Python Programmieren

Eine kurze Einführung in die Programmierung mit Python

Dr. Tobias Kohn

1 Turtle-Grafik

Einführung: Die Turtle Die Turtle war ursprünglich ein kleiner Roboter mit einem Stift, der von einem Computer gesteuert wurde. Für TigerJython verwenden wir eine Simulation dieser Turtle, die ihre Bilder in einem Fenster zeichnet.

Turtle-Bewegungen Zunächst einmal kennt die Turtle die drei Grundbefehle `left(angle)`, `right(angle)` und `forward(length)`. Mit `left` und `right` dreht sich die Turtle an Ort um den angegebenen Winkel (in Grad), bei `forward` läuft sie die entsprechende Anzahl Pixel vorwärts. Damit lässt sich bereits eine beeindruckende Fülle an Bildern zeichnen.

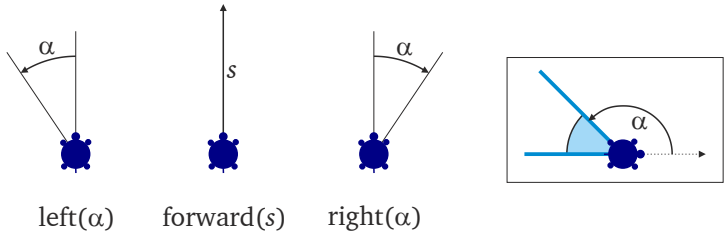


Abbildung 1: Die Grundbefehle der Turtle.

Bereits aus der Zeit der Roboterturtle stammen die Befehle `penUp()` und `penDown()`, mit denen der Stift hochgehoben bzw. wieder auf das Blatt gesenkt wird. Damit lässt sich steuern, ob die Turtle eine Linie zeichnet oder sich bewegt, ohne eine Spur zu hinterlassen.

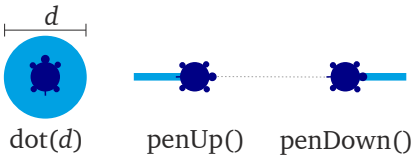


Abbildung 2: `dot(d)` zeichnet an der aktuellen Position einen Punkt mit dem Durchmesser d. Zudem kannst du mit `penUp()` und `penDown()` steuern, ob die Turtle eine Linie zeichnen soll.

Das Turtle-Programm Jedes Turtle-Programm muss zunächst das Turtle-Modul (Bibliothek) laden bzw. importieren. Danach wird mit `makeTurtle()` ein neues Fenster mit globaler Turtle erzeugt.

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

Hilfreich sind zudem die Befehle `speed(-1)` für maximale Animationsgeschwindigkeit und `hideTurtle()`, um die Turtle ganz auszublenden und Zeichnungen ohne Animation auszuführen.

Beispiel: Ein Dreieck Als erstes vollständiges Beispiel eines Python-Programms mit Turtle-Grafik zeichnen wir hier ein einfaches Dreieck. Beachte, dass die Winkel, um die sich die Turtle jeweils drehen muss *nicht* die Innenwinkel des Dreiecks sind!

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

Farben Mit `setPenColor(color)` lässt sich die Farbe setzen. Dabei ist `color` ein X11-Farbname (der in Anführungszeichen stehen muss, z. B.: `setPenColor("blue")`). Über `setPenWidth(width)` wird schliesslich die Breite des Stifts gesetzt.

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

Abbildung 3: Das ist eine Auswahl der Farben, die du mit `setPenColor("farbe")` auswählen kannst.

Füllen: Flächen überstreichen Bei `fillToPoint()` wechselt die Turtle in einen Modus, in dem sie nicht nur eine Spur zeichnet, sondern ganze Flächen überstreicht. Die aktuelle Position der Turtle dient dabei als Ankerpunkt. So können sehr einfach gefüllte Flächen gezeichnet werden. Mit `fillOff()` wird das Überstreichen der Flächen wieder ausgeschaltet.

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

Beispiel: Die Ampel Das nächste Beispiel zeigt die Verwendung von Farben und wie gefüllte Rechtecke auch als breite Linien gezeichnet werden können. Mit dem Hash `#` werden Kommentare eingeleitet, die von Python ignoriert werden.

```
1  from gturtle import *
2  makeTurtle()
3
4  setPenColor("black") # Schwarz als Stiftfarbe
5  penWidth(40)         # Stiftbreite soll 40 Pixel sein
6  forward(80)          # 80 Pixel forwärts (n. oben) gehen
7
8  left(180)            # Die Turtle drehen
9  penUp()              # Keine Linien mehr zeichnen
10
11 setPenColor("red")   # Rot als Stiftfarbe
12 dot(30)              # Punkt mit Durchmesser 30 Pixel
13 forward(40)          # 40 Pixel vorwärts (n. unten) gehen
14
15 setPenColor("yellow")
16 dot(30)
17 forward(40)
18
19 setPenColor("green")
20 dot(30)
21 forward(40)
```

