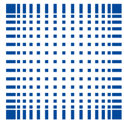


Python Programmierung (PYP)
Vorlesung - Hochschule Mannheim

Grundlagen

Prof. Thomas Smits

Sommersemester 2021

21. März 2022

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Python Programmierung (PYTHON) an der Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Inhaltsverzeichnis

1 Einführung **1**

1.1 Algorithmus [5] 1
1.2 Programmierung [8] 3
1.3 XKCD [9] 3
1.4 Eigenschaften von Python [10] 4
1.5 Ziele von Python [13] 6
1.6 Interaktiver Modus [14] 6
1.7 Skriptmodus [17] 7
1.8 Kommentare [20] 9
1.9 Werte und Typen [21] 9
1.10 Anweisung [22] 10
1.11 Ausdruck [23] 11
1.12 Ganzzahlen (Integer) [24] 11
1.13 Fließkommazahlen (Float) [26] 13
1.14 Variablen [27] 14
1.15 Referenz vs. Kopie [28] 15
1.16 Schlüsselwörter [30] 16
1.17 Vergleichsoperatoren [31] 17
1.18 Logische Operatoren [33] 18
1.19 Lesen von der Console [35] 18

2 Kontrollstrukturen **20**

2.1 Überblick [37] 20
2.2 If-Statement [38] 21
2.3 If-Else-Statement [40] 22
2.4 If-Elif-Else-Statement [42] 23
2.5 Ternäres If [44] 24
2.6 While-Schleife [45] 25
2.7 For-Schleife [46] 26
2.8 continue und break [47] 28
2.9 Pass [49] 30

3 Funktionen 31
3.1 Funktion definieren [51] . . . 31
3.2 Rückgabewerte [53] . . . 33
3.3 Funktionen aufrufen [55] . . . 35
3.4 Vararg-Funktion [58] . . . 37
3.5 Named Arguments [59] . . . 38
3.6 Default Argumente [60] . . . 39
3.7 Funktionen sind Objekte [61] . . . 40
3.8 Gültigkeitsbereich [62] . . . 41
3.9 Schlüsselwort global [63] . . . 43
3.10 Funktionen in Funktionen [64] . . . 44
3.11 Lambda [65] . . . 45
3.12 Docstring [66] . . . 46

4 Listen und Tuple 48
4.1 Listen [69] . . . 48
4.2 Slicing und Verketten [73] . . . 51
4.3 Multiplikation [74] . . . 53
4.4 Kopieren einer Liste [75] . . . 53
4.5 Packing und Unpacking [77] . . . 55
4.6 Funktionen für Listen [79] . . . 56
4.7 Über Listen iterieren [81] . . . 57
4.8 sum [83] . . . 58
4.9 filter [84] . . . 59
4.10 map [85] . . . 59
4.11 reduce [86] . . . 61
4.12 List Comprehensions [87] . . . 62
4.13 Tuples als unveränderliche Listen [88] . . . 62
4.14 Packing / Unpacking [89] . . . 63

5 Dictionaries und Sets 64
5.1 Dictionaries [92] . . . 64
5.2 Wichtige Funktionen [95] . . . 66
5.3 Sets [97] . . . 67
5.4 Wichtige Funktionen [99] . . . 69
5.5 Set Comprehension [101] . . . 70

6 Strings 72
6.1 Zeichenketten [103] . . . 72
6.2 Umwandlung von/in String [106] . . . 74
6.3 Escape-Sequenzen [109] . . . 77
6.4 String-Formatierung [111] . . . 78
6.5 Weitere Operationen für Strings [113] . . . 79

Inhaltsverzeichnis

Index iii

Kapitel 1

Einführung

1.1 Algorithmus [5]

Ein zentraler Begriff in der Programmierung ist „Algorithmus". Deswegen wollen wir uns zum Einstieg zwei Definitionen dieses Begriffes anschauen und deren Essenz herausziehen.

> Ein *Algorithmus* *(algorithm)* ist eine allgemeingültige endliche Folge von elementaren Operationen, die maschinell durchgeführt werden können. Die Folge enthält eine Beschreibung, wie diese elementaren Operationen nacheinander durchgeführt werden sollen. Insbesondere ist dabei eine Start- und Ende-Bedingung angegeben.
>
> G. Büchel, Praktische Informatik

Für Büchel ist also entscheidend, dass sich um eine

- endliche Folge
- von Operationen
- die maschinell ausgeführt werden können

handelt. Somit scheidet eine Anleitung wie „nach Gefühl mit Salz bestreuen" aus, weil sie nicht maschinell ausgeführt werden kann – zumindest nicht, solange Maschinen kein Gefühl haben. Genauso wenig kann „bestimme die Summe bis ins Unendliche" ein Algorithmus sein, weil hier keine *endliche* Folge von Anweisungen vorliegt.

Eine andere Definition liefert Boles, der aber auch auf die Präzision und die Ausführbarkeit durch einen Computer abhebt.

> *Algorithmus* *(algorithm)* – Arbeitsanleitung zum Lösen eines Problems bzw. einer Aufgabe, die so präzise formuliert ist, dass sie von einem Computer ausgeführt werden kann.
>
> Dietrich Boles

- Formulierung von Algorithmen
  - Umgangssprachlich
  - Programmiersprache
  - Aktivitätsdiagramme
- Ausführung von Algorithmen durch einen Prozessor (Mensch / Computer) → *Prozess* (*process*)

Generell ist erst einmal nicht festgelegt in welcher Form ein Algorithmus formuliert werden muss. Man kann ihn in umgangssprachliche Worte fassen, als Computerprogramm in einer Programmiersprache schreiben oder grafisch als Aktivitätsdiagramm darstellen. Entscheidend ist, dass man die verschiedenen Formen ineinander überführen kann und bei der Formulierung in Umgangssprache keine Doppeldeutigkeiten vorkommen, die eine Maschine nicht auflösen könnte.

Bei der Formulierung in einer Programmiersprache kann man auf jeden davon ausgehen, dass der Algorithmus maschinell ausführbar ist. Bei den anderen Formen bleibt ein Restrisiko, dass die Darstellung nicht präzise genug ist oder Dinge enthält, die man nicht einfach in ein Programm umwandeln kann.

Aufgabe: Berechne die Summe der Zahlen von 1 bis n

Umgangssprachliche Formulierung

```
Addiere für eine vorgegebene natürliche Zahl n
die Zahlen von 1 bis n. Dies ist das Resultat.
```

Programmiersprache: Python

```python
n = int(input())
erg = 0
aktZahl = 1

while (aktZahl <= n):
    erg = erg + aktZahl
    aktZahl = aktZahl + 1

print(erg)
```

Diesen Algorithmus könnte man auch problemlos als mathematische Formel darstellen, weil er hinreichend einfach ist.

$$e = \sum_{i=1}^{n} i$$

Eine direkte Darstellung als geschlossene mathematische Formel ist aber nur bei den wenigsten Algorithmen möglich, weil wir häufig Wiederholungen (Schleifen) und Auswahlen benötigen, um ein komplexeres Problem zu lösen.

1.2 Programmierung [8]

Aus einem Algorithmus wird ein Programm, indem wir ihn *programmieren*. Dabei geht es nicht nur darum, das eigentliche Programm in den Computer zu tippen, sondern in einem ersten Schritt das Problem so zu fassen, dass man ein Lösungsverfahren (Algorithmus) dazu entwickeln kann. Erst im letzten Schritt erfolgt die Formulierung in einer Programmiersprache und die Ausführung auf dem Computer.

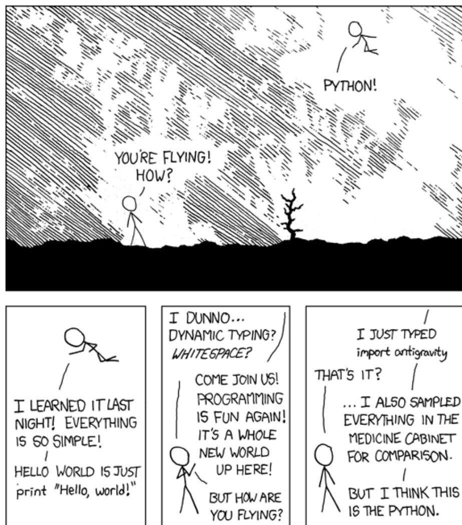
Gerade für Einsteiger in die Programmierung bietet es sich an, mehr Zeit in die Formulierung des Problems und die Bestimmung des Lösungsverfahrens zu stecken.

Programmierung: Weg von der Formulierung des Problems bis zum erfolgreichen Einsatz des ausführbaren Programms

1. Formulierung des Problems
2. Bestimmung des Lösungsverfahrens (Algorithmus)
3. Programmierung des Algorithmus in einer Programmiersprache

1.3 XKCD [9]

In diesem Kurs werden wir die Programmierung anhand der Programmiersprache Python lernen. Als Einstieg in das Thema bietet sich der folgende Comic von XKCD an.



Source: https://xkcd.com/353

Python löst also offensichtlich alle Probleme; es hilft sogar dabei, die Schwerkraft zu überwinden – man muss nur das richtige Modul (siehe unten) importieren.

1.4 Eigenschaften von Python [10]

Was ist Python?

- Entwickelt seit 1991 von Guido van Rossum in den Niederlanden
- Erste Veröffentlichung 1994 ⇒ genauso alt wie Java
- Extrem populär für wissenschaftliche Anwendungen

Guido van Rossum

Warum bietet sich Python als Programmiersprache für Einsteiger an? Gerade bei Menschen, die keine Informatiker sind (Ingenieure, Wissenschaftler etc.), ist Python ausgesprochen beliebt. Einer der Gründe ist, dass sich die Programme sehr schnell erstellen und testen lassen, weil man keinen Compiler benötigt, um sie auszuführen. Das Programm ist direkt nach dem Eintippen – oder sogar während der Eingabe – ausführbar. Python bietet eine relativ hohe Abstraktion und erlöst den Programmierer deshalb von einer ganzen Anzahl an Details, um die er sich in anderen Programmiersprachen kümmern müsste.

Natürlich kommen diese Vorteile nicht ohne Nachteile Nebenwirkung, so bietet sich Python nicht an, wenn man viel Kontrolle über die Details oder eine besonders hohe Performance benötigt. Außerdem ist die Sprache nicht typsicher, sodass bei größeren Programmen die Fehlersuche komplizierter werden kann als in einer typisierten Sprache, wie z. B. Java.

Vorteile von Python

- *Einfach* zu lernen
- Programme lassen sich *schnell* erstellen
- *Intuitive* Verwendung
- *Vielseitig* (↔ Matlab oder R)

Nachteile

- *Weniger Kontrolle* über Details (↔ C)
- Schlechtere *Performance* (↔ C)
- *Nicht typsicher*, Nutzer ist verantwortlich (↔ Java)

Was sind die zentralen Eigenschaften von Python?

- Skriptsprache (⇒ kein Compiler)
- multiparadigmatisch
  - objektorientiert
  - prozedural
  - funktional
  - Metaprogrammierung
  - aspektorientierte Programmierung
- alles ist ein Objekt
- dynamische Typisierung
- plattformunabhängig

Viele der hier genannten Eigenschaften klingen für einen Nichtinformatiker komplex und unverständlich.

Mit *Skriptsprache* ist gemeint, dass man Python-Programme direkt ausführen kann, ohne sie erst compilieren zu müssen. Der Programmquelltext wird direkt vom *Interpreter* ausgeführt; man benötigt keinen *Compiler*.

Multiparadigmatisch bedeutet, dass die Sprache nicht einem einzigen Programmierparadigma folgt, sondern eine ganze Reihe von verschiedenen Programmierstilen unterstützt. So kann man sowohl prozedural, als auch objektorientiert programmieren.

Bezüglich der Datentypen ist Python deutlich konsequenter als Sprachen, wie z.b. Java: Es wird nicht zwischen primitiven Datentypen und Objekten unterschieden; *alles ist ein Objekt*, auch die einfachen Datentypen, wie z. B. Ganzzahlen.

Außerdem wird der Entwickler davon verschont, sich Gedanken über die Typen seiner Variablen zu machen. Die Typen werden automatisch bestimmt, was als *dynamische Typisierung* bezeichnet wird. Eine Variable kann zu verschiedenen Zeiten, Werte unterschiedlichen Typs aufnehmen.

Beispiel: Dynamische Typisierung

```
a = "Hallo"
a = 3
```

In anderen Programmiersprachen muss man den Typ einer Variablen bei deren *Deklaration* festlegen, und kann ihn danach nicht mehr ändern. Man spricht hier von einer *statischen Typisierung*, weil zur Laufzeit keine Anpassungen der Typen möglich sind.

Beispiel: Statische Typisierung

```
String a = "Hallo";
int b = 3;
```

Des weiteren ist Python *plattformunabhängig*: Ein Programm, das unter einem Betriebssystem programmiert wurde, kann auf jedem anderen Betriebssystemen ausgeführt werden, für das es einen Python-Interpreter gibt.

1.5 Ziele von Python [13]

Guido van Rossum hat selbst einmal seine Ziele für Python formuliert.

- einfach
- so einfach zu lesen wie reines Englisch
- intuitiv
- gleichwertig mit den Konkurrenten bezüglich Mächtigkeit
- Open Source
- für tägliche Aufgaben geeignet sein
- kurze Entwicklungszeiten ermöglichen

Nach: van Rossum, Computer Programming for Everybody

Es ging ihm darum, eine Sprache zu entwickeln, die möglichst einfach ist, aber trotzdem für alle Aufgaben zu verwenden, für die auch die Konkurrenten von Python (Ruby, Perl, etc.) eingesetzt werden. Dabei wollte er die Syntax so gestalten, dass sich Python-Programme ähnlich wie englische Sätze lesen lassen.

Python ist *Open Source*, d. h. jeder kann sich den Quelltext der Programmiersprache herunterladen und ihn nach eigenen Bedürfnissen anpassen. Außerdem muss man für die Verwendung von Python kein Geld bezahlen, sondern erhält es kostenlos.

1.6 Interaktiver Modus [14]

Für Experimente mit Python bietet sich der interaktive Modus an. In ihm kann man die Python-Befehle direkt eintippen und sieht sofort das Ergebnis. Eine Besonderheit des interaktiven Modus ist, dass die Ergebnisse der Eingaben – anders als im Skriptmodus – direkt ausgegeben werden.

- Programme direkt interaktiv ausprobieren
- *REPL* = Read Evaluate Print Loop
- Start mit dem Kommando `python3`

```
~$ python3
Python 3.8.6 (default, Jan 27 2021, 15:42:20)
>>> print('Hello World')
Hello World
```

```
>>> 2**3
8
>>> exit()
~$
```

Im Skriptmodus müsste man `print(2**3)` schreiben, um das Ergebnis zu sehen. Im interaktiven Modus wird der Rückgabewert jedes Funktionsaufrufs und das Ergebnis jeden Ausdrucks direkt ausgegeben.

`exit()` ruft eine Python-Funktion auf (mehr dazu später), die den Interpreter beendet.

Wenn Sie in ältere Bücher oder Anleitungen zu Python schauen, werden Sie dort manchmal die Form `print 2**3` anstatt `print(2**3)` finden. Die Form ohne Klammern ist veraltet und wird nur in Python, Version 2 unterstützt.

Aufgabe:

Starten Sie den interaktiven Modus von Python und berechnen Sie die Anzahl der Reiskörner, die man auf dem letzten Feld eines Schachbrettes findet, wenn links oben mit einem Korn anfängt und dann die Anzahl mit jedem Feld verdoppelt. Oder anders: Berechnen Sie 2^{64}.

Lösung:

```
~$ python3
Python 3.8.6 (default, Jan 27 2021, 15:42:20)
>>> 2**64
18446744073709551616
```

1.7 Skriptmodus [17]

Der interaktive Modus ist nur dazu geeignet kurze Schnipsel auszuprobieren oder als Ersatz für einen Taschenrechner benutzt zu werden. Er ist ungeeignet für ernsthafte Programme – allein schon deshalb, weil man das Programm jedes Mal neu eintippen müsste. Will man Python-Programme entwickeln, verwendet man den Skriptmodus. Hierbei wird das Python-Programm in einer Textdatei abgelegt, die dann vom Interpreter ausgeführt werden kann.

- Die Befehle werden in ein Skript geschrieben (z. B. `hello-world.py`)
- Skript wird mit `python3 SKRIPTNAME` ausgeführt
- Auf Unix kann das Skript direkt gestartet werden, wenn die erste Zeile `#!/usr/bin/python3` ist (*Shebang*)

Datei: hello-world.py

```
#!/usr/bin/python3
print("Hello World")
```

Aufruf

```
~$ python3 hello-world.py
Hello World
```

Mit welchem Programm Sie die Datei erstellen ist egal, solange die Software wirklich reine Textdateien erzeugt. Textverarbeitungsprogramme, wie Word, scheiden also aus. Üblicherweise verwendet man einen *Editor*, der speziell für das Schreiben von Programmen entwickelt wurde, z. B. Visual Studio Code.

Die Dateiendung `.py` ist eine Konvention, an die man sich halten sollte, weil viele Werkzeuge nach der Endung schauen.

