Python-Schulung (5)

Dateien und Objekte

Wegen der langen Pause erst einmal eine kurze Rekapitulation

Was ist Python?

- Python ist eine Interpreter-Sprache
- Python kennt keine statische Typzuweisung, stattdessen werden die Typen nach dem Prinzip des »Duck-Typings« dynamisch während der Laufzeit vergeben und können auch während der Laufzeit verändert werden
- Python kennt folgende »einfache« Datentypen: Integer (Ganzzahl), Float (Fließkommazahl), Strings (Zeichenketten) und Bool (wahr oder falsch)

- Python kennt die üblichen mathematischen **Operatoren**, wie Addition (`+`), Substraktion (`-`), Multiplikation (`*`), Division (`/`) und das Potenzieren (`**`)
- Daneben gibt es noch die Ganzzahl-Divison (`//`) und den Modulo-Operator (`%`)
- Die Operatoren `+` und `*` können auch auf den Datentyp String und auf Listen angewandt werden

Für den **Programmfluß** kennt Python folgende Abläufe:

- Die *Sequenz*, bei der das Programm von oben nach unten abgearbeitet wird (der Normalfall)
- Die *Schleife*, bei der eine Sequenz kontrolliert oft wiederholt wird

```
while <bedingung>:
```

```
for i in range(x):
```

- Blöcke werden in Python nicht geklammert, sondern durch *Einrückungen* gekennzeichnet

- Die *Verzweigung*

```
if <bedingung1>:
    tue  irgendwas
elif <bedingung2>:
    tue etwas anderes
elif <bedingung3>:
    tue noch etwas anderes
else:
    nutze den Notausgang
```

- Die `elif`- und `else`-Abfragen sind optional
- Da Python eine Interpretersprache ist, die sequenziell abgearbeitet wird, müssen alle Variablen und Objekte **vor** ihrer ersten Nutzung vereinbart werden.

Um Bedingungen abzufragen, kennt Python die folgenden **boolschen Operatoren**:

- Abfrage auf Gleichheit (`==`)
- Abfrage auf Ungleichheit (`!=`)
- Abfrage auf kleiner (`<`) und kleiner-gleich (`<=`)
- Abfrage auf größer (`>`) und größer-gleich (`>=`)
- Die Operatoren können mit `and` und `or` zu kombinierten Ausdrücken verknüpft werden
- Die Operatoren liefern entweder `True` oder `False` zurück

Python kennt neben den einfachen Datentypen auch noch **komplexe Datentypen**: Die wichtigsten sind

- die **Liste** `[a, b, c]`, die beliebige Datentypen (auch kombiniert und auch weitere Listen) enthalten kann. Listen sind veränderbar.
- Das **Dictionary** (ebenfalls veränderbar), das aus Schlüssel-/Wertpaaren besteht. Dabei können die Werte wiederum Dictionaries sein.
- Das Tupel, im Prinzip eine unveränderbare Liste

Funktionen sind in Python *first-order-Objekte* wie jeder andere Datentyp auch.

- Sie werden mit `def()` definiert:

```
def meineFunktion(a, b):
    meinWert = a*b
    return(meinWert)
```

- Da Funktionen *first-order-Objekte* sind, können sie sich auch selber aufrufen *(Rekursion)*.
- Funktionen können, müssen aber keinen Rückgabewert besitzen
- Alle innerhalb einer Funktion definierten Werte sind *lokal*

Dateien und Persistenz

Textdateien

Um eine Datei zu lesen, benötigt man erst einen *File Descriptor*, den bekommt man mit dem Befehl `open()`

```
fd = open("kant.txt")
```

Man beachte, daß Python die Datei auch finden kann! Das ist nicht immer so einfach, wie es scheint: Python erwartet die Datei im *current working directory*, also im aktuellen Arbeitsverzeichnis. Das ist normalerweise das Verzeichnis, in dem das Programm gestartet wird, IDEs und auch TextMate biegen dieses Verzeichnis oft um (Wurzelverzeichnis des Projekts).

