Python Schulung (3)

Immer noch ein Parforceritt durch die Sprache

Zur Auflockerung noch einmal die Schildkröte

Wie wäre es mit einem Polygon?

```
def polygon(t, n, length):
    angle = 360.0/n
    for i in range(n):
        t.forward(length)
        t.left(angle)

polygon(alex, 7, 70)
```

Oder mit einem Kreis?

```
import math

def circle(t, r):
    circum = 2*math.pi*r
    # n = 50
    n = int(circum/3) + 1
    length = circum/n
    polygon(t, n, length)
```

Woraus besteht ein Programm?

1. Sequenzen (werden von oben nach unten abgearbeitet)
2. Verzweigung (entweder dies oder das)
3. Schleifen (Mehrfachdurchläufe)

Boolsche Ausdrücke

Python kennt diese boolschen Operatoren:

- `x == y`: x ist **gleich** y
- `x != y`: x ist **ungleich** y
- `x > y`: x ist **größer** y
- `x < y`: x ist **kleiner** y
- `x >= y`: x ist **größer gleich** y
- `x <= y`: x ist **kleiner gleich** y

Logische Operatoren

- Es gibt drei logische Operatoren, mit denen boolsche Ausdrücke kombiniert werden können, `and`, `or` und `not`.
- Das Ergebnis von boolschen und logischen Operatoren ist immer `True` (wahr) oder `False` (falsch).

Boolsche Funktionen

Funktionen können auch Wahrheitswerte liefern:

```
def ist_teilbar(x, y):
    if x%y == 0:
        return(True)
    else:
        return(False)
```

oder:

```
def is_even(x):
    if x%2 == 0:
        return(True)
    else:
        return(False)
```

Verzweigungen

- Mit Hilfe der logischen und boolschen Operatoren können **Verzweigungen** programmiert werden:

```
if x > 0:
   print("x ist positiv")
else:
    print("x ist negativ oder null")
```

Oder auch:

```
if x%2 == 0:
    print("x ist gerade")
else:
    print("x ist ungerade")
```

Verkettete Bedingungen

Manchmal gibt es mehr als zwei Möglichkeiten:

```
if x < y:
    print("x ist kleiner als y")
elif x > y:
    print("x ist größer als y")
elif x == y:
    print("x ist gleich y")
else:
    print("Das Ende des Universums ist nahe!")
```

`elif` steht für `else if`.

Rekursion

Eine Funktion darf nicht nur eine andere Funktion, sondern auch sich selber aufrufen. Dies nennt man eine **Rekursion**:

```
def countdown(n):
    if n <= 0:
        print("Whammm …      !!")
    else:
        print(n)
        countdown(n-1)
```

Schleifen

Die `while`-Schleife ist eigentlich die einzige Schleife, die man braucht. Beispiel:

```
def countdown(n):
    while n > 0:
        print(n)
        n -= 1
    print(""Whammm …      !!")

countdown(10)
```

Die Endlos-Schleife

```
while True:
    print("From here to internity")
    # Hektische Suche nach dem Kill-Befehl
```

Die Endlos-Schleife ist aber nicht immer ein Programmfehler, sie wird zum Beispiel benutzt

- bei einem Server (der soll schließlich »immer« laufen)
- bei GUIs und/oder Spielen (warte auf Benutzer-Eingaben)

Callback (Ruf uns nicht an, wir rufen zurück)

```
def exit_prog():
    global keepGoing
    keepGoing = False

t.listen()
t.onkey(exit_prog, "Escape") # Escape beendet Programm

keepGoing = True
while keepGoing:
    pass  # Mache irgendetwas
```

(Beispielprogramm: `particle1.py` in `turtlepy`.)

Die `for`-Schleife

Python kennt noch die `for`-Schleife:

```
for i in range(10):
    print(i)
```

Der Endwert ist »exklusiv« (mathematisch gesprochen wird das halboffene Intervall `[0 … 10[` abgearbeitet, das heißt die Schleife zählt von Null bis Neun.

