Mathematische Operatoren & Funktionen

by Woche 2

Python ermöglicht alle grundlegenden mathematischen Berechnungen, die auch auf einem Taschenrechner durchgeführt werden können, mit einer Syntax[1], die einem z.B. durch Microsoft Excel zumindest teilweise geläufig ist.

Symbolische Operatoren

Die Operatoren für Addition, Subtraktion, Multiplikation und Division sind erwartungsgemäß:

```
10 + 3
```

```
13
```

```
10 - 3
```

```
7
```

```
10 * 3
```

```
30
```

```
10 / 3
```

```
3.3333333333333335
```

Für die Potenzierung wird ** genutzt:

```
10 ** 3
```

```
1000
```

[1]Syntax ist ein Satz von Regeln, die festlegen, wie Code geschrieben werden muss, damit der Computer ihn verstehen kann. Es ist wie die Grammatik für Computercode.

Mathematische Ausdrücke in Python folgen der normalen arithmetischen Reihenfolge der Operationen, also

1. `**`
2. `*` und `/`
3. `+` und `-`

Es können aber Klammern eingesetzt werden um eine andere Reihenfolge zu erreichen:

```
1 + 2 * 3 ** 2
```

```
19
```

```
((1 + 2) * 3) ** 2
```

```
81
```

Der Vollständigkeit halber soll erwähnt sein, dass es weitere, nicht so gängige Operatoren gibt. Beispielsweise wird mit `//` erst dividiert und dann der Quotient auf die nächste ganze Zahl abgerundet, sodass `10 // 3` = 3. Mit dem Modulus-Operator `%` hingegen wird der Rest, der bei der Division zweier Zahlen entsteht, zurückgegeben, sodass `10 % 4` = 2.

Funktionen

Funktionen in Python, ähnlich wie in Excel, sind vordefinierte Operationen, die bestimmte Aufgaben ausführen und dabei oft Eingaben (Argumente) benötigen. Die Syntax einer Funktion besteht aus ihrem Namen, gefolgt von Klammern, in denen die Argumente übergeben werden.

Erstes Beispiel sei die `round()` Funktion, welche in Basis-Python enthalten ist und mit der Zahlen standardmäßig auf die nächste ganze Zahl gerundet werden können. Man übergibt also eine Zahl als Argument und erhält die gerundete Zahl zurück. Die Funktion erlaubt aber auch ein zweites Argument, um auf eine bestimmte Dezimalstelle zu runden. Gibt man kein zweites Argument vor, so wird der entsprechende default-Wert verwendet, was in diesem Fall zu keiner Nachkommastelle führt.

```
# keine Nachkommastelle
round(123.456)
```

```
123
```

```
# zwei Nachkommastellen
round(123.456, 2)
```

```
123.46
```

> **Hinweis**
>
> Fügt man eine Raute `#` in den Code ein, so wird alles, was in derselben Zeile danach kommt, als **Kommentar** interpretiert und nicht ausgeführt. Kommentare sind demnach sozusagen nur für Menschen und nicht für den Computer. Kommentare sind eine Möglichkeit, um den Code zu dokumentieren, übersichtlicher zu machen und zu erklären, was er tut.
>
> Da wir Jupyter Notebooks inklusive Markdown-Zellen verwenden, kann argumentiert werden, dass diese #-Kommentare innerhalb des Codes weniger nützlich sind. Nichtsdestotrotz können sie eine gute Ergänzung sein. Außerdem muss auch klar sein, dass nicht alle Python-Programmierer eine Jupyter-Umgebung verwenden und demnach nur ggf. #-Kommentare und eben keine Markdown-Kommentare verwenden können.