Um sicherzugehen gibt man entweder den vollständigen Pfad zum Verzeichnis an, oder man läßt sich das *current working directory* von Python anzeigen:

```
import os
print(os.getcwd())
```

Dann kann man sich mit `os.join()` den Pfad sicher zusammensstellen:

```
path = os.path.join(os.getcwd(), "sources/kant.txt")
```

Und dann die Datei sicher öffnen:

```
fd = open(path)
```

Wenn man die Datei geöffnet hat, kann man ihren Inhalt in eine Variable schreiben und dann den Inhalt dieser Variable manipulieren:

```
content = fd.read()
print(content)
```

Vieles davon kann man sich aber ersparen, wenn man statt Apfel-R (Run File) Shift-Apfel-R (Run File in Terminal) nutzt.

Dateien schreiben

Um in eine Text-Datei zu schreiben, gibt es zwei verschiedene Modi:

```
fout = open("file1.txt", "w")
content = fout.write("Alles neu mächt der Mai!")
```

Wenn man mit dem Schreiben fertig ist, sollte man die Datei schließen:

```
fout.close()
```

Hier ist die Datei im `write`-Mode und wird bei jedem Schreibvorgang neu erstellt, das heißt der alte Inhalt wird überschrieben.

Daneben gibt es den `append`-Mode, hier wird der neue Inhalt an den alten angehängt:

```
fapp = open("file2.txt", "a")
content = fapp.write("Alles neu mächt der Mai!")
fapp.close()
```

Natürlich sollte man auch hier nicht vergessen, das Datei-Handle am (Programm-) Ende zu schließen.

Ausnahmen abfangen

Da beim Lesen und Schreiben von Dateien eine Menge schiefgehen kann, empfiehlt es sich, dieses abzufangen:

```
try:
    fin = open("boese_datei.txt")
    for zeile in fin:
        print(zeile)
    fin.close()
except:
    print("Es ist etwas faul im Staate Dänemark!")
```

Die Syntax ist ähnlich der einer `if`-Anweisung.

Andere Dateiformate

Für fast alle möglichen Dateiformate gibt es Python-Bibliotheken, die spezielle Methoden dafür bieten, die das Lesen und Schreiben dieser Dateien vereinfachen:

- Für Excel- und/oder CSV-Dateien gibt es das Modul `csv` aus der Standard-Bibliothek
- In der Regel nutze ich diese Bibliothek, wobei ich die Excel-Datei schon aus Excel heraus entweder als Komma-separierte (Text-) Datei (CSV) oder als Tab-separierte (TSV) Datei exportiere.

Falls das `csv`-Modul nicht ausreicht, bietet `pandas` nahezu alles, was man braucht, um mit Excel-Files und Dateien aus anderen Tabellenkalkulationen umzugehen:

```
import pandas as pd

data = pd.read_csv("meine_csv_datei.csv")
```

pandas nimmt per default die erste Spalte als *Label* der Reihe (und nicht als Wert). Dieses Verhalten kann man durch setzen von `index_col = False` abschalten.

Um eine CSV-Datei zu schreiben, reicht die `to_csv`-Methode:

```
data.to_csv("meine_csv_datei.csv")
```

Man kann mit pandas aber auch direkt Excel-Files lesen und parsen:

```
data = pd.read_excel("datei.xls", sheetname = "Sheet 1")
```

Und wenn man alle Sheets auf einmal lesen will, liest man eben alle auf einem Rutsch ein:

```
data = pd.ExcelFile("file.xls")
```

Und natürlich kann man mit pandas auch Excel-Dateien schreiben:

```
data.to_excel("file.xls", sheet = "Sheet1")
```

Pandas ist eine umfangreiche Bibliothek zur Datenmanipulation und -visualisierung. Sie operiert nicht auf Texten, sondern auf `Data Frames`, einer Datenstruktur, die sehr viel Ähnlichkeit mit Tabellen aus einer Tabellenkalkulation aufweist.

