Erste Python-Grundlagen

Hier finden Sie eine kurze Einführung in ausgewählte Python-Grundlagen. Ziel des Tutorials ist es, Ihnen eine erste Einführung in Python zu geben, sodass Sie dem Workshop folgen können. Zögern Sie nicht uns bei Fragen rund um dieses Tutorial direkt zu kontaktieren (buschhue@uni-potsdam.de, peter.wulff@uni-potsdam.de).

Führen Sie dieses Tutorial am besten gleich in Ihrer Python-Installation aus. Nutzen Sie dabei die Entwicklungsumgebung Spyder (oder eine Umgebung Ihrer Wahl).

1. Module

Analog zur Rs Paketen gibt es bei Python sogenannte Module (siehe auch die Installationsanleitung). Hierin befinden sich zum Beispiel nützliche Funktionen. Anders als bei R ist es in Python üblich die Module unter einem *alias* zu laden (`import Modulename as Aliasname`).

```
# erst laden wir einige Module

import pandas as pd # insbesondere für data frames
import sklearn as sk # für shallow machine learning
```

So lässt sich z.B. auf Funktionen zurückgreifen, ohne dass es zu Konflikten kommt, weil zwei Funktionen aus verschiedenen Modulen einen gleichen Namen haben.

Eine elementare Datenstruktur in python sind Pandas' DataFrames (analog zu Rs data.frame). Im Folgenden erstellen wir ein DataFrame aus dem Modul pandas:

```
# nutze die Klasse DataFrame aus pandas und erstelle daraus das Objekt DataFrame
dataframe = pd.DataFrame([1,2,3,4])
```

2. Zuweisungen

In Python lassen sich mit einem Gleichheitszeichen Variablen einen Wert zuweisen.

Im folgenden Beispiel wird der Variable x der Wert 12 zugewiesen.

```
x = 12
```

3. Daten Typen

In Python gibt es verschiedene Datentypen. Die für unsere Zwecke wichtigsten Datentypen sind:

- int: 1 (Integer, Ganze Zahl)
- float: 0.3 (Float: Gleitkommazahlen)
- bool: True, False (Boolean: Boolsche Werte)
- str: "Hello" (String: Zeichenkette)

```
# integer
a = 1
# float
b = 0.3
# boolean
c = True
# string
d = "Hello"
```

Der Typ eines Objektes (z.B. einer Variable) lässt sich mit der Funktion `type` ermitteln

```
type(b)
```

```
float
```

4. Komplexere Datenstrukturen: Listen (lists) und Wörterbücher (Dictionairies)

Im Folgenden finden Sie einige Datentypen, die sich aus obigen Datentypen zusammensetzen können

```
firstnames = ["David","Peter","Christina", "Tobias"]
lastnames = ["Buschhueter","Wulff","Meyer", "Schneider"]
ages = [35,30,12, 20]
mix = ["2",1] # die Elemente der Liste können auch diverse Typen enthalten
```

Listen sind geordnet. Das heißt, man kann über einen Index auf die einzelnen Elemente der Liste zugreifen.

```
firstnames[1]
```

```
'Peter'
```

Dabei zeigt sich schon ein Unterschied zu R. Python fängt bei null an zu zählen.

Zudem funktioniert das "Slicing" etwas anders: Hier hilft es die Kommas zu zählen (das erste Komma folgt nach David, das dritte nach Christina):

```
firstnames[1:3]
```

```
['Peter', 'Christina']
```

Im Folgenden erstellen wir ein dictionairy, das einem eindeutigen Key (hier der Vorname) einen Nachnamen zuordnet. Elemente im Python-Dictionary sind sog. `key`-`value`-Pairs. Der Key ist hier der Vorname und die Values sind die Nachnamen.

```
fullnames = {"David": "Buschhüter","Peter": "Wulff", "Christina": "Meyer"}
type(fullnames)
```

```
dict
```

Dictionairies sind ungeordnet. Das heißt wir können nicht wie oben (im Beispiel der Liste) mittels eines Index auf einzelne Elemente zurückgreifen. Wir können aber auf die Werte (z.B. Meyer) zugreifen in dem wir den Key (z.B. Christina) nutzen:

```
fullnames.get("Christina") # alternativ: fullnames['Christina']
```

```
'Meyer'
```

5. Methoden und Funktionen

In R gibt es im Allgemeinen Funktionen und Objekte. Die Funktionen können auf Objekte angewendet werden. In Python sind diese Operationen aber z.T. in den Objekten "versteckt". Dies haben wir z.B. schon oben gesehen : `fullnames.get("Christina")`. Die Methode ist hier Teil des Objekts `fullnames`. Man nennt diese Operationen **Methoden**

Dennoch gibt es auch **Funktionen**. Diese stehen für sich allein z.B. `len` (gibt die Länge eines objects an).

Für Experten: In Python gibt es sog. Dunder-Methods, die solche Operationen definieren. Bspw. kann mit `firstnames.__len__()` dieselbe Funktion ausgeführt werden.

```
len(firstnames)
```

```
4
```

```
len("David")
```

```
5
```

6. Vergleiche

In Python sind (wie in den meisten Programmiersprachen) Kontrollstrukturen definiert wie etwa Schleifen und konditionale Abfragen. Bspw. können Wenn-dann Abfragen sich z.B. wie folgt nutzen lassen (siehe: https://www.hdm-stuttgart.de/~maucher/Python/html/Kontrollstrukturen.html).

