,2,3))$). Die Klammern werden von Innen nach Aussen gelöst und hier soll $1+2$ zuerst berechnet werden.

AUFGABEN

7. Erweitere die Auswertung so, dass auch der Potenzoperator «^» ausgewertet wird. Achtung: In Python wird das als $**$ geschrieben, ^ selbst hat eine andere Bedeutung.

8. Bringe die folgenden Rechnungen in die polnische Notation:

(a) $12+7\cdot 2-9$

(b) $4\cdot 7-3\cdot x+14$

(c) $5-6-7$

(d) $1\cdot 2\cdot 3+(4-5)\cdot 6$

9. Werte die folgenden Ausdrücke von Hand aus.

(a) $(+,(\cdot,5,3),4)$

(b) $(/,(-,29,11),(+,2,4))$

10. Gibt es einen Fall bzw. eine Rechnung, bei der du auch in der polnischen Notation zusätzliche Klammern brauchst?

11. Neben den *binären* brauchen wir auch *unäre* Operationen, die neben dem Operator nur einen Wert haben. Der wichtigste davon ist die Negation. Der Ausdruck -42 wird also korrekt als $(-,42)$ geschrieben, etwas wie $-(3\cdot 4)$ wird zu $(-,(\cdot,3,4))$.

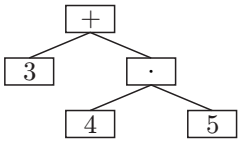
Füge eine weitere Möglichkeit in der Funktion ein, so dass auch ein Tupel der Länge 2 mit dem Operator «−» korrekt ausgewertet wird. Prüfe deine Lösung zum Beispiel mit $(-,(\cdot,3,4))$.

4 Die Grammatik einer Sprache

Lernziele In diesem Abschnitt lernst du:

- Was ein Strukturbaum ist.
- Wie du eine formale Sprache mit einer Grammatik beschreibst.
- Aus einer Grammatik ein Programm zu schreiben, das eine Sprache mit der Grammatik in einen Strukturbaum umwandelt.

Einführung In den vorhergehenden Abschnitten hast du gesehen, wie du eine Eingabe in einzelne Tokens zerlegst und wie du eine Rechnung mit einem *Strukturbaum* darstellst. Den Strukturbaum haben wir allerdings mit Tupeln geschrieben.



Der Strukturbaum für die Rechnung $3 + 4 \cdot 5$. In Python haben wir das als $(+, 3, (\cdot, 4, 5))$ geschrieben.

Das Herzstück fehlt aber noch: Wie kommst du nun von der Liste mit den Tokens zum Strukturbaum? Und wie stellst du dabei sicher, dass die Punkt-vor-Strich-Regel richtig eingehalten wird?

Die Grammatik für arithmetische Terme Du hast sicher gelernt, dass eine *Summe* aus einzelnen *Summanden* und ein *Produkt* aus einzelnen *Faktoren* besteht. Die einzelnen Summanden werden durch $+$ getrennt, die Faktoren durch $\cdot$. Das können wir so schreiben:

```
summe ::= summand '+' summand '+' summand '+' ...
produkt ::= faktor '*' faktor '*' faktor '*' ...
```

Das Zeichen `::=` steht hier für «wird definiert als». Die genaue Anzahl der Summanden und Faktoren kennen wir nicht. Im Prinzip kann aber eine Summe auch aus nur einem Summanden und ein Produkt aus nur einem Faktor bestehen.

Die Wiederholungen fassen wir jetzt mit Potenzen zusammen. Anstatt $x\,a\,x\,a\,x$ schreiben wir $x\,(a\,x)^2$. Wenn wir die genaue Anzahl nicht kennen, dann schreiben wir $x\,(a\,x)^*$, also:

```
summe ::= summand ('+' summand)*
produkt ::= faktor ('*' faktor)*
```

Die korrekte Reihenfolge der Operationen lässt sich jetzt sehr einfach beschreiben: Jeder Summand ist ein Produkt. Jeder Faktor ist eine Zahl. Damit wird unsere Grammatik zu:

```
rechnung ::= summe
summe ::= produkt ('+' produkt)*
produkt ::= faktor ('*' faktor)*
faktor ::= NUMBER
```

Am Ende fehlt nur noch eine Erweiterung: Die einzelnen Summanden in einer Summe können auch durch Minuszeichen getrennt sein. $a - b$ ist auch eine Summe, nämlich $a + (-b)$. Um das auch in der Grammatik deutlich zu machen brauchen wir den senkrechten Strich | mit der Bedeutung «oder»:

```
rechnung ::= summe
summe ::= produkt (('+' | '-') produkt)*
produkt ::= faktor ('*' faktor)*
faktor ::= NUMBER
```

Wiederum haben wir in diesem Beispiel die unären Operatoren (Vorzeichen) + und − noch nicht berücksichtigt. Dazu definieren wir den `faktor` um:

```
faktor ::= ('+' | '-') faktor | NUMBER
```

Das Programm Dieses Programm erstellt den Strukturbaum zur Rechnung $3 + 4 \cdot 5$. Die Rechnung geben wir der Einfachheit halber bereits in Token zerlegt an (Zeile 1).

Zur Erstellung des Strukturbaums (sogenanntes «parsen») brauchen wir zwei wichtige Hilfsfunktionen: `peek()` in Zeile 4 und `next()` in Zeile 10. Während `peek` nur das nächste Zeichen (Token) in der Eingabe zurückgibt, erhöht `next()` auch noch den aktuellen Index. Das heisst, `next()` liest das Zeichen auch ein und geht dann zum nächsten weiter.

Die Funktion `parse()` in Zeile 19 ist die wichtige Funktion, bei der das eigentliche Parsen, bzw. die Erstellung des Strukturbaums beginnt. Du siehst hier, wie wir die folgende Grammatik umgesetzt haben:

```
rechnung ::= summe
summe ::= produkt (('+' | '-') produkt)*
produkt ::= faktor (('*' | '/') faktor)*
faktor ::= atom
atom ::= ('+'|'-') atom | NUMBER
```

Beachte schliesslich, dass wir auch Fehler als solche erkennen und angeben (Zeilen 49 und 52).

```
1  token_input = [3, '+', 4, '*', 5]
2  index = 0
3
4  def peek():
5      if index < len(token_input):
6          return token_input[index]
7      else:
8          return None
9
10 def next():
11     global index
```

```
12      if index < len(token_input):
13          result = token_input[index]
14          index += 1
15          return result
16      else:
17          return None
18
19  def parse():
20      return parse_summe()
21
22  def parse_summe():
23      result = parse_produkt()
24      while peek() in ['+', '-']:
25          operator = next()
26          next_summand = parse_produkt()
27          result = (operator, result, next_summand)
28      return result
29
30  def parse_produkt():
31      result = parse_faktor()
32      while peek() in ['*', '/']:
33          operator = next()
34          next_faktor = parse_faktor()
35          result = (operator, result, next_faktor)
36      return result
37
38  def parse_faktor():
39      return parse_atom()
40
41  def parse_atom():
42      token = next()
43      if token in ['+', '-']:
44          next_token = parse_atom()
45          return (token, next_token)
46      elif type(token) is int:
47          return token
48      elif token is None:
49          print "FEHLER: unerwartetes Ende der Eingabe"
50          return None
51      else:
52          print "FEHLER: ungueltiges Zeichen"
53          return None
54
55  print parse()
```

Struktur und Bedeutung, Syntax and Semantik Mit dem Parsen wird eine Liste von Tokens in einen Strukturbaum umgewandelt. Dabei haben die einzelnen Zeichen und Operatoren aber keine *Bedeutung*! Das Programm hier weiss also nichts davon, dass + für die Addition steht. Auch die Grammatik gibt nur die Struktur einer Sprache an und

nicht die Bedeutung der einzelnen Elemente.