Dabei gibt es auch Merkwürdigkeiten und Fallstricke. Daher solltet Ihr vor der Nutzung einen Blick ins Handbuch nicht scheuen.

Für *JSON*-Dateien gibt es das Modul `json`, ebenfalls aus der Standard-Bibliothek. JSON-Dateien werden meist via einer API direkt aus dem Netz geladen, z.B so:

```
import json
import urllib2

weatherUrl = "http://api.openweathermap.org/data/…"
weatherData = json.load(urllib2.urlopen(weatherUrl))
```

JSON-Daten werden beim Laden in Dicitionaries umformatiert. Daher gibt es keine Garantie auf die Reihenfolge der Daten, die Ihr gegebebenfalls in der Dokumenation zur API findet!

HTML- und XML-Dateien parsen

- Um (X)HTML-Dateien zu parsen, reicht oft das Modul `html.parser` aus der Standard-Bibliothek aus
- Ein weiteres, viel verwendetes Moduls für HTML und XML ist die Bibliothek `Beautiful Soup`
- Da es aber noch viele andere, teils hochspezialisierte Parser gibt, lohnt sich – bevor man sich in das Abenteuer stürzt – eine Recherche

Objektorientierte Programmierung

Python ist eine **objektorientiert** Programmiersprache.

- Das heißt, Python kennt **Klassen**, **Objekte** und **Methoden**
- Objekte können Eigenschaften an Unterobjekte **vererben**
- Und leider kennt Python auch die **Mehrfach-Vererbung**

Klassen und Methoden

- Eine **Klasse** ist eine Objektdefinition. Sie definiert die Datenstrukturen und Methoden eines Objektes
- Eine Klasse »lebt« erst, wenn sie einem Objekt zugewiesen wurde
- Eine **Methode** ist einer Funktion ähnlich, jedoch
  - werden Methoden *innerhalb* einer Klasse definiert und
  - die Syntax für den Aufruf einer Methode unterscheidet sich von der Syntax für den Aufruf einer Funktion

Beispiel: Die aus Processing bekannte Klasse *PVector* in Python (Auszug):

```
import math

class PVector():

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self, v):
        self.x += v.x
        self.y += v.y

    def sub(self, v):
        self.x -= v.x
        self.y -= v.y
```

- Mit `class Klassenname()` wird eine Klassendefinition eingeleitet
- Wann `class Klassenname(object)` wirklich nötig ist, hat sich mir bisher nicht erschlossen. Es scheint so, daß es notwendig ist, wenn die Klasse einer *Oberklasse* sein soll
- Klassennamen werden per Konvention mit (mindestrns) einem Großbuchstaben am Anfang geschrieben
- Die Methode `__init__(parameter)` ist der *Konstruktor* einer Klasse (zwei Unterstriche rechts und links)

- Der Parameter `self` macht aus einer Funktion eine Methode. Er muß immer als *erster* Parameter übergeben werden.

Ein Objekt wird aus einer Klasse wie folgt erzeugt:

```
location = PVector(100, 100)
velocity = PVector(1.0, 3.3)

location.add(velocity)
print(location.x, location.y)
```

- Im Gegenssatz zu einer Funktion werden Methoden durch die Punkt-Notation, zum Beispiel `location.add()` aufgerufen.

- Das gilt auch für alle anderen Eigenschaften eines Objektes, sofern sie mit `self` zur Objekteigenschaft erklärt wurden:

```
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

- Der erste Parameter `self` wird also beim Aufruf mithilfe der Punktnotation vor dem Aufruf geschoben, aus *self* wird daher der Objektname:
  - `self.x` -> `location.x`
  - `add(self)` -> `location.add()`