Die magische Zeile `#!/usr/bin/python3` hat folgenden Zweck: Unter Unix kann man ein Python-Skript direkt als ausführbar markieren (`chmod a+x DATEINAME`). Dazu muss die erste Zeile des Skripts den Pfad zum Python-Interpreter enthalten. Den speziellen Ausdruck `#!` bezeichnet man als Shebang). Er muss in der ersten Zeile des Skriptes stehen, um zu funktionieren. Hat man einen entsprechendes Shebang in der Skript-Datei untergebracht, kann man sie als ausführbar markieren und direkt aufrufen:

```
~$ chmod a+x hello-world.py
~$ ./hello-world.py
Hello World
```

Die Angabe des Interpreters entfällt. Das `./` ist nötig, da Linux aus Sicherheitsgründen keine Dateien im aktuellen Verzeichnis (`.`) ausführt, außer man weist es mit `./` explizit dazu an.

Aufgabe:

Öffnen Sie einen Editor, schreiben Sie ein Skript `bob.py`, das die Ausgabe „Hello Bob“ macht und starten Sie das Skript auf der Kommandozeile. Versuchen Sie das Skript über einen Shebang direkt ausführbar zu machen.

Lösung:

```
#!/usr/bin/python3
print("Hello Bob")
```

```
~$ chmod a+x bob.py
~$ ./bob.py
Hello Bob
```

1.8 Kommentare [20]

Nicht immer ist ein Programm vollkommen selbsterklärend, insbesondere für Personen, die es nicht programmiert haben. Deswegen hat man die Möglichkeit, das Programm mit Kommentaren zu versehen. Diese werden vom Python-Interpreter ignoriert, dienen aber dem Menschen dazu das Programm besser zu verstehen.

Kommentare werden in Python durch das `#`-Zeichen begonnen und erstrecken sich immer bis zum Ende der Zeile. D. h. alles zwischen `#` und dem Zeilenende wird vom Interpreter einfach ignoriert.

Alles nach einem `#` ist ein Kommentar

```python
# Ein Kommentar am Anfang der Zeile

print("Hello") # Ein Kommentar am Ende einer Zeile

# Ein mehrzeiliger
# Kommentar
```

Man kann Kommentare auch verwenden, um Programmteile für Tests oder während der Entwicklung kurzzeitig *auszukommentieren*. Im fertigen Programm sollten solche Teile aber nicht mehr enthalten sein, weil sie einen unfertigen und undurchdachten Eindruck erwecken und als *schlechter Stil* gelten.

1.9 Werte und Typen [21]

Wenn man ein Programm abstrakt betrachtet, dann bekommt es Eingabe-Daten, verarbeitet diese und produziert Ausgabe-Daten. Dieses einfache Modell gilt für alle Arten von Programmen, von der Textverarbeitung bis hin zu Cyberpunk 2077.

Daten (egal ob Eingabe- oder Ausgabedaten) haben zwei Eigenschaften:

- Sie haben einen *Wert*
- Sie haben einen *Typ*

- Programme manipulieren *Werte*, z. B. `"Hello World!"`, `42`, `6.023E23`, `True`
- Werte haben einen *Typ*

- Python kann man mit `type(WERT)` nach dem Typ fragen

```
>>> type("Hello World!")
<class 'str'>
>>> type(42)
<class 'int'>
>>> type(6.023E23)
<class 'float'>
>>> type(True)
<class 'bool'>
```

Die Unterscheidung ist wichtig, weil der *Typ* bestimmt, welche Operationen auf den Daten möglich sind und der *Wert* das Ergebnis der Operationen. Sie kennen diese strikte Trennung aus der Physik, bei der Sie auch immer die *Einheit* und die *Größe* betrachten. Eine Kraft wird in Newton angegeben und eine Strecke in Metern. Zur Berechnung eines Drehmomentes dürfen Sie die beiden Größen multiplizieren und erhalten als Einheit Newtonmeter (Nm). Eine Addition von Kraft und Strecke ist unsinnig und kommt deswegen nicht vor. Auch hier bestimmt die *Einheit* die möglichen Operationen und die *Größe* das Ergebnis.

Betrachten wir die Operation *Addition*: Diese ist nur sinnvoll für Daten, deren Typ numerisch ist, also `int` und `float` im Beispiel oben. Eine Addition zweier Wahrheitswerte (`True + False`) ist sinnlos. Im Gegensatz ergibt die Operation *logisches UND* keinen Sinn bei Zahlen, wohl aber bei Wahrheitswerten.

Deswegen müssen wir beim Programmieren nicht nur die Werte betrachten, sondern auch die Typen. In den meisten Fällen denkt man beim Programmieren nicht so intensiv über die Typen nach, weil man die richtigen Operationen intuitiv durchführt. Der Python-Interpreter muss hier mehr aufpassen und die Typen prüfen, damit keine sinnlosen Operationen erfolgen. Deswegen kann man den Interpreter mit der `type` auch danach fragen, welchen Typ ein Wert hat.

Die Schreibweise `type(...)` ist aus der Mathematik entlehnt, bei der man Funktionen definieren $f(x) = x^2$ und dann anwenden kann ($f(2) = 4$). Die Definition der Funktion `type` kennen wir nicht, sehen hier aber die Anwendung der Funktion.

1.10 Anweisung [22]

Eine *Anweisung* *(statement)* ist die kleinste ausführbare Einheit eines Programms und sagt dem Computer was er tun soll

Beispiele

- Zuweisen eines Wertes: `a = 5`
- Berechnung der Summe zweier Zahlen: `a = 5 + 6`
- Ausgabe eines Textes: `print("Hallo")`
- ...

In Python erstreckt sich eine Anweisung normalerweise auf genau eine Zeile. Das Zeilenende beendet auch die Anweisung. In der nächsten Zeile findet man dann die nächste Anweisung. Ausnahmsweise können sich Anweisungen über mehrere Zeilen erstrecken, dann gelten besondere Regeln, die hier erst einmal übersprungen werden. Andere Programmiersprachen, wie Java oder C, haben ein spezielles Zeichen (`;`), das die Anweisung eindeutig beendet. Bei diesen ist es egal, wie viele Anweisungen man in eine Zeile packt.

1.11 Ausdruck [23]

Ein *Ausdruck* *(expression)* verknüpft Operanden mit Hilfe eines Operators

- *Operand* *(operand)* ist der Wert der verknüpft werden soll
- *Operator* *(operator)* legt die Art der Verknüpfung fest
  (z. B. Addition mit +)

```
19 + 7
```

Um rechnen zu können, verwenden wir *Ausdrücke*. Mit ihnen können wir aus bekannten Werten neue Werte berechnen. Auch dieses Konzept ist bereits aus der Mathematik bekannt, so ist z. B. $x + 8$ ein Ausdruck, der aus einem beliebigen Wert x einen neuen Wert, der um 8 größer ist berechnet. Ein Ausdruck besteht aus Operatoren und Operanden. Die Operatoren werden auf die Operanden angewendet, um ein neues Ergebnis zu erhalten. Im Ausdruck $x + 8$ ist + der Operator und `x` und 8 sind die Operanden.

Ein Ausdruck besteht aus

- *Operanden* die Werte miteinander verknüpfen (im Beispiel `19` und `7`)
- *Operatoren* welche die Art der Verknüpfung festlegen (im Beispiel `+`)

Auch hier stellt sich wieder die Frage, welchen *Typ* das Ergebnis der Operation hat. Bei vielen Operatoren hängt der Typ des Ergebnisses vom Typ der Operanden ab. So ist z. B. der Typ von `5 + 2` eine Ganzzahl, weil beide Operanden Ganzzahlen sind. Der Typ von `5 + 2.0` ist allerdings vom Typ `float`, weil ein Operant vom Typ `float` ist.

Ein Ausdruck ist nicht zwingend identisch mit einer Anweisung, weil eine Anweisung aus mehreren Ausdrücken bestehen kann. So besteht z. B. die *Anweisung* `a = 4 + 5` aus zwei Ausdrücken: dem Ausdruck `4 + 5` und der Zuweisung `a = ...`.

1.12 Ganzzahlen (Integer) [24]

Einer der gängigsten Datentypen ist die ganze Zahl (Integer oder `int`). Python definiert eine ganze Reihe von Operatoren, die man auf ihnen anwenden kann.

- *Ganzzahlen* (*Integer*) stellen beliebig große, ganze Zahlen dar

- Operatoren
  - +, -: Addition und Subtraktion
  - *: Multiplikation
  - **: Potenzierung
  - /: Division
  - //: Ganzzahlige Division
  - %: Modulo

Eine Besonderheit, die man aus der Mathematik wahrscheinlich nicht kennt, sind die drei unterschiedlichen Divisions-Operatoren. Man kann dem Interpreter bei der Division zweier Ganzzahlen sagen, welchen Wert man als Ergebnis haben möchte haben möchte:

1. das Ergebnis als Fließkommazahl (`5 / 2` → `2.5`)
2. das Ergebnis als Ganzzahl (`5 // 2` → `2`)
3. den Divisionsrest (`5 % 2` → `1`)

Die anderen Operatoren sind wie aus der Mathematik bekannt. Da man Computerprogramme üblicherweise nur mit den Buchstaben des amerikanischen Alphabets schreibt, verwendet man für die Operatoren andere Zeichen, als man diese im Mathebuch finden würde.

```
## Addition und Subtraktion
5 + 2 # -> 7
5 - 2 # -> 3

# Division
5 / 2 # -> 2.5
5 // 2 # -> 2

# Modulo
5 % 2 # -> 1

# Multiplikation
5 * 2 # -> 10

# Potenzierung
5 ** 2 # -> 25
```

Eine Besonderheit von Python ist, dass Ganzzahlen keine Beschränkung bezüglich ihrer Länge haben, d. h. sie können beliebig große positive oder negative Zahlen darstellen.

Atome im Universum

```
>>> 10**89
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Andere Sprachen, wie Java, C oder C++, können dies nicht. Hier müssen Sie auf spezielle Bibliotheken zurückgreifen, wenn Sie Zahlen größer als 2^{64} verarbeiten wollen.

1.13 Fließkommazahlen (Float) [26]

Der zweite Datentyp für Zahlen in Python sind die Fließkommazahlen. Mit ihnen lassen sich rationalen Zahlen darstellen und Berechnungen auf Ihnen durchführen. Eine Darstellung irrationaler Zahlen ist nicht möglich, weil die Anzahl der Nachkommastellen begrenzt ist. Allerdings kann man auch irrationale Zahlen so weit annähern, dass es für die allermeisten Berechnungen ausreichend ist.

- *Fließkommazahlen* (*Float*) stellen rationalen Zahlen dar
- Genauigkeit und Wertebereich sind beschränkt
- Werden als Binärbrücke gespeichert ⇒ Rundungsfehler
- Operationen
  - `+`, `-`: Addition und Subtraktion
  - `*`, `**`: Multiplikation, Potenzierung
  - `/`, `//`: Division, ganzzahlige Division

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

Bei der Verwendung von Fließkommazahlen ist zu beachten, dass sie zum einen einen beschränkten Wertebereich haben, d. h. nicht beliebig große oder kleine Zahlen darstellen können. Zum anderen sind sie auch in der Genauigkeit beschränkt, können also nur eine bestimmte Anzahl von Nachkommastellen abbilden.

Da Fließkommazahlen im Computer in einer bestimmten binären Darstellung gespeichert werden, kann nicht jede rationale Zahl auch als Fließkommazahl dargestellt werden. Dies sieht man im obigen Beispiel: Obwohl `0.1 + 0.1 + 0.1` `0.3` ergeben sollte, hat das Ergebnis einen Rundungsfehler. Dieser basiert auf der internen Speicherung im Computer und ist nicht zu vermeiden.

Für technische Berechnungen sind die sehr klein Rundungsfehler im allgemein unwichtig, in der Finanzmathematik können sie aber erheblich stören, weil sie sich aufschaukeln können, weswegen man dort auf Ganzzahlen zurückgreift und einfach statt Euro 1/1000 oder 1/10000 Euro verwaltet.

1.14 Variablen [27]

Programme sollen möglichst universell funktionieren, d. h. mit beliebigen Eingabedaten benutzt werden können. Was nützt mir eine Umrechnung von kW in PS, wenn ich jedes Mal das Programm ändern muss, um einen anderen Wert einzugeben. Deswegen verwendet man anstatt fester Werte sogenannte *Variablen*. Diese können zu verschiedenen Zeiten verschiedene Werte annehmen, sodass man das Programm mit ganz unterschiedlichen Werten eingesetzt werden kann.

Variablen ordnen einem Wert einen Namen zu

```
nachricht = "Hallo, Welt!"
n = 42
k = 23
e = 2.71

print(nachricht) # -> "Hallo, Welt!"
print(n * k * e) # -> 2617.86
```

Es ist meist besser, eine Variable anstatt eines Wertes zu verwenden

- Code ist leichter zu korrigieren
- Namen sind leichter zu verstehen

Aus Sicht des Programmierers sind Variablen Platzhalter für Werte, die erst später (also nachdem das Programm fertiggestellt wurde) bestimmt werden. Mit Variablen kann man dann – wieder ähnlich zur Mathematik – mit den Platzhaltern anstatt mit den konkreten Werten arbeiten.

Die Syntax für die Verwendung von Variablen ist einfach: Wie in der Mathematik wird der Variabel mit = ein Wert zugewiesen (*Zuweisung*). Das Gleichheitszeichen heißt deswegen auch *Zuweisungsoperator*. Im Programm kann man dann anstelle der Werte die Variablen verwenden; zur Laufzeit werden die Variablen durch ihren konkreten Wert ersetzt.

```
a = 2
b = 4
print(a * b) # -> 8
b = 12
print(a * b) # -> 24
```

An dieser Stelle ein Wort der Vorsicht: Das Gleichheitszeichen (=) in der Mathematik und das Gleichheitszeichen (=) in der Programmierung haben eine unterschiedliche Bedeutung: In der Mathematik drückt = aus, dass die Variabel den Wert *hat*, in der Programmierung *weist* es der Variabel den Wert. Will man im Programm ausdrücken, dass eine Variable einen Wert *hat*, also sicherstellen, dass sie diesen Wert trägt, dann verwendet man das doppelte Gleichheitszeichen (==).

Im folgenden mathematischen Ausdruck, hat x von Anfang an den Wert 3, was wir durch elementare Umformungen zeigen

$$3x = 9 \Leftrightarrow x = 3$$

In einem Programm weist man der Variable einen Wert zu, um diese Variable dann zu verwenden. Das = ist also keine Prüfung eines Wertes, sondern eine Zuweisung. Das folgende Programm zeigt dies.

```
# Wir weisen x einen Wert zu
x = 4

# Wir prüfen, ob die Gleichung erfüllt ist
print(3*x == 9)
```

Ausgabe

```
False
```

Ein weiterer wichtiger Unterschied zur Mathematik besteht darin, dass Variablen ihren Wert beliebig oft ändern können. Eine Variable kann also mehr als einmal *zugewiesen* werden. Dies ist in der Mathematik nicht möglich. Das Konstrukt `i = i + 1` wäre in einer mathematischen Formel nicht zulässig. Hier müsste man für jede Veränderung eine neue Variable einführen.

1.15 Referenz vs. Kopie [28]

Eine kleine Komplexität versteckt sich noch hinter den Variablen: Abhängig von der Art der Variable wird sie unterschiedlich gespeichert. Bei den einfachen Datentypen in (`int`, `float`, …) enthält die Variable direkt den zu speichernden Wert. Bei komplexeren Datentypen (*Referenzdatentypen*) ist die Variabel nur eine Referenz auf den Wert. *Referenz* bedeutet, dass die Variable die Speicherstelle des eigentlichen Wertes enthält und man somit nur indirekt darauf zugreift.

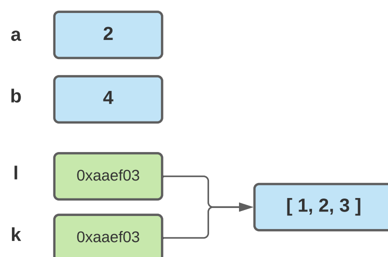
Dieser Unterschied könnte einem eigentlich egal sein, er zeigt sich aber, wenn man z. B. zwei Variablen für einen Wert hat. Bei den einfachen Datentypen, handelt es sich dann um eine wirkliche Kopie, bei den Referenzdatentypen verweisen beide Variablen auf dasselbe Datenobjekt. Dies wird dann sichtbar, wenn man die Daten über eine Variable verändert.

- Bei einfachen Datentypen (`int`, `float`, …) wird der Wert der Variable bei einer Zuweisung kopiert
- Bei komplexen Typen (Liste, Sequenzen, Dictionaries, …) ist die Variable nur eine Referenz auf das Objekt

```
a = 2
b = a
```

```
b = b + 2
print(a) # 2
print(b) # 4

l = [1, 2]
k = l
k.append(3)
print(l) # [1, 2, 3]
print(k) # [1, 2, 3]
```

Das Beispiel zeigt das beschriebene Verhalten deutlich. Im ersten Fall wird `b` durch die Zuweisung `b = a` eine Kopie des Wertes (2) aus `a` zugewiesen. Änderungen an `b` haben keine Auswirkungen auf `a`. Im zweiten Fall, sind `l` und `k` nur Referenzen, die auf eine einzige Liste mit dem Inhalt `[1, 2]` zeigen. Verändere ich die Liste über `k`, so ist sie auch bei einem Zugriff über `l` verändert.



1.16 Schlüsselwörter [30]

Welche Namen darf ich für Variablen verwenden? Um diese Frage zu beantworten, müssen wir uns zuerst überlegen welchen Namen wir *nicht* verwenden dürfen. Jede Programmiersprache definiert einen Satz sogenannter *Schlüsselwörter*. Das sind Wörter, die für die Programmiersprache reserviert sind und nicht als Variablennamen verwendet werden dürfen.

Python hat relativ wenige Schlüsselwörter verglichen mit anderen Programmiersprachen.

Schlüsselwörter sind reserviert und dürfen nicht als Variablennamen verwendet werden

```
False       def         if          raise
None        del         import      return
True        elif        in          try
and         else        is          while
as          except      lambda      with
assert      finally     nonlocal    yield
```

```
break          for          not
class          from         or
continue       global       pass
```

Jedes Schlüsselwort hat in Python eine besondere Bedeutung, so zeigt z. B. `if` an, dass eine Fallunterscheidung vorgenommen werden soll oder `return`, dass eine Funktion beendet wird. Die einzelnen Schlüsselwörter werden wir im folgenden noch behandeln.

1.17 Vergleichsoperatoren [31]

Algorithmen Leben davon, dass man *Fallunterscheidung* treffen kann, wie das genau geht, sehen wir im Kapitel zu den Kontrollstrukturen. Deswegen bietet Python eine Reihe von sogenannten *Vergleichsoperatoren* an, mit denen ich Werte miteinander vergleichen kann. Das Ergebnis eines Vergleichs ist immer eine Wahrheitswert (`True` oder `False`), der anzeigt, ob der Vergleich erfolgreich war oder nicht.

Vergleichsoperatoren vergleichen zwei Werte und geben `True` (wahr) oder `False` (falsch) zurück

- ==: Gleich
- !=: Ungleich
- >: Größer
- <: Kleiner
- >=: Größer-gleich
- <=: Kleiner-gleich

Warum für den Test auf Gleichheit das doppelte Gleichheitszeichen (==) benutzt wird und nicht das einfache (=) wurde bereits bei den Variablen erläutert: Wir haben das einfache schon für die Zuweisung „verbraucht“ und brauchen deswegen für den Vergleich ein anderes Zeichen.