AUFGABEN

1. Programmiere die drei Figuren aus der Abbildung 4 mit der Turtle.

(a) (b) (c)

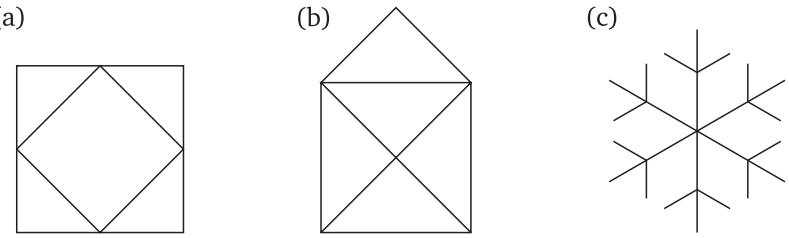


Abbildung 4: In der Mitte das «Haus von Nikolaus» und rechts eine einfache Schneeflocke.

Koordinatengrafik Die Turtle kann sich nicht nur relativ zu ihrer aktuellen Position und Ausrichtung bewegen, sondern auch absolute Punkte im Fenster mit Koordinaten ansteuern. Die zwei wichtigsten Befehle dazu sind `setPos(x, y)` und `moveTo(x, y)`. Bei `setPos` springt die Turtle direkt an den angegebenen Punkt, bei `moveTo` zeichnet sie eine Linie vom aktuellen Standort aus.

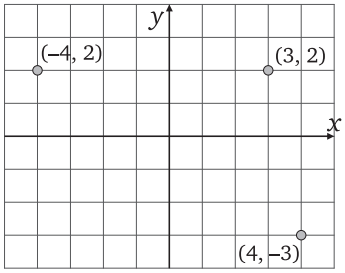


Abbildung 5: Die Turtle befindet sich in einem Koordinatensystem: Jeder Punkt wird mit zwei Zahlen x und y angegeben. Die Mitte ist $(0, 0)$. Daneben sind hier noch drei weitere Punkte eingezeichnet.

Beispiel: Ein Kreuz (X) Das kurze Programm zeichnet ein Kreuz.

```
1 from gturtle import *
2 makeTurtle()
3
4 setPos(50, 100)
5 moveTo(-50, 0)
6 setPos(-50, 100)
7 moveTo(50, 0)
```

Ausrichtung Neben der absoluten Position lässt sich auch die Ausrichtung der Turtle mit `setHeading(angle)` steuern. Ein Winkel von 0 zeigt direkt nach oben. Zusammen mit der Funktion `towards(x, y)` lässt sich die Turtle auf einen bestimmen Punkt (x, y) hin ausrichten: `setHeading(towards(50, 60))`.

AUFGABEN

2. Mit der Funktion `randint` aus dem Modul «`random`» kannst du Zufallszahlen aus einem gegebenen Bereich ziehen und damit zufällig einen Punkt auf den Bildschirm setzen:

```
from random import randint
x = randint(-300, 300)
y = randint(-200, 200)
setPos(x, y)
dot(20)
```

Verwende diese Technik, um das Fenster der Turtle mit 200 zufälligen Punkten zu füllen. Wähle dabei auch jeweils den Radius der Punkte zufällig.

2 Schleifen

Einführung Schleifen gehören zu den wichtigsten Programmstrukturen überhaupt. Zunächst einmal lernen die Schüler, einen festen Programmteil zu wiederholen. Später kommen Variablen dazu, und damit die Möglichkeit, jede Wiederholung ein wenig anders zu gestalten.

Mehr als Python: Repeat Für Anfänger werden Schleifen schnell schwierig, wenn Variablen involviert sind. Aus diesem Grund bietet TigerJython eine variablenfreie Schleife an, die von Logo übernommen wurde und nicht zum eigentlichen Python gehört: `repeat`.

