python_jupyter_notebook

April 24, 2023

1 Python Einführung

Willkommen! Dies ist eine interaktive Umgebung namens *jupyter notebook*, die es erlaubt, Python Code schnell auszuführen. jupyter notebooks bieten sich vor allem an, um mit mehreren Personen zusammenzuarbeiten und Code zu präsentieren.

In diesem Notebook behandeln wir folgende Themen: 0. Einführung in die Nutzung von jupyter notebooks 1. Programmierung mit Python 3. Weiterführende Python Konzepte

Jedes Kapitel kannst du auf- und zuklappen. Sie enthalten #TODOs, sogenannte Platzhalter, die du mit deinem eigenen Code füllen kannst. Wir gehen heute jeden Schritt gemeinsam durch, du kannst jedoch auch gerne selbst in deinem Tempo daran arbeiten.

Happy coding!

1.1 0. Einführung in die Nutzung von jupyter notebooks (Wenn du in der Thonny IDE programmierst kannst du dieses Kapitel ignorieren)

Ein notebook besteht aus mehreren Zellen, wobei eine Zelle eine Text- oder Code-zelle sein kann. Du kannst Zellen hinzufügen, indem du die Maus mittig über diese Zelle führst und auf `+Code` oder `+Text` klickst.

Code-Zellen können..

- aus einem Kommentar bestehen oder
- Programmcode ausführen

[ ]:
```
print("Das ist ein Beispiel Programmcode, der etwas ausgiebt.")
```

[ ]:
```
# Dies ist ein Kommentar, d.h. das Programm macht nichts
```

1.2 1. Programmierung mit Python

In diesem Kapitel sehen wir uns die Programmiersprache Python an. Eine erste Anweisung hast du oben bereits geschrieben. Mit Python kannst du viele weitere Anweisungen schreiben, um somit bspw. Pixel aufleuchten zu lassen. Bevor wir dies tun, lernen wir erste Python Konzepte kennen:

- Befehle
- Variablen

- Konditionen
- Listen
- Schleifen
- Funktionen

1.2.1 Befehle

Beim Programmieren schreiben wir Anweisungen, die wir auch Befehle nennen. Du kannst einfache Befehle, wie Addition, Division oder Multiplikation, schreiben.

Dies ist ein print Befehl mit einer Addition, d.h. der Befehl schreibt das Ergebnis der Addition aus. Kommentiere den Befehl aus, indem du `#` löschst und führe ihn mit Play aus.

[ ]:
```
#print(1+2)
```

[ ]:
```
# TODO: Füge weitere Befehle hinzu
#print()
```

1.2.2 Variablen

Die Ergebnisse obiger Befehle kannst du in sogenannten *Variablen* speichern, die du beliebig nennen kannst. Variablen sind dafür da, um Werte zu speichern, und diese später wiederzuverwenden. Variablen können Zahlen, Texte, Bedingungen und mehr speichern.

Du kannst sie wie folgt definieren:

`name_der_variable = <Befehl>`

Wir gehen in diesem Kapitel auf die folgenden Datentypen ein: - Numbers (Zahlen) - Strings (Texte) - Bool (Werte `True` oder `False`)

Numbers (Zahlen) Wir nennen Zahlen *Numbers*, welche ganze Zahlen, Kommazahlen und mehr sein können. Ganze Zahlen nennen wir *integer*, kurz *int*.

Speichere ein Befehl in einer Variable namens `summe`. Du kannst sie auch umbenennen.

[ ]:
```
#summe = 1+2
#print(summe)
```

Du kannst einer Variable auch einen einfachen Wert zuweisen.

[ ]:
```
#alter = 27
#print(alter)
```

Du kannst Variablen wiederverwenden, indem du beispielsweise Berechnungen mit ihnen ausführst.

[ ]:
```
#alter_meines_bruders = alter - 2
#print(alter_meines_bruders)
```

Strings (Texte) Wir können nicht nur Zahlenwerte, sondern auch einen Text in einer Variablen speichern. Diese Texte nennen wir *string*. Strings werden mit Anführungsstrichen markiert.

Hier definieren wir die Variable namens `stadt` mit dem Text `Muenchen`.

[ ]:
```
#stadt = "Muenchen"
#print(stadt)
```

Wenn wir eine Variable mit mehreren Wörtern benennen möchten, trennen wir sie mit einem Untenstrich ab.