Die `for`-Schleife kann natürlich auch rückwärts zählen:

```
for i in range(10, 0, -1):
    print(i)
```

- Beachtet dabei, daß auch hier der Endwert exklusiv ist, die Schleife also von 10 bis 1 rückwärts zählt.
- Es gibt auch noch eine Abwandlung der `for`-Schleife für Strings und Listen. Dazu später mehr.

Die `range()`-Funktion

```
for identifier in range([start, ] stop [, step])
```

Dabei gilt:

- Alle Parameter sind Integer
- Alle Parameter können positiv oder negativ sein
- Wie alles in Python beginnt der Index mit `0`, daher ist der Stop-Wert »exklusiv«:

```
for i in range(5, 10):
    print i
```

Die `range()`-Funktion (2)

In Python 2.x gab es noch eine `xrange()`-Funktion. Der Unterschied war:

- `range()` gab als Ergebnis eine Liste zurück
- `xrange()` gab als Ergebnis einen `iterator` zurück

In Python 3 wurde `xrange()` zu `range()` und das originale `range()` als veraltet erklärt *(deprecated)*, das heißt `range()` liefert nun immmer einen `iterator` zurück.

Komplexe Datentypen und Strukturen

- Strings (noch einmal)
- Listen
- Tupel (nächste Woche)
- Dictionaries (Hashes) (auch nächste Woche)

Strings

Ein String ist eine Folge von Zeichen. Auf die einzelnen Zeichen kann man mit dem (eckigen) Klammer-Operator zugreifen:

```
frucht = "Banane"
zeichen = frucht[1]
print(zeichen)
```

Den Ausdruck in den eckigen Klammern nennt man `Index`. Ein Index beginnt immer mit `0`.
Der Indexwert kann berechnet werden, muß aber immer ein `Integer`-Wert sein.

`len` liefert die Anzahl der Zeichen eines Strings.

```
frucht = "Banane"
print(len(frucht))
```

Aber das gibt einen Fehler:

```
laenge = len(frucht)
letzter_buchstabe = frucht[laenge]
```

Richtig ist:

```
letzter_buchstabe = frucht[laenge - 1]
print(letzter_buchstabe)
```

Traversierung mit einer Schleife

```
i = 0
while i < len(frucht):
    zeichen = frucht[i]
    print(zeichen)
    i += 1
```

Oder (einfacher):

```
for zeichen in frucht:
    print(zeichen)
```

Enten

In Robert McCloskeys Buch *Make Way for Ducklings* gibt es Entchen mit den Namen Jack, Kack, Lack, Mack, Nack, Ouack, Pack und Quack:

```
praefixe = "JKLMNOPQ"
for praefix in praefixe:
    if praefix == "O" or praefix == "Q":
        print(praefix + "u" + suffix)
    else:
        print(praefix + suffix)
```

String-Teile

```
s = "Monty Python"
print(s[0:5])
print(s[6:12])
```

Der Operator `[n:m]` gibt den Teil des Strings vom »n-ten« bis zum »m-ten« Zeichen zurück und zwar einschließlich des ersten, aber ausschließlich des letzten Zeichens. Auch hier gilt wieder das halboffene Intervall `[n, m[`.

Slices

```
frucht = "banane"
print(frucht[:3])
print(frucht[3:])
print(frucht[:])
print(frucht[-1])
```

Wird der erste Index vor dem Doppelpunkt weggelassen, beginnt das Slice am Anfang des Strings, wird der zweite Index weggelassen, reicht das Slice bis zum Ende des Strings.

Negative Indizes zählen von hinten.

Strings sind unveränderlich (immutable)

Man kann aus »Jörg« kein »Jürg« machen:

```
name = "Jörg"
name[1] = "ü"
TypeError: 'str' object does not support item assignment
```

Man muß stattdessen einen neuen String erzeugen:

```
neuer_name = name[0] + "ü" + name[2:]
print(neuer_name)
```

String-Methoden

Diese Methode spricht für sich:

```
frucht = "banane"
neue_frucht = frucht.upper()
```

Man kann in Strings auch suchen:

```
frucht = "banane"
i = frucht.find("a")
print(i)
```

Die Methode find gibt immer die erste Position im String zurück, wo der zu suchende Teilstring gefunden wurde, andernfalls `-1`.