Das nächste Programm zeichnet ein Quadrat mit Hilfe von `repeat` und macht die Turtle anschliessend unsichtbar. Der Schleifenkörper unter dem `repeat` muss *eingerückt* sein, damit Python erkennt, was zur Schleife gehört und wiederholt werden soll.

```
1 from gturtle import *
2 makeTurtle()
3 repeat 4:
4     forward(100)
5     left(90)
6 hideTurtle()
```

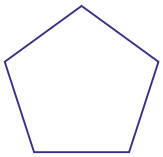
Variablenwerte ändern Schleifen werden besonders interessant, wenn wir bei jeder Wiederholung den Wert einer Variable ändern. Damit lässt sich zum Beispiel eine Spirale programmieren:

```
1 from gturtle import *
2 makeTurtle()
3 side = 10
4 repeat 18:
5     forward(side)
6     left(60)
7     side += 2
```

While-Schleifen Die Spirale lässt sich natürlich auch mit einer `while`-Schleife programmieren. Wird die Anweisung `side += 2` in die Zeile 5 verschoben, wird das Programm für Anfänger schwieriger.

```
1 from gturtle import *
2 makeTurtle()
3 side = 10
4 while side < 46:
5     forward(side)
6     left(60)
7     side += 2
```

Beispiel: Ein Fünfeck Um ein Fünfeck zu zeichnen muss die Turtle 5 Mal das gleiche tun: Eine Linie zeichnen und sich um $72°$ drehen.



```
from gturtle import *
makeTurtle()
right(90)
repeat 5:
    forward(100)
    left(72)
```

Kreise zeichnen Computer können keine Kreise zeichnen. Mit der Turtle approximieren wir einen Kreis als Polygon mit $n \geq 36$.

```
from gturtle import *
makeTurtle()
repeat 36:
    forward(2)
    left(360 / 36)
```

Flächen füllen Beim Befehl `fillToPoint()` merkt sich die Turtle die aktuelle Position. Danach verbindet sie alle weiteren Punkte mit dieser aktuellen Position und überstreicht so Flächen. Mit `fillOff()` wird das wieder ausgeschaltet.

```
from gturtle import *
makeTurtle()

fillToPoint()
forward(100)
left(120)
repeat 6:
    forward(100)
    left(60)
fillOff()
```

AUFGABEN

3. Zeichne mit der Turtle einen fünfzackigen Stern oder ein Pentagramm wie in der Abbildung 6.

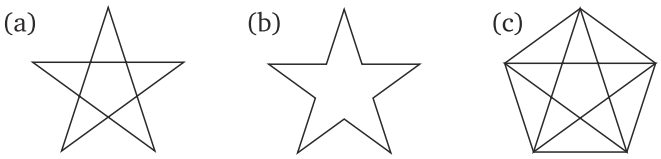


Abbildung 6: Fünfzackige Sterne mit dem Pentagramm.

4. Lass deine Turtle den PacMan (a), das Yin-Yang-Symbol (b) und ein beliebiges Smiley (c) zeichnen wie in der Abbildung 7.

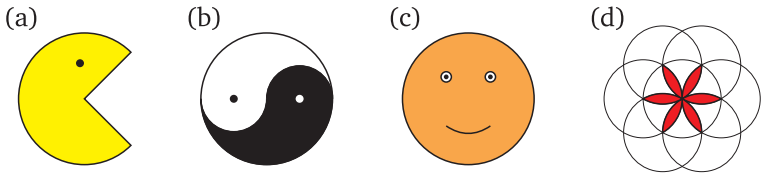


Abbildung 7: Kreisfiguren.

5. Die «Blume» in der Abbildung 7(d) setzt sich aus sieben gleich grossen Kreisen zusammen. Verwende einen eigenen Befehl `circle()` und zeichne damit diese Blume.

6. Die Funktion `makeColor("rainbow", value)` mit `value` zwischen 0.0 und 1.0 erzeugt eine Farbe aus dem Farbspektrum. Schreibe damit ein Programm, das (a) das Turtlefenster mit dem Farbspektrum ausfüllt bzw. (b) einen Kreis mit den entsprechenden Farben zeichnet.

7. `makeColor(r, g, b)` erzeugt eine Farbe aus den *Rot*-, *Grün*- und *Blau*-Anteilen zwischen 0.0 und 1.0. Programmiere damit einen Farbverlauf von Schwarz nach Gelb.

8. Zeichne die Figur aus Abbildung 8.

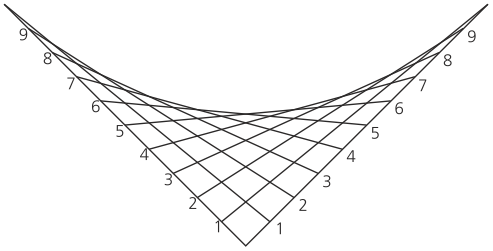


Abbildung 8: Eine Figur aus der «Fadengrafik».

3 Funktionen ohne Rückgabewerte

Einführung: Neue Befehle und Funktionen Die eigentliche Programmierung beginnt mit der Abstraktion, wenn wir neue Befehle und Funktionen definieren. Für Schüler ist das Definieren eines neuen Turtlebefehls zunächst einfach und natürlich. Probleme stellen sich aber im Zusammenhang mit Parametern und vor allem mit dem Rückgabewert von Funktionen.