```
5 == 2 # -> False
3 == 3 # -> True

5 != 4 # -> True
3 != 3 # -> False

5 < 3 # -> False
5 > 3 # -> False

3 > 3 # -> False
3 >= 3 # -> True
```

1.18 Logische Operatoren [33]

Die Vergleichsoperatoren liefern mir für jeden Vergleich einen Wahrheitswert. Wenn ich mehrere Wahrheitswerte miteinander verknüpfen möchte, muss ich zu den logischen Operatoren greifen. Diese bekommen einen oder zwei Wahrheitswert(e) und bilden daraus einen neuen Wahrheitswert.

Logische Operatoren (*logical operators*) verknüpfen Wahrheitswerte (`True`, `False`) miteinander

- `and`: logisches UND
- `or`: logisches ODER
- `not`: logisches NICHT

```
True and False # -> False
True and True # -> True

True or False # -> True
False or False # -> False

not True # -> False
not False # -> True
```

Vergleichsoperatoren und logische können verknüpft werden

```
# Ich
lieblingsfilm = "Avengers: Endgame"
geld = 12 # Euro

# Kino
film = "Avengers: Endgame"
eintritt = 13 # Euro

# Eisdiele
preis_spaghetti_eis = 6 # Euro

# Kann ich ins Kino?
(film == lieblingsfilm) and (eintritt <= geld) # -> False

# Kann ich Eis essen?
preis_spaghetti_eis <= geld # -> True
```

1.19 Lesen von der Console [35]

Nicht immer will man alle Werte fest im Programm hinterlegt haben, sondern man möchte sie von außen bekommen. Die einfachste Möglichkeit hierzu ist es, die Daten von der Konsole einzulesen.

- Die Funktion `input(prompt)` erlaubt es, Eingaben von der Konsole zu Lesen
- Man kann die Eingabe mit `int` und `float` in entsprechende Zahlen umwandeln

```
name = input("Wie heisst du?: ")
print("Hallo {}".format(name))
```

Ausgabe

```
Wie heisst Du?: Thomas
Hallo Thomas
```

Bei `input` handelt es sich um eine Funktion, die einen Wert zurückgibt. Diese Rückgabe ist dann genau die Eingabe des Benutzers. Man kann den Rückgabewert einer Variablen zuweisen und weiter verwenden.

In `print` wird die seltsame Konstruktion `"Hallo {}".format(name)` benutzt. Sie leistet folgendes: Alle Platzhalter `{}` in der Zeichenkette (String) werden durch die Variablen ersetzt, die in Klammern beim `.format()` angegeben sind. Die Details zu dieser Syntax können wir jetzt noch nicht erläutern, deswegen sollten Sie das vorläufig einfach so hinnehmen und benutzen. Details zu Strings – einem weiteren Datentyp – folgen später.

Das folgende Beispiel zeigt, wie der Datentyp der Eingabe angepasst werden kann, indem man `int(...)` benutzt.

Umwandlung mit int

```
name = input("Wie heisst Du?: ")
alter = int(input("Wie alt bist du?: "))

print("Hallo {}, du bist {} Jahre alt".format(name, alter))
```

Kapitel 2

Kontrollstrukturen

2.1 Überblick [37]

Selbst die einfachsten Algorithmen brauchen die Möglichkeit, Fallunterscheidungen zu treffen. Beispielsweise muss man für die Berechnung des Absolutbetrages einer Zahl unterscheiden, ob die Zahl positiv oder negativ ist.

Algorithmus: Absolutbetrag von a

```
1. Wenn a >= 0: gib a zurück
2. Wenn a < 0: gib -a zurück
```

In der Programmierung werden solche Fallunterscheidungen durch *Kontrollstrukturen* realisiert. Sie erlauben es, Anweisungen abhängig von einer Bedingung auszuführen. Trifft die Bedingung zu, werden die Anweisungen ausgeführt, trifft sie nicht zu, dann werden sie übersprungen.

- Programme müssen Entscheidungen treffen oder Aufgaben wiederholen können
- Hierzu dienen *Kontrollstrukturen* (*control structures*)
  - `if`, `elif`, `else`
  - `for`
  - `while`

Man unterscheidet bei den Kontrollstrukturen

- *Bedingungen*: Sie führen eine Gruppe von Anweisungen abhängig von einer Bedingung aus: `if`, `elif`, `else`
- *Schleifen*: Sie wiederholen eine Gruppe von Anweisungen abhängig von einer Bedingung: `for`, `while`

Der Unterschied zwischen den Bedingungen und Schleifen liegt darin, was mit den Anweisungen passiert, die von der Kontrollstruktur *kontrolliert* werden. Die Bedingung entscheidet, ob sie ausgeführt werden, die Schleife, wie oft sie ausgeführt werden. Da bei der Schleife auch die Möglichkeit besteht, die Anweisungen kein Mal auszuführen, umfasst sie auch die Entscheidung über das ob.

2.2 If-Statement [38]

Die einfachste Kontrollstruktur ist das `if`, das dazu dient eine Reihe von Anweisungen abhängig von einer Bedingung auszuführen. Die Bedingung wird nach dem Schlüsselwort `if` angegeben, die kontrollierten (bedingten) Anweisungen eingerückt in den folgenden Zeilen.

- Das *If-Statement* führt Anweisungen bedingt aus, d. h. nur wenn `<BEDINGUNG>` wahr ist, wird `<ANWEISUNGEN>` ausgeführt

Syntax If-Statement

```
if <BEDINGUNG>:
    <ANWEISUNGEN>
```

```
if ampel_farbe == 'grün':
    fahr_weiter()

print("Wird immer ausgeführt")
```

Im Beispiel wird die Variable `ampel_farbe` darauf geprüft, ob sie den Wert `'grÃ¼n'` enthält. Wenn dem so ist, wird die Anweisung `fahr_weiter()` ausgeführt. Enthält `ampel_farbe` einen anderen Wert, werden die eingerückten Anweisungen übersprungen und das Programm danach fortgesetzt, hier also bei dem `print`.

- Welche Anweisungen zur Kontrollstruktur gehören wird über die Einrückung festgestellt

```
bedingung = True

if bedingung:
    # Wenn bedingung wahr ist, wird dieser Code hier ausgeführt,
    # andernfalls nicht
    print('Bedingung ist wahr')
    print('sonst würde das hier nicht ausgegeben')

# Nicht eingerückte Zeilen werden auf jeden Fall ausgeführt
print('Hallo, ich komme immer raus')
```

Damit hat Python eine absolute Besonderheit, die keine andere Programmiersprache hat. Die Formatierung des Quelltextes ist relevant für die Semantik des Programms. In anderen Sprachen kann man das Programm formatieren, wie man will, da die Zuordnung zu den Kontrollstrukturen über spezielle Zeichen ({ und } in Java, C, C++) oder Schlüsselworte (`end` in Ruby) erfolgt.

Damit man auch bei Komplexen Kontrollstrukturen den Überblick behält, muss man eine konsistente Einrückung verwenden. Am besten immer zwei der vier Leerzeichen. Tabs sollte man nicht verwenden, da sie, sobald sie mit Leerzeichen gemischt werden, sehr verwirrend sind.

Das folgende Beispiel zeigt dies für ein *geschachteltes if*.

```
a = True
b = False
if a:
    print('a ist wahr')
    if b:
        print ('b ist wahr')
    print('gehört zu a')

print('wird immer ausgegeben')
```

Ausgabe

```
a ist wahr
gehört zu a
wird immer ausgegeben
```

2.3 If-Else-Statement [40]

Es gibt viele Fälle, in denen man nicht nur dann etwas ausführen möchte, wenn eine Bedingung erfüllt ist, sondern auch etwas tun möchte, wenn sie nicht erfüllt ist.

Für diesen Zweck kann man das *If-Else-Statement* verwenden, bei dem zusätzliche Anweisungen unter dem `else:` angegeben werden können, die ausgeführt werden, wenn die Bedingung nicht zutrifft.

Das *If-Else-Statement* erlaubt, weitere Bedingungen zu prüfen

- Wenn <BEDINGUNG> wahr ist, werden die <ANWEISUNGEN1> ausgeführt
- Wenn <BEDINGUNG> falsch ist, werden die <ANWEISUNGEN2> ausgeführt

Syntax If-Else-Statement

```
if <BEDINGUNG>:
    <ANWEISUNGEN1>
else:
    <ANWEISUNGEN2>
```

```
if ampel_farbe == 'grün':
    fahr_weiter()
else:
    halt_an()

print("Fahr'n, fahr'n, fahr'n auf der Autobahn")
```

In diesem Beispiel wird die Farbe der Ampel geprüft. Ist sie `'grÃ¼n'`, wird `fahr_weiter()` ausgeführt. Andernfalls wird `halt_an()` ausgeführt.

2.4 If-Elif-Else-Statement [42]

Welche Lösung gibt es jetzt für Probleme, bei denen mehr als eine Möglichkeit bestehen? Z. B. kann eine Ampel grün, gelb oder rot sein.

Will man mehr als eine Bedingung prüfen, kommt das *If-Elif-Else-Statement* zum Einsatz. Es erlaubt, beliebig viele Bedingungen zu testen und abhängig davon Statements auszuführen. Trifft keine Bedingung zu, werden die Statements im *else-Zweig* ausgeführt. Der else-Zweig kann auch fehlen, d. h. er ist nicht zwingend notwendig.

Syntax If-Else-Statement

```
if <BEDINGUNG1>:
    <ANWEISUNGEN1>
elif <BEDINGUNG2>:
    <ANWEISUNGEN2>
else:
    <ANWEISUNGEN3>
```

Das *If-Else-Statement* erlaubt, weitere Bedingungen zu prüfen

- Wenn <BEDINGUNG1> wahr ist, werden die <ANWEISUNGEN1> ausgeführt. <BEDINGUNG2> wird dann nicht mehr geprüft
- Wenn <BEDINGUNG1> falsch ist, wird <BEDINGUNG2> geprüft. Ist <BEDINGUNG2> wahr, werden die <ANWEISUNGEN2> ausgeführt.
- Sind weder <BEDINGUNG1> noch <BEDINGUNG2> wahr, werden die <ANWEISUNGEN3> ausgeführt

```
if ampel_farbe == 'grün':
    fahr_weiter()
elif ampel_farbe == 'gelb':
    gib_gas()
else:
    halt_an()
```

```
print("Fahr'n, fahr'n, fahr'n auf der Autobahn")
```

Das Beispiel prüft die Variable ampel_farbe erst auf den Wert 'grÃ¼n'. Ist dieser Test erfolgreich, wird fahr_weiter() ausgeführt und das Programm läuft bei dem print weiter.

Ist die ampel_farbe nicht 'grÃ¼n', wird sie auf 'gelb' getestet. Wenn das Ergebnis True ist, wird gib_gas() ausgeführt und es geht beim print weiter. Ist das Ergebnis allerdings False, dann wird die Anweisung nach dem else ausgeführt, d. h. halt_an, bevor es beim print weiter geht.

2.5 Ternäres If [44]

In der Programmierung kommt es häufig vor, dass man einen Wert abhängig von einer Bedingung setzen möchte. Deswegen gibt es für diesen Zweck eine spezielle, verkürzte Variante des if. Da es sich wie ein Operator mit drei Operanden verhält, wird es auch als *ternäres If* bezeichnet.

Die Syntax ist:

```
<WERT1> if <BEDINGUNG> else <WERT2>
```

Wenn die Bedingung <BEDINGUNG> wahr ist, wird <WERT1> zurück gegeben, wenn sie falsch ist <WERFT2>.

- Wenn if nur zwischen zwei Werten unterscheiden soll, kann man eine verkürzte Form nutzen: das *ternäre If*

Normales if

```
if a > b:
    max = a
else:
    max = b
```

Ternäres if

```
max = a if a > b else b
```

Beide Varianten sind vom Ergebnis her identisch. Der Unterschied besteht nur in der verkürzten Schreibweise und darin, dass bei der zweiten Variante sofort klar ist, dass die Variable max in Abhängigkeit von der Bedingung gesetzt werden soll. Bei der ersten Variante muss man etwas genauer hinschauen, damit das klar wird.

2.6 While-Schleife [45]

Will man eine Anweisung *mehrfach* ausführen, verwendet man eine Schleife. Mit „mehrfach“ kann

- kein Mal
- ein Mal
- n Mal

gemeint sein.

Die einfachste Schleife in Python ist die `while`-Schleife. Sie prüft eine Bedingung und wiederholt die zugeordneten Anweisungen (*Schleifenrumpf* genannt), solange diese Bedingung wahr ist.

Ist die Bedingung beim ersten Aufruf der Schleife nicht wahr, wird der Rumpf der Schleife überhaupt nicht ausgeführt. Deswegen spricht man von einer *kopfgesteuerten Schleife*.

- Die *While-Schleife* wiederholt `<ANWEISUNGEN>` so lange, wie `<BEDINGUNG>` wahr ist

Syntax While-Schleife

```
while <BEDINGUNG>:
    <ANWEISUNGEN>
```

Beispiel: While-Schleife

```
i = 1
while i < 5:
   print(i**2) # -> 1,4,9,16
   i = i + 1 # erhöht i um eins
```

Wenn man die Variablen, die in die Bedingung der Schleife verwendet wird, innerhalb der Schleife nicht verändert, dann liegt ein Programmierfehler vor: Die Schleife kann nicht enden, es liegt eine *Endlosschleife* vor.

Fehler: Endlosschleife

```
i = 1
while i < 5:
   print(i**2)

print("Diese Stelle wird nie erreicht")
```

Die While-Schleife prüft die Bedingung vor der ersten Ausführung, sodass diese am Anfang wahr sein muss, damit der Schleifenrumpf mindestens einmal ausgeführt werden soll. Will man, dass die Schleife mindestens einmal ausgeführt wird, also eine *fußgesteuerte Schleife* ist, dann muss man eine weitere Variable einführen, welche die Schleife kontrolliert.

Fußgesteuerte Schleife in Python

```
weiter = True
i = 0
while weiter:
    print(i**2)
    i = i + 1
    weiter = (i < 5)
```

Andere Programmiersprachen haben für diesen Zweck eine andere Form der Schleife (do-while-Schleife), Python verzichtet aber darauf und überlässt es der Programmiererin sich eine entsprechende Schleife selbst zu bauen. In C sähe das Beispiel mit einer do-while-Schleife wie folgt aus:

Fußgesteuerte Schleife in C

```
do {
    printf("%d\n", i * i);
    i = i + 1;
} while (i < 5);
```

2.7 For-Schleife [46]

Einer der häufigsten Anwendungszwecke von Schleifen besteht darin, eine Operation auf die Elemente einer Liste anzuwenden, z. B. um die Durchschnittsnote aller Studierenden der Vorlesung zu bestimmen:

Algorithmus: Durchschnittsnote