Es geht also zunächst darum, die *Syntax* (Struktur, Aufbau) einer Sprache mit einer Grammatik zu beschreiben. Ein Text oder Programm muss immer zunächst einmal syntaktisch korrekt sein (zum Beispiel hat $3++$ keinen Sinn, es ist syntaktisch nicht korrekt). Erst danach können wir uns Gedanken über die *Semantik* (d. h. Bedeutung) machen. In diesem Abschnitt haben wir die Semantik der einzelnen Zeichen aber noch gar nicht berücksichtigt.

AUFGABEN

12. Im Moment arbeitet das Programm nur mit ganzen Zahlen (Datentyp `int`). Erweitere das Programm so, dass es auch Fliesskomma-Zahlen (vom Datentyp `float`) erkennt und zulässt.

13. Erweitere das Programm so, dass es auch Potenzen erkennt und korrekt verarbeitet.

14. Im Moment kann unser Programm noch keine Rechnungen mit Klammern verarbeiten. Dazu müssen wir die Grammatik zuerst etwas abändern und `atom` so definieren, dass auch Summen in Klammern ein `atom` sind:

```
atom ::= ('+'|'-') atom | NUMBER | '(' summe ')'
```

Programmiere diese Erweiterung ein, so dass das Programm auch Klammerausdrücke erkennt und korrekt umsetzt. Achtung: Die Klammern siehst du im ausgegebenen Strukturbaum nicht mehr. Sie sind nur in der Eingabe wichtig.

15. In den vorangegangenen Abschnitten hast du gelernt, einen Text in eine Liste von Tokens zu zerlegen, und einen Strukturbaum auszuwerten (auszurechnen). Füge diese Teile mit dem vorliegenden Programm zusammen, so dass ein kompletter Rechner entsteht.

Über `inputString()` kann man eine Rechnung eingeben und erhält dann das korrekte Ergebnis.

Teste dein Programm auch ausgiebig. Überlege dir möglichst «gemeine» oder «schwierige» Fälle und stelle sicher, dass dein Programm diese korrekt bearbeitet. Ein Programm, das nicht korrekt arbeitet ist relativ wertlos!

16.* Füge die Symbole `pi` und `e` für $\pi \approx 3.1416$ und $e \approx 2.7128$ hinzu.

17.* Erweitere deinen Parser so, dass er auch die Wurzelfunktion `sqrt()` (z. B. `sqrt(3)` für $\sqrt{3}$) erkennt (vgl. Aufgabe 2.14).

```
atom ::= ('+' ... summe ')' | 'sqrt' '(' summe ')'
```

5 Modularisieren

Lernziele In diesem Abschnitt lernst du:

- Einzelne Programmteile in Modulen zu verwalten.
- Variablennamen einzulesen.

Einführung Sobald die Programme etwas grösser werden, ist es sinnvoll, einzelne Teile in *Module* auszulagern. In Python ist es sehr einfach, ein Modul zu erstellen, dass du dann mittels `import` in einem anderen Programm verwendest kannst.

Das Programm Tatsächlich handelt es sich hier um zwei Programme, die aber zusammengehören. Speichere also die beiden Programme *im selben Ordner* unter den Namen `scanner.py` und `parser.py`.

Das erste Programm hier ist unser «Modul»: Es enthält nur Funktionen und keinen eigenen Programmcode (wenn du das Programm so ausführst, dann passiert zunächst einmal nichts). Ähnlich wie die Funktion `str_to_num` auf Seite 32 definieren wir eine Funktion, die aus einem Text einen Variablennamen (engl. «Identifier» für «Bezeichner») ausliest.

In Zeile 16 nutzen wir eine Spezialität von Python. Um ein Teilstück aus einem String oder einer Liste zu extrahieren, verwendest du die Syntax `text[anfang:ende]`. Achtung: Der Buchstabe beim Index «Ende» gehört nicht mehr dazu. `"Hallo"[1:4]` ergibt also «all» – das «o» an 4. Stelle gehört nicht mehr dazu.

Hin und wieder kommt es vor, dass eine Zeile in Python zu lang wird. Weil das Zeilenende aber grundsätzlich immer auch das Ende der Anweisung ist, musst du einen Backslash \ vor das Zeilenende setzen, um anzugeben, dass die Anweisung noch weitergeht.

```
1  # scanner.py
2  def is_digit(char):
3      return '0' <= char <= '9'
4
5  def is_letter(char):
6      return ('A' <= char <= 'Z') or ('a' <= char <= 'z') \
7             or (char in ['_'])
8
9  def is_letter_or_digit(char):
10     return is_letter(char) or is_digit(char)
11
12 def read_identifier(text, start):
13     i = start
14     while (i < len(text)) and is_letter_or_digit(text[i]):
15         i += 1
16     return text[start:i], i
```

Ein Modul unterscheidet sich eigentlich nicht von einem Hauptprogramm. Nachdem wir also das Programm oben als `scanner.py` gespeichert haben, importieren wir es wie üblich über **`import`**. Falls du dem Modul einen anderen Namen gegeben hast, so musst du natürlich auch hier den Namen anpassen.

```
1 # parser.py
2 import scanner
3
4 text = "meine_variable_1 = 123"
5 (name, i) = scanner.read_identifier(text, 0)
6 print name
```

Im Prinzip könntest du auch **`from`** `scanner` **`import`** `*` verwenden. Das hat aber den Nachteil, dass man im Programm viel schlechter sieht, woher die einzelnen Funktionen (hier z. B. `read_identifier`) stammen. Wenn du zudem in beiden Modulen jeweils eine Funktion mit dem selben Namen hast, dann wird eine davon überschrieben, was zu ziemlich unlesbarem Programmcode führt.

AUFGABEN

18. Verwende die Technik mit Modulen, um das bisherige Projekt sinnvoll aufzuteilen. Traditionellerweise gibt es ein Modul «scanner» (manchmal auch «lexer» oder «tokenizer») genannt, das den Text in eine Liste von Tokens umwandelt. Das nächste Modul «parser» erstellt aus der Liste von Tokens einen Strukturbaum. Das letzte Modul «interpreter» setzt die Rechnungen im Strukturbaum um und führt das Programm aus. Neben diesen drei Modulen gibt es dann ein Hauptprogramm, das selbst sehr wenig Code enthält, aber die drei anderen Module verwendet, um eine eingegebene Rechnung auszuwerten.

19. Füge auch die Funktion `read_identifier` aus dem Beispielprogramm oben zu deinem Projekt hinzu, so dass du nun auch Variablennamen einlesen kannst. Die Grammatik unserer Sprache wird dabei ein wenig verändert:

```
atom ::= ('+'|'-') atom | NUMBER | NAME
```

6 Programme aus mehrere Anweisungen

Lernziele In diesem Abschnitt lernst du:

- Ein Programm aus mehreren Anweisungen zu parsen und die Strukturbäume zu erstellen.
- Ein Programm aus mehreren Anweisungen auszuführen.

Einführung Bis jetzt kann unser Programm eine einzelne Rechnung einlesen und auswerten. Nun wollen wir das um zwei wichtige Komponenten erweitern. Zum einen soll die Eingabe aus mehreren Anweisungen bestehen – wobei jede Anweisung eine Rechnung sein kann. Zum anderen möchten wir auch Variablen definieren und verwenden können, um die Rechnungen zu vereinfachen.