Funktionen definieren Funktionen werden über `def Name(Param)` definiert. Der Funktionskörper ist wiederum eingerückt. Mit `return` kann ein Rückgabewert angegeben werden. Ansonsten ist der Rückgabewert immer `None`.

```
1  from gturtle import *
2  makeTurtle()
3
4  def triangle():
5      repeat 3:
6          forward(100)
7          left(120)
8
9  def square(side):
10     repeat 4:
11         forward(side)
12         left(90)
13
14 square(100)
15 triangle()
```

Beispiel: Polygon In diesem Beispiel demonstrieren wir gleich beide Funktionsarten: Mit und ohne Rückgabewert. Das Programm zeichnet ein regelmässiges 7-Eck.

```
1  from gturtle import *
2  makeTurtle()
3
4  def getSideFromCircumference(n, c):
5      return c / n
6
7  def polygon(n, c):
8      side = getSideFromCircumference(n, c)
9      repeat n:
10         forward(side)
11         left(360 / n)
12
13 polygon(7, 294)
```

Beispiel: Ein Zeichenprogramm Eine Funktion mit zwei Parametern für x- und y-Koordinate lässt sich sehr einfach als Callback für einen Mausklick definieren. Damit lässt sich sehr schnell ein einfaches Zeichenprogramm schreiben.

```
1  from gturtle import *
2
3  @onMousePressed
4  def onPress(x, y):
5      setPos(x, y)
6
7  @onMouseClicked
8  def onClick(x, y):
9      dot(10)
10
11 @onMouseDragged
12 def onDrag(x, y):
13     moveTo(x, y)
14
15 makeTurtle()
16 hideTurtle()
```

AUFGABEN

9. Definiere einen Befehl `gitter(breite, hoehe)`, der ein Gitter wie in Abbildung 9(b) zeichnet.

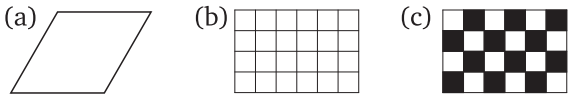


Abbildung 9: Rhombus, Gitter und Schachbrettmuster.

10. Definiere einen Befehl `schachbrett(breite, hoehe)`, um ein Schachbrett wie in Abbildung 9(c) zu zeichnen.

11. Definiere eine Funktion `circle(x, y, radius)`, die einen Kreis mit dem angegebenen Mittelpunkt und Radius zeichnet.

12. Baue das Zeichenprogramm oben so aus, dass es bei jedem Mausklick die Farbe wechselt. Beachte, dass globale Variablen in einer Funktion definiert werden müssen (ansonsten sind alle Variablen in einer Funktion automatisch lokal).

```
i = 0
@onMouseClicked
def onClick(x, y):
    global i
    i += 1
```

4 Primzahlen

Einführung Primzahlen lassen sich sehr gut im Informatikunterricht einsetzen. Zum Beispiel bei einem kleinen Programm, das prüft, ob eine gegebene Zahl eine Primzahl ist.

Die Funktion `exit()` aus dem `sys`-Modul beendet das Programm sofort. Wenn wir im folgenden Beispiel wissen, dass die Zahl keine Primzahl ist, müssen wir nicht weiter nach einem Teiler suchen.

Das Prozentzeichen wird in Python verwendet, um den Rest einer ganzzahligen Division zu berechnen. So ergibt z. B. `17 % 3` den Wert 2 und `17 % 4` den Wert 1.

```
1  from sys import exit
2  zahl = 2000003
3  teiler = 2
4  while teiler < zahl:
5      rest = zahl % teiler
6      if rest == 0:
7          print "Falsch"
8          exit()
9      teiler += 1
10 print "Wahr"
```

Übrigens: Hier sind einige grössere Primzahlen zum Ausprobieren (Quelle: “the prime pages”, `http://primes.utm.edu/`):

$$9\,949,\quad 10\,333,\quad 1\,500\,007,\quad 32\,452\,843,\quad 49\,979\,693,\quad 86\,028\,121$$

AUFGABEN

13. Im Moment geht das Programm alle möglichen Teiler der Reihe nach durch. Dabei würde es bei ungeraden Zahlen genügen, nur ungerade Zahlen zu prüfen. Ändere das Programm entsprechend ab.

14.* Schreibe das Programm in eine Funktion `isPrime(zahl)` um, die mit `return` ‘wahr’ oder ‘falsch’ zurückgibt. Dann brauchst du auch die `exit()`-Funktion nicht mehr.

15.* Schreibe ein Programm, das der Reihe nach alle Primzahlen heraussucht und auf dem Bildschirm ausgibt.