```
noten_summe = 0
anzahl = 0

Für jeden Studierenden s in der Liste ls
  noten_summe = noten_summe + Note von S
  anzahl = anzahl + 1

durchschnittsnote = noten_summe / anzahl
```

Listen werden in einem späteren Kapitel erläutert. An dieser Stelle muss als Einführung reichen, dass man Listen durch eckige Klammern symbolisiert und die einzelnen Elemente darin auflistet. Die Liste mit den Noten aus dem Algorithmus oben könnte man in Python schreiben als: `[ 1.3, 2.3, 1.7, 2.7, 4.0, 1.3 ]`.

- Die *For-Schleife* erlaubt es Anweisungen eine definierte Anzahl von Malen zu wiederholen. Dabei nimmt die `<VARIABLE>` nacheinander die Werte aus `<LISTE>` an.

Syntax For-Schleife

```
for <VARIABLE> in <LISTE>:
    <ANWEISUNGEN>
```

```
for i in [1, 2, 3]:
    print(i)
```

Ausgabe

```
1
2
3
```

Wer `for`-Schleifen aus anderen Programmiersprachen kennt, wird sich über die sehr beschränkten Möglichkeiten dieser Schleife in Python wundern. Dahinter steckt die Philosophie von Guido van Rossum, der die Sprache sehr schlank und einfach halten wollte.

Der Algorithmus zur Durchschnittsnote sieht in Python dann wie folgt aus:

```
noten_liste = [ 1.3, 2.3, 1.7, 2.7, 4.0, 1.3 ]
noten_summe = 0
anzahl = 0

for note in noten_liste:
    noten_summe = noten_summe + note
    anzahl = anzahl + 1

durchschnittsnote = noten_summe / anzahl
print(durchschnittsnote) # -> 2.216666666666667
```

Will man mit `for` über fortlaufende ganze Zahlen laufen, kann man anstatt einer Liste die `range()`-Funktion verwenden, die entsprechende Elemente liefert.

Mit range

```
# Laufe über die Elemente 1 bis 4
for i in range(1, 5):
    print(i)
```

Mit Liste

```
# Laufe über die Elemente 1 bis 4
for i in [1, 2, 3, 4]:
    print(i)
```

Bei `range` ist zu beachten, dass die obere Grenze exklusiv ist, d. h. `range(a, b)` liefert die Elemente von `a` bis `b - 1`.

2.8 continue und break [47]

Es gibt in der Programmierung eine Reihe von Fällen, in denen man eine Schleife vorzeitig beenden möchte. Ein typisches Beispiel ist die Suche nach einem Wert (z. B. in einer Liste oder bei einer Berechnung): sobald der Wert gefunden ist, kann man die Schleife beenden und braucht nicht weiterzusuchen. Dies wird in Python mit dem Schlüsselwort `break` erreicht.

Ein anderer Fall liegt vor, wenn man die Schleife zwar weiterlaufen lassen möchte aber der aktuelle Schleifendurchlauf nicht mehr benötigt wird, z. B. weil man eine Element vor der Flinte hat, dass nicht bearbeitet werden muss. Hier kann man mit dem Schlüsselwort `continue` die Schleife dazu auffordern, sofort den nächsten Durchlauf zu starten.

- `break` springt aus der umgebenden Schleife heraus

```
N = 64
for i in range(0, 10):
    if i**2 == N:
        print("Die Wurzel von", N, "ist", i)
        break # Wurzel gefunden, springe aus Schleife

print("Nach der Schleife")
```

Ausgabe

```
Die Wurzel von 64 ist 8
Nach der Schleife
```

Man kann sich `break` als einen Sprung zum ersten Statement nach der Schleife vorstellen, hier im Beispiel das `print`.

- `continue` springt zu nächsten Iteration der umgebenden Schleife

```
for num in range(2, 10):
    if num % 2 == 0:
        print(num, " ist gerade")
        continue
    print(num, " ist ungerade")
```

Ausgabe

```
2  ist gerade
3  ist ungerade
4  ist gerade
5  ist ungerade
6  ist gerade
...
```

Das Beispiel ist etwas konstruiert, weil man normalerweise einfach ein `if-else` verwenden würde:

```
for num in range(2, 10):
    if num % 2 == 0:
        print(num, " ist gerade")
    else:
        print(num, " ist ungerade")
```

Trotzdem kann man die Funktionsweise von `continue` gut erkennen.

Generell ist die Verwendung von `break` und `continue` umstritten. Die einen sehen darin eine willkommene Möglichkeit, Schleifen effizient zu programmieren, die anderen eine versteckte Form des unbedingten Sprungs (`goto`), der seit 1968 verpönt ist (siehe Edgar Dijkstra: Go To Statement Considered Harmful).

Man kann jedes Problem auch ohne Verwendung von `break` oder `continue` lösen, indem man eine weitere Kontrollvariable einführt. Ob das immer elegant und sinnvoll ist, muss jede Entwicklerin selbst einscheiden.

Wurzelberechnung ohne break

```
N = 64
i = 2
cont = True
while i < 10 and cont:
    if i**2 == N:
        print("Die Wurzel von", N, "ist", i)
        cont = False
    i = i + 1

print("Nach der Schleife")
```

Zumindest bei der Wurzelberechnung scheint die Variante mit `break` die bessere zu sein.

2.9 Pass [49]

Eine Besonderheit von Python, die man in anderen Programmiersprachen so nicht kennt, ist das pass-Statement. Es ist eine Anweisung, die nichts tut und nur dazu da ist, syntaktische Anforderungen der Sprache zu erfüllen.

- Python benötigt immer mindestens eine Anweisung in der Kontrollstruktur
- pass ist eine Anweisung, die nichts tut

```
if ampel_farbe == 'grün':
    pass # Kommt später, TODO
elif ampel_farbe == 'gelb':
    gib_gas()
else:
    halt_an()
```

Sie benutzen also pass immer dann, wenn die Syntax von Python eine Anweisung erfordert, Sie aber keine angeben wollen. Ein völlig korrektes, allerdings nutzloses, Python-Programm könnte so aussehen:

```
pass
pass
pass
pass
```

Kapitel 3

Funktionen

3.1 Funktion definieren [51]

Die Programmierung wäre ein mühsames Geschäft, wenn man nur Anweisungen und Kontrollstrukturen zur Verfügung hätte, weil man so das Rad immer wieder neu erfinden müsste. So wie wir in der Mathematik vorgefertigte Berechnungen in Form von Funktionen (z. B. sin() oder cos()) benutzen, möchten wir in einem Programm wiederkehrende Aufgaben und Algorithmen zusammenfassen und beliebig oft nutzen können.

- Gewisse Berechnungen oder Aktionen benötigt man immer wieder
- *Funktionen* (*functions*) erlauben es, wiederverwendbare Folgen von Anweisungen zu schreiben

```
import math # Mathematische Funktionen laden

radius = 2.5
umfang = 2 * math.pi * radius

## Später im Programm
radius = 3.7
umfang = 2 * math.pi * radius
```

Das Beispiel zeigt das Problem deutlich: Die Berechnung des Umfangs erfolgt immer wieder nach demselben Muster. Hat man einen Fehler gemacht, muss man ihn an allen Stellen finden und beheben.

Die Bedeutung des `import` am Anfang des Programms wird im Kapitel zu Modulen erläutert. Hier sei nur erwähnt, dass es dafür sorgt, dass man auf die Konstante `math.pi` für π zugreifen kann.

In Python können wir mit einer *Funktion* eine beliebige Anzahl von Anweisungen zusammenfassen und wiederverwenden. Dazu hat eine Funktion

- einen *Namen*,

- eine *Argumentliste* und
- eine *Rückgabe*.

Der Name dient dazu, die Funktion später aufzurufen, d. h. wir sprechen Funktionen über ihren Namen an, so wie Sie in der Mathematik die Sinus-Funktion über den Namen `sin` ansprechen.

Die Argumentliste benutzen wir, damit Funktionen mit unterschiedlichen Eingabewerten arbeiten können. Auch das ist keine Überraschung, denn auch in der Mathematik nutzt uns eine Funktion wenig, wenn wir keine Werte übergeben können, z.b. $\sin(2\pi)$. Ein anderes Wort für *Argument* ist *Parameter*. Im Unterschied zu mathematischen Funktionen haben wir es bei der Programmierung oft mit Funktionen zu tun, die mehr als einen Parameter bekommen (z. B. `range`) oder gar keine Parameter haben.

Wenn eine Funktion mit ihrer Arbeit fertig ist, kann sie etwas zurückgeben, also eine Rückgabe machen. Python hat hier die Besonderheit (siehe unten), dass eine Funktion mehr als einen Wert zurückgeben kann. Die Rückgabe erfolgt mit dem Schlüsselwort `return`.

- Eine Funktionsdefinition wird mit dem Schlüsselwort `def` eingeleitet,
- gefolgt von einem *Namen*,
- einer *Argumentliste* (0–n Parameter)
- und einem Doppelpunkt :

Syntax Funktionsdefinition

```
def <NAME>(<ARG1>, <ARG2> ...):
     <ANWEISUNGEN>
```

Beispiel: Umfang

```python
def umfang(r):
    ergebnis = 2 * r * math.pi
    return ergebnis
```

Das Beispiel zeigt, wie die Berechnung des Umfangs in eine Funktion ausgelagert werden kann. Diese Funktion kann man dann überall verwenden, wo eine solche Berechnung benötigt wird:

```python
r = 2.5
u = umfang(r)

# Später im Programm
r = 3.7
u = umfang(r)
```

3.2 Rückgabewerte [53]

Das Schlüsselwort `return` dient dazu, eine Rückgabe aus der Funktion zu machen. Der Wert, den die Funktion zurückgeben möchte, wird nach dem `return` angegeben. So gibt z. B. `return 5` den Wert 5 zurück oder `return n` den Wert, der aktuell in der Variable `n` steht.

- Mit `return` wird angegeben, was die Funktion zurückgeben soll
- Ohne `return` hat die Funktion den Rückgabewert `None`
- `return` beendet die Funktion

```
def wurzel(z):
    for n in range(0, 100):
        if n**2 == z:
            return n

    return -1
```

Eine Funktion kann beliebig viele (auch kein) `return` enthalten. Wenn das `return`-Statement ausgeführt wird, wird die Funktion sofort beendet und der Wert hinter dem `return` wird als Rückgabewert der Funktion an den Aufrufer geliefert.

```
def abs(x):
    if x >= 0:
        return x
    else:
        return -x

print(abs(-23)) # 23
print(abs(42)) # 42
```

Eine Verwendung mehrerer `return` in einer Funktion gilt bei manchen Programmierern als schlechter Stil. Sie folgen der „single-entry-single-exit“-Philosophie: Funktionen haben einen Einstiegspunkt (am Anfang) und einen Ausstiegspunkt (am Ende). Eine so gestaltete Funktion gilt bei ihnen als besser verständlich. Die `abs`-Funktion sähe nach dieser Schule wie folgt aus:

```
def abs(x):
    if x >= 0:
        result = x
    else:
        result = -x
    return result
```

```
print(abs(-23)) # 23
print(abs(42)) # 42
```

Funktionen müssen kein `return` enthalten. In diesem Fall geben sie auch keinen (echten) Wert zurück, sondern liefern `None`. `None` zeigt in Python das Fehlen eines Wertes an, so wie `null` in Java oder `nil` in Ruby.

```
def empty():
    pass

a = empty()
print(a) # -> None
```

Es gibt bei Funktionen eine Asymmetrie: Wieso kann eine Funktion beliebig viele Parameter (Argumente) bekommen, aber nur einen Rückgabewert haben. Das ist doch irgendwie unfair…

Glücklicherweise kann man in Python über einen kleinen Umweg Funktionen auch beliebig viele Werte zurückgeben lassen. Dazu verwendet man das automatische Auspacken von Werten aus Listen (zu Listen, siehe unten).

- Über das Packing/Unpacking (siehe Listen) kann eine Funktion auch mehrere Rückgabewerte zurückgeben

```
def multi_return():
    return (42, 23)

a, b = multi_return()
print(a) # -> 42
print(b) # -> 23
```

Die Funktion `multi_return` gibt zwar auch in diesem Beispiel nur einen Wert zurück, dieser ist aber vom Typ einer Liste, die mehrere Werte enthalten kann. Der Aufrufer kann dann die Elemente der Liste in einer *Mehrfachzuweisung* an die Variablen `a` und `b` zuweisen. Damit fühlt es sich für den Verwender der Funktion so an, als ob sie mehr als einen Wert zurückgeben könnte.

Der Verwender einer Funktion muss wissen, wie viele Werte sie zurückgibt, da es andernfalls zu einem Fehler kommt. Die Funktion sollte die Anzahl der Werte als in einer Weise dokumentieren, die für die Verwender unmissverständlich ist.

Fehler

```
def multi_return():
    return (42, 23)
```

```
a, b, c = multi_return()
```

Fehlermeldung

```
ValueError: not enough values to unpack (expected 3, got 2)
```

3.3 Funktionen aufrufen [55]

In den Beispielen sieht man bereits, wie eine Funktion aufgerufen wird: Man verwendet den Namen der Funktion und gibt die Argumente in Klammern, durch Komma getrennt an.

- Eine Funktion wird über ihren Namen aufgerufen und den Parametern (entsprechend der Reihenfolge)

```
a = wurzel(2)
print(a) # -> -1

b = wurzel(4)
print(b) # -> 2
```

Dieses Beispiel ist relativ klar und sollte niemanden überraschen, der schon einmal mit mathematischen Funktionen zu tun hatte. Bei den Argumenten (Parametern) einer Funktion muss man allerdings etwas spitzfindig sein, und noch zwei Begriffe einführen, die für das Verständnis wichtig sind.

- *Formalparameter* Parameter bei der Definition der Funktion (*Variablen*)
    - sind innerhalb der Funktion Variablen
    - werden ein Mal definiert
- *Aktualparameter* aktueller Wert beim Aufruf der Funkion (*Wert*)
    - Anzahl entspricht Formalparametern
    - Belegen die Variablen mit Werten
    - können von Aufruf zu Aufruf der Funktion variieren

Was bedeuten die beiden Begriffe Formal- und Aktualparameter genau? Nehmen wir eine Funktion als Beispiel:

```
def f(a, b, c):
    print(a, b, c)
```

Bei der Funktion `f` sehen wir drei Parameter: `a`, `b`, `c`. Diese sind die *Formalparameter* der Funktion. Wir kennen den Wert dieser Parameter nicht, da sie ihren Wert erst beim Aufruf der Funktion bekommen. Wir wissen nur, dass es beim Aufruf der Funktion drei Werte geben wird, die wir in den drei Parametern finden können. Die Parameter sind für uns also einfach nur Variablen, mit denen wir, wie mit jeder anderen Variable auch, hantieren können; die Werte interessieren uns nicht. Die Formalparameter legen also fest, wie viele Werte einer Funktion in der Parameterliste übergeben werden können.

Wenn wir die Funktion `f` aufrufen, z. B. mit `f(1, 2, 3)`, dann legen wir die Werte fest, die innerhalb der Funktion in den Variablen `a`, `b`, `c` zu finden sind, nämlich `1`, `2` und `3`. Wir könnten die Funktion auch mit `f("x", "y", "z")` aufrufen und somit die Variablen `a`, `b`, `c` mit `"x"`, `"y"`, `"z"` belegen. Die konkreten Werte beim Aufruf der Funktion bezeichnen wir als *Aktualparameter*, weil sie die *aktuelle* Belegung der Variablen beim Funktionsaufruf bestimmen.

Eine Funktion hat nur *einen* Satz von Formalparametern, die bei der Definition der Funktion bestimmt werden. Sie kann aber *beliebig viele* Aktualparameter haben – nämlich jedes Mal, wenn sie aufgerufen wird.

Was passiert in folgendem Beispiel?

```
def f(a, b, c):
    print(a, b, c)

a = 1
b = 2
c = 3
f(a, b, c)
```

Hier hilft uns die strenge Unterscheidung zwischen Formal- und Aktualparametern weiter: Die Funktion `f` hat die drei Formalparameter `a`, `b`, `c`. Beim Aufruf tauchen auch wieder `a`, `b` und `c` auf. Dies sind aber die Aktualparameter und dass die Namen identisch sind ist Zufall – genaugenommen ist es kein Zufall, sondern hier aus didaktischen Gründen extra so gemacht.

Es passiert folgendes: Die *Werte* der Aktualparameter werden in die Variablen *kopiert*, die als Formalparameter angegeben sind. Es wird also geschaut, welchen Wert hat das `a` im Aktualparameter und dieser Wert (1) wird in den Parameter `a` der Funktion geschrieben. Dasselbe dann mit `b` und `c`.

Das Programm ist damit identisch mit einem Programm, das wie folgt aussieht:

```
def f(a, b, c):
    print(a, b, c)
```

```
x = 1
y = 2
z = 3
f(x, y, z)
```

Da die Werte der Aktualparameter in die Variablen der Formalparameter kopiert werden, ist es egal, wie die Aktualparameter heißen.

Übergabe von Daten an Funktionen erfolgt per *Wertübergabe* *(pass by value)* (im Gegensatz zu *Referenzübergabe* *(pass by reference)*)

- Wert des Aktualparameters wird kopiert
- Methode arbeitet auf einer Kopie
- Wert beim Aufrufer bleibt unverändert

Dies gilt auch für Referenzvariablen

Die *Wertübergabe* hat zur Konsequenz, dass eine Funktion den Wert der Variablen, die beim Aufrufer als Aktualparameter benutzt wurden, nicht verändern kann.