In Bezug auf die Grammatik erweitern wir die Sprache wie folgt:

```
program ::= statement (';' statement)*
statement ::= assignment | rechnung
assignment ::= 'let' NAME '=' rechnung
```

Die erste Zeile bedeutet, dass ein Programm aus Anweisungen (engl. *statements*) besteht, die durch Semikolon getrennt werden. In Python steht hier anstelle des Semikolons eigentlich ein Zeilenumbruch `'\n'` – du kannst das für deine Sprache beliebig anpassen.

In Bezug auf das Semikolon gibt es einen kleinen feinen Unterschied: In einigen Programmiersprachen muss jede Anweisung mit einem Semikolon abgeschlossen werden. In der Grammatik hier trennt das Semikolon aber Anweisungen und die letzte Anweisung muss nicht mit einem Semikolon abgeschlossen sein.

Die zweite Zeile definiert eine Anweisung als entweder eine Zuweisung (engl. *assignment*) oder eine Rechnung. Das Schlüsselwort `let`, das wir hier verwenden kommt aus der englischen Formulierung «let x equal 3» für «sei x gleich 3».

Das Programm I: Parsen Um die Grammatik umzusetzen müssen wir die Funktionen zum Parsen anpassen. Während in den Beispielen früher die Funktion `parse()` einfach `parse_rechnung()` aufgerufen hat, so ist diese Funktion jetzt dafür verantwortlich, eine beliebig lange Liste von Anweisungen einzulesen.

Beachte dass die Funktion `parse()` nicht mehr direkt einen Strukturbaum (engl. *AST* für «abstract syntax tree») zurückgibt, sondern eine Liste von Strukturbäumen.

```
1 def match_token(token):
2     if peek() == token:
3         next()
4     else:
```

```
5           print "FEHLER: '" + str(token) + "' erwartet"
6
7   def parse():
8       result = []
9       while peek() is not None:
10          stmt = parse_statement()
11          result.append(stmt)
12          if peek() == ';':
13              next()
14          elif peek() is not None:
15              print "FEHLER: Semikolon erwartet"
16      return result
17
18  def parse_statement():
19      if peek() == 'let':
20          return parse_assignment()
21      else:
22          return parse_rechnung()
23
24  def parse_assignment():
25      match_token("let")
26      name = next()
27      match_token('=')
28      rechnung = parse_rechnung()
29      return ('let', name, rechnung)
30
31  def parse_rechnung():
32      return parse_summe()
```

Das Programm II: Auswertung Die Funktion `eval_tuple()` kennst du bereits aus den vorherigen Abschnitten. Allerdings haben wir einige Änderungen vorgenommen, so dass sich nun eine Liste von Anweisungen auswerten lässt.

Die Funktion `eval_program()` wertet eine Liste von Anweisungen aus, wie sie von der Funktion `parse()` geliefert wird. Der Rückgabewert von `eval_program()` ist der letzte Wert, der berechnet wird. `eval_stmt()` wertet eine einzelne Anweisung aus und prüft dabei, ob es eine Zuweisung mit `let` oder eine Rechnung ist.

Expression ist der englische Begriff für einen «Ausdruck» – also das, was wir bis jetzt als Rechnung bezeichnet haben. Die Funktion `eval_expr()` kann nun nicht nur Tupel, sondern auch Zahlen auswerten. Was noch fehlt: Wenn `expr` eine Variable ist, dann sollte diese Funktion den Wert der Variablen zurückgeben.

```
1   def eval_tuple(t):
2       if len(t) == 3:
3           op, left, right = t
4           left = eval_expr(left)
```

```
5           right = eval_expr(right)
6           if op == '+':
7               return left + right
8           elif op == '-':
9               return left - right
10          elif op == '*':
11              return left * right
12          elif op == '/':
13              return left / right
14
15  def eval_expr(expr):
16      if type(expr) is tuple:
17          return eval_tuple(expr)
18      elif type(expr) is int or type(expr) is float:
19          return expr
20      else:
21          return 0
22
23  def eval_stmt(stmt):
24      if type(stmt) is tuple and stmt[0] == 'let':
25          op, left, right = stmt
26          right = eval_expr(right)
27          print "LET", left, "=", right
28          return None
29      return eval_expr(stmt)
30
31  def eval_program(ast_list):
32      result = None
33      for stmt in ast_list:
34          result = eval_stmt(stmt)
35      return result
```

AUFGABEN

20. Füge die obigen Teile mit deinem bisherigen Projekt zusammen, so dass du «Programme» mit mehreren Anweisungen korrekt abarbeiten und auswerten kann.

21. Die Sprache erlaubt im Moment zwar, Variablen zu definieren. Tatsächlich definiert werden die Variablen dann aber nicht. Baue die Funktionen `eval_expr()` und `eval_stmt()` so aus, dass man für die drei Variablen x, y und z Werte definieren und verwenden kann.

Hinweis: Du wirst dafür wahrscheinlich `global` verwenden müssen.

22.* Ändere die Sprache so ab, dass jede Anweisung mit einem Semikolon abgeschlossen werden muss.

7 Variablen definieren

Lernziele In diesem Abschnitt lernst du:

- Im Programm beliebige Variablen zu definieren und auszuwerten.
- Mit Tabellen und Wörterbüchern zu arbeiten.

Einführung Unsere Sprache erlaubt es zwar, mit `let` neue Variablen zu definieren und diese dann auch zu verwenden. Allerdings hast du erst einmal nur die drei Variablen x, y und z zur Verfügung.

Um beliebige Variablen zu definieren verwenden wir ein *Dictionary* (engl. für «Wörterbuch»). Das ist eine der wichtigsten Datenstrukturen in Python! Ein Dictionary ist eine Tabelle, die für jeden Schlüssel (*key*) einen Wert (*value*) enthält.

In anderen Programmiersprachen heissen Dictionaries auch «table», «map», «hashtable» oder manchmal «associative array».

Key	Value
Merkur	4 879.4
Venus	12 103.6
Erde	12 756.3
Mars	6 752.4

In Python schreiben wir diese Tabelle/Wörterbuch so:

```
planets_diameter = {
  "Merkur": 4879.4, "Venus": 12103.6,
  "Erde": 12756.3, "Mars": 6752.4
}
print "Durchmesser der Venus:", planets_diameter["Venus"]
```

Das Programm Wir zeigen dir hier nur einen kleinen Teil des ganzen Projekts. Du siehst an diesem Beispiel aber, wie wir ein Dictionary `variables` verwenden, um die Werte von Variablen zu speichern. Die erste Variable, die wir bereits definiert haben ist natürlich π.

Das definieren neuer Variablen ist sehr einfach. Mit `variables["x"] = 123` wird zum Beispiel ein neuer Eintrag mit dem Schlüssel `"x"` und dem Wert 123 angelegt. Bevor wir aber einen Eintrag auslesen müssen wir mit `if "x"in variables:` prüfen, ob das Dictionary überhaupt einen entsprechenden Eintrag enthält (Zeile 15).

```
1 import math
2 variables = { 'pi': math.pi }
3
4 def define_variable(name, value):
```

```
5       variables[name] = value
6
7   def eval_expr(expr):
8       if type(expr) is tuple:
9           return eval_tuple(expr)
10      elif type(expr) is int or type(expr) is float:
11          return expr
12      elif type(expr) is str:
13          if expr in variables:
14              return variables[expr]
15          else:
16              print "FEHLER: Unbekannte Variable", expr
17              return 0
18      else:
19          print "FEHLER: Unbekannter Wert"
20          return 0
21
22  def eval_stmt(stmt):
23      if type(stmt) is tuple and stmt[0] == 'let':
24          op, left, right = stmt
25          right = eval_expr(right)
26          define_variable(left, right)
27          return None
28      return eval_expr(stmt)
```

AUFGABEN

23. Füge die neuen Teil in dein Projekt ein, so dass du nun beliebige Variablen definieren und verwenden kannst.

24. Einige Variablen wie `pi` sollte man nicht überschreiben können. Ändere die Funktion `define_variable()` so ab, dass man `pi` nicht neu definieren kann, sondern eine Fehlermeldung erhält.

