Einführung in die Programmierung mit Python

MARS – Center for Entrepreneurship

05.05.2022

Contents

Der Kurs richtet sich an Personen, die über keinerlei Erfahrung im Programmieren verfügen, dies aber lernen möchten.

Wir werden gemeinsam in entspannter Atmosphäre die ersten kleinen Programme erstellen und dabei die Grundlagen der Programmiersprache Python kennen lernen.

Kurzvorstellung

- Wer bin ich?
- Wer seid ihr und was wollt ihr am Ende des Kurses machen?

Los gehts

- Jupyter
  - https://jupyter.org/try-jupyter/lab/
  - Notebook Pyolite
  - Python im Webbrowser
  - Input-Zeile bedienen
  - Python Programm (Code) und Dokumentation (Markdown)

CHAPTER 1

Taschenrechner

Wir fangen ganz einfach an und schauen, wie sich Python als Taschenrechner macht:

```
1 + 2
```

```
3
```

```
2 * 3 * 4
```

```
24
```

```
1 + 3 * 14
```

```
43
```

```
(1+3) * 14
```

```
56
```

1.1 Aufgabe

? Jetzt seid ihr dran! Denkt an den Mathe-Unterricht zurück und versucht

- Minus
- Division
- Potenzieren
- Quadratwurzel

1.2 Lösung

```
5 - 2
```

```
3
```

```
5 / 2
```

```
2.5
```

Kommazahlen gibt's auch. Diese müssen, wie im Englischen üblich, mit Punkt statt mit Komma angegeben werden.

```
2 ** 6
```

```
64
```

Potenzieren war schon tricky. Die ** kann man sich aber einfach merken wenn man daran denkt das Potenzieren mehrfaches Multiplizieren ist.

Für die Quadratwurzel müssen wir aber noch erst noch ein bisschen über Python lernen.

CHAPTER 2

Packages

Der Kern der Programmiersprache Python kann relativ wenig. Die meiste Funktionalität liegt in Packages.

So ein Package muss mittels `import` verfügbar gemacht werden bevor man dessen Funktionen nutzen kann.

```
import math
math.sqrt(4)
```

```
2.0
```

2.1 Aufgabe

? Das Package `math` enhält noch eine Reihe weiterer Funktionen, die aus dem Mathe-Unterricht bekannt sind. Welche sind das?

2.2 Lösung

Durch ausprobieren kann man `sin`, `cos` und `tan` finden.

Das ist aber aufwändig. Einfacher geht's mit der Hilfe-Funktion.

CHAPTER 3

Hilfe

Hilfe kann man sich für Packages oder Funktionen anzeigen lassen:

`help(math)`

`help(math.sin)`

```
help(math.sin)
```

```
Help on built-in function sin in module math:

sin(x, /)
    Return the sine of x (measured in radians).
```

Das **/** ist kein Parameter sondern eine Info für fortgeschrittene Programmierer (Details). Wir ignorieren dies.

ℹ `<TAB>` zeigt verfügbare Funktionen an. Probiert:

`math.<TAB>`

`math.c<TAB>`

3.1 Aufgabe

? Berechne sin und cos von 0°, 90° und 180°!

3.2 Lösung

War das Ergebnis richtig?

Die Hilfe sagt, dass man den Parameter von `sin`/`cos` in Radians (Bogenmaß) angeben muss. Gegeben waren aber Grad. D.h. wir müssen dies erst umrechnen. Dazu gibt es auch eine Funktion `math.radians()`. Somit wäre die Lösung:

```
math.sin(math.radians(90))
```

```
1.0
```

ℹ Funktionen sind schachtelbar

ℹ Ein Programm besteht aus ineinander geschachtelten Funktionsaufrufen.

```
math.sin(math.radians(180))
```

```
1.2246467991473532e-16
```

Mmh, das sollte 0 sein.

ℹ Beim Rechnen mit Kommazahlen ist die Genauigkeit begrenzt.

? Was wäre, wenn es `math.radians()` nicht geben würde?

CHAPTER 4

Funktionen

Wenn es `math.radians()` nicht gäbe, dann könnten wir die uns selbst schreiben:

```
import math
def rad(x):
    return x * 2 * math.pi / 360
```

Zur Unterscheidung heißt sie einfach nur `rad()`. Das Package `math` muss importiert sein um `pi` zu verwenden.

```
rad(180)
```

```
3.141592653589793
```

4.1 Aufgabe

? Das sieht schon toll aus. Geht das aber auch besser?

4.2 Lösung

Beim Programmieren ist es oft nicht leicht zu entscheiden was **besser** ist.