5 Listen

Einführung Die Primzahlen eignen sich hervorragend als Beispiel-Liste, weil die Zahlen nicht regelmässig verteilt sind. In diesem Abschnitt stellen wir verschiedene Techniken vor, um die Summe einer Liste von Primzahlen (oder anderem) zu berechnen.

For-Schleife For-Schleifen in Python arbeiten *immer* mit Listen: Die Laufvariable geht der Reihe nach alle Elemente einer Liste durch. Damit ist diese Technik prädestiniert, um die Summe zu bilden.

```
1 primes = [2, 3, 5, 7, 11, 13, 17, 19]
2 summe = 0
3 for p in primes:
4     summe += p
5 print summe
```

Funktionen In Python wird eine Funktion mit `def` definiert und gibt mit `return` ein Resultat zurück. Indem wir den Code oben in eine Funktion verpacken, haben wir flexibleren, wiederverwendbaren Code.

```
1 def summe(liste):
2     result = 0
3     for zahl in liste:
4         result += zahl
5     return result
6
7 primes = [2, 3, 5, 7, 11, 13, 17, 19]
8 print summe(primes)
```

Zufallszahlen Für viele Zwecke ist es nützlich, eine Liste mit Zufallszahlen erstellen zu können, um dann mit einer solchen zufälligen Liste weiterzuarbeiten. Dazu laden wir die Funktin `randint` aus dem `random`-Modul und erzeugen dann mit `randint(0, 20)` eine Zufallszahl zwischen 0 und 20.

Indem wir mit `in` bzw. `not in` zuerst prüfen, ob eine Zahl bereits in der Liste vorkommt, vermeiden wir doppelte Einträge.

```
1 from random import randint
2 liste = []
3 repeat 30:
4     zahl = randint(0, 20)
5     if zahl not in liste:
6         liste.append(zahl)
7 print liste
```

Sortieren Nachdem wir eine Liste erzeugt haben wollen wir sie auch sortieren. Hier ist ein einfacher Algorithmus, der demonstriert, wie man Elemente aus einer ersten Liste entfernen und an eine andere Liste anhängen kann.

```
1 def my_sort(liste):
2     result = []
3     while liste != []:
4         x = min(liste)
5         liste.remove(x)    # x aus alter Liste entfernen
6         result.append(x)   # x an neue Liste anhängen
7     return result
8
9 print my_sort([14, 3, 4, 5, 17, 7, 18, 11, 19])
```

AUFGABEN

16.* Schreibe eine Lotto-Simulation. Wähle zuerst 6 Zahlen aus 42 aus und führe dann 10 000 Ziehungen mit 7 Zufallszahlen durch. Das Programm gibt dann jedes Mal die 7 Zahlen aus, wenn 4, 5 oder 6 «richtige» dabei sind und zählt zusammen, wie oft du mit deinen 6 Glückszahlen gewonnen hast.

17.* Gegeben ist eine Liste, die nur die Zahlen `0` und `1` enthält. Finde darin die längste Abfolge von Nullen oder Einsen. Für die Eingabe `[0, 1, 1, 0, 0, 1, 1, 1, 0]` wäre die längste Folge `[1, 1, 1]`.

Das Programm soll drei Informationen dazu ausgeben: Die Länge der Folge, ob es Nullen oder Einsen sind und an welcher Stelle in der Liste die Folge auftritt.

6 Simulationen: Die Uhr

Wir erarbeiten am Beispiel einer Uhr ein grundsätzliches Spiralcurriculum zu den Simulationen, wobei wir immer «abstraktere» Elemente in unser Design mit aufnehmen. Nach den hier präsentierten Schritten lassen sich dann auch komplexere Simulationen besprechen.

Einführung Die Idee der Simulation gehört zu den wichtigsten Beiträgen der Informatik zum allgemeinbildenden Unterricht: Simulationen spielen heute eine enorm wichtige Rolle (Wettervorhersage, Modellprüfung in den Naturwissenschaften, etc.). Aus Sicht der Schüler handelt es sich zunächst einmal um «Animationen». Indem wir diese aber schrittweise aufbauen, können wir schliesslich sogar naturwissenschaftlich interessante Simulationen besprechen.

Neben dem Anwendungsbereich verdeutlichen Simulationen aber auch schön ein Grundprinzip der Algorithmen mit dem zeitlichen, sequentiellen Ablauf und der schrittweisen Änderung des Zustands.

Wir bauen Simulationen oder Animationen grundsätzlich immer nach dem folgenden Schema auf:

```
initialize()  # Anfangszustand herstellen
repeat:
    escape()  # Simulation verlassen?
    update()  # Objekte Bewegen
    render()  # Objekte darstellen
    wait()    # Warten
```

Als Abbruchbedingung kann man auch einbauen, dass die Simulation bei Tastendruck abgebrochen werden soll. Zu Beginn wählen wir mit den Schülern aber eine feste Anzahl Wiederholungen.

```
if getKeyCode() != 0:
    break
```

Übrigens: Dieses Schema ist zeitlich nicht exakt, weil vor allem die beiden Schritte «update» und «render» selbst auch Zeit benötigen. Für unsere Zwecke reicht es aber.