Daneben gibt es auch andere Variablennamen, die man nicht verwenden darf, z. B. `let`. Schütze auch diese entsprechend.

25. Erweitere deine Sprache um eine **`print`**-Anweisung, die einen Wert auf den Bildschirm ausgibt. Ob du das in Python selbst mit **`print`** oder `msgDlg` umsetzst ist natürlich dir überlassen.

26.* Erweitere deine Sprache um eine Anweisung `read x`, die z. B. mit `input()` einen Wert einliest und in der Variable x speichert.

Schreibe schliesslich in deiner eigenen Sprache ein Programm, um quadratische Gleichungen zu lösen. Es liest also die drei Koeffizienten a, b und c von $ax^2 + bx + c = 0$ ein und berechnet daraus die beiden Lösungen x_1 und x_2.

8 Blöcke und Verzweigungen

Lernziele In diesem Abschnitt lernst du:

- In der Sprache Verzweigungen mit `if` und Blöcke von Anweisungen zu unterstützen.

Einführung Eines der wichtigsten Elemente in einer Programmiersprache ist eine Verweigung, z. B. mit `if`. Damit diese Verzweigung aber auch sinnvoll funktioniert brauchen wir zwei Zutaten: Zum einen müssen wir die Sprache so erweitern, dass wir Vergleiche ausdrücken können. Zum anderen brauchen wir Programm-Blöcke.

```
statement ::= assignment | conditional | rechnung
conditional ::= 'if' vergleich statement
vergleich ::= summe ('='|'<'|'>') summe
```

Insgesamt soll also die Grammatik für unsere Sprache so aussehen:

```
program ::= statement (statement)*
statement ::= assignment ';' | conditional ';'
            | rechnung ';' | block
block ::= '{' statement (';' statement)* '}'
conditional ::= 'if' vergleich statement
assignment ::= 'let' NAME '=' rechnung
vergleich ::= summe ('='|'<'|'>') summe
rechnung ::= summe
summe ::= produkt (('+' | '-') produkt)*
produkt ::= faktor (('*' | '/') faktor)*
faktor ::= atom
atom ::= ('+'|'-') atom | NUMBER | NAME
```

Beachte, dass wir hier das Semikolon nicht mehr verwenden, um Anweisungen zu trennen. Vielmehr muss jetzt jede Anweisung, ausser Blöcken, mit einem Semikolon abgeschlossen werden.

Das Programm I: Parsen Um die Verzweigungen und Blöcke parsen zu können erweitern wir die Funktionen `parse_statement()` und `parse_rechnung()`.

```
1 def parse_statement():
2     if peek() == 'let':
3         result = parse_assignment()
4     elif peek() == 'if':
5         result = parse_if()
6     elif peek() == '{':
```

```
7           return parse_block()
8       else:
9           result = parse_rechnung()
10      match_token(';')
11      return result
12
13  def parse_if():
14      match_token('if')
15      cond = parse_rechnung()
16      match_token('then')
17      stmt = parse_statement()
18      return ('if', cond, stmt)
19
20  def parse_block():
21      match_token('{')
22      result = []
23      while peek() != '}':
24          stmt = parse_statement()
25          result.append(stmt)
26      match_token('}')
27      return result
28
29  def parse_vergleich():
30      left = parse_summe()
31      op = next()
32      if op not in ['=', '<', '>']:
33          print "FEHLER: Vergleichsoperator erwartet"
34      right = parse_summe()
35      return (op, left, right)
```

Das Programm II: Auswerten Um das eingelesene Programm auszuwerten braucht es zwei Anpassungen. Erstens muss die Funktion `eval_tuple()` auch die Vergleiche richtig auswerten können. Zweitens muss `eval_stmt()` sowohl mit `if`-Verzweigungen als auch mit Blöcken richtig umgehen können. Beachte, dass wir Blöcke intern einfach als Liste von Anweisungen darstellen.

```
1   def eval_tuple(t):
2       if len(t) == 3:
3           op, left, right = t
4           left = eval_expr(left)
5           right = eval_expr(right)
6           if op == '+':
7               return left + right
8           #...
9           elif op == '=':
10              return left == right
11          elif op == '<':
12              return left < right
13          elif op == '>':
```

```
14              return left > right
15
16  def eval_stmt(stmt):
17      if type(stmt) is tuple:
18          if stmt[0] == 'let':
19              op, left, right = stmt
20              right = eval_expr(right)
21              variables[left] = right
22              print "LET", left, "=", right
23              return right
24          if stmt[0] == 'if':
25              op, cond, stmt = stmt
26              cond = eval_expr(cond)
27              if cond:
28                  eval_stmt(stmt)
29      if type(stmt) is list:
30          return eval_program(stmt)
31      return eval_expr(stmt)
```

AUFGABEN

27. Baue auch die neuen Teile wiederum in dein Projekt ein und teste es mit einigen Beispielen aus.

28. Führe eine einfache Schleife mit folgender Grammatik ein:

```
loop ::= 'repeat' NUMBER STATEMENT
```

Zum Beispiel könnte man dann mit dem folgenden Programm die drei Zahlen 1, 2, 3 ausgeben lassen.

```
let x = 0; repeat 3 { let x = x+1; print x; };
```

29.* Ergänze dein Programm um die Vergleichsoperatoren `<=`, `>=`, `!=`.

30.* Ergänze dein Programm um eine **`while`**-Schleife. Achtung: Du kannst zwar die Syntax für die **`if`**-Verzweigung von oben übernehmen. Bei der Auswertung in `eval_stmt` muss `cond` aber vor jeder Wiederholung nochmals neu mittels `eval_expr` ausgewertet werden!

31.* Ergänze das Programm um eine **`if-else`**-Anweisung. Dafür kann die **`if`**-Verzweigung noch einen optionalen (freiwilligen) **`else`**-Teil haben. In der Grammatik wird ein optionaler Teil in eckige Klammern geschrieben.

```
conditional ::= 'if' vergleich statement ['else' statement]
```

32.* Füge deiner Sprache mindestens die Grundanweisungen `forward`, `left` und `right` hinzu, um die Turtle zu steuern. Achtung: Jeder dieser drei Anweisungen braucht natürlich auch einen Parameter!

K a p i t e l D r e i

PROGRAMME AUSFÜHREN

Nachdem du weisst, wie du eine Eingabe analysieren und auswerten kannst, widmen wir uns in diesem Kapitel vor allem der Frage, wie du auch komplexere Programme ausführen kannst. Die einfachen Techniken aus dem letzten Kapitel stossen relativ schnell an ihre Grenzen.

Als Grundmodell für die Ausführung verwenden wir eine Stack-Machine. Zunächst einmal arbeiten wir mit dem «PostScript»-Format, das auf graphische Darstellungen ausgelegt ist. Gegen Ende des Kapitels wirst du dann aber genug Grundlagen kennen, um auch ein einfaches Modell der Java Virtual Machine programmieren zu können.

1 Werte stapeln

Lernziele In diesem Abschnitt lernst du:

- ▷ Mit der umgekehrten polnischen Notation zu arbeiten.
- ▷ Mit einem Stack Rechnungen auszuführen.