[ ]:
```
#mein_name = "Mai"
#print(mein_name)
```

1.2.3 Bedingungen und der Datentyp bool

Mithilfe von Bedingungen können wir bestimmte Befehle ausführen oder ignorieren. Eine Bedingung kann entweder `True` (wahr) oder `False` (falsch) sein. Die Werte `True` und `False` sind vom Datentyp `bool`. Eine Bedingung resultiert also in ein `bool`. Sie werden wie folgt definiert:

```
if (Bedingung):
  # tu etwas
else:
  # tu etwas anderes
```

Eine Bedingung kann bspw. anhand von numerischen Vergleichen geschrieben werden.

Beispiel:

```
Wenn (das Alter größer ist als 12 Jahre):
  dann schreibe aus, dass ich älter als 12 Jahre bin und am Girlsday
  teilnehmen kann
andernfalls
  schreibe aus, dass ich 12 Jahre oder jünger bin und leider noch nicht am
  Girlsday teilnehmen kann
```

Vergleiche zwischen zwei Zahlen kannst du mit den Zeichen $<$, $<=$ oder $>$ schreiben.

[ ]:
```
# TODO: Ergänze die untenstehende Bedingung im Code
#if alter > 12:
#  print("Ich bin älter als 12 Jahre und darf beim Girlsday teilnehmen.")
#else:
#  print("Ich bin 12 Jahre oder jünger und muss mich noch gedulden bis ich am␣
 ↪Girlsday teilnehmen kann.")
```

1.2.4 Listen

Wir haben einer Variable bisher nur einen Wert übergeben. Wir können jedoch auch mehrere Werte in einer Art Liste sammeln, was wir in Python `list` nennen. Eine `list` ist eine Datenstruktur, die mehrere Werte unterschiedlicher Typen (bspw.`int`, `string`) beinhalten kann. Man

kann eine Liste im Nachgang auch verändern, indem man Werte hinzufügt oder wegnimmt, und viele weitere Operationen mit Listen machen. Falls es euch interessiert, könnt ihr [hier](https://www.programiz.com/python-programming/list) reinklicken. Wir werden uns heute jedoch nur den Listenzugriff ansehen. Eine Liste mit Werte des Typen `string` kann wie untenstehend definiert werden:

```
name_der_liste = ["eintrag1", "eintrag2"]
```

Erstelle eine Liste Hier erstellen wir eine Liste mit `strings`, die wir in der Variable `familien_namen` speichern. Jeder Name bzw. Eintrag wird als *Element* oder im Englischen *Item* bezeichnet.

[ ]:
```
# TODO: Ersetze die Namen oder füge weitere hinzu, indem du diese mit einem␣
↪Komma abtrennst
#familien_namen = ["Maria", "Martin", "Mai"]
## index                          0                          1              ␣
↪        2
#print(familien_namen)
```

Zugriff auf Einträge einer Liste Listen sind sehr vorteilhaft, um mehrere Daten in einer Struktur zu speichern. Somit brauchen wir nicht für jeden Namen (oder auch das Alter) eine neue Variable zu definieren. Listen sind geordnet, d.h. sie enthalten eine Reihe von Einträgen in einer bestimmten Reihenfolge. Dabei nummerieren wir die Einträge einfach durch, d.h. der erste Eintrag wäre im obigen Beispiel der Name “Maria”, der zweite Eintrag der Name “Martin” und der dritte sowie letzte Eintrag ist der Name “Mai”. Diese Durchnummerierung nennen wir in der Programmierung *Index*. Ein Index ist eine Nummer, die wir verwenden, um auf die Einträge einer Liste zuzugreifen. Wir Informatiker starten beim Zählen mit der 0, d.h. das erste Eintrag besitzt den Index 0, der zweite den Index 1, usw. Der Zugriff erfolgt bspw. mit

```
name_der_liste[i] # i steht für eine Zahl zwischen 0 und der Länge der Liste - 1
```

Hiermit greifen wir auf den ersten Eintrag der Liste zu.

[ ]:
```
#print(familien_namen[0]) # Der index des ersten Eintrags ist 0
```

1.2.5 Schleifen

Schleifen sind dafür da, um Code, den wir oft schreiben, zu vereinfachen. Somit sparen wir uns die Schreibarbeit und der Code ist einfacher zu lesen. Du kannst Schleifen wie folgt definieren:

```
for eintrag in einer_liste:
  # tu etwas
```

Beispiel: Grüße alle Familienmitglieder, indem du ausschreibst “Hallo, `name`”. Dafür kannst du den print Befehl und die oben definierte Liste verwenden.