1. Schritt: Eine Minute abzählen Das erste Beispiel lässt einen Zeiger der Uhr im Sekundentakt im Kreis laufen. Beachte, dass das Beispiel zwar eine Simulation ist, aber ohne Variablen auskommt! Der Zustand ist geschickt in der Turtle gespeichert. Zum Zweiten wird das Prinzip des selbst definierten Befehls motiviert: Ein und dieselbe Funktion wird zum Zeichnen und zum Löschen des Zeigers verwendet.

```
1 from gturtle import *
2
3 def zeiger():
4     forward(100)
5     back(100)
6
7 makeTurtle()              # <- initialize
```

```
8  hideTurtle()
9  repeat 60:
10     right(6)              # <- update
11     setPenColor("black")  # <- render (I)
12     zeiger()
13     delay(1000)           # <- wait
14     setPenColor("white")  # <- render (II)
15     zeiger()
```

2. Schritt: Zwei Zeiger Bei zwei Zeigern (für Sekunden und Minuten) kann der Zustand nicht mehr einfach in der Turtle gespeichert werden. Wir brauchen zwingend Variablen. Das Darstellen der Szene findet in einem eigenen Befehl statt und wir achten darauf, die Turtle am Ende wieder zurückzusetzen (Zeile 7).

```
1  from gturtle import *
2
3  def zeiger(winkel, laenge):
4      right(winkel)
5      forward(laenge)
6      back(laenge)
7      left(winkel)
8
9  def szene(sekunden, minuten):
10     clear()
11     zeiger(minuten * 6, 60)
12     zeiger(sekunden * 6, 100)
13
14 sekunden = 36                 # <- initialize
15 minuten = 14
16 makeTurtle()
17 hideTurtle()
18 repeat 60:
19     sekunden += 1             # <- update
20     if sekunden == 60:
21         sekunden = 0
22         minuten += 1
23     szene(sekunden, minuten)  # <- render
24     delay(1000)               # <- wait
```

3. Schritt: Ungleichmässige Bewegung Hier kommt ein schwingendes Pendel zur Uhr hinzu, dessen Schwingung eigentlich von der Schwerkraft bestimmt wird.

Das Pendel muss schneller schwingen als die Zeiger. Wir haben hier 10 Frames pro Sekunde gewählt. Der Wert in `delay()` verringert sich damit auf 100 Millisekunden (Zeile 37). Den Wert der Variable `sekunden` erhöhen wir in 0.1-Schritten. Mit `int(sekunden)` (Zeile 36) verwenden wir für die Darstellung aber nur den ganzzahligen Anteil. Beachte

auch, dass ein Wert von 0.1 zu Rundungsfehlern führt. Deshalb ändert sich der Test in Zeile 31.

Wenn wir das Bild mehr als einmal pro Sekunde neu zeichnen beginnt die Darstellung zu flackern, weil wir das kurzzeitige Löschen des Bildschirms in Zeile 15 wahrnehmen. Um das zu verhindern schalten wir das automatische Zeichnen in Zeile 28 ab und verwenden explizit `repaint()` (Zeile 20), um die Szene neu zu zeichnen.

Achtung: Dass die Position des Pendels die Geschwindigkeit beeinflusst, und diese wiederum die Position bestimmt ist für Schüler schwierig! Eine weitere (kleinere) Schwierigkeit besteht in den Rundungsfehlern.

```
1  from gturtle import *
2
3  def zeiger(winkel, laenge):
4      ...
5
6  def pendel(alpha):
7      setPos(0, -13)
8      right(180 + alpha)
9      forward(150)
10     dot(42)
11     back(150)
12     left(180 + alpha)
13
14 def szene(sekunden, minuten, alpha):
15     clear()
16     setPos(0, 100)
17     zeiger(minuten * 6, 60)
18     zeiger(sekunden * 6, 100)
19     pendel(alpha)
20     repaint()                    # <- zeichnen
21
22 sekunden = 36                    # <- initialize
23 minuten = 14
24 alpha = 30                       # <- alpha: Ort
25 phi = 0                          # <- phi: Geschwindigkeit
26 makeTurtle()
27 hideTurtle()
28 enableRepaint(False)             # <- nicht zeichnen
29 repeat 600:
30     sekunden += 0.1              # <- update
31     if sekunden >= 59.9:
32         sekunden = 0
33         minuten += 1
34     phi -= alpha / 10            # <- Beschleunigung
35     alpha += phi                 # <- Pendel bewegen
36     szene(int(sekunden), minuten, alpha)
37     delay(100)                   # <- warten
```