Einführung Erinnerst du dich an die «polnische Notation», in der Rechnungen immer so dargestellt werden, dass zuerst der Operator kommt und danach die dazugehörigen Werte. Die Rechnung $3+4$ wird dann zu $(+,3,4)$. Anstelle der polnischen Notation können wir auch die *umgekehrte polnische Notation* verwenden, d. h. die Rechnung $3+4$ wird dann zu $(3,4,+)$.

Der grosse Vorteil der umgekehrten polnischen Notation *UPN* besteht darin, dass sich sehr einfach Machinen bauen lassen, die genauso rechnen. Tatsächlich verwendet eine solche Machine eine Liste mit Werten («stack» genannt) und arbeitet ein Programm dann nach folgenden Regeln ab:

(R1) Wenn eine Zahl kommt, dann füge sie ans Ende der Liste an.

(R2) Wenn ein (binärer) Operator kommt, dann nimm die zwei letzten Werte aus der Liste, verrechne sie und füge sie wieder zur Liste hinzu.

Lass uns das am Beispielprogramm `3 4 5 + *` ansehen. Wir geben für jedes Eingabe-Symbol in diesem Programm an, wie der *stack* aussieht und was passiert:

1. Symbol: `3`, Stack: [] – Die Zahl wird zum Stack hinzugefügt (R1).
2. Symbol: `4`, Stack: $[3]$ – Die Zahl wird zum Stack hinzugefügt (R1).
3. Symbol: `5`, Stack: $[3,4]$ – Die Zahl wird zum Stack hinzugefügt (R1).
4. Symbol: `+`, Stack: $[3,4,5]$ – Die beiden letzten Werte werden vom Stack genommen, addiert und das Resultat wird wieder auf den Stack gelegt (R2).
5. Symbol: `*`, Stack: $[3,9]$ – Die beiden letzten Werte werden vom Stack genommen, multipliziert und das Resultat wird wieder auf den Stack gelegt (R2).

Am Ende enthält der Stack einfach den Wert 27, d. h. der Stack ist $[27]$.

Das Programm Dieses Programm führt die Rechnung aus dem Beispiel durch. Wir haben hier allerdings anstatt den Operationszeichen `+` und `*` die Operationen als `add` und `mul` bezeichnet. Das ändert allerdings nichts am Prinzip.

Wichtig sind die zwei Listen-Methoden `append` und `pop`. Mit `append` wird ein Element ans Ende der Liste angehängt. Mit `pop` wird das letzte Element der Liste von der Liste entfernt und zurückgegeben.

```
1  program = [3, 4, 5, "add", "mul"]
2  stack = []
3
4  for item in program:
5      if type(item) is int:
6          stack.append(item)
7      elif type(item) is str:
8          if item == "add":
9              b = stack.pop()
10             a = stack.pop()
11             stack.append(a + b)
12         elif item == "mul":
13             b = stack.pop()
14             a = stack.pop()
15             stack.append(a * b)
16 print stack
```

Mit `if type(item) is int:` prüfen wir im Programm, ob der Wert der Variable `item` den Datentyp `int` (für «Integer», d. h. ganze Zahl) hat. So können wir unterscheiden, ob der gegebene Wert eine Zahl ist, und sie dann auf den Stack legen.

Der Stack Der Stack arbeitet wie ein Stapel (tatsächlich heisst «stack» Stapel). Das letzte Element, das auf den Stack gelegt wurde, wird auch als erstes wieder entfernt. Das funktioniert wie bei einem Stapel Teller. Du nimmst jeweils den obersten Teller als erstes wieder weg – also genau den, den du als letztes auf den Stapel gelegt hast. Dieses «Last In First Out»-Prinzip (LIFO) ist das bestimmende Merkmal eines Stacks.

Das Gegenstück wäre übrigens ein *Buffer*, der nach dem «First In First Out»-Prinzip (FIFO) arbeitet. Zum Beispiel verwendet die Tastatur einen solchen Buffer: Die Reihenfolge, in der die Tasten gedrückt werden soll beibehalten werden. Beim Stack wird die Reihenfolge hingegen umgedreht.

Der Stack gehört zu den wichtigsten Datenstrukturen in der Informatik. Zum Beispiel arbeiten sowohl die Java Virtual Machine (JVM), als auch das PDF-Format nach genau diesem Prinzip. In PDF würdest du eine Linie im Prinzip so zeichnen/programmieren:

```
30 42 moveto 150 60 lineto
```

Damit würde eine Linie vom Punkt $(30, 42)$ zum Punkt $(150, 60)$ gezeichnet. Der folgende Code erreicht genau dasselbe:

```
150 60 30 42 moveto lineto
```

(Tatsächlich verwendet PDF allerdings für die Befehle Kürzel wie `m` und `l`, um die Dateigrösse klein zu halten).

Terminologie Weil der Stack ja ein «Stapel» ist, spricht man davon, dass Werte *auf den Stack gelegt werden* oder *vom Stack genommen werden*. Im Englischen bezeichnet man das hinzufügen von Werten auch als `push` und das entfernen eines Wertes als `pop`.

AUFGABEN

1. Beantworte ohne Computer: Was ist das Resultat dieser «Programme» in UPN?

(a) `15 7 - 3 7 * +` (b) `96 5 1 8 - - /`

2. Erweitere das Beispielprogramm so, dass auch die Subtraktion mit dem Befehl `sub` unterstützt wird. Achte darauf, dass das «Programm» `8 5 sub` als Ergebnis 3 angibt und nicht -3. Dazu musst du dir also überlegen, welchen Wert du zuerst vom Stack nimmst und ob du $a - b$ oder $b - a$ rechnen musst.

3. Erweitere das Beispielprogramm so, dass es neben Integerwerten (ganzen Zahlen, Datentyp `int`) auch gebrochene Zahlen verarbeiten kann (Datentyp `float`). Zudem sollen auch die Befehle `div`, `mod` und `neg` funktionieren. `div` dividiert zwei Zahlen, `mod` berechnet den Rest und `neg` negiert die letzte Zahl.

4. Üblicherweise gehören zum Stack nicht nur arithmetische Befehle (d. h. solche zum Rechnen), sondern auch `pop` (entfernen/löschen des letzten Elements), `exch` (vertauschen der letzten zwei Elemente), sowie `dup` (verdoppeln/duplizieren des letzten Elements). Implementiere auch diese drei Befehle.

5. Schreibe ein Programm, dass eine Rechnung von der polnischen Notation in die umgekehrt polnische Notation umschreibt. Insbesondere soll also aus dem Ausdruck

$$(+, (*, (-, 15, 7), 3), (/, 60, -3))$$

das «Programm»

```
[15, 7, "sub", 3, "mul", 60, -3, "div", "add"]
```

entstehen. Das Resultat der Rechnung wäre natürlich 4.

2 PostScript

Lernziele In diesem Abschnitt lernst du:

- Einen einfachen Interpreter zu schreiben, der Programme ausführen kann, die ein Bild zeichnen.
- Mit dem Stack Variablen zu definieren und zu verwenden.

Einführung Die Sprache «PostScript» wurde entwickelt, um die genaue Darstellung einer Seite angeben zu können. Mit einem PostScript-Programm wird also genau angegeben, wo auf der Seite was gezeichnet oder geschrieben werden soll. Später wurde PostScript dann vereinfacht und verkürzt (wie erwähnt wurde z. B. aus «moveto» einfach «m»), und so entstand das PDF-Format. Im Grunde genommen ist ein PDF-Dokument aber noch immer ein Programm, das vom PDF-Reader (oder Drucker) ausgeführt wird.