Man kann in Strings auch Teilstrings suchen:

```
i = frucht.find("na")
print(i)
```

Als zweites Argument kann man optional den Startindex angeben:

```
i = frucht.find("na", 3)
print(i)
```

Und als drittes Argument kann man den Index angeben, an dem die Suche beendet werden soll:

```
frucht.find("e", 1, 4)
print(i)
```

Der in-Operator

Das Wort `in` ist ein Boolscher Operator, der zwei Strings erwartet und `True` zurückliefert, wenn der erste las Teilstring im zweiten vorkommt:

```
print("a" in frucht)
print("samen" in frucht)
```

Beispiel:

```
def in_beiden(wort1, wort2):
    for zeichen in wort1:
        if zeichen in wort2:
            print(zeichen)

in_beiden("apfel", "orange")
```

Listen

- Nach Strings sind **Listen** die wichtigste Datenstruktur in Python (meine ich zumindest)
- Genau wie Strings ist eine Liste eine Folge von Werten. In einem String sind die Werte Zeichen, in einer Liste können die Werte beliebig sein und auch andere Listen enthalten:

```
liste1 = [10, 20, 30, 40]
liste2 = ["apfel", "banane", "zitrone", "orange"]
liste3 = ["SPAM", 2.0, 5, [10, 20]]
leere_liste = []
```

Listen sind veränderbar (mutable)

```
kaesesorten = ["Edamer", "Gouda", "Brie", "Cheddar"]
print(kaesesorten)
kaesesorten[1] = "Camenbert"
print(kaesesorten)
```

Indizes für Listen funktionieren genauso wie für Strings:

- Indizes können berechnet werden, müssen aber immer ganzzahlige Werte (Integer) sein
- Wird versucht, auf ein Element zuzugreifen, das nicht existiert, gibt es einen Index-Error
- Slices sind ebenfalls möglich [n:m], [:m], [n:], [-1]

Der `in`-Operator funktioniert auch mit Listen:

```
print("Edamer" in kaesesorten)
True
print("Gouda" in kaesesorten)
False
```

Löschen von Elementen aus einer Liste

Es gibt mehrere Methoden, um Elemente aus einer Liste zu löschen:

- Wenn man den Index des Elements kennt, kann man die Methode `pop` verwenden
- `pop` ändert die Liste und liefert das gelöschte Element zurück

```
fruits = ["Apple", "Tomato", "Banana", "Orange", "Lemon"]

for i in range(len(fruits)):
if fruits[i] == "Banana":
    a = i

fruits.pop(a)
print(fruits)

['Apple', 'Tomato', 'Orange', 'Lemon']
```

Braucht man den gelöschten Wert nicht, kann der `del`-Operator verwendet werden

```
fruits = ["Apple", "Tomato", "Banana", "Orange", "Lemon"]
del(fruits[fruits.index("Apple")])
print(fruits)
["Tomato", "Banana", "Orange", "Lemon"]
```

Wenn man das Element kennt, aber nicht den Index, kann auch `remove` verwendet werden

```
fruits = ["Apple", "Tomato", "Banana", "Orange", "Lemon"]
fruits.remove("Apple")
print(fruits)
["Tomato", "Banana", "Orange", "Lemon"]
```

Ein buntes Beispiel:

```
fruits = ["Apple", "Tomato", "Banana", "Orange", "Lemon"]
print(fruits)

for i in range(len(fruits)):
    if fruits[i] == "Banana":
        a = i

fruits.pop(a)
print(fruits)

del(fruits[fruits.index("Apple")])
print(fruits)
```