```
def change(a):
    a = a + 1

a = 3
change(a)
print(a) # -> 3
```

3.4 Vararg-Funktion [58]

Funktionen können beliebig viele Parameter bekommen aber wir müssen bei der Definition der Funktion angeben, welche Formalparameter sie hat. Wir müssen also wissen, wie viele Parameter diese eine konkrete Funktion bekommen soll.

Wieso ist es dann möglich, die `print`-Funktion mal mit einem `print("Hello")` und mal mit mehr Parametern aufzurufen `print(1, 2, 3)`? Die Antwort ist, dass man Funktionen schreiben kann, bei denen man die Anzahl der Formalparameter nicht kennt.

- Wenn die Funktion beliebig viele Parameter annehmen soll, kann man einen `*`-Parameter definieren
- Die Werte werden dann als Liste (siehe unten) übergeben

```
def max(*a):
    result = a[0]
    for i in a:
        if result < i:
            result = i
    return result

max(3) # -> 3
max(4, 3) # -> 4
max(4, 3, 5) # -> 5
```

Eine Funktion, die man mit beliebig vielen Argumenten aufrufen kann, bezeichnet man als *Vararg-Funktion*, für variable arguments. Will man ein solches Verhalten, muss man einen Formalparameter am Ende der Parameterliste mit einem * kennzeichnen. Alle verbliebenen Aktualparameter werden dann in eine Liste verpackt und diesem Parameter zugewiesen.

```
def f(a, *b):
    print(a, b)

f(1)
f(1, 2)
f(1, 2, 3)
f(1, 2, 3, 4)
```

Ausgabe

```
1 ()
1 (2,)
1 (2, 3)
1 (2, 3, 4)
```

Man sieht, dass alle Aktualparameter (außer dem ersten) als Liste im Parameter `b` zu finden sind.

Die einzelnen Elemente der Liste kann man mit der entsprechenden Syntax (siehe unten) aus der Liste extrahieren.

3.5 Named Arguments [59]

Bei Skriptsprachen findet keine Überprüfung der Typen von Aktualparametern beim Aufruf einer Funktion statt. Wenn die Typen nicht zu dem passen, was im Inneren der Funktion passiert, dann kommt es erst zur Laufzeit zu einem Fehler. Der Aufrufer muss also wissen, welche Typen eine Funktion bei den jeweiligen Parametern verarbeiten kann.

```
def printer(text, wert):
    print(text.format(wert, wert**2))

printer("{}^2 = {}", 8) # -> 8^2 = 64
printer("8, {}^2 = {}") # Fehler
# TypeError: printer() missing 1 required positional argument: 'wert'
```

Ein weiteres Problem ist, dass Funktionen mit langen Parameterlisten schnell unübersichtlich werden. Leicht vertauscht man einen Parameter oder vergisst ihn beim Aufruf.

Die genannten Probleme lassen sich vermeiden, wenn man die Parameter nicht über ihre Position, sondern über ihren Namen adressiert.

- Es ist möglich anstatt der Reihenfolge auch die Namen der Parameter im Aufruf zu verwenden: *Benannte Argumente* (*named arguments*)

```
def potenz(basis, exponent):
    return basis ** exponent

# Positionale Parameter
a = potenz(2, 4) # -> 16

# Benannte Parameter
b = potenz(exponent = 4, basis = 2) # -> 16
```

Das Beispiel zeigt, dass man eine Python-Funktion sowohl aufrufen kann, indem man die Aktualparameter der Reihe nach angibt als auch die Parameter explizit über ihren Namen anspricht. Verwendet man den Namen, spielt die Reihenfolge keine Rolle mehr.

3.6 Default Argumente [60]

Wenn ein Formalparameter definiert ist, dann muss er beim Aufrufen mit einem Wert belegt werden. Das ist aber nicht immer komfortable, weil es Funktionen gibt, bei denen man Standardwerte hat, die man nur ab und zu übersteuern möchte.

Ein solches Beispiel wäre eine Logarithmus-Funktion, die normalerweise den Logarithmus zur Basis e bestimmen soll, es sei denn, der Verwender gibt eine andere Basis an. Dieses Problem könnte man lösen, indem man zwei Funktionen anbietet:

```
def log_e(x):
    # ...
```

```
def log(basis, x):
    # ...
```

Dieser Ansatz ist aber nicht sehr elegant.

Python bietet als Lösung an, dass man Parameter mit Standardwerten belegen kann. Wird ein Parameter nicht belegt, hat er den Standardwert.

- Parameter von Methoden können mit *Standardwerten* (*default values*) belegt werden
- Gibt der Aufrufer keinen Wert an, wird der Standardwert verwendet
- Über benannte Argumente kann jeder Parameter ausgewählt werden

```
def potenz(basis = 2, exponent = 1):
    return basis ** exponent

# Positionale Parameter
a = potenz() # -> 2
b = potenz(4) # -> 4
c = potenz(2, 4) # -> 16

# Benannte Parameter
d = potenz(exponent = 8) # -> 256
```

Das Beispiel zeigt, wie man die Funktion `potenz` mit keinem, einem oder zwei Parametern aufrufen kann. Wird der Parameter nicht belegt, dann wird einfach der Standardwert (2 für `basis` bzw. 1 für `exponent`) verwendet.

Die Standardwerte funktionieren auch, wenn man benannte Argumente verwendet, wie die letzte Zeile im Beispiel zeigt.

3.7 Funktionen sind Objekte [61]

Ein komplexeres Feature von Python ist, dass Funktionen selbst wieder Objekte sind; der Objektbegriff wird später noch erläutert. Man kann Funktionen also in Variablen speichern und an andere Funktionen übergeben.

```
def p(x):
    print(x)

f = p
print(f) # -> <function p at 0x7f0842a78dc0>
p("Hallo") # -> Hallo
```

Das Beispiel definiert eine Funktion `p` und weist diese dann der Variable `f` zu. Die Funktion, auf welche die Variable `f` zeigt, kann man mit dem `()`-Operator aufrufen. `()` wird auch als *Aufrufoperator* bezeichnet.

- Alles in Python ist ein Objekt, auch Funktionen
- Man kann Funktionen an Funktionen übergeben (später mehr dazu)

```
def twice(f, x):
    return f(f(x))

def quadriere(x):
    return x*x

print(twice(quadriere, 2))
```

Ausgabe

```
16
```

Im Beispiel ruft die Funktion `twice` einfach die ihr übergebene Funktion `f` zweimal mit dem Parameter `x` auf. Hierbei wird das Ergebnis des ersten Aufrufs an den zweiten übergeben (`f(f(x))`). `twice` ist es hierbei egal, was die übergebene Funktion macht.

Man kann sie mit jeder anderen Funktion aufrufen, die einen Wert nimmt:

```
def hello(name):
    return "Hallo {}".format(name)

print(twice(hello, "Thomas"))
```

Ausgabe

```
Hallo Hallo Thomas
```

3.8 Gültigkeitsbereich [62]

Wenn wir in einem Programm (außerhalb von Funktionen) mit Variablen arbeiten, dann ist die Variable von ihrer ersten Zuweisung (*Definition*) bis zum Ende des Programms sichtbar und kann benutzt werden. Der *Gültigkeitsbereich* der Variable recht also von der Definition bis zum Ende des Programms.

Wenn jetzt Funktionen ins Spiel kommen, verändert sich die Situation. Es wäre sehr unpraktisch, wenn Variablen innerhalb von Funktionen genauso behandelt würden, wie die außerhalb. Sehr

schnell hätten wir das Problem, dass uns keine neuen Namen mehr einfallen und der Speicher voller unnützer Variablen wäre. Deswegen werden Funktionen besonders behandelt.

- Der *Gültigkeitsbereich* (*scope*) einer Variable gibt an, wo sie sichtbar und verwendbar ist
- Variablen in Funktionen sind nur in diesen sichtbar und gültig
- Können denselben Namen wie Variablen außerhalb der Funktion haben

```
x = 1

def add_one(x):
    x = x + 1 # lokales x
    return x

y = add_one(x)
# x = 1, y = 2
```

In diesem Beispiel sehen wir eine Variable `x`, die außerhalb der Funktion definiert wurde. Die Funktion selbst hat einen Parameter, der ebenfalls `x` heißt und der innerhalb der Funktion wie eine Variable benutzt werden kann. Die Namensgleichheit ist hier egal, es handelt sich um zwei vollkommen unterschiedliche Variablen. Schreiben wir das Programm um, um das deutlicher zu machen:

```
g = 1

def add_one(l):
    l = l + 1
    return l

y = add_one(g)
# g = 1, y = 2
```

Die Variable `g` ist global und ihr Gültigkeitsbereich erstreckt sich von der Zeile `g = 1` bis zum Ende des Programms, es handelt sich um eine *globale Variable*. Die Variable `l` ist nur innerhalb der Funktion `add_one` gültig. Nachdem die Funktion beendet wurde, wird die Variable verworfen. Dasselbe gilt für alle weiteren Variablen, die in einer Funktion definiert werden. Schreiben wir `add_one` um zu

```
def add_one(l):
    k = l + 1
    return k
```

dann hat die Funktion jetzt zwei *lokale Variablen*, nämlich `l` und `k`, die nur in der Funktion gelten. Es gilt als schlechter Programmierstil, die Parameter einer Funktion innerhalb der Funktion zu verändern, sodass das obige Beispiel von vielen Entwicklerinnen als das bessere angesehen würde.

Jeder Versuch, auf die lokalen Variablen einer Funktion außerhalb dieser zuzugreifen, führt zu einer Fehlermeldung:

```
def f():
  l = 3

f()
print(l) # Error
# NameError: name 'l' is not defined
```

3.9 Schlüsselwort global [63]

Globale Variablen sind überall im Programm sichtbar, also auch in den Funktionen.

Beispiel: Zugriff auf globale Variable aus Funktion

```
g = "global"

def f():
  print(g)

f() #-> global
```

Python verbietet aber standardmäßig, dass eine Funktion den Wert einer globalen Variable ändert, weil dies eine häufige Quelle von schwer zu findenden Fehlern ist. Ein Schreibzugriff erzeugt einfach eine neue, lokale Variable mit demselben Namen, wie das folgende Beispiel zeigt.

Beispiel: Versuch, eine globale Variable zu schreiben

```
g = "global"

def f():
  g = "lokal"

f()
print(g) #-> global
```

- Globale Variablen sind in Funktionen sichtbar, können aber nicht verändert werden → Zuweisung legt eine neue lokale Variable an
- Mit dem Schlüsselwort `global` können globale Variablen in Funktionen zugewiesen werden (*Vorsicht!*)

```
x = 0

def incr_x():
    x = x + 1 # Fehler

def incr_x2():
    global x
    x = x + 1 # funktioniert
```

Die Funktion `incr_x()` kann man nicht aufrufen, weil Python einen Fehler meldet („UnboundLocalError: local variable ’x’ referenced before assignment“). Das Problem ist, dass die Zuweisung `x =` eine neue lokale Variable `x` erzeugt, die dann aber rechts der Zuweisung auftaucht, bevor sie einen Wert hatte.

Der Ausweg aus dem Problem ist, die globale Variable mit dem Schlüsselwort `global` in der Funktion zu kennzeichnen. Damit wird sie schreibbar und eine Zuweisung legt keinen neue, lokale Variable mit demselben Namen an, sondern verwendet wirklich die globale Variable.

Beispiel: Zugriff mit global

```
g = "global"

def f():
  global g
  g = "lokal"

f()
print(g) #-> lokal
```

3.10 Funktionen in Funktionen [64]

Variablen können sowohl global, als auch lokal sein. Lokale Variablen sind nur in der jeweiligen Funktion sichtbar, in der sie definiert wurden.

Analog kann man auch Funktionen in Funktionen definieren. Diese sind dann ebenfalls nur innerhalb der jeweiligen Funktion sichtbar und können auch nur dort aufgerufen werden.

- Funktionen können Funktionen enthalten (analog zu Variablen)

```
def function1(x):
    def function2(y):
        print(y + 2)
        return y + 2
    return 3 * function2(x)
```

```
a = function1(2) # 4
print(a) # 12

b = function2(2.5) # error: undefined name
```

Die Funktion `function2` ist innerhalb von `function1` definiert und nur dort sichtbar. Der Versuch, sie von außerhalb mit `b = function2(2.5)` aufzurufen führt deswegen zu einem Fehler. Innerhalb von `function1` kann man sie aber problemlos nutzen.

3.11 Lambda [65]

Die Syntax, um Funktionen in Python zu definieren ist relativ kompakt. Trotzdem braucht man selbst für eine einfache Additionsfunktion zwei Zeilen und die Schlüsselwörter `def` und `return`. Für kurze, einfache Funktionen, gibt es deswegen die Möglichkeit, ein sogenanntes *Lambda* zu verwenden.

- Für kurze Funktionen (mit nur einem Ausdruck) steht mit dem *Lambda* eine verkürzte Schreibweise zur Verfügung

```
# Klassische Funktionsdefinition
def add(a, b):
    return a + b

## Definition über Lambda
add2 = lambda a, b: a + b

print(add(1, 2)) # -> 3
print(add2(2, 3)) # -> 5
```

Die Syntax für ein Lambda ist:

```
lambda <ARGUMENTLISTE>: <AUSDRUCK>
```

Es ist kein `return` notwendig, weil das Ergebnis des Ausdrucks <AUSDRUCK> als Rückgabewert aus dem Lambda geliefert wird.

Präzise muss man sagen, dass `lambda` ein namenloses Funktionsobjekt erzeugt, das man dann über eine Zuweisung an eine Variable zuweist. Der Ausdruck `add2 = lambda a, b: a + b` besteht somit aus zwei Teilen

- erzeugen eines Funktionsobjektes mit `lambda a, b: a + b`

- zuweisen des Objektes an eine Variable `add2 = ...`

Die vielfältigen Anwendungen von Lambdas sprengen den Rahmen dieser Vorlesung. Sie sind aber ein wichtiges Element für die funktionale Programmierung in Python, die wir bei den Listen noch einmal sehen werden. Dort findet sich das folgende Beispiel:

```
filme = [ 'Pulp Fiction', 'Kill Bill', 'Reservoir Dogs' ]
filme_gross = list(map(lambda f: f.upper(), filme))

print(filme_gross)
# -> ['PULP FICTION', 'KILL BILL', 'RESERVOIR DOGS']
```

Hier sieht man, wie ein Lambda benutzt wird, um eine Funktion auf die Elemente einer Liste anzuwenden.

3.12 Docstring [66]

Wir haben schon an einigen Stellen gesehen, dass die Verwenderin einer Funktion wissen muss, was die Funktion genau macht, welche Bedeutung die Parameter haben und welche(n) Rückgabewert(e) die Funktion haben wird. Da es viel zu mühselig wäre, dies immer aus dem Quelltext der Funktion zu schließen, werden Funktionen in Python mit einer entsprechenden Dokumentation versehen, dem *Docstring*.

Funktionen sollten dokumentiert werden, damit

- andere sie korrekt benutzen können
- Sie sich nach drei Monaten noch erinnern können, was sie machen

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

Der Docstring wird durch drei Anführungszeichen (`"""`) eingeleitet und durch dieselbe Markierung wieder beendet. Zwischen den Markierungen beschreibt man, das die Funktion macht, welche Parameter mit welcher Bedeutung sie hat und was sie zurückgeben wird.

- Die Informationen aus dem Docstring können über `help(FUNKTIONSNAME)` ausgegeben werden

```
>>> help(complex)

Help on function complex in module __main__:

complex(real=0.0, imag=0.0)
    Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
```

Ein Beispiel aus der Python-Bibliothek für die `print`-Funktion:

```
>>> help(print)

Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Kapitel 4

Listen und Tuple

4.1 Listen [69]

Programme bearbeiten häufig große Mengen Daten identischer Struktur. Am Anfang wurde schon das Beispiel gegeben, die Durchschnittsnote aller Teilnehmer der Veranstaltung „Python-Programmierung“ zu bestimmen. Hierzu muss man eine Reihe von Noten durchlaufen und das arithmetische Mittel bestimmen. Die einzelnen Datenpunkte sind alle vom selben Typ und es wäre sehr mühselig, wenn man jede Note in einer einzelnen Variable speichern würde:

```
n1 = 1.3
n2 = 2.3
n3 = 1.7
n4 = 2.7
n5 = 4.0
n6 = 1.3
pypnote = (n1 + n2 + n3 + n4 + n5 + n6) / 6
```

Käme eine weitere Teilnehmerin dazu, müsste man eine weitere Variable (`n7`) einführen und die Formel für den Durchschnitt an zwei Stellen ändern. Das kann nicht sinnvoll sein.

Deswegen gibt es in jeder Programmiersprache eine Möglichkeit gleichartige Daten zusammenzufassen und dann „auf einen Rutsch“ zu verarbeiten. In Java oder C/C++ heißen diese Datenstrukturen *Arrays* in Python spricht man von *Listen*.

- Oft möchte man größere Mengen von Daten gleichartig bearbeiten
- *Listen* (*lists*) speichern mehrere Werte geordnet
- Daten können von unterschiedlichen Typen sein
- Zugriff erfolgt über `[]` mit Index

```
my_list = [ 2, 3, 5, 7, 11, 13, 'Hugo' ]
```

```
my_list[0] # -> 2
my_list[1] # -> 3
```

Eine Liste kann beliebig viele Elemente enthalten und die Elemente können von unterschiedlichen Typen sein. In der Praxis wird man aber normalerweise darauf achten, dass nur Elemente eines Typs in der Liste vorkommen, weil sonst die Verarbeitung zu kompliziert wird.