Wir arbeiten hier mit einer etwas vereinfachten Variante von PostScript. Das Ziel soll aber sein, einen einfachen PostScript-Interpreter zu schreiben (damit also auch die Vorstufe eines PDF-Readers). Dazu soll unser Interpreter neben `add`, `sub`, `mul`, `div`, etc. (siehe letzter Abschnitt) folgende graphische Befehle verstehen:

`x y moveto`	An die Position (x, y) springen.
`x y lineto`	Linie zum Punkt (x, y) ziehen.
`x y rlineto`	Linie zum relativen Punkt $(+x, +y)$ ziehen.
`r g b setcolor`	Farbe (r, g, b) setzen.
`w setlinewidth`	Linienbreite setzen.
`x show`	Den Wert x auf dem Bildschirm ausgeben.

Mit `rlineto` ist folgendes gemeint: Wenn die aktuelle Position gerade $(15, 7)$ ist, dann zeichnet `18 9 rlineto` eine Linie zum Punkt $(15 + 18, 7 + 9) = (33, 16)$.

Am einfachsten verwendest du für die Implementation (Umsetzung) die Turtle-Graphik. Die Turtle kennt bereits Befehle `setPos` und `moveTo` zum Setzen der Position und Zeichnen der Linie. Mit `label()` kannst du zudem Texte ins Turtle-Fenster schreiben.

AUFGABEN

6. Ergänze das Programm aus dem letzten Abschnitt um die angegebenen graphischen Befehle.

Variablen definieren In PostScript lassen sich natürlich auch Variablen definieren und verwenden. Das folgende PostScript-Programm definiert in der ersten Zeile die Variable x. In der zweiten Zeile wird die Variable dann verwendet.

```
/x 123 def
x 23 sub 10 div
```

Warum braucht es bei der Definition den Schrägstrich `/`? Überlege dir dazu, was der folgende Programmausschnitt bewirken könnte:

```
/x x 1 add def
```

In PostScript gilt: Wenn im Programm eine Variable auftritt, dann wird nicht die Variable, sondern ihr Wert auf den Stack gelegt (gepusht). Wenn du den Namen der Variable auf den Stack legen möchtest, dann stellst du einen Schrägstrich davor, um diese automatische Ersetzung zu verhindern. Angenommen, die Variable x habe den Wert 123. Dann gilt:

Symbol	Wirkung auf den Stack
`/x`	`[...] -> [..., "x"]`
`x`	`[...] -> [..., 123]`

Das Programm Neben dem Stack und dem «Programm» verwenden wir ein Dictionary für die Variablen. Um das Beispiel kurz zu halten gibt es als einzige Rechenoperation hier `add`. Dafür lassen sich mit `def` Variablen definieren.

```
1  program = ["/myvar", 123, "def", "myvar", -23, "add"]
2  stack = []
3  variables = {}
4
5  for item in program:
6      if type(item) is int:
7          stack.append(item)
8      elif type(item) is str:
9          if item == "add":
10             b = stack.pop()
11             a = stack.pop()
12             stack.append(a + b)
13         elif item == "def":
14             value = stack.pop()
15             name = stack.pop()
16             variables[name] = value
17         else:
18             if item[0] == '/':
19                 stack.append(item[1:])
20             else:
21                 stack.append(variables[item])
22 print stack
```

Beachte: In Python gibt `item[0]` das erste Zeichen des Strings `item` zurück. Umgekehrt gibt `item[1:]` alles von der Position «1» an, also alles ohne das erste Zeichen (das an Position «0» steht).

AUFGABEN

7. Füge alles zusammen, so dass du einen soweit vollständigen Interpreter hast, der alle arithmetischen Operationen und die Graphik-Befehle, sowie beliebige Variablen definieren und verwenden kann. Füge auch noch Befehle wie `sqrt` oder `sin` hinzu, um das Ganze abzurunden.

8. Im Moment fehlen vor allem Fehlerausgaben. Zum Beispiel kann es ja sein, dass eine Variable gar nicht definiert ist. Oder bei `def` wurde kein Name angegeben. Oder bei `add` ist nur ein einziger Wert auf dem Stack. In diesen Fällen muss dein Programm eine sinnvolle Fehlermeldung ausgeben!

9. Füge einen Befehl `shift` hinzu, der die letzten n Elemente auf dem Stack um Eins rotiert. Mit `3 shift` würden die drei letzten Elemente rotiert und `2 shift` hätte die gleiche Wirkung wie `exch`. Beispiel zu `3 shift`:

$$[1,2,3,4,5,6,7] \rightarrow [1,2,3,4,7,5,6]$$

10. Tatsächlich wird ein PostScript-Programm nicht als Liste angegeben, sondern als Text bzw. String. Als Trennzeichen zwischen den Symbolen steht jeweils eine beliebige Anzahl von Leerzeichen und/oder Zeilenumbrüchen. Im Beispielprogramm oben würde das PostScript-Programm eigentlich so definiert sein:

```
program = "/myvar 123 def\nmyvar -23 add"
```

Schreibe den PostScript-Interpreter so, dass er als Eingabe-Programm einen Text/String akzeptiert anstelle der Liste. Der Interpreter erstellt die Liste also zunächst einmal selbst. Dazu brauchst du die ganzen Techniken, die du im letzten Kapitel gelernt hast.

11. Im letzten Kapitel hast du gelernt, beliebige mathematische Ausdrücke wie $3 \cdot (5+8)$ zu analysieren. Verwende die dort gelernten Techniken, um einen «Compiler» zu bauen, der aus einem solchen mathematischen Ausdruck ein PostScript-Programm erstellt. Das Beispiel hier würde also zum Programm `3 5 8 add mul`.

Du kannst natürlich neben mathematischen Ausdrücken auch eine umfangreichere Sprache in das PostScript-Format compilieren.

3 Eigene Funktionen definieren

Lernziele In diesem Abschnitt lernst du:

- Wie du in einem PostScript-Programm eigene Funktionen definieren und aufrufen kannst.

Einführung: Code ersetzen Eine besonders wichtige Technik in jeder Programmiersprache ist die Möglichkeit, eigene Funktionen oder Unterprogramme zu definieren. Damit lässt sich der Wortschatz einer Programmiersprache beliebig erweitern und an das Problem anpassen.

In PostScript funktionieren selbst definierte Funktionen nach einem «Einsetzungs-Prinzip» (in der Regel als «Macros» bezeichnet). Die Idee ist, dass der Name f einer Funktion im Code durch die Definition ersetzt wird.

Ein einfaches Beispiel: Nehmen wir an, wir haben eine eigene Funktion `sqr` geschrieben, die die oberste Zahl auf dem Stack quadriert. Das machen wir mit der folgenden Sequenz: `dup mul`. Die oberste Zahl wird mit `dup` dupliziert und dann werden die beiden obersten Zahlen miteinander multipliziert. Der Interpreter ersetzt nun im folgenden Programm den Namen `sqr` durch die beiden Befehle `dup mul`:

```
[123, "sqr", 1, "sub"]  ->  [123, "dup", "mul", 1, "sub"]
```

Das Programm I: Code einsetzen Die eigenen Funktionen – in diesem Fall «sqr» – speichern wir einfach auch im Dictionary der Variablen. Der «Code» von Funktionen wird immer eine Liste sein, auch wenn diese Liste dann vielleicht nur ein einziges Element enthält.