Dabei sind die Operatoren:

- == gleich
- != ungleich
- < kleiner als
- > größer als
- <= kleiner als oder gleich
- >= größer als oder gleich

Hier ein Beispiel mit Strings:

```
a = "Flora"
b = "Tanja"
a == b
```

```
False
```

Hier ein Beispiel mit Integeren:

```
a = 2
b = 42
b > a
```

```
True
```

7. Logische Abfragen

Die Operatoren für logische Abfragen sind

- and: und
- or: oder
- not: nicht (Verneinung)

```
# and
```

```
a = 1
b = 1
a == 1 and b == 1
```

```
True
```

```
# not - nicht (Verneinung)
not a == 1
```

```
False
```

```
# or - oder
a=1
c=2
a!= 1 or c==2
```

```
True
```

8. Wenn-Dann-Abfragen

Konditionale Abfragen werden ähnlich wie in R umgesetzt. Allerdings werden in Python Einrückungen z.B. Tabstobs, oder vier Leerzeichen verwendet (Good-Pratice: Immer die gleiche Art der Einrückung verwenden). Dazu werden die Befehle if (wenn), elif (anderenfalls, wenn) und else (sonst) verwendet. Es lassen sich so viele elif-Abfragen nutzen wie gewünscht.

```
a = 200
b = 200
if b > a: # wenn b größer a schreibe "b ist größer als a"
  print("b ist größer als a")
elif a == b: # wenn b gleich a schreibe "b ist größer als a"
  print("a und b sind gleich groß")
else: # sonst muss es wohl b ist kleiner als a sein
  print("a ist größer als b")
```

```
a und b sind gleich groß
```

9. Schleifen

Wie in vielen anderen Sprachen gibt es auch in Python Schleifen. Hier sei eine sogenannte for-Schleife dargestellt. Man beachte auch hier den Doppelpunkt und die Einrückung, hier ein Tabstop.

Die Schleife läuft über die Liste `ages`. Im ersten Durchlauf ist `i` das erste Element der Liste. Im zweiten Durchlauf, das zweite Element usw.

```
# wie alt sind die Personen dieses Jahr?
print("Alter:")
print(ages)

# wie alt sind sie nächstes Jahr?
for i in ages: # die Schleife läuft über die Liste ages
  age_x = i + 1 # ist das ursprüngliche Alter i+1 das neue Alter
  print("Neues Alter:")
  print(age_x)
```

```
Alter:
[35, 30, 12, 20]
Neues Alter:
36
Neues Alter:
31
Neues Alter:
13
Neues Alter:
21
```

10. Eigene Funktionen schreiben

Um eine Funktion selbst zu definieren, werden auch wieder Einrückungen und Doppelpunkte verwendet. Hier wird die Funktion `add_two_func` geschrieben, die den Wert `x` als (Input-)Parameter annimmt. `add_two_func` addiert den Wert zwei auf die Zahl `x` und gibt diesen neuen Wert `<(=x+2)` aus.

```
# diese Funktion addiert den Wert zwei auf eine Zahl
def add_two_func(x=42): # 42 ist die Defaulteinstellung
  y=x+2
  return y

print(add_two_func(x=1))
```

```
3
```

Und hier noch ein weiteres Beispiel zum selbst nachvollziehen:

```
# ein anderes Beispiel

def my_func_text(dein_name):
  out = dein_name + " ist super in Python."
  return out

print(my_func_text(dein_name = "Tobias"))
```

```
Tobias ist super in Python.
```

11. Klassen und Objekte (Instanzen)

Klassen sind so etwas wie Baupläne für Objekte. Wir schreiben hier einen einfachen "Bauplan" für Gemüse mit den Eigenschaften Farbe (`farbe`) und Geschmack (`geschmack`) den Methoden: Zeige den Geschmack des Gemüses an (`zeige_geschmack`) sowie zeige die Farbe des Gemüses an (`zeige_farbe`).

Die Methode `__init__` wird automatisch ausgeführt, wenn wir das Objekt initialisieren. Dabei werden dem Objekt die Eigenschaften Farbe und Geschmack zugeordnet.

```
class gemuese():
    """
    Eine einfache klasse
    """

    def __init__(self, farbe, geschmack):
        """
        intialisiere neues Gemüse

        * farbe (string): Farbe
        * geschmack (string): Geschmack des Gemüses
        """
        self.farbe = farbe # weise der Instanz den die Farbe zu
        self.geschmack = geschmack # ... und den Geschmack

    # definiere einfach die Methode des Zeigens der Farbe
    def zeige_farbe(self):
        return self.farbe

    # diese Methode fügt auch noch zwei ausrufzeichen hinzu
    def zeige_geschmack(self):
        return self.geschmack + "!!!"

# erstelle Instanz der Klasse
saure_Gurke = gemuese("grün", "sauer")
# wende die Methode zeige_farbe auf die Instanz an
print(saure_Gurke.zeige_farbe())
# wende die Methode zeige_geschmack auf die Instanz an
print(saure_Gurke.zeige_geschmack())
```

```
grün
sauer!!!
```

Das Arbeiten mit Klassen ist für R-User sicher etwas ungewohnt. Man kann sich aber vorstellen, dass damit interessante Möglichkeiten der Programmierung eröffnet werden. Wir werden sehen, wie Klassen verwendet werden können, um Textvorverarbeitung in sogenannten Pipelines zu systematisieren.

12. Aufgabe

Nachdem Sie den Code oben ausprobiert haben. Hier zwei kleine Aufgaben. Seien Sie dabei kreativ:

1) Überlegen Sie sich eine Aufgabe für eine Schleife. Programmieren Sie diese selbstständig oder passen Sie obige Schleife an.

2) Überlegen Sie sich eine Aufgabe für eine eigene Klasse. Programmieren Sie diese selbstständig oder passen Sie obige Klasse an.