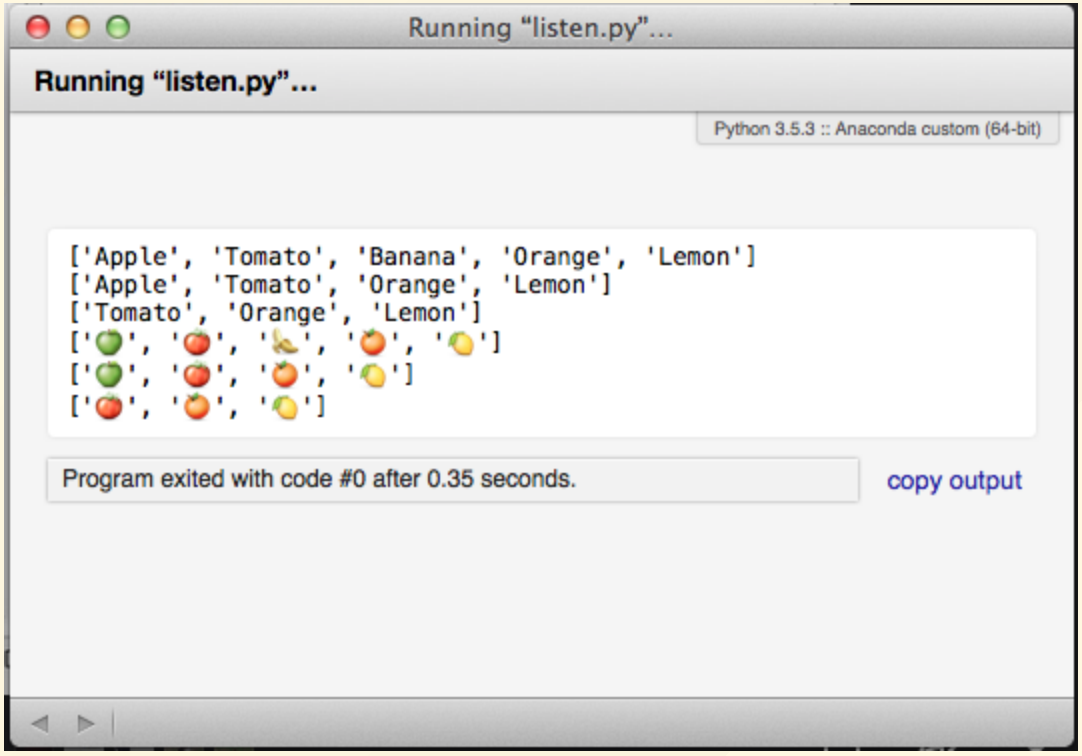
-->

```
fruits2 = ["  ", "  ", "  ", "  ", "  "]
print(fruits2)

for i in range(len(fruits2)):
    if fruits2[i] == "  ":
        a = i

fruits2.pop(a)
print(fruits2)

del(fruits2[fruits2.index("  ")])
print(fruits2)
```


Running “listen.py”...
Running “listen.py”...
Python 3.5.3 :: Anaconda custom (64-bit)
['Apple', 'Tomato', 'Banana', 'Orange', 'Lemon']
['Apple', 'Tomato', 'Orange', 'Lemon']
['Tomato', 'Orange', 'Lemon']
['🍏', '🍎', '🍌', '🍊', '🍋']
['🍏', '🍎', '🍊', '🍋']
['🍎', '🍊', '🍋']
Program exited with code #0 after 0.35 seconds.
copy output

Operationen mit Listen

Ähnlich wie bei Strings sind auch bei Listen die Operatoren `+` und `*` defininiert:

```
fruits1 = ["Apple", "Tomato", "Banana"]
fruits2 = ["Orange", "Lemon"]
fruits3 = fruits1 + fruits2
print(fruits3)
['Apple', 'Tomato', 'Banana', 'Orange', 'Lemon'

print([0, 1]*4
[0, 1, 0, 1, 0, 1, 0, 1]
```