Um den Code dann tatsächlich einsetzen zu können, haben wir den Umgang mit dem Programm selbst geändert. Der Interpreter entfernt jeden Befehl vor der Abarbeitung aus der `program`-Liste (das geschieht mit `pop(0)` in Zeile 6). Wenn wir jetzt nämlich neuen Code einfügen wollen, dann hängen wir ihn einfach am Anfang der Liste an (Zeile 26).

```
1 program = [11, "sqr", 1, "sub"]
2 stack = []
3 variables = { "sqr": ["dup", "mul"] }
4
5 while len(program) > 0:
6     item = program.pop(0)
7     if type(item) is int:
8         stack.append(item)
```

```
    elif type(item) is str:
        if item == "sub":
            b = stack.pop()
            a = stack.pop()
            stack.append(a - b)
        elif item == "mul":
            b = stack.pop()
            a = stack.pop()
            stack.append(a * b)
        elif item == "dup":
            stack.append(stack[-1])
        else:
            if item[0] == '/':
                stack.append(item[1:])
            else:
                value = variables[item]
                if type(value) is list:
                    program = value + program
                else:
                    stack.append(value)
print stack
```

Das Programm II: Code definieren Den Code für eine Funktion definierst du in PostScript mit `def` wie schon Variablen zuvor. Allerdings folgt nach dem Namen ein *Code-Block* in geschweiften Klammern. Für die «sqr»-Funktion oben sieht das so aus:

```
/sqr { dup mul } def
```

Die geschweiften Klammern haben zwei Bedeutungen. Erstens werden die Elemente zwischen den Klammern in eine Liste verpackt. Zweitens verhindern die geschweiften Klammern, dass diese Elemente (in diesem Fall `dup` und `mul`) schon bei der Definition ausgeführt werden.

Im Programm haben wir das so umgesetzt, dass die Funktion `read_code()` die Elemente im Programm direkt in einer Liste speichert, ohne diese Elemente auszuführen.

```
program = ["/sqr", "{", "dup", "mul", "}", "def",
           13, "sqr", 1, "sub"]
stack = []
variables = { }

def read_code():
    result = []
    while len(program) > 0:
        item = program.pop(0)
        if item == "}":
```

```
            break
        else:
            result.append(item)
    return result

while len(program) > 0:
    item = program.pop(0)
    if type(item) is int:
        stack.append(item)
    elif type(item) is str:
        if item == "sub":
            ...
        elif item == "mul":
            ...
        elif item == "dup":
            stack.append(stack[-1])
        elif item == "{":
            stack.append(read_code())
        elif item == "def":
            value = stack.pop()
            name = stack.pop()
            variables[name] = value
        else:
            if item[0] == '/':
                stack.append(item[1:])
            else:
                value = variables[item]
                if type(value) is list:
                    program = value + program
                else:
                    stack.append(value)
print stack
```

AUFGABEN

12. Erweitere deinen eigenen Interpreter so, dass du beliebige Funktionen definieren und anwenden kannst.

13. Mit der Technik des Code-Blocks kannst du nun auch Schleifen bauen. Erweitere deinen Interpreter um eine einfache `repeat`-Schleife, die einen Code-Block n Mal wiederholt. Das Resultat des folgenden Beispiels wäre dann $390\,625 = \left((5^2)^2\right)^2$:

```
5 3 { dup mul } repeat
```

4 Referenzen auf Funktionen

Lernziele In diesem Abschnitt lernst du:

- Mit Funktionen als Werte zu arbeiten.
- Funktionen mit der Lambda-Darstellung direkt einzubetten.

Einführung Bei vielen Alternativen kann eine `if-elif-else`-Struktur in Python ganz schön lang und unübersichtlich werden. In solchen Fällen kann es ein grosser Gewinn sein, den Code so umzuschreiben, dass kaum noch `if`-Anweisungen vorkommen. Stattdessen soll Python in einer Tabelle nachschlagen, welcher Code ausgeführt werden soll.

In Python kannst du mit Funktionen arbeiten wie mit anderen Variablen auch. Zum Beispiel kannst du eine Funktion «kopieren», in diesem Beispiel z. B. `sqrt`:

```
root = sqrt
print root(36)
```

Die Klammern hinter dem Namen entscheiden dabei, ob die Funktion aufgerufen und ausgeführt, oder nur als *Referenz* (Verweis) verwendet wird.

Dieses Konzept von Referenzen oder «Funktionen als Werte» erlaubt uns auch, verschiedene Funktionen in einer Liste zusammenzufassen. Das folgende Beispielprogramm hat keinen tiefen Sinn, illustriert aber, wie man zwei Funktionen als Werte in einer Liste speichert und dann eine der Funktionen aus der Liste ausführt, um schliesslich den Wert 4 auszugeben.

```
def inc(x):
    return x + 1

def dec(x):
    return x - 1

functions = [inc, dec]
f = functions[0]
print f(3)
```

Das Programm I Du erkennst sofort die Grundelemente wie den Stack oder das PostScript-Programm, das ausgeführt werden soll. Um

das Beispiel einfach zu halten haben wir den Stack bereits mit Zahlenwerten gefüllt. Zudem werden nur zwei Operationen unterstützt: Die Addition und die Multiplikation.

In den Zeilen 4 und 7 werden zunächst zwei Funktionen definiert. In Zeile 10 erstellen wir dann eine Tabelle (ein Dictionary), das zu drei Strings die entsprechende Referenz auf eine Funktion enthält. Damit ist `operations["add"]` gleichwertig wie ein einfaches `add`. Um die zwei Zahlen 3 und 4 zu addieren könnten wir also schreiben:

```
print operations["add"](3, 4)
```

Das mag etwas umständlich erscheinen. Ab Zeile 16 wird aber die Mächtigkeit dieser Technik deutlich. Das einzige `if`, das wir noch brauchen prüft, ob der Befehl (`"add"` bzw. `"mul"` in unserem Fall) überhaupt definiert ist. Wenn ja, dann wird die Funktion aus dem Dictionary geholt (Zeile 17) und in Zeile 20 ausgeführt.

```
1  program = ["add", "mul", "add"]
2  stack = [1, 2, 3, 4]
3
4  def add(x, y):
5      return x+y
6
7  def mul(x, y):
8      return x*y
9
10 operations = {     # Uebersetzt einen String
11     "add": add,    # in eine Funktion
12     "mul": mul
13 }
14
15 for command in program:
16     if command in operations:
17         function = operations[command]
18         y = stack.pop()
19         x = stack.pop()
20         result = function(x, y)
21         stack.append(result)
22     else:
23         print "Unbekannter Befehl:", command
24 print stack
```

AUFGABEN

14. Erweitere das Beispiel-Programm um weitere Funktionen wie die Subtraktion oder Division. Baue es so aus, dass auch Zahlen im PostScript-Programm vorkommen dürfen und dann auf den Stack gelegt werden.

Die Lambda-Notation Das Beispiel-Programm I hat einen Schönheits-Fehler. Die Technik mit dem Dictionary und den Funktions-Referenzen erfordert, dass Du viele kleine Funktionen definierst. Anders gesagt: Für eine einfache Addition brauchen wir ganz schön viele Code-Zeilen.

Als Abhilfe gibt es die Möglichkeit, eine Funktion direkt im Dictionary zu definieren. Eine solche Funktion hat dann keinen eigentlichen Namen und kann nur über das Dictionary aufgerufen werden. Das funktioniert auch nur mit Kleinstfunktionen, die direkt einen Wert ausrechnen.