[ ]:
```
#print("Hallo, " + familien_namen[0])
#print("Hallo, " + familien_namen[1])
```

```
#print("Hallo, " + familien_namen[2])
```

Du hast nun denselben Code mehrmals geschrieben, bloß, dass jede Zeile einen anderen Index verwendet. Schleifen ermöglichen es dir, einen bestimmten Code mehrmals laufen zu lassen. Dafür musst du vorab wissen: Welchen Befehl möchtest du ausführen und wie oft soll der Befehl ausgeführt werden? Die Schreibweise hierfür wäre:

[ ]:
```
# familien_namen = ["Maria", "Martin", "Mai"]
#for name in familien_namen: # für jeden Namen in der Liste familien_namen
#  # im 1. Durchlauf ist name="Maria"
#  # im 2. Durchlauf ist name="Martin"
#  # im 3. Durchlauf ist name="Mai"
#  print("Hallo, " + name) # schreibe "Hallo, <name>" aus
```

[ ]:
```
# TODO: Hier kannst du eigene Schleifen bauen, bspw. mit einer neuen Liste␣
↪familien_alter
```

1.2.6 Funktionen

Bisher hast du wahrscheinlich gemerkt, dass wir vieles wiederholt und isoliert geschrieben haben, wie z.B. Variablendefinitionen. Wir möchten große Codeblöcke, wie Variablendefinitionen und Schleifen gruppieren, damit wir sie öfter wiederverwenden können. Dies bilden wir mit sogenannten Funktionen ab, die wir wie folgt definieren:

```
def name_der_funktion():
  # tu etwas
```

Du rufst eine Funktion einfach mit den Namen und den geschweiften Klammern auf

```
name_der_funktion()
```

Erstelle eine Funktion Diese Funktion heißt greet und schreibt ein einfaches Hallo aus.

[ ]:
```
#def sag_hallo():
#  print("Hallo")
```

Rufe obig definierte Funktion auf.

[ ]:
```
#sag_hallo()
```

Funktionen mit Parametern Funktionen kannst du sogar Informationen (sog. *Parameter*) mitgeben, die so benamt werden können, wie du möchtest. Sie sehen wie folgt aus:

```
def meine_funktion(information):
  # tu etwas
  print(information) # Beispiel: schreibe die information aus
  # tu noch etwas
```

Du rufst die Funktion dann einfach mit einer Beispiel Information auf:

`meine_funktion("Informationsbeispiel")`

Schreibe eine Funktion, die deinen Namen ausgibt.

[ ]:
```
#def sag_hallo(name): # die Funktion heißt sag_hallo und bekommt die␣
 ↪Information über einen Namen mit
#  print("Hallo, " + name + "!")
```

Rufe die Funktion mit deinem Namen auf.

[ ]:
```
#sag_hallo("Mai")
```

Aufruf von Funktionen in Funktionen Beim Programmieren versuchen wir so wenig wie möglich Befehle doppelt zu schreiben. Nehmen wir an, zu einer Begrüßung können neben dem Namen noch detailliertere Informationen zu einer Person angegeben werden. Dafür ziehen wir das Alter als Beispiel heran. Hier kannst du eine weitere Funktion schreiben. Eine Funktion kann dann auch in einer anderen Funktion aufgerufen werden.

```
def haupt_funktion(x):
  # Tu etwas
  eine_andere_funktion(x)
```

Dafür definieren wir zunächst eine Funktion, die dein Alter ausgiebt

[ ]:
```
#def schreibe_alter(alter):
#  # str() ist eine Standard Python-Funktion, die aus einem Integer einen␣
 ↪string-Typ erzeugt
#  # dies ist wichtig, wenn wir mehrere strings mit einem + zusammenführen␣
 ↪möchten
#  print("Ich bin " + str(alter) + "Jahre alt.")
```

[ ]:
```
# TODO: rufe diese Funktion mit einem Beispiel Alter auf
#schreibe_alter(27)
```

Nun wollen wir die Begrüßung sowohl mit einem Namen aus auch mit dem Alter schreiben. Dafür können wir eine neue Funktion schreiben, die beides macht.

Wenn wir für `sag_hallo` den Namen als Information brauchen, und für `schreibe_alter` das Alter, können wir in der neuen Funktion beide Informationen mitgeben.

Die Informationen `name` und `age` sind die Parameter der Funktion.