Listen-Slices

Der Slice-Operator funktioniert auch bei Listen analog zu Strings:

```
fruits = ["Apple", "Tomato", "Banana", "Orange", "Lemon"]
print(fruits[1:3])
['Tomato', 'Banana']
print(fruits[:4])
['Apple', 'Tomato', 'Banana', 'Orange']
print(fruits[3:])
['Orange', 'Lemon']
```

Methoden für Listen

Python bietet Methoden für Listen. `append` hängt ein neues Element an das Ende einer Liste:

```
fruits.append("Pear")
print(fruits)
['Apple', 'Tomato', 'Banana', 'Orange', 'Lemon', 'Pear']
```

`extend` erwartet eine Liste als Argument und hängt alle Elemente an eine andere Liste an:

```
commanders1 = ["Christopher Pike", "James T. Kirk"]
commanders2 = ["Jean-Luc Picard", "Jonathan Archer"]
commanders1.extend(commanders2)
print(commanders1)
```

In diesem Beispiel bleibt `commanders2` unverändert.

`sort` sortiert die Elemente einer Liste von unten nach oben:

```
fruits = ["Apple", "Tomato", "Banana", "Orange", "Lemon"]
fruits.sort()
print(fruits)
['Apple', 'Banana', 'Lemon', 'Orange', 'Tomato']
```

Keine der Methoden einer Liste hat einen Rückgabewert. Sie verändern die Liste und liefern `None`. Die Eingabe von

```
t = t.sort()
```

ist also sinnlos.

Vorsicht beim Entfernen von Elementen einer Liste in einer Schleife

Schreibt man folgende Schleife:

```
fruits = ["Apple", "Tomato", "Banana", "Orange", "Lemon"]

for i in range(len(fruits)):
if fruits[i] == "Banana":
    fruits.pop()
```

Dann erhält man den Fehler:

```
IndexError: 'list index out of range'
```

Daher sollte man in diesen (und ähnlichen Fällen) Listen besser rückwärts durchlaufen:

```
fruits = ["Apple", "Tomato", "Banana", "Orange", "Lemon"]
print(fruits)

for i in range(len(fruits) - 1, -1, -1):
        if fruits[i] == "Banana":
        fruits.pop(i)

print(fruits)
```

Listen und Strings

Ein String ist eine Sequenz von Zeichen und eine Liste eine Sequenz von Werten. Aber eine Liste mit Zeichen ist etwas anderes als ein String:

```
s = "SPAM"
l = list(s)
print(l)
['S', 'P', 'A', 'M']
```

`list` konvertiert einen String in eine Liste aus einzelnen Zeichen.

Soll ein String in eine Liste aus einzelnen Wörtern aufgeteilt werden, ist dafür die Methode `split` zuständig:

```
s = "Veronika der Lenz ist da"
l = s.split()
print(l)
['Veronika', 'der', 'Lenz', 'ist', 'da']
```

Wird `split()` ohne Argumente aufgerufen, ist ein Leerzeichen der Trenner. Optional kann aber auch ein Trenner mitgegeben werden, z.B bei CSV-Strings:

```
s = "Alex,Berta,Chris,Doris"
l = s.split(",")
print(l)
['Alex', 'Berta', 'Chris', 'Doris']
```

`join` ist das Gegenteil von `split`. `join` ist eine String-Methode, die das gewünschte Trennzeichen als String übernimmt und für diesen String die Methode mit der Liste als Parameter aufruft:

```
l = ['Veronika', 'der', 'Lenz', 'ist', 'da']
t = " "
s = t.join(l)
print(s)
Veronika der Lenz ist da
```

Sollen die Elemente ohne Leerzeichen zusammengesetzt werden, kann der Leerstring `""` als Trennzeichen genutzt werden.

Das war's für heute ...

Nun solltet Ihr in der Lage sein, einfache Programme in Python selber zu schreiben. In der nächsten Woche fahren wir mit Dictionaries und Tupeln fort und wenden uns dann dem Lesen und Schreiben von Dateien zu.

Fragen?