Der Fachbegriff für solche namenlose Kleinstfunktionen ist *Lambda-Funktion* (nach dem griechischen Buchstaben λ). Das widerspiegelt auch die Syntax von Python:

```
lambda x, y: x + y
```

In der Mathematik würdest du das als $\lambda(x, y) = x + y$ schreiben. Und natürlich kannst du die Parameter auch ganz anders bennenen oder ihre Anzahl ändern:

```
lambda t: t**2
```

Das wäre die Quadrat-Funktion $\lambda(t) = t^2$. Traditionell in Python würdest du dafür schreiben (wir verwenden die Fragezeichen, um anzudeuten, dass `lambda`-Funktionen keinen Namen haben):

```
def ???(t):
    return t**2
```

Das Programm II Das Programm ist immer noch das gleiche. Lediglich das Dictionary mit den Funktionen haben wir jetzt kürzer geschrieben, indem die eigentlichen Funktionen direkt eingebettet werden.

```
1  program = ["add", "mul", "add"]
2  stack = [1, 2, 3, 4]
3
4  operations = {
5      "add": lambda x, y: x+y,
6      "mul": lambda x, y: x*y
7  }
8
9  for command in program:
10     if command in operations:
11         function = operations[command]
12         y = stack.pop()
13         x = stack.pop()
14         result = function(x, y)
15         stack.append(result)
16     else:
17         print "Unbekannter Befehl:", command
18
19 print stack
```

Natürlich kannst du Lambda-Funktionen nicht nur in Dictionaries verwenden, sondern etwa auch, um ganz reguläre Funktionen zu definieren. Die folgenden zwei Definition für die Quadratfunktion `sqr` sind zum Beispiel gleichwertig.

```
sqr = lambda x: x**2

def sqr(x):
    return x**2
```

Theoretisch dürfen Lambda-Funktionen auch beliebig komplex sein. Aber guter Programmierstil heisst, dass du sie nur für kleine Aufgaben verwendest und grössere Funktionen nach wie vor mit **`def`** definierst.

AUFGABEN

15. Baue die neue Technik in deinen Interpreter ein. Funktionen, die etwas ausführen und nicht etwas berechnen (wie `"def"` in PostScript) kannst du nicht sinnvoll als Lambda-Funktion schreiben. Achte also darauf, deinen Code nur dort zu optimieren, wo es auch wirklich eine Verbesserung ist.

16. Soweit basiert diese Technik darauf, dass alle Funktionen zwei Parameter haben. Wie könntest du trotzdem auch Funktionen wie `neg` einbauen, die nur ein Argument haben? Suche nach einer möglichst eleganten Möglichkeit.

Hier einige Ideen: Eine Funktion könnte auch zwei Werte zurückgeben, z. B. so:

```
lambda a, b: [a+1, b+1]
```

Oder du verwendest den Ansatz:

```
lambda stack: stack.pop() + stack.pop()
```

Prüfe deine Ausarbeitung auch immer auf Korrektheit!

17.* Mit Lambda-Funktionen kannst du sehr einfach verschachtelten Code schreiben, der kaum noch lesbar ist. Dafür brauchen wir im vorliegenden Fall kein einziges **`if`** und unterscheiden trotzdem zwei Fälle.

Untersuche die hier definierte Funktion $f(z)$ und schreibe sie in gleichwertigen, aber verständlichen «traditionellen» Python-Code um, der keine Lambda-Funktionen verwendet.

```
f = lambda z: [lambda x: 0, lambda x: x**0.5][z >= 0](z)
```

5 Reflection

Lernziele In diesem Abschnitt lernst du:

- Während der Laufzeit Informationen über das Programm selber zu ermitteln.

Einführung Die Verwendung der Funktionsreferenzen im letzten Abschnitt ist mit einem Problem verbunden. Damit diese Technik funktioniert müssen alle verwendeten Funktionen die gleiche Anzahl an Parametern haben (vgl. Aufgabe 3.16). Python bietet aber die Möglichkeit, dass dein Programmcode selbst ermitteln kann, wie viele Parameter eine Funktion tatsächlich hat.

Die Technik, wenn ein Programm den eigenen Code analysiert heisst *Reflection*. In Python hat z. B. jede Funktion `f` ein «verstecktes» Feld `__code__`, das unter anderem die Anzahl der Parameter enthält. Genau das nutzen wir im Programm unten aus.

Das Programm Das Beispielprogramm enthält gerade genug, um den Flächeninhalt eines Kreises zu berechnen. Für einen Radius r (hier 12) wird die Formel $A = r^2 \cdot \pi$ angewendet.

In Zeile 13 prüft das Programm zunächst ob `function` überhaupt *aufrufbar* ist (engl. «callable»). Falls es keine Funktion ist, dann wird der Wert in Zeile 19 direkt als Resultat gesetzt. Das ist der Fall bei `"pi"`.

In Python heisst die Syntax `stack[2:6]` um einen Teil der Liste `stack` zu kopieren slice. *Beachte, dass der zweite Index (hier 6) immer das erste Element ist, das* nicht *mehr dazugehört. Fehlt einer der beiden Indices, dann ist damit der Anfang bzw. das Ende der Liste gemeint. `stack[:]` erzeugt damit eine Kopie der ganzen Liste.*

Aus dem Feld `__code__` holen wir in Zeile 14 die Anzahl der Parameter (Argumente). Zur Erinnerung: Mit `stack[-1]` gibt uns Python das letzte Element aus der Liste zurück. Mit `stack[-5:-2]` gibt uns Python eine Teilliste mit dem fünftletzten, dem viertletzten und dem drittletzten Element zurück und mit `stack[-3:]` erhalten wir die letzten drei Elemente. Das nutzen wir in Zeilen 15 und 16 aus, um die Argumente vom Stack in die Liste `args` zu übertragen.

Schliesslich bewirkt die Syntax `function(*args)`, dass die Elemente aus der Liste `args` als einzelne Argumente übergeben werden. In anderen Worten: `f(*[2, 4, 5])` ist das gleiche wie `f(2, 4, 5)`.

```
1  program = ["sqr", "pi", "mul"]
2  stack = [12]
3
4  operations = {
5      "mul": lambda x, y: x * y,
```

```
6       "sqr": lambda x: x*x,
7       "pi": 3.1415926536
8   }
9
10  for command in program:
11      if command in operations:
12          function = operations[command]
13          if callable(function):
14              argcount = function.__code__.co_argcount
15              args = stack[-argcount:]
16              stack = stack[:-argcount]
17              result = function(*args)
18          else:
19              result = function
20          if result is not None:
21              stack.append(result)
22      else:
23          print "Unknown command", command
24
25  print stack
```

AUFGABEN

18. Was passiert, wenn der Stack nicht genügend Argumente für eine Funktion hat? Ergänze das Beispielprogramm so, dass dein Programm eine brauchbare Fehlermeldung ausgibt. Behandle auch den Fall einer Funktion, die gar keine Parameter braucht. In einem solchen Fall funktioniert die hier vorgestellte Technik mit `stack[-argcount:]` nicht.

19. Sieh dir den Quelltext an und baue den PostScript-Befehl `pop` mit möglichst kleinem Aufwand ein, d. h. als eine Lambda-Funktion in den `operations`.

`pop` nimmt einfach das oberste Element vom Stack und entfernt es. Aus `[3, 4, 5]` wird also `[3, 4]`.

20. Den PostScript-Befehl `dup`, der das letzte Element auf dem Stack verdoppelt kannst du nicht sinnvoll über die obige Technik einbauen. Dafür gibt es aber die `if`-Abfrage in Zeile 11 (bzw. 22). Ergänze das Programm oben um `dup` und die üblichen Befehle.

Alternativ kannst du `dup` auch als Funktion `lambda x: [x, x]` umsetzen, musst dann allerdings dein Programm anderweitig ausbauen.

21.* Verbinde das Dictionary `operations` mit `variables` aus dem Abschnitt 3.3 zu einem einzigen Dictionary, das sowohl PostScript-Funktionen, Variablen als auch Lambda-Funktionen und einfache Zahlenwerte enthalten kann.

INDEX