Bei einer Liste hat man dann eine Variable (`my_list` im Beispiel) über die man auf alle Elemente zugreifen kann. Das einzelne Element wählt man über seinen Index aus, den man in eckigen Klammern (`[]`) nach dem Namen der Variable angibt, z. B. `my_list[3]` für das vierte Element.

Wieso liefert `my_list[3]` das 4. und nicht das 3. Element?

- Index beginnt bei `0`
- Negative Indices zählen von hinten

```
my_list = [ 1, 2, 3, 4, 5 ]

my_list[0] # -> 1
my_list[1] # -> 2

my_list[-1] # -> 5
my_list[-2] # -> 4

my_list[5] # IndexError: list index out of range
```

In fast allen Programmiersprachen (mit Ausnahme von Lua) wird bei einem Array bzw. einer Liste immer bei 0 mit dem Zählen begonnen. D. h. das erste Element hat den Index 0, das zweite den Index 1 usw.

Verwendet man einen Index, der außerhalb der möglichen Indices für die vorhandenen Elemente liegt, dann bricht das Programm mit einem Fehler (*IndexError*) ab.

Python bietet die Möglichkeit, die Elemente auch von hinten zu adressieren, indem man negative Indices verwendet. Hierbei ist `-1` das letzte Element, `-2` das vorletzte usw. Haben wir also ein Array `a` mit drei Elementen, dann gilt `a[0] = a[-3]`, `a[1] = a[-2]` und `a[2] = a[-1]` oder ganz allgemein für ein Array mit n Elementen: `a[i] = a[-(n - i)]`.

Eine Liste kann jederzeit verändert werden: Man kann Elemente hinzufügen oder vorhandene Elemente ersetzen.

- Listen sind veränderlich
- Zuweisung über den Index

```
# Element ersetzen
my_list = [0, 23, 42]

my_list[0] = 7
print(my_list) #-> [7, 23, 42]

# Element anfügen
my_list.append(128)
print(my_list) #-> [7, 23, 42, 128]

# Elemente einfügen bei Index 1
my_list[1:1] = [ -1, -2]
print(my_list) #-> [7, -1, -2, 23, 42, 128]
```

- `my_list[0] = 7`: Ein einzelnes Element ersetzt man, indem man es mit dem Index auswählt und dann einen neuen Wert zuweist.
- `my_list.append(128)`: An das Ende der Liste kann man mit `.append(WERR)` einen neuen Wert anfügen. Die Schreibweise mit dem Punkt `.` ist neu und vielleicht etwas verwirrend. Sie bedeutet, dass man „auf der Liste eine Methode aufruft". Die Details werden später bei den Klassen noch etwas genauer erläutert, hier reicht es erst einmal die Syntax zu akzeptieren, wie sie ist.
- `my_list[1:1] = [ -1, -2]`: Man kann auch mehrere Elemente in der Liste einfügen/ersetzen. Hierzu gibt man in den eckigen Klammern durch Doppelpunkt (`:`) getrennt das erste und letzte zu ersetzende Element an, wobei die zweite Zahl exklusive ist. Hier im Beispiel wird mit `[1:1]` kein Element ersetzt, sondern die Elemente werden vor dem Element mit dem Index 1 eingefügt. Will man das Element am Index 1 ersetzen, muss man `[1:2]` schreiben.

Man kann auch mehrere Elemente auf einmal ersetzen:

```
# Element am Index 1 ersetzen
my_list = [0, 23, 42]
my_list[1:2] = [ -1, -2]
print(my_list) #-> [0, -1, -2, 42]

# Elemente am Index 1 und 2 ersetzen
my_list = [0, 23, 42]
my_list[1:3] = [ -1, -2]
print(my_list) #-> [0, -1, -2]
```

Wenn man in der Zuweisung eine Variable verwendet, dann wird in die Liste übernommen

- bei einfachen Datentypen der Wert der Variablen
- bei Referenzvariablen die Referenz

```
x = -2
my_list = [0, 23, 42]
my_list[1:2] = [x, -3]
print(my_list) #-> [0, -2, -3, 42]

# Variable x wird verändert, wirkt sich aber nicht auf die Liste aus
x = 4
print(my_list) #-> [0, -2, -3, 42]
```

Das Beispiel zeigt, dass man die Variable `x` nach dem Eintragen in die Liste ändern kann, ohne dass sich dies auf die Liste auswirkt.

Dies gilt aber nicht bei Referenzvariablen, da bei diesen kein Wert, sondern eine Referenz auf die Daten vorliegt. Eine Liste ist z. B. etwas, das über eine Referenzvariable verwaltet wird, wie das folgende Beispiel zeigt:

Referenztyp in Liste

```
x = [ -1, -2, -3 ]
my_list = [0, 23, 42]
my_list[1:2] = [x, -3]
print(my_list) #-> [0, [-1, -2, -3], -3, 42]

x.append("Ooops")
print(my_list) #-> 0, [-1, -2, -3, 'Ooops'], -3, 42]
```

4.2 Slicing und Verketten [73]

Manchmal möchte man von einer Liste einen Ausschnitt erhalten, eine Liste abschneiden oder zwei Listen miteinander verketten. Hierzu bietet Python das *Slicing* und den +-Operator an.

- Aus Listen können Sublisten erstellt werden (*Slicing*): `[von:bis]` (exklusive `bis`)
- Listen können mit + verkettet werden

```
my_list = [ 1, 2, 3, 4, 5 ]
my_list[0:2] # -> [1, 2]
my_list[2:3] # -> [3]
my_list[2:4] # -> [3, 4]
my_list[1:] # -> [2, 3, 4, 5]
my_list[:3] # -> [ 1, 2, 3]

new_list = [ 'a', 'b', 'c' ]
```

```
verkettet = my_list + new_list
# [ 1, 2, 3, 4, 5, 'a', 'b', 'c' ]
```

Das Beispiel zeigt, wie durch Slicing aus der Liste `my_list` neue Listen (*Slices*) erstellt werden. Es handelt sich hierbei um echte Kopien der Liste, d. h. ein Slice ist nicht nur eine Sicht auf die vorhandene Liste, sondern eine neue Liste. Dies sieht man an folgendem Beispiel:

Slices sind Kopien

```
my_list = [ 1, 2, 3, 4, 5 ]
slice = my_list[0:2] # -> [1, 2]

my_list[0] = 99
print(my_list) # -> [ 99, 2, 3, 4, 5 ]
print(slice) # -> [1, 2]
```

Dasselbe gilt auch für die Verkettung von Listen mit +:

Verkettungen sind Kopien

```
a = [ 1, 2, 3 ]
b = [ 4, 5, 6 ]
new_list = a + b # -> [ 1, 2, 3, 4, 5, 6 ]

a[0] = 99
print(a) # -> [ 99, 2, 3 ]
print(new_list) # -> [ 1, 2, 3, 4, 5, 6 ]
```

Man kann bei der Angabe des Slices noch eine Schrittweite angeben, sodass die vollständige Syntax zur Erzeugung eines Slices `[von:bis:schrittweite]` ist.

Slice mit Schrittweite

```
my_list = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
slice = my_list[0:8:3] # -> [1, 2]
print(slice) # -> [1, 4, 7]
```

Hier werden die Elemente vom Index 0 bis zum Index 7 (die 8 ist exklusive) adressiert, von diesen Elementen aber nur jeder dritte Index, also 0, 3, 6 und damit die Elemente `[1, 4, 7]`. Man kann sich die Funktionsweise des durch folgende Funktion verdeutlichen.

Slice als Funktion

```
def slice(list, start, end, step = 1):
    i = start
    result = []
```

```
    while i < end:
        result.append(list[i])
        i = i + step
    return result

my_list = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
s = slice(my_list, 0, 8, 3)
print(s)
```

Im Vergleich zu diesem Beispiel ist die eingebaute Syntax von Python schon deutlich kompakter.

4.3 Multiplikation [74]

Nicht nur der +-Operator ist für Listen erlaubt, sondern auch der *-Operator. Man kann Listen also multiplizieren.

Möglich ist nur eine Multiplikation einer Liste mit einer Ganzzahl n. Die Liste wird dann n-Mal mit sich selbst verkettet.

- Listen können multipliziert werden

Beispiel: Listen multiplizieren

```
my_list = [ 'a', 'b' ]

my_list * 2
# [ 'a', 'b', 'a', 'b' ]

2 * my_list
# [ 'a', 'b', 'a', 'b' ]
```

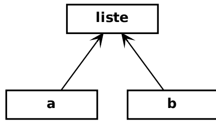
4.4 Kopieren einer Liste [75]

Listen sind Datentypen, die über Referenzvariablen verarbeitet werden. Das bedeutet, dass man eine Liste durch die Zuweisung an eine andere Variable nicht kopiert, sondern nur die Referenz auf die Liste.

Kopie der Referenz auf die Liste

```
a = [ 1, 2, 3 ]
b = a

a[0] = -3
print(a) # -> [ -3, 2, 3 ]
print(b) # -> [ -3, 2, 3 ]
```

Es gibt in diesem Beispiel nur eine Liste mit den Elementen `[ 1, 2, 3 ]`. Die Zuweisung `b = a` kopiert nicht die Liste mit ihrem Inhalt, nur die Referenz wird kopiert. Beide Variablen (`a` und `b`) zeigen auf dasselbe Listen-Objekt. Eine Änderung, die ich über die Referenz `a` an der Liste vornehme, ist damit auch über die Variable `b` sichtbar.

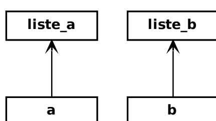


Will ich eine „echte" Kopie der Liste, d. h. eine Kopie nicht nur der Referenz, sondern auch der Elemente, kann ich den Slice-Operator dafür verwenden. Da Slices immer Kopien der Listen sind, führt ein Slice `[0:lÃ¤nge]` zu einer vollständigen Kopie der Liste. Jedes Mal vorher die Länge bestimmen zu müssen ist aber recht unhandlich, sodass mit `[:]` eine verkürzte Form zur Verfügung steht, die alle Elemente kopiert.

Kopie der Liste

```
a = [ 1, 2, 3 ]
b = a[:] # oder b = list(a)

a[0] = -3
print(a) # -> [ -3, 2, 3 ]
print(b) # -> [ 1, 2, 3 ]
```

In diesem Beispiel wird die Liste kopiert, sodass `a` und `b` auf zwei unterschiedliche Listen-Objekte verweisen. Eine Änderung über `a` betrifft nur dessen Kopie der Liste, `b` bleibt davon unbeeindruckt.



4.5 Packing und Unpacking [77]

Im Kapitel zu den Funktionen haben wir bereits gesehen, dass Funktionen über den Umweg der Liste mehrere Werte auf einmal zurückgeben können. Hierbei handelt es sich um einen ganz allgemeinen Mechanismus, den man als *packing* und *unpacking* bezeichnet.

Beim *unpacking* weist man eine Liste mehreren Variablen zu.

- *unpacking*: Listen werden automatisch entpackt, wenn auf der linken Seite der Zuweisung mehrere Variablen stehen
- Die Anzahl der Variablen muss mit den Listenelementen übereinstimmen

Unpacking einer Liste

```
my_list = [ 1, 2, 3, 4 ]
a, b, c, d = my_list # a=1, b=2, c=3, d=4

x, y, z = my_list
# ValueError: too many values to unpack (expected 3)

a, b, c, d, e = my_list # a=1, b=2, c=3, d=4
# ValueError: not enough values to unpack (expected 5, got 4)
```

Gegensatz zum *unpacking* werden beim *packing* beliebig viele Variablen zu einem Tuple umgewandelt, wenn links von einer Zuweisung mehr als eine Variable steht.

Tuple wurden an dieser Stelle noch nicht erläutert und kommen später in der Vorlesung vor; man kann sie sich aber einfach als Listen vorstellen, die nicht verändert werden können.

- *packing*: Variablen werden zu einem Tuple (siehe unten) gepackt, wenn sie auf der rechten Seite der Zuweisung stehen

Packing von Variablen zu einem Tuple

```
my_tuple = a, b, c, d
# (1, 2, 3, 4)
```

Nicht immer will man, dass die Variablen als Tuple verpackt werden, sondern hätte lieber eine Liste. In diesem Fall muss man die Variablen in eckige Klammern setzen. Es handelt sich hierbei nicht mehr um ein echtes packen der Variablen, da wir einfach eine Liste mit ihnen als Elementen erzeugen aber vom Ergebnis her gibt es keinen Unterschied.

Variablen zu einer Liste packen

```
new_list = [ a, b, c, d ]
# [1, 2, 3, 4]
```

4.6 Funktionen für Listen [79]

Es gibt in Python eine Reihe von Funktionen, mit denen man Listen ver- und bearbeiten kann. Weiter oben wurde bereits die `append`-Methode eingeführt, es gibt aber noch weitere Funktionen.

Einige Funktionen bekommen die Liste als Argument (z. B. `len`), andere werden auf der Liste aufgerufen (z. B. `append`). Wenn eine Funktion auf einem Objekt aufgerufen werden kann, dann sprechen wir von einer *Methode*. Somit sind `append`, `count`, `insert` etc. *Methoden* der Liste. Bei Methoden steht vor dem Punkt die Variable, die auf das Objekt verweist, auf dem die Methode aufgerufen werden soll und rechts vom Punkt der Name der Methode. Man kann Methoden mental auf Funktionen abbilden, indem man ganz allgemein `o.methode(parameter)` durch `methode(o, parameter)` ersetzt.

Python geht mit der Mischung aus Funktionen und Methoden einen sehr eigenwilligen Weg, dem andere Programmiersprachen nicht folgen. Hier ist entweder alles über Funktionen abgebildet (C) oder alles über Methoden (Ruby, Java). Es gibt auch keinen tieferen Grund für diese Entscheidung, außer dass der Erfinder der Sprache es so eleganter fand. Wir müssen also einfach damit leben.

- `len(xs)`: Länge der Liste
- `xs.append(x)`: `x` an `xs` anhängen
- `xs.count(x)`: Anzahl der `x` in `xs` zählen
- `xs.insert(i, x)`: `x` an der Stelle `i` in `xs` einfügen
- `xs.sort()`: `xs` sortieren
- `xs.remove(x)`: `x` aus `xs` entfernen
- `xs.pop()`: letztes Element on `xs` entfernen und zurückgeben
- `x in xs`: testen, ob `xs` ein `x` enthält
- `del list[i]`: entfernt Element mit dem Index `i` aus `list`

```
my_list = [ 5, 4, 3, 2 ]

len(my_list) # -> 4

my_list.append(1) # -> [ 5, 4, 3, 2, 1 ]

my_list.count(3) # -> 1

my_list.insert(2, 3.5) # -> [ 5, 4, 3.5, 3, 2, 1 ]

my_list.sort() # -> [1, 2, 3, 3.5, 4, 5]

my_list.remove(3.5) # -> [1, 2, 3, 4, 5]
my_list.pop() # -> [1, 2, 3, 4]
del my_list[1] # -> [1, 3, 4]
```

```
4 in my_list # -> True
```

4.7 Über Listen iterieren [81]

Im Kapitel zu den Kontrollstrukturen wurde bereits die `for`-Schleife vorgestellt. Sie dient in Python dazu, über die Elemente einer Liste zu laufen, was als *iterieren* bezeichnet wird.

- Mit `for` kann man über die Elemente einer Liste laufen

```
filme = [ 'Pulp Fiction', 'Kill Bill', 'Reservoir Dogs' ]

for f in filme:
    print(f)
```

Ausgabe

```
Pulp Fiction
Kill Bill
Reservoir Dogs
```

Die `for`-Schleife liefert uns bei jedem Durchlauf das gerade aktuelle Element der Liste in der sogenannten *Laufvariablen*. Dies ist sehr komfortabel, weil wir uns nicht mit den Indices der Elemente herumschlagen müssen und auch die Frage, wann die Liste zu Ende ist kein Problem darstellt.

In manchen Fällen benötigt man aber neben den Elementen der Liste bei der Iteration auch deren Index. Hier kann man die `enumerate`-Funktion verwenden.

- Mit `enumerate` kann man über Elemente und Index gleichzeitig iterieren

```
filme = [ 'Pulp Fiction', 'Kill Bill', 'Reservoir Dogs' ]

for index, f in enumerate(filme):
    print(index, f)
```

Ausgabe

```
0 Pulp Fiction
1 Kill Bill
2 Reservoir Dogs
```

Die enumerate-Funktion liefert ein Objekt zurück, das man in der for-Schleife anstelle der Liste verwenden kann. Bei jedem Schleifendurchlauf bekommt man dann ein Tuple, bestehend aus Index und Element, wie das folgende Beispiel zeigt:

```
filme = [ 'Pulp Fiction', 'Kill Bill', 'Reservoir Dogs' ]

for t in enumerate(filme):
    print(t)
```

Ausgabe

```
(0, 'Pulp Fiction')
(1, 'Kill Bill')
(2, 'Reservoir Dogs')
```

Sinnvollerweise packt man dieses Tuple aber direkt per unpacking wieder aus, sodass die for-Schleife zwei Variablen vor dem in hat: `for index, f in enumerate(filme):`

4.8 sum [83]

Wenn eine Liste numerische Daten (Ganzzahlen oder Fließkommazahlen) enthält, kann man alle Elemente mit Hilfe der sum-Funktion aufaddieren lassen.