Man könnnte `x * math.pi / 180` schreiben. Das spart evtl. die Berechnung von `2 / 360`. Warum `evtl`? Weil manche Programmiersprachen dies selbständig optimieren. Warum ist das kürzere nicht besser? Die Formel, die man in der Schule gelernt hat ist so nicht direkt wiederzuerkennen und somit das Programm schwerer zu verstehen.

Wichtig ist aber immer eine gute Dokumentation von Funktionen.

```
help(rad)
```

```
Help on function rad in module __main__:

rad(x)
```

Mmh, das sieht nicht so toll aus.

Es braucht einen Docstring, um hier etwas vernünftiges angezeigt zu bekommen.

4.3 Aufgabe

Schreibt die Funktion `rad()` so um, dass `help(rad)` einen hilfreichen Text anzeigt.

4.4 Lösung

Wir verwenden einfach den Hilfetext von `math.rad`.

```
def rad(x):
    """Convert angle x from degrees to radians."""
    return x * 2 * math.pi / 360
```

4.5 Aufgabe

Schreibt eine Funktion, die die Seitenlänge eines rechtwinkligen Dreiecks anhand des Satzes von Pythagoras berechnet. Zur Erinnerung, der lautet $a^2 + b^2 = c^2$.

Überlegt:

- Was sind die Parameter?
- Was soll das Ergebnis sein?
- Wie kann man das Berechnen?
- Wie soll die Funktion heißen?
- Was soll die Hilfe zeigen

```
def pyth(a, b):
    """Berechnet Hypothenuse eines rechtwinkligen Dreiecks mit Seitenlängen a und  b""
↪"
    return math.sqrt(a*a + b*b)
```

```
pyth(3, 4)
```

```
5.0
```

4.6 Aufgabe

? Schreibt eine Funktion, die die kleinere von zwei Zahlen zurück gibt.

4.7 Lösungen

Lösung 1

Das ist unnötig. Die Funktion gibt es schon. Sie heißt `min()`

Lösug 2

Das ist zwar unnötig, aber wenn es denn sein muss:

```
def my_min(a, b):
    return min(a, b)
```

Lösung 3

Wie man das selbst implementiert lernen wir erst im nächsten Kapitel.

CHAPTER 5

Bedinungen & Variablen

5.1 Bedingungen

Achtet auf Doppelpunkte und Einrückung!

```
def my_min(a, b):
    if a < b:
        return a
    else:
        return b
```

```
my_min(-2, -5)
```

```
-5
```

5.2 Kommentare

Programme können durch Kommentare leichter verstanden werden.

```
""" Das ist ein
    mehrzeiliger
    Kommentar """

sin(0) # Kommentar bis Ende der Zeile
```

ℹ Vernünftige Funktionsnamen sind aber noch wichtiger!

5.3 Variablen

In Funktionen habt ihr schon Variablen verwendet. Diese lassen sich aber auch direkt zuweisen:

```
a = 7
b = 5
c = a * b
# welchen Wert hat c?
c
```

```
35
```

```
b = 8
# welchen Wert hat c jetzt?
c
```

```
35
```

```
# c ist immer noch 35 weil es noch nicht neu berechnet wurde
c = a * b
# jetzt sollte es 56 sein
c
```

```
56
```

ℹ Warum muss `c` eingegeben werden um das Ergebnis zu sehen?

Ein Ergebnis wird nur angezeigt, wenn der Aufruf irgendetwas zurück gibt:

- Berechnung wie 1 + 2 -> Ergebnis
- Funktionaufruf -> das return-te
- Zuweisung a = b -> nix
- Variablenname -> Wert der Variable

CHAPTER 6

Texte

Bisher haben wir nur mit Zahlen gearbeitet, d.h. Python als besseren Taschenrechner verwendet. Jetzt wollen wir auch Texte verwenden.

Text wird auch Zeichenkette oder String genannt.

Texte haben wir auch schon gesehen, z.B.:

- Variablennamen `a`, `b`, `c`
- Funktionsnamen wie `my_min` und `cos`

Um Texte, mit denen wir rechnen wollen, von Variablen und Funktionen zu unterscheiden, müssen wir diese in " oder ' einschließen.

```
a = 'Wort'
b = "Zwei Worte"
c = """Ein mehrzeiliger Text
muss in
drei Hochkommatas
eingeschlossen werden."""
```

```
a
```

```
'Wort'
```

```
c
```

```
'Ein mehrzeiliger Text\nmuss in\ndrei Hochkommatas\neingeschlossen werden.'
```

ℹ mit \ lassen sich Sonderzeichen darstellen

- \ = \
- \n = Zeilenumbruch

- \t = Tabulator

6.1 Aufgabe

? Was kann man alles mit Texten machen?

Tipp:

- `c.<TAB>`
- `help(str)`

6.2 Lösung

Die Aufgabe war so einfach, die braucht keine Lösung.