[ ]:
```
#def begruesse_mit_alter(name, alter): # die Funktion heißt begruesse_mit_alter␣
 ↪und nimmt die Parameter name und alter an
#  sag_hallo(name) # Hier rufen wir die Funktion sag_hallo mit dem Parameter␣
 ↪name auf
#  schreibe_alter(alter) # Hier rufen wir die Funktion schreibe_alter mit dem␣
 ↪Parameter alter auf
```

[ ]:
```
# TODO: Rufe die Funktion begruesse_mit_alter mit einem Beispiel Namen und
↪Alter auf

#begruesse_mit_alter("Mai", 27)
```

1.3 2. Weiterführende Python Konzepte

- Bibliotheken
- Tupel und Listen

1.3.1 Bibliotheken

Es gibt allgemein bekannte Berechnung, die sehr häufig im Alltag verwendet werden, wie die Zeitabfrage oder arithmetische Funktionen. Damit du nicht alle Funktionalitäten neu schreiben musst, bietet Python sogenannte Bibliotheken an, die Programmcode in Funktionen beinhalten, welche du einfach verwenden kannst. Du rufst sie wie deine eigenen Funktionen auf. Damit deine Python Datei jedoch die Funktionsnamen kennt, müssen sie einmal - meist zu Beginn einer Datei - vorher importiert werden. Die Importe schreibst du mit

- `from <package> import <module>` oder
- `import <package>`

Bibliotheksfunktionen kannst du dann mit einer Punkt-Notation aufrufen

`<module>.<function_name>()`

Welche Bibliotheken verfügbar sind, findest du online (https://docs.python.org/3/library/index.html). Da Python recht viele davon hat, kannst du heute einfach uns fragen :) Alle Bibliotheken, die wir brauchen, haben wir jedoch für heute vorgegeben.

Die `datetime` Bibliothek bietet dir Funktionen an, um das Datum oder die Zeit abzufragen. Zunächst importieren wir das Modul der Bibliothek.

[ ]:
```
from datetime import datetime
```

Frage sowohl das heutige Datum als auch die jetzige Zeit ab und speichere sie in der Variable `jetzt`.

[ ]:
```
jetzt = datetime.now()
print(jetzt)
```

1.3.2 Tupel und Listen

Bisher hast du die Datenstruktur `list` kennengelernt. Eine weitere ähnliche Datenstruktur nennt sich `tuple` und kann so wie die Liste mehrere Einträge beinhalten. Sie wird folgendermaßen definiert

`name_des_tuples = (1,2,3)`

Du siehst, dass wir hier statt eckige, geschweifte Klammern verwenden und die Einträge auch mit einem Komma abtrennen. Im Gegensatz zu Listen sind Tupel unveränderbar, d.h. man kann im Nachgang keine weiteren Einträge hinzufügen.

Wann würden wir Tupel eher als Listen verwenden?

- Wenn wir wissen, dass wir nur eine bestimmte Anzahl von Werten speichern wollen.
- Wenn wir Werte speichern möchten, die sich nicht mehr ändern.
- Oft verwenden wir unterschiedliche Datentypen (string, integers, ..) in Tupel und gleiche Datentypen in Listen (obwohl auch diese unterschiedliche Datentypen beinhalten können).

Erstelle ein Tupel Wir möchten Farbwerte in einem Tupel darstellen. Hierfür geben wir RGB (=Rot, Grün, Blau) Werte an. Das Tupel besteht also aus 3 Elementen, die für die Farben Rot, Grün, Blau stehen. Jedes Element kann einen Wert zwischen 0 und 255 annehmen, wobei 0 für wenig und 255 für einen hohen Anteil der Farbe steht.

Beispiele für Farben sind - rot: (255,0,0) - grün: (0,255,0) - blau: (0,0,255)

Wenn du eine andere Farbe darstellen möchtest, kannst du sie aus den drei Farben mischen, bspw: - gelb: (255,255,0)

Die RGB Codes kannst du auch online einsehen: https://www.rapidtables.com/web/color/RGB_Color.html

Hier definieren wir einen Tupel mit den RGB Werten für die Farbe Rot.

[ ]:
```
#rot = (255,0,0)
```

[ ]:
```
# TODO: definiere weitere Farben
```

Zeige deine Farben an (nur im Notebook) Die Farben, die du oben definiert hast, möchten wir nun visuell anzeigen.

Dafür importieren wir ein Modul der Bibliothek matplotlib.

[ ]:
```
#import matplotlib.pyplot as plt
```

Dieser Code zeigt deine Farbe.

[ ]:
```
# TODO: ersetze rot mit deiner Farbe
#plt.imshow([[rot]])
```