7 Die Turtle im Labyrinth

Das Grundgitter Im Gegensatz zur Turtle-Grafik lädst du hier die Datei «tjgrids» (*tj* steht für *TigerJython*, den Namen der Programmierumgebung), bevor du eine Turtle mit Fenster erzeugst. Das Turtle-Fenster sieht denn auch anders aus: Du kannst mit der Maus Blöcke erzeugen bzw. entfernen und die Turtle an eine beliebige Stelle ziehen. Mit den Pfeiltasten der Tastatur steuerst du die Turtle manuell.

```
1 from tjgrids import *
2 makeTurtle()
```

Du kannst auch die Grösse des Grundgitters angeben (Breite, Höhe):

```
1 from tjgrids import *
2 makeTurtle(20, 15)
```

Die Turtle steuern Die Steuerung der Turtle funktioniert grundsätzlich wieder gleich: Du hast `forward(s)`, `left()` und `right()` zur Verfügung. Im Unterschied zur Grafik kann sich die Turtle aber nur noch um $90°$ oder $180°$ drehen und sie zeichnet keine Spur mehr.

```
1 from tjgrids import *
2 makeTurtle()
3
4 right()
5 forward(3)
6 left()
```

Zellen färben Jede Zelle des Gitters enthält eine (unsichtbare) Zahl. Mit `setCell(zahl)` setzst du diese Zahl auf einen neuen Wert. «0» steht für eine leere Zelle, «1» ist ein Block und die Zahlen 2 bis 6 stehen für die Farben: rot (2), gelb (3), grün (4), blau (5), schwarz (6). Dieses Programm hier zeichnet ein kleines rot-gelbes Karomuster.

```
1  from tjgrids import *
2  makeTurtle()
3
4  setCell(2)
5  forward()
6  setCell(3)
7  right()
8  forward()
9  setCell(2)
10 right()
11 forward()
12 setCell(3)
```

Die Turtle programmieren: Muster aus Regeln In diesem Programm erzeugt die Turtle ein Bild, das wir nicht im Voraus kennen! Vielmehr geben wir nur die Regel an, wie sie das Bild zeichnen soll. Die Grundstruktur sieht dabei so aus: Wir definieren einen Befehl `oneStep()`, der angibt, was die Turtle in *einem* Schritt tun soll. Danach starten wir die Turtle mit `start(oneStep)`.

```
1  from tjgrids import *
2  makeTurtle(15, 15)
3  setPos(7, 7)         # In die Mitte setzen
4
5  def oneStep():
6      if isCell(0):    # 0: Leere Zelle
7          setCell(2)   # -> rot färben
8          right()
9          forward()
10     elif isCell(2): # 2: Rote Zelle
11         setCell(3)   # -> gelb färben
12         left()
13         forward()
14     else:
15         forward()
16
17 start(oneStep)       # Los geht's!
```

Die Regel in diesem Beispiel sieht so aus:

- Wenn die aktuelle Zelle leer ist (den Wert «0» hat), dann setze den Wert auf «2» (rot), drehe dich nach rechts und gehe einen Schritt vorwärts.
- Wenn die aktuelle Zelle hingegen den Wert «2» (rot) hat, dann setze den Wert auf «3» (gelb), drehe dich nach links und gehe einen Schritt vorwärts.
- In allen anderen Fällen: Gehe einen Schritt vorwärts.

Bedingungen prüfen Ein Kernstück der Regeln ist das Prüfen einer Reihe von Bedingungen. Das Grundmuster dabei ist **`if`** für die eigentliche Bedingung und **`else`** für «andernfalls». Bei mehr als zwei Möglichkeiten wird das **`else if`** (wenn andernfalls) zu einem **`elif`** verkürzt.

```
if isCell(0):
    # Zelle ist leer
elif isCell(2):
    # Zelle ist rot
elif isCell(3):
    # Zelle ist gelb
elif isCell(4):
    # Zelle ist grün
else:
    # ALLE anderen Fälle
```

Der Weg im Labyrinth Die Technik mit den Regeln, die bei jedem Schritt ausgeführt werden, eignet sich hervorragend, um die Turtle durch ein Labyrinth zu manövrieren. In diesem ersten Beispiel ist die Regel denkbar einfach: «Wenn du vorwärts gehen kannst, dann gehe vorwärts. Ansonsten drehe dich nach rechts.»

```
1  from tjgrids import *
2  makeTurtle()
3
4  def oneStep():
5      if canGoForward():
6          forward()
7      else:
8          right()
9
10 startOnEnter(oneStep)
```

Zudem haben wir das `start` durch ein `startOnEnter` ersetzt. Du kannst also zuerst mit der Maus ein Labyrinth für die Turtle zeichnen. Sobald du «*Enter*» bzw. «*Return*» drückst, beginnt die Turtle sich zu bewegen.