6.3 Aufgabe

? Schreibt eine Funktion, die als Parameter ein englisches Wort akzeptiert und dessen Pluralform zurückliefert.

6.4 Lösung

Wir ignorieren die Sonderfälle und beschränken uns darauf, dBuchstaben **s** anzufügen wenn das Wort nicht damit endet.

```
def plural(s):
    if s.endswith('s'):
        return s
    else:
        return s + 's'
```

```
plural("horse")
```

```
'horses'
```

6.5 Rechnen mit Texten

Texte kann man addieren und multiplizieren:

```
a = "Hallo"
b = "Python"
a + '   ' + b
```

```
'Hallo   Python'
```

```
a * 3
```

```
'HalloHalloHallo'
```

Das kann man z.B. nutzen, um einen langen Strich zu erzeugen:

```
'_' * 80
```

```
'________________________________________________________________________________'
```

CHAPTER 7

Schleifen und Arrays

7.1 For-Schleifen

Addiere die Zahlen 1 bis 100 ist eine Aufgabe, die man effizient mit der Gaußschen Summenformel lösen kann. Wir wollen aber den Computer die einzelnen Berechnungen durchführen lassen. Hierfür muss dieser

- von 1 bis 100 Zählen
- und diese Zahl jeweils auf die bisherige Summe addieren

Als Funktion sieht das so aus:

```
def sum(start, end):
    sum = 0
    for i in range(start, end):
        sum = sum + i
    return sum
```

```
sum(1,101)
```

```
5050
```

ℹ Bei der Programmierung hält man sich i.d.R. daran, dass bei einem Bereich

- der erste Wert dazu gehört
- der zweite Wert aber nicht mehr.

Damit ist ein Jahr z.B. definiert als 1.1.2022 - 1.1.2023. Das gilt auch für die Funktion `range()`, die einen Zahlenbereich liefert.

7.2 Arrays

Bisher hatten Variablen genau einen Wert, eine Zahl oder einen Text. Manchmal ist es aber praktisch, mehrere Werte darin zu speichern, z.B. den zu erwartenden Kontostand in den nächsten Jahren wenn man Geld mit einem bestimmten Zinssatz angelegt hat.

Für so etwas gibt es Arrays.

```
# Array mit den Werten 1, 2 und 3
a = [1,2,3]
# 8 mal der Arry [1], d.h. [1, 1, 1, 1,1, 1, 1, 1]
b = [1] * 8
# Arrays kann man aneinanderfügen
c = a + b
c
```

```
[1, 2, 3, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
# die Reihenfole umdrehen - Achtung: es wird nichts zurückgeliefert, c wird intern␣
↪geändert
c.reverse()
c
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 3, 2, 1]
```

```
# Auf einzelne Elemente des Arrays kann man zugreifen
a[0] + a[1]
```

```
3
```

```
# Ein neues Element hinten an den Array anhängen
a.append(4)
a
```

```
[1, 2, 3, 4]
```

```
# Einen Array kann man statt range() auch in einer for-Schleife verwenden
# Hier erhält die Variable i in jeden Durchlauf der Schleife einen Wert des Arrays a
# Abhängig von der Berechung in der Schleife muss das Ergebnis (hier prod)
# vorher initialisiert werden
prod = 1
for i in a:
    prod = prod * i

prod
```

```
24
```

7.3 Aufgabe

? Schreibe eine Funktion, die als Parameter einen **Anlagebetrag**, einen **Zins** und die **Laufzeit** in Jahren entgegennimmt, die Zinsrechnung durchführt (Zinsen werden wieder angelegt) und einen Array zurückliefert, bei dem das 0te Element den aktuellen Geldbetrag enthält und die darauffolgenden Elemente den Kontostand nach jeweils n Jahren.

7.4 Lösung

```
def zinsrechnung(anlagebetrag, zins, jahre):
    """Berechnet den Kontonstand der nächsten Jahre

    anlagebetrag
      Der inital angelegte Betrag
    zins
      Der vereinbarte Zins. Dieser bleibt über die gesamte Laufzeit gleich. Ein Zins␣
↪von 1.5% muss als 1.5 angegeben werden.
    jahre
      Die Anlagedauer in Jahren.

    Das Ergebis ist ein Array mit jahre + 1 Elementen.
    Das 0. Element enthält den Anlagebetrag, das 1. Element den Gesamtbetrag nach␣
↪Verzinsung nach dem ersten Jahr, ...
    """

    result = [anlagebetrag]    # Initialisieren mit anlagebetrag
    factor = (1 + zins / 100) # ist für jeden Schleifendurchlauf gleich, muss man nur␣
↪einmal berechnen

    for i in range(0, jahre):
        result.append(result[i] * factor)

    return result
```

```
zinsrechnung(1000, 1, 3)
```

```
[1000, 1010.0, 1020.1, 1030.301]
```

7.5 Aufgabe

? Jetzt erweitern wir die vorherige Aufgabe. Der Zinssatz kann variabel sein. Er wird als Array neben dem anlagebetrag der Funktion übergeben.