Zwar können wir Argumente wie oben gezeigt einfach in der richtigen Reihenfolge getrennt durch Kommas übergeben, tatsächlich hat jedes Argument aber auch einen Namen, der in der Funktionssignatur definiert ist. Um die Argumentnamen einer Funktion in Erfahrung zu bringen, kann die Funktion `help()` verwendet werden:

```
help(round)
```

```
Help on built-in function round in module builtins:

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.  Otherwise
    the return value has the same type as the number.  ndigits may be
negative.
```

Das erste Argument heißt demnach `number` und das zweite `ndigits`. Wir können auch die Argumentnamen explizit angeben. Dies macht zum Einen den Code ggf. besser lesbar, zum Anderen können wir die Argumente dann auch in beliebiger Reihenfolge übergeben. Anders ausgedrückt: Nur wenn wir die Argumente explizit benennen, können wir sie in beliebiger Reihenfolge übergeben. Geben wir keine Argumentnamen an, so

müssen wir die Argumente in der Reihenfolge übergeben, in der sie in der Funktionssignatur definiert sind.

Schließlich sei noch erwähnt, dass auch eine negative Zahl an `ndigits` übergeben werden, um auf eine 10er-Stelle zu runden:

```
round(123.456, -1)
```

```
120.0
```

```
round(ndigits = -1, number = 123.456)
```

```
120.0
```

Selbstverständlich ist es auch möglich z.B. den Logarithmus einer Zahl zu berechnet. Allerdings gibt es dafür keine direkt verfügbare Funktion, sondern wir müssen auf ein *Modul* zurückgreifen, das diese Funktion enthält. Diese und andere Funktionen sind im Standardmodul `math` enthalten, sodass wir dieses zwar nicht installieren, wohl aber zunächst laden müssen.

```
import math # Das math-Modul laden
```

Erst nachdem wir diesen Code zum laden des Moduls haben laufen lassen, können wir die Funktion `math.log()` verwenden, um den Logarithmus einer Zahl zu berechnen. Demnach müssen wir hier das Modul `math` voranstellen und mit einem `.` mit der eigentlichen Funktion `log()` verbinden.

> **i** Hinweis
>
> Für besseren Lesefluss wird hier vorerst nicht detaillierter auf Module eingangen. Eine detailliertere Einführung kommt in einem späteren Kapitel.

Standardmäßig wird der *natürliche* Logarithmus von 100 berechnet, wobei die Basis des Logarithmus die Euler'sche Zahl e (≈ 2,718) ist. Es kann aber auch der Logarithmus zu einer beliebigen Basis berechnet werden, solange die als zweites Argument in der Funktion angegeben wird:

```
# Logarithmus zur Basis e
math.log(100)
```

```
4.605170185988092
```

```
# Logarithmus zur Basis 10
math.log(100, 10)
```

```
2.0
```

Analog kann auch die Exponentialfunktion (e^x) berechnet werden, wobei e wieder die Euler'sche Zahl ist. Schließlich kann auch die Quadratwurzel $\sqrt{(x)}$ gezogen werden:

```
math.exp(10)
```

```
22026.465794806718
```

```
math.sqrt(9)
```

```
3.0
```

Es können auch `math.ceil()` und `math.floor()` verwendet werden, um immer auf- bzw. abzurunden.

```
# Aufrunden
math.ceil(123.456)
```

```
124
```

```
# Abrunden
math.floor(123.456)
```

```
123
```

Module

Wie gerade gesehen, sind nicht alle Funktionen in Python direkt verfügbar. Stattdessen haben wir uns gerade einige zusätzliche Funktionen verfügbar gemacht, indem wir das `math`-Modul geladen haben. Module sind Sammlungen von Funktionen und Klassen, die in Python-Dateien gespeichert sind. Das `math` Modul ist dabei in der Python Standard

Library enthalten, also direkt nach Installation von Python verfügbar - muss aber eben noch geladen bzw. aktiviert werden.