- Die Funktion sum summiert alle Elemente einer Liste

```
l = [ 1, 2, 3, 4 ]
summe = sum(l) # -> 10

# Alternativ zu Fuß
summe = 0
for e in l:
    summe += e
```

Bei nicht-numerischen Daten bekommt man bei sum einen Fehler.

sum bei nicht-numerischen Daten

```
a = ["A", "B"]
s = sum(a)
# TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

4.9 filter [84]

Man kann eine Liste filtern, d. h. Elemente entfernen, indem man über die gesamte Liste läuft und nur die gewünschten Elemente in das Ergebnis übernimmt.

Filtern einer Liste

```
my_list = [ 1, 2, 3, 4, 5, 6, 7, 8 ]
gerade = []

for e in my_list:
    if e % 2 == 0:
        gerade.append(e)

print(gerade) # -> [2, 4, 6, 8 ]
```

Da es sich hierbei aber um einen sehr häufigen Vorgang handelt, hat Python eine Methode `filter`, die diese Aufgabe elegant übernimmt.

- Mit der Funktion `filter` kann man die Elemente einer Liste filtern
- Alle Elemente, für die die Lambda-Funktion `True` zurückgibt, werden übernommen

```
my_list = [ 1, 2, 3, 4, 5, 6, 7, 8 ]
gerade = list(filter(lambda e: e % 2 == 0, my_list))

print(gerade)
# -> [2, 4, 6, 8]
```

Entscheidend in dem Beispiel ist der Rückgabewert der Lambda-Funktion `lambda e: e % 2 == 0`: Gibt sie `True` zurück, weil der Wert ohne Rest durch zwei Teilbar, also gerade ist, wird das Element in die neue Liste übernommen. Andernfalls wird das Element verworfen.

4.10 map [85]

- Die Funktion `map` erlaubt es, die Elemente einer Liste zu bearbeiten

```
filme = [ 'Pulp Fiction', 'Kill Bill', 'Reservoir Dogs' ]
filme_gross = list(map(lambda f: f.upper(), filme))

print(filme_gross)
# -> ['PULP FICTION', 'KILL BILL', 'RESERVOIR DOGS']
```

In dem Beispiel taucht eine neue Methode auf: `list`. Sie nimmt ein Objekt, das iterierbar ist, d. h. in einer `for`-Schleife nach dem `in` verwendet werden kann, läuft über alle Elemente und gibt sie als

Liste zurück. Beispiele für solche Objekte sind ein `range`-, `map`-, `filter`- oder `enumerate`-Objekt, die von den gleichnamigen Methoden erzeugt werden. Die Implementierung von `list` kann man sich also wie folgt vorstellen:

```
# Implementierung von list
def list(iterable):
    result = []
    for e in iterable:
        result.append(e)
    return result
```

Die Methode *map* ist ein mächtiges Werkzeug zur Bearbeitung von Daten in Python (und anderen Programmiersprachen).

`map`

- erzeugt eine neue Liste von der Größe der ursprünglichen,
- läuft über die Liste und wendet die übergebene Funktion auf das aktuelle Element an und
- schreibt den Rückgabewert des Blockes an die korrespondierende Position der neuen die.

Nach Ablauf von `map` hat man eine neue Liste, bei der jedes Element der ursprünglichen entspricht allerdings nach Anwendung der Funktion.

Wollte man die `map`-Methode mit einer Schleife nachbauen, würde aus dem Beispiel

map

```
a = list(map(lambda x: x*x, [ 1, 2, 3, 4 ]))
```

folgender Python-Code entstehen:

map von Hand programmiert

```
f = lambda x: x*x
a = []
for x in [ 1, 2, 3, 4 ]:
    a.append(f(x))
```

Warum liefert `map` keine Liste, sondern ein spezielles Objekt, das man erst mit `list` wieder in eine Liste konvertieren muss? Der Grund liegt darin, dass man mehrere Operationen auf einer Liste mit `map`, `sum` etc. hintereinander schalten kann und es nicht effizient wäre, jedes Mal eine Liste als Zwischenwert zu erzeugen.

map und sum

```
l = [ 1, 2, 3, 4 ]
s = sum(map(lambda x: x*x, l))
print(s) # -> 30
```

In diesem Beispiel wird die Summe über die Quadrate der Elemente bestimmt, ohne extra eine Liste als Zwischenwert zu benötigen.

4.11 reduce [86]

map verwendet man normalerweise zusammen mit einer anderen Funktion reduce, und kann so ein Verfahren namens *Map/Reduce* implementieren. Es ist ein mächtiges Verfahren, um große Datenmengen zu verarbeiten. Bekannt geworden ist es durch Google, das seinen Suchindex mithilfe von Map/Reduce bearbeitet, siehe *Jeffrey Dean and Sanjay Ghemawat (2004).* MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation.

- Als Gegenstück zu map reduziert reduce die Werte mehrerer Elemente auf ein Element

```
import functools as f
a = [ 1, 2, 3, 4 ]
r = f.reduce(lambda sum, x: sum + x, a)
print(r) # => 10
```

reduce reduziert alle Werte einer Liste auf einen einzigen Wert. (Dieser einzige Wert kann natürlich selbst auch wieder eine Liste sein, sodass man mit den Methoden sehr vielseitige Dinge tun kann).

reduce(lambda a, b: , list, startvalue) geht dabei wie folgt vor:

1. a wird mit dem Wert s initialisiert
2. Die Lambda-Funktion wird mit dem Wert a und dem aktuellen Element b aufgerufen
3. Der Rückgabewert der Lambda-Funktion wird in a gespeichert
4. Weiter bei 2. bis die Liste komplett durchlaufen wurde
5. Liste Collection durchlaufen, gibt den letzten Rückgabwert der Lambda-Funktion zurück

Am Beispiel

```
import functools as f
a = [ 1, 2, 3, 4 ]
r = f.reduce(lambda sum, x: sum + x, a, 100)
print(r) # => 110
```

soll dies schrittweise aufgezeigt werden:

#	sum	x	Lamdba-Rückgabe
1	100	1	101
2	101	2	103
3	103	3	106
4	106	4	*110*

Guido van Rossum mag persönlich weder map noch reduce und konnte sich insofern durchsetzen, als dass reduce in das Modul functools verbannt wurde.

4.12 List Comprehensions [87]

Python hat ein außergewöhnliches Feature, das man aus keiner anderen Programmiersprache kennt: die List Comprehensions. Sie erlauben es, Liste durch Angabe einer erzeugenden Funktion mit Werten zu befüllen. Diese Funktion muss allerdings nicht als Funktion oder Lambda formuliert werden, sondern kann direkt in den eckigen Klammern bei der Listenerzeugung genutzt werden.

- *List Comprehensions* sind ein originäres Python-Feature zum Erstellen von Listen

```
my_list = [ i**2 for i in range(5) ]
print(my_list) # -> [0, 1, 4, 9, 16]

ints = [1, 3, 10]
[i*2 for i in ints] # [2, 6, 20]

[[i, j] for i in ints for j in ints if i != j]
# [[1, 3], [1, 10], [3, 1], [3, 10], [10, 1], [10, 3]]
```

Im ersten Beispiel wird über den Bereich von 0–4 mit Hilfe der `range`-Funktion iteriert und die Liste wird aus den Quadraten dieser Werte erzeugt.

Das zweite Beispiel zeigt, dass für eine List Comprehension eine andere Liste als Ausgangspunkt dienen kann, hier die Liste `ints`.

Im dritten Beispiel wird eine mehrdimensionale Liste erzeugt, wobei zusätzlich zu den Iterationen noch eine Bedingung verwendet wird, die nur ungleiche Werte für `i` und `j` zulässt.

4.13 Tuples als unveränderliche Listen [88]

Mit Tuples hat Python einen weiteren Datentyp zur Darstellung von Sammlungen von gleichartigen Daten. Von der Verwendung sind Tuple analog zu Listen, nur dass man sie nach dem Anlegen nicht mehr verändern kann.

Um Tuples und Listen unterscheiden zu können, definiert man Tuple mit runden Klammern (`()`) und Listen mit eckigen Klammern (`[]`).

- *Tuples* sind ähnlich wie Listen aber unveränderlich, d. h. die Elemente können später nicht mehr geändert werden

```
my_tuple = (1, 2, 3, 4)
my_tuple[0] # -> 1
my_tuple[1] # -> 2

my_tuple[1:3] # -> (2, 3)
```

```
my_tuple[:3] # -> (1, 2, 3)
my_tuple[2:] # -> (3, 4)

my_tuple[0] = 7 # Fehler
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# TypeError: 'tuple' object does not support item assignment
```

4.14 Packing / Unpacking [89]

- Tuples erlauben es, Werte auf mehrere Variablen zu verteilen (*unpacking*) oder
- Werte aus mehreren Variablen in einem Tuple zu sammeln (*packing*)

```
# Packing
t = 'A', 'B', 'C'
print(t) # -> ('A', 'B', 'C')

# Unpacking
a, b, c = t
print(a) # -> 'A'
print(b) # -> 'B'
print(c) # -> 'C'
```

Wenn eine Funktion in Python mehrere Rückgabewerte liefert, geschieht dies hinter den Kulissen über den Umweg von Tuplen (siehe oben).

- Über Tuples können Funktionen mehr als einen Wert zurückgeben

```
def my_function():
    return 'A', 'B', 'C'

x = my_function()
print(x) # -> ('A', 'B', 'C')

a, b, c = my_function()
print(a) # -> 'A'
print(b) # -> 'B'
print(c) # -> 'C'
```

Kapitel 5

Dictionaries und Sets

5.1 Dictionaries [92]

Liste (und Tuple) sind die wichtigsten Datenstrukturen für die Sammlung von gleichartigen Daten. Solche Sammlungen bezeichnet man auch nach dem englischen Begriff als *Collections*. Bei Listen dient eine Zahl (der Index), also die Position in der Liste, als Schlüssel, um auf die Daten zuzugreifen.

Es gibt eine Reihe von Problemen, bei denen sich eine Zahl als Schlüssel für das Wiederfinden der Daten nicht anbietet. Ein typisches Beispiel ist ein Wörterbuch, bei dem ein Wort als Schlüssel für den Zugriff auf die entsprechende Übersetzung dient, so wird man z. B. unter dem Schlüssel „go" die deutschen Begriffe „gehen", „fahren", „laufen" etc. finden. Ein solches Wörterbuch kann man nur schlecht mit Listen abbilden, insbesondere, wenn man die Einträge schnell finden möchte und nicht die ganze Liste durchsuchen will.

Das folgende Beispiel zeigt, wie kompliziert es wird, ein Wörterbuch mithilfe von Listen aufzubauen. Besonders störend ist die lineare Suche in der Liste (`for entry in dictionary:`), deren Laufzeit mit der Anzahl der gespeicherten Wörter linear zunimmt.

```
woerterbuch = [ ["go", [ "gehen", "fahren", "laufen" ] ],
                ["fly", [ "fliegen" ] ]
              ]

def search(dictionary, word):
    for entry in dictionary:
        if entry[0] == word:
            print(entry[1])

search(woerterbuch, "go") # -> ['gehen', 'fahren', 'laufen']
search(woerterbuch, "fly") # -> ['fliegen']
```

Deswegen gibt es in allen modernen Programmiersprachen eine weitere Art von *Collection*, die speziell für die Speicherung unter einem nicht-numerischen Schlüssel konstruiert sind.

- *Dictionaries* (*Wörterbücher*) sind Sammlungen von Name-Wert-Paaren

- Andere Bezeichnungen
  - *assoziatives Array*
  - *Hash*
  - *Map*
- Ein Wert wird mit einem Namen als Schlüssel abgelegt
- Der Name kann ein beliebiger Datentyp sein

```
d = {}
d['go'] = 'gehen'
d['fly'] = 'fliegen'
print(d['go']) # -> gehen
```

Anders als in einer Liste, werden die Elemente in einem Dictionary nicht über einen Index, sondern über einen *Schlüssel* (*Key*) adressiert. D. h. im Dictionary sind genaugenommen nicht einzelne Werte, sondern *Key/Value-Paare* abgelegt.

Man fügt ein neues Element, ähnlich wie bei den Listen, durch Angabe des Schlüssels in eckigen Klammern ein (`d['go'] = 'gehen'`). Analog liest man einen Wert aus dem Dictionary aus, indem man den Schlüssel in eckigen Klammern verwendet (`d['go']`)

Beachten Sie, dass auch Zahlen als Key in einem Dictionary verwendet werden können.

Zahlen als Schlüssel in einem Dictionary

```
d = {}
d[1] = 'hallo'
d[3] = 'du'
print(d[1]) # -> hallo
```

Anders als bei einer Liste müssen diese numerischen Schlüssel nicht fortlaufend sein.

Während eine Liste durch eckige Klammern (`[1, 2, 3]`) und ein Tuple durch runde (`(1, 2, 3)`) gekennzeichnet wird, verwendet man beim Dictionary geschweifte Klammern (`{'go': 'gehen'}`), um es direkt im Quelltext anzulegen.

- Ein *Literal* gibt einen Wert direkt im Quelltext an, ohne ihn zu berechnen
- Dictionaries können über ein Literal auch direkt beschrieben werden

Beispiel: Dictionary als Literal

```
d1 = { 'go': 'gehen', 'fly': 'fliegen' }
d2 = { 1: 'eins', 2: 'zwei', 3: 'drei' }

d1['go'] # -> 'gehen'
d2[2] # -> 'zwei'
```

Allgemein ist die Syntax für Dictionary-Literale: `{ key1: value1, key2, value2, ... }`. Der Schlüssel wird vom Wert durch einen Doppelpunkt (`:`) getrennt.

Der Zugriff auf die Elemente eines Dictionaries erfolgt über den jeweiligen Schlüssel. Damit stellt sich die Frage, welche Datentypen als Schlüssel in einem Dictionary zulässig sind.

- Der Schlüssel muss *unveränderlich* sein
- Jeder Schlüssel ist eindeutig
  ⇒ neue Werte mit demselben Schlüssel ersetzen die alten

```
d = {}
d['go'] = 'gehen'
d['fly'] = 'fliegen'
d['fly'] = 'Fliege' # ersetzt 'fliegen'
print(d) # {'go': 'gehen', 'fly': 'Fliege'}
```

Obwohl in Python häufig Strings als Schlüssel verwendet werden, kann jedes beliebige Objekt als Schlüssel in einem Dictionary verwendet werden. Generell sollte man aber – in allen Programmiersprache – als Schlüssel nur Objekte verwenden, die nicht verändert werden können, da es andernfalls vorkommen kann, dass das Objekt nicht wieder gefunden wird.

5.2 Wichtige Funktionen [95]

Es gibt eine Reihe von Methoden und Funktionen für den Umgang mit Dictionaries.

- `len(dict)` Anzahl der gespeicherten Werte
- `key in dict` testet, ob ein Schlüssel vorhanden ist
- `dict.key()` liefert alle Schlüssel
- `dict.values()` liefert alle Werte
- `dict.items()` liefert eine Liste mit den Schlüssel-Wert-Paaren

```
d = {}
d['go'] = 'gehen'
d['fly'] = 'fliegen'
'go' in d # -> True
'gehen' in d # -> False
list(d.keys()) # -> ['go', 'fly']
list(d.values()) # -> ['gehen', 'fliegen']
```

Wenn man über ein Dictionary iterieren möchte, dann wäre der naive Weg, dies über die `key`-Methode zu machen:

Iteration mit keys()

```
d = { 'go': 'gehen', 'fly': 'fliegen' }
for key in d.keys():
    e = d[key]
    print(e)
```

Ausgabe

```
gehen
fliegen
```

Dieser Weg ist aber nicht elegant, weil man einen unnötigen Zugriff über den Key erzeugt. Deswegen sollte man zum Iterieren über ein Dictionary die `items()`-Methode verwenden.

- Mit der `for`-Schleife kann man über ein Dictionary iterieren
- Man sollte die `items()`-Methode verwenden

Iteration mit items()

```
d = { 'go': 'gehen', 'fly': 'fliegen', 'sleep': 'schlafen' }

for key, value in d.items():
    print(key, '->', value)
```

Ausgabe

```
go -> gehen
fly -> fliegen
sleep -> schlafen
```

5.3 Sets [97]

Die letzte Collection in Python wird durch *Sets* realisiert. Sie dienen dazu, Mengen von Objekten oder Werten abzubilden. Wie in einer mathematischen Menge auch, kann in einem Set jeder Wert nur ein mal enthalten sein, es sind keine Duplikate erlaubt.

Man kann Sets in Python entweder über `{}` als Literal angeben oder sie über die Funktion `set` aus einer Liste oder einer anderen Collection erzeugen.