7.6 Lösung

```
def variabler_zins(anlagebetrag, zinsen):
    """Berechnet den Kontonstand der nächsten Jahre

    anlagebetrag
      Der inital angelegte Betrag
    zinsen
      Array mit den Zinssätzen der nächsten Jahre. [1.0, 1.5] bedeutet,
      dass der Kontostand für 2 Jahre berechnet werden soll und im 1. Jahr 1.0%
      und im 2. Jahr 1.5% Zinsen gezahlt werden und die Zinsen wiederangelegt werden.

    Das Ergebis ist ein Array mit jahre + 1 Elementen.
    Das 0. Element enthält den Anlagebetrag, das 1. Element den Gesamtbetrag nach␣
 ↪Verzinsung nach dem ersten Jahr, ...
    """
    jahre = len(zinsen)
    result = [anlagebetrag]
    for i in range(0, jahre):
        result.append(result[i] * (1+ zinsen[i]/100))
    return result
```

```
variabler_zins(1000, [1.0, 1.5, 2.0])
```

```
[1000, 1010.0, 1025.1499999999999, 1045.6529999999998]
```

CHAPTER 8

Simulationen

Aufbauend auf der letzten Aufgabe möchten wir jetzt simulieren, welches Ertrag wir erwarten können, wenn der Zinnsatz zufällig schwankt. Dafür benötigen wir eine Möglichkeit, diesen Zufall zu berechnen.

8.1 Zufallszahlen

Das Package `random` enthält Funktionen zu Erzeugung von Zufallszahlen. Probieren wir es aus:

```
import random
```

`help(random)` liefert eine lange Dokumentation. Wir brauchen nur einen kleinen Teil davon.

```
# Initialisierung - gleicher seed liefert gleiche Zufallszahlen!
random.seed(42)
samples = 10
for i in range(samples):
    tmp = random.randint(0, 400)
    print(tmp / 100)
```

```
3.27
0.57
0.12
3.79
1.4
1.25
1.14
0.71
3.77
0.52
```

8.2 Aufgabe

Was tut das kleine Beispiel? Baue eine Funktion, die einen Array der Länge **n** zurückliefert. Der Inhalt sollen Zufallszahlen mit 2 Nachkommastellen sein die zwischen **from** und **to* liegen.

8.3 Lösung

```
def rand_numbers(n, start, end):
    digits = 2; # falls es mal nur 1 oder 3 Nachkommastellen werden sollen
    factor = 10 ** digits;
    result = []
    for i in range(n):
        result.append(random.randint(start * factor, end * factor) / factor)

    return result
```

```
rand_numbers(5, 2, 44)
```

```
[9.12, 36.56, 4.6, 4.44, 9.67]
```

8.4 Aufgabe

Schreibe eine neue Funktion, die diese Funktion als Zinsen in der Funktion zur variablen Zinsberechnung verwendet.

```
variabler_zins(100, [1.5, 2.5, 3.5])
```

```
[100, 101.49999999999999, 104.03749999999998, 107.67881249999998]
```

8.5 Lösung

```
def variabler_zufallszins(anlagebetrag, jahre, min_zins, max_zins):
    return variabler_zins(anlagebetrag, rand_numbers(jahre, min_zins, max_zins))
```

```
variabler_zufallszins(1000, 5, 0, 2)
```

```
[1000,
 1005.5000000000001,
 1011.4324500000001,
 1024.479928605,
 1040.256919505517,
 1040.8810736572204]
```

8.6 Aufgabe

Eine einzelne Simulation hat nicht viel Aussagekraft. Schreibe eine Funktion, die die Simulation **n** mal ausführt und von jeder Ausführung das Endergebnis zurückgibt.

8.7 Lösung

```
def simulate(anlagebetrag, jahre, min_zins, max_zins, n):
    result = []
    for i in range(n):
        tmp = variabler_zufallszins(anlagebetrag, jahre, min_zins, max_zins)
        result.append(tmp[-1]) # letztes Element in Array hat Index -1, vorletztes -2,
 ↪ usw.

    return result
```

```
results = simulate(1000, 5, 0, 4, 10)
results.sort()
results
```

```
[1074.1497323214344,
 1087.9676471345958,
 1088.0664280686938,
 1093.414803351509,
 1100.0531206370113,
 1100.331147723317,
 1102.3034250690189,
 1118.6554589470022,
 1131.6023469749832,
 1152.8333883377713]
```