Die Rechte-Hand-Regel Wie findest du deinen Weg durch ein Labyrinth? Eine einfache Möglichkeit (nicht unbedingt die schnellste): Berühre mit der rechten Hand die Wand und fahre der Wand entlang. Für die Turtle heisst das als Regel:

- Wenn rechts frei ist, dann gehe nach rechts (drehe dich also und mach einen Schritt vorwärts).
- Wenn hingegen geradeaus frei ist, dann gehe einen Schritt nach vorne.
- In allen anderen Fällen (wenn also rechts und vorne blockiert ist), drehe dich nach Links.

```
1  from tjgrids import *
2  makeTurtle()
3
4  def oneStep():
5      if canGoRight():
6          right()
7          forward()
8      elif canGoForward():
9          forward()
10     else:
11         left()
12
13 startOnEnter(oneStep)
```

8 Strings: Text analysieren

Einführung Python kann relativ gut mit Strings (Zeichenketten) umgehen und bietet eine Vielzahl von Funktionen. Dabei unterscheidet Python nicht, ob ein String in einfache oder doppelte Anführungszeichen eingeschlossen wird:

```
"Python ist toll!" == 'Python ist toll!'
```

Die Länge des Strings lässt sich mit `len("...")` bestimmen.

Buchstaben zählen In vielerlei Hinsicht unterscheiden sich Strings in Python kaum von Listen. So können wir auch mit einer `for`-Schleife die einzelnen Buchstaben/Zeichen eines Strings durchgehen und so die vorkommenden «E» zählen.

```
1  def count_e(text):
2      count = 0
3      for letter in text:
4          if letter in ["e", "E"]:
5              count += 1
6      return count
7
8  print count_e("Python lernen macht Spass!")
```

Frequenzanalyse Bei einfachen Text-Verschlüsselungen lohnt es sich, eine Frequenzanalyse des Textes vorzunehmen und die Häufigkeiten der Buchstaben zu zählen. Hier lassen wir das Python erledigen und verwenden dazu ein «Dictionary».

```
1  def freq_analysis(text):
2      # Tabelle enthält zur Zeit nur Eintrag für 'a' und 'e'.
3      letters = {"a": 0, "e": 0}
4      for ch in text:
5          ch = ch.lower()     # Alles in Kleinbuchstaben
6          if ch in letters:
7              # Vorhandenen Wert erhöhen:
8              letters[ch] += 1
9          else:
10             # Neuer Eintrag wird automatisch erstellt:
11             letters[ch] = 1
12     return letters
13
14 print freq_analysis("Python lernen ist cool!")
```

Wörter zählen Python kann einen gegebenen String mit `split` direkt zerlegen und liefert dann eine Liste mit den einzelnen Teilstücken. Das nutzen wir, um die Wörter in einem String zu zählen.

```
1  def count_words(text):
2      words = text.split(" ")
3      return len(words)
4
5  print count_words("Python ist auch eine Schlange.")
```

Parsen einer Zahl Dieses Programm liest aus einem gegebenen String eine hexadezimale Zahl heraus. Aus `"3F"` wird damit 63. In Python selber lassen sich hexadezimale Zahlen als `0x3F` direkt eingeben.

Mit `ord('A')` ermitteln wir den Ascii-Code eines einzelnen Zeichens. Die Umkehrung dazu wäre `chr(65)`, die aus dem Ascii-Code wiederum einen String erzeugt.

```
1  eingabe = inputString()
2  zahl = 0
3  for ch in eingabe.upper():
4      zahl *= 0x10
5      if '0' <= ch <= '9':
6          zahl += (ord(ch) - ord('0'))
7      elif 'A' <= ch <= 'F':
8          zahl += (ord(ch) - ord('A') + 10)
9      else:
10         print "Fehler: Ungültiges Zeichen", ch
11         break
12 print zahl
```

AUFGABEN

18. Schreibe die Funktion `count_words` so um, dass sie Wörter wie «Python-Kurs» als zwei eigene Wörter zählt.

19. Schreibe ein Programm, das einen Text mit der Caesar-Chiffre verschlüsselt.

20. Schreibe ein Programm, das eine Längen-Eingabe wie «14 cm» oder «5 in» entgegennimmt und die Länge in Meter ausgibt.

Hinweis: $1\,\text{in} = 2.54\,\text{cm}$, $1\,\text{ft} = 30.48\,\text{cm}$, bzw. in Python:

```
factors = {"in": 0.0254, "ft": 0.3048}
```