Darüber hinaus gibt es viele Module, die von Dritten erstellt wurden und einmalig installiert werden müssen, bevor sie geladen und verwendet werden können. Die Installation solcher Module erfolgt dabei z.B. über den Befehl `pip install` (siehe Python Package Index (PyPI); später mehr). Eigentlich gehören viele der Module, die wir in diesem Kurs verwenden werden, zu diesen sogenannten *Third-Party-Modulen*, sodass wir sie erst installieren müssten. Da wir aber Python via Anaconda verwenden, sind viele dieser Module bereits installiert und können direkt verwendet werden.

Es sei noch erwähnt, dass statt “Modul” in diesem Kontext auch Begriffe wie “Bibliothek” oder “Paket/Package” verwendet werden. Diese Begriffe sind mehr oder weniger Synonyme, wobei “Modul” eher für einzelne Dateien steht, “Bibliothek” für eine Sammlung von Modulen und “Paket/Package” für eine Sammlung von Bibliotheken. So ist z.B. `pandas` ein einzelnes Modul, wohingegen `scikit-learn` eine Bibliothek/ein Paket aus mehreren Modulen ist.

Importieren von Modulen

Oben ist für `math` eine gängige Art gezeigt, ein Modul zu laden und dessen Funktionen zu nutzen. Hier sind alle Möglichkeiten, wie ein Modul geladen werden kann:

Option 1

Modul mit `import` laden und Funktion mittels Punkt-Notation nutzen.

```
import math

math.sqrt(9 + math.pi)
```

```
3.4844788209414896
```

Option 2

Modul mit `import` und `as` unter einem anderen Namen (Alias) laden und Funktion mittels Punkt-Notation nutzen.

```
import math as m

m.sqrt(9 + m.pi)
```

```
3.4844788209414896
```

Option 3a

Spezifische Funktion aus Modul via `from` und `import` laden und direkt nutzen. Option a: Funktionsnamen in einer Zeile durch Kommas getrennt.

```
from math import sqrt, pi

sqrt(9 + pi)
```

```
3.4844788209414896
```

Option 3b

Spezifische Funktion aus Modul via `from` und `import` laden und direkt nutzen. Option b: Ein Funktionsname je Zeile durch Kommas getrennt und in Klammern.

```
from math import (sqrt,
                  pi)
sqrt(9 + pi)
```

```
3.4844788209414896
```

Option 4

Alle Funktionen aus Modul via `from` und `import *` laden und direkt nutzen.

```
from math import *

sqrt(9 + pi)
```

```
3.4844788209414896
```

> **Hinweis**
>
> Optionen 1 und 2 sind die gängigsten und empfohlenen, da sie zwar mehr Code benötigen, dafür aber stets klar ist, welche Funktionen aus welchem Modul stammen. Das ist nicht nur für die Menschen nachvollziehbarer, sondern vermeidet darüber hinaus noch sogenannte Namenskonflikte. Ein Namenskonflikt tritt auf, wenn zwei der geladenen Module (zufälligerweise) Funktionen mit demselben Namen haben. In diesem Fall würde Python automatisch die Funktion des zuletzt geladenen Moduls verwenden, was zu unerwarteten Ergebnissen führen kann.
>
> Darüber hinaus sind sogar die Aliase (Option 2) für bestimmte Module so gängig, dass sie in der Python-Community als “Standard” gelten. So wird z.B. `import pandas as pd` und `import numpy as np` so häufig verwendet, dass es fast schon ungewöhnlich wäre, es nicht zu tun.

> **Weitere Ressourcen**
>
> - What are Python modules? **nur bis 3:53!**
> - Please NEVER Do THIS In Python :(

Übungen

Was ergibt round(111, −1)

- (A) −1
- (B) 110
- (C) 111.0

Was ergibt round(-1, 111)

- (A) −1
- (B) 110
- (C) 111.0

Welcher dieser Befehle führt nicht zu dem gleichen Ergebnis der anderen drei?

- (A) round(1, 1.33)
- (B) round(ndigits=1, number=1.33)
- (C) round(1.33, 1)
- (D) round(number=1.33, ndigits=1)