```
# Set als Literal
s = { 1, 3, 2}
print(s) # -> {1, 2, 3}
```

```
# Set aus Liste
s = set([ 1, 2, 3, 1, 2 ])
print(s) # -> {1, 2, 3}

# Set aus Dictionary
s = set({'go': 'gehen', 'fly': 'fliegen' })
print(s) # -> {'fly', 'go'}

# Set aus Range
s = set(range(1, 5))
print(s) # -> {1, 2, 3, 4}
```

Sets entsprechen mathematischen Mengen

- Jedes Element kann nur einmal enthalten sein (→ Keys von Dictionaries)
- Reihenfolge der Elemente wird nicht erhalten

```
s = set(['a', 'b', 'c', 'a', 'c', 'd', 'b'])
print(s) # -> {'b', 'd', 'c', 'a'}
```

Will man ein leeres Set anlegen, dann kann dies nicht über `{}` erfolgen, weil dieses Literal eine leeres Dictionary erzeugt.

```
s = {}
print(type(s)) # -> <class 'dict'>
```

Deswegen muss man in diesem Fall die `set()`-Funktion verwenden.

```
s = set()
print(type(s)) # -> <class 'set'>
```

Wichtig für die Arbeit mit Sets ist, dass die Werte keine *Ordnung* haben. *Ordnung* bedeutet in diesem Zusammenhang, dass die Elemente innerhalb der Collection in einer festen Reihenfolge abgelegt sind, die sich aus der Reihenfolge des Einfügens ergibt. Wenn eine Ordnung vorhanden ist, kann man den gespeicherten Objekten stabile *Indices* zuordnen, jedes Objekt bekommt einen Index, der über die Zeit stabil bleibt. Hält die Collection keine Ordnung, können die Elemente nicht indiziert werden und bei aufeinanderfolgenden Iterationen können die Objekte in unterschiedlicher Sequenz ausgegeben werden. Insbesondere kommen die Elemente in einer anderen Reihenfolge heraus, als man sie hineingesteckt hat.

- Iteration über Sets efolgt mit `for`-Schleife

Iteration über Set

```
s = { 'A', 'B', 'C', 'D'}

for e in s:
    print(e) # -> D A B C
```

5.4 Wichtige Funktionen [99]

Es gibt eine Reihe von Methoden und Funktionen für den Umgang mit Sets.

- `len(s)` Anzahl der gespeicherten Werte
- `key in s` testet, ob ein Schlüssel vorhanden ist
- `s.add(x)` fügt Element `x` hinzu
- `s.update(list)` fügt mehrere Elemente aus der Liste `list` hinzu
- `s.discard(x)` entfernt das Element `x`
- `s.remove(x)` entfernt das Element `x`
  gibt Fehler, wenn es nicht vorhanden war
- `s.pop()` liefert ein zufälliges Element aus dem Set und entfernt es
- `s.clear()` löscht das Set

```
s = { 'A', 'B', 'C', 'D'}
print('E' in s) # -> False

s.add('E')
print(s) # -> {'A', 'C', 'B', 'E', 'D'}

s.update([ 'F', 'G', 'H' ])
print(s) # -> {'H', 'A', 'F', 'C', 'G', 'B', 'E', 'D'}

s.discard('H')
print(s) # -> {'A', 'F', 'C', 'G', 'B', 'E', 'D'}

s.remove('H') # KeyError 'H'

e = s.pop()
print(e) # -> A
print(s) # -> {'F', 'C', 'G', 'B', 'E', 'D'}

s.clear()
print(s) # -> set()
```

Bei einem leeren Set wird `set()` und nicht `{}` ausgegeben, damit es nicht zu Verwechselungen mit einem Leeren Dictionary kommt.

Mengenoperationen

- `s1.union(s2)` Vereinigungsmenge
- `s1.intersection(s2)` Schnittmenge
- `s1.difference(s2)` Differenzmenge
- `s1.symmetric_difference(s2)` Symmetrische Differenz

```
a = { 'A', 'B', 'C', 'D' }
b = { 'C', 'D', 'E', 'F' }

print(a.union(b)) # -> {'A', 'F', 'C', 'B', 'E', 'D'}
print(a.intersection(b)) # -> {'D', 'C'}
print(a.difference(b)) # -> {'A', 'B'}
print(a.symmetric_difference(b)) # -> {'F', 'B', 'E', 'A'}
```

Anstatt der hier genannten Methoden kann man auch spezielle Operatoren auf Sets anwenden:

- `s1 | s2` Vereinigungsmenge
- `s1 & s2` Schnittmenge
- `s1 - s2` Differenzmenge
- `s1 ^ s2` Symmetrische Differenz

```
a = { 'A', 'B', 'C', 'D' }
b = { 'C', 'D', 'E', 'F' }

print(a | b) # -> {'A', 'F', 'C', 'B', 'E', 'D'}
print(a & b) # -> {'D', 'C'}
print(a - b) # -> {'A', 'B'}
print(a ^ b) # -> {'F', 'B', 'E', 'A'}
```

Dass man Operatoren, die normalerweise für Zahlen benutzt auch für Sets anwenden kann, wird durch ein Sprachfeature ermöglicht, das man als *operator overloading* bezeichnet.

Es gibt auch die Methoden `isdisjoint()`, `issubset()` und `issuperset()`, um zwei Sets miteinander zu vergleichen.

5.5 Set Comprehension [101]

- Sets können ähnlich wie Listen über *Set Comprehensions* erzeugt werden

```
s = { x for x in 'python rocks' if x not in 'aeiou ' }

print(s)
# {'h', 's', 'y', 'n', 'p', 'r', 't', 'c', 'k'}
```

Kapitel 6

Strings

6.1 Zeichenketten [103]

Einer der wichtigsten Datentypen ist die Zeichenkette, auch als *String* bezeichnet. Programme lesen Eingaben von der Tastatur, die als Strings verarbeitet werden müssen oder erzeugen Ausgaben auf dem Bildschirm, die ebenfalls Strings sind. Ein signifikanter Anteil der Programmierung geht deshalb in die Stringverarbeitung.

Strings (*Zeichenketten*)

- sind eine Sequenz von Zeichen (→ Listen)
- sind unveränderlich
- können wie Listen und Tupel „gesliced“ werden
- stehen zwischen einfachen (') oder doppelten Anführungszeichen (")

```
s1 = 'Hallo, Welt!'
s2 = "Hola el mundo!"

s1[7:] # -> 'Welt!'
s2[0:4] # -> 'Hola'
```

Python-Strings werden als Kette von einzelnen Zeichen aufgefasst. Ein Zeichen in einem String kann hierbei jedes Unicode-Zeichen sein; wir sind also nicht auf die 127 Zeichen des US-ASCII-Zeichensatzes beschränkt.

Ein wichtiger Unterschied zu den Liste ist, dass Strings unveränderlich sind. Nach seiner Erzeugung kann man einen String nicht mehr abändern.

Strings sind unveränderbar

```
s = "Hallo"
s[1] = "o" # TypeError: 'str' object does not support item assignment
```

Analog zu den Listen kann man aber Strings slicen, d. h. Sub-Strings erzeugen.

```
s = "Hallo Python-Programmierung"
print(s[-21:]) # -> Python-Programmierung
print(s[:5]) # -> Hallo
```

Durch das Slicen entsteht ein neues, wieder unveränderliches String-Objekt.

Es gibt keinen besonderen Unterschied zwischen den beiden unterschiedlichen *Begrenzern* `'` und `"`. Python unterstützt zwei verschiedene Zeichen, damit man leichter Anführungszeichen innerhalb Strings verwenden kann, indem man das jeweils andere als Begrenzer verwendet.

```
s1 = 'Thomas sagte "Python rocks".'
s2 = "Thomas' Idee war es nicht."
```

Intern bezeichnet Python den Datentyp für Strings als `str`.

```
print(type("ich bin ein String")) # -> <class 'str'>
```

Wir haben bereits bei den Sets gesehen, dass in Python Operatoren überladen werden können, d. h. auch für nicht-numerische Datentypen eingesetzt werden können. Für Strings ist der +-Operator so umdefiniert, dass er zwei Strings verkettet.

- Strings können mit + verkettet werden

```
s1 = "Hallo"
s2 = ", "
s3 = "Welt"

ergebnis = s1 + s2 + s3 + "!"

print(ergebnis) # -> Hallo, Welt!
print(s1) # -> Hallo
```

Bei der Stringverkettung entsteht ein neuer String, die ursprünglichen Strings (`s1`, `s2` und `s3` im Beispiel) bleiben unverändert.

Beim +-Operator müssen beide Operanden Strings sein.

```
s = "Hallo"
ergebnis = s + 3 # Fehler
# TypeError: can only concatenate str (not "int") to str
```

Ein weiterer Operator, der bei Strings eine besondere Bedeutung hat, ist der `*`-Operator. Er dient dazu, die Zeichen eines Strings n-Mal zu wiederholen.

- Strings können mit `*` wiederholt werden

```
chicken = "chicken " * 5

print(chicken)
```

Ausgabe

```
chicken chicken chicken chicken chicken
```

Anders als bei dem +-Operator muss hier einer der Operanden eine Ganzzahl sein. Hierbei ist es egal, ob zuerst die Zahl und dann der String kommt oder umgekehrt.

```
s = "Hallo "
print(s * 3) # -> Hallo Hallo Hallo
print(3 * s) # -> Hallo Hallo Hallo
```

6.2 Umwandlung von/in String [106]

Wie geht man vor, wenn man einen Datentyp hat, der kein String ist und diesen aber als String benötigt?

- Jeder Datentyp in Python kann mit der Funktion `str()` in einen String umgewandelt werden
- Wenn keine sinnvolle Darstellung existiert, wird der Datentyp und seine Adresse ausgegeben

```
t = ('a', 'b', 'c')

s = str(t) # -> "('a', 'b', 'c')"
s[0:3] # -> "('a"

n = 42
s = str(n) # -> '42'
```

`str()` funktioniert immer, liefert aber nicht zwangsweise sinnvolle Ergebnisse. Im folgenden Beispiel wird eine eigene Klasse `X` definiert und davon wird ein Objekt angelegt. Verfüttert man dieses an die `str()`-Methode bekommt man Informationen zu Typ des Objektes (`X`) und seiner Speicherstelle (`0x7f62b4247970`>). Details zu Klassen folgen im entsprechenden Kapitel.

```
class X:
    pass

x = X()
print(str(x)) # -> <__main__.X object at 0x7f62b4247970>
```

Will man dieses Verhalten ändern, muss man der Klasse eine `__str__`-Methode spendieren.

```
class X:
    def __str__(self):
        return "Ich bin ein X ;-)"

x = X()
print(str(x)) # -> Ich bin ein X ;-)
```

Was hier genau passiert, brauchen Sie nicht unbedingt zu verstehen – es nur der Vollständigkeit halber gezeigt.

Bei Listen ist es nicht immer attraktiv, sie einfach mit `str()` in einen String umzuwandeln, weil man dann keine Kontrolle darüber hat, wie die Elemente verkettet werden.

```
l = ['a', 'b', 'c', 'd']
print(str(l)) # -> ['a', 'b', 'c', 'd']
```

Deswegen gibt es eine alternative Möglichkeit Listen mit `join()` in Strings umzuwandeln und dabei deutlich mehr Kontrolle über die Darstellung zu haben, weil man das Trennzeichen angeben kann.

- Listenelemente können mit `join()` zu einem String zusammengesetzt werden
- Syntax: `trennzeichen.join(liste)`

Beispiel: Listenelemente zusammenfügen

```
l = ['a', 'b', 'c', 'd']

print(''.join(l)) # -> 'abcd'
print(' '.join(l)) # -> 'a b c d'
print(', '.join(l)) # -> 'a, b, c, d'
```

Liegen Daten in einem String vor, z. B. weil eine Tabelle mit Messwerten aus einer Textdatei gelesen wurde, dann möchte man diesen String in seine einzelnen Elemente zerlegen und in einer Liste speichern.

Beispiel: Messwerte

```
2.333 1.778 0.882 3.231
```

Für diesen Zweck bietet Python die `split()`-Methode an, der man (optional) ein Trennzeichen übergeben kann an dem der String aufgebrochen und in einzelne Listenelemente überführt wird. Gibt man kein Trennzeichen an, wird an *Whitespaces* (`' '`, `'\n'`, `'\t'`, `'\r'`, `'\f'`, `'\v'`) getrennt.

- Strings können über `split()` in Listen aufgeteilt werden
- standardmäßig wird an Whitespace getrennt

Beispiel: Trennen an Whitespaces

```
s = "Hallo, Python.\nWie geht es dir?"
l = s.split()
print(l) # -> ['Hallo,', 'Python.', 'Wie', 'geht', 'es', 'dir?']
```

Beispiel: Trennen an beliebigem Zeichen

```
s = "1.3, 2.5, 1.4, 6.7"
l = s.split(', ')
print(l) # -> ['1.3', '2.5', '1.4', '6.7']
```

Im vorherigen Beispiel werden die Zahlenwerte per `split()` zwar korrekt auf eine Liste verteilt, es handelt sich aber immer noch um Strings. Deswegen muss man sie, falls sie verarbeitet werden sollen, in einem zweiten Schritt in Fließkommazahlen umwandeln. Dies kann elegant mit einer List-Comprehension erfolgen, ohne eine Schleife zu verwenden.

Umwandlung in Fließkommazahlen

```
s = "1.3, 2.5, 1.4, 6.7"
l = s.split(', ')
n = [float(i) for i in l]
print(n) # -> [1.3, 2.5, 1.4, 6.7]
```

Man kann die Umwandlung auch in einem einzigen Schritt machen.

Direkte Umwandlung in Fließkommazahlen

```
s = "1.3, 2.5, 1.4, 6.7"
n = [float(i) for i in s.split(', ')]
print(n) # -> [1.3, 2.5, 1.4, 6.7]
```

6.3 Escape-Sequenzen [109]

Nicht alle Zeichen, die man ausgeben möchte, kann man einfach in einem Quelltext darstellen, z. B. das Zeichen für eine neue Zeile oder einen Tabulator. Ein anderes Problem sind die Zeichen `"` und `'` die als Stringbegrenzer dienen und deswegen nicht in einem String vorkommen können, der durch sie begrenzt wird.

Damit man diese Zeichen trotzdem in Stringliteralen verwenden kann, gibt es sogenannte *Escape Sequenzen*, die mit einem Backslash \ beginnen und dann das Zeichen definieren, das an Stelle ihrer eingesetzt werden soll. Durch das *Escape-Zeichen* \ bekommt das nächste Zeichen eine spezielle Bedeutung. Da der Backslash ebenfalls zu einem speziellen Zeichen wird, muss man ihn durch \\ darstellen.

Strings können spezielle Zeichen enthalten (*Escape Sequenzen*)

- \\ - Backslash
- `\n` - Neue Zeile (*newline*)
- `\f` - Seitenvorschub (*form feed*)
- `\r` - Wagenrücklauf (*carriage return*)
- `\v` - Vertikaler Tabulator
- `\b` - Backspace (löscht Zeichen links)
- `\t` - Tabulator
- `\'`, `\"` - Anführungszeichen

Beispiel: Escape-Sequenzen

```
print('c:\\windows')
print('vor newline\nnach newline')
print('vor formfeed\fnach formfeed')
print('a\tb')
print('vor\bnach')
print('Ich habe einen 26\'7" Monitor')
```

Ausgabe

```
c:\windows
vor newline
nach newline
vor formfeed
            nach formfeed
a              b
vonach
Ich habe einen 26'7" Monitor
```

6.4 String-Formatierung [111]

Will man Daten ausgeben, kann die Konvertierung mit `str()` und das Verketten mit + umständlich werden. Insbesondere, wenn man die Daten noch in einem speziellen Format darstellen möchte, z. B. bei Fließkommazahlen nur eine begrenzte Anzahl von Nachkommastellen.

```
f = 0.23
s = 'Hello'
i = 12

ausgabe = "s: " + s + " f: " + str(f) + " i: " + str(i)
print(ausgabe) # -> s: Hello f: 0.23 i: 12
```

Um diese Art von Ausgaben zu vereinfachen, bietet Strings über die `format()`-Methode die Möglichkeit mit Platzhaltern zu arbeiten, die im String durch Werte ersetzt werden.

- Formatierung von Strings über `format()`
- Platzhalter (`{}`) werden durch Daten in der Parameterliste ersetzt
- Ersetzung erfolgt von links nach rechts

```
f = 0.23
s = 'Hello'
i = 12
print("s: {} f: {:0.1f} i: {}".format(s, f, i))
```

Ausgabe

```
s: Hello f: 0.2 i: 12
```

In den Platzhaltern (`{}`) können Anweisungen untergebracht werden, wie die Daten einzufügen sind, so sorgt z. B. `{:0.1f}` dafür, dass der Wert als Fließkommazahl mit nur einer Nachkommastelle ausgegeben wird. Die verschiedenen Möglichkeiten hier zu erläutern geht zu weit, sodass auf die Dokumentation verwiesen sei.

Eine weitere Möglichkeit, Strings zu formatieren, besteht darin, die Werte von Variablen direkt in den String zu übernehmen. Damit es nicht zu Fehler kommt, muss man Python explizit dazu anweisen, diese Ersetzungen zu machen, indem man den String durch ein vorangestelltes `f` als *Format-String* kennzeichnet.

Format-Strings erlauben es, Variablen direkt im Text zu ersetzen

- werden mit `f"..."` deklariert
- Variablen stehen in geschweiften Klammern `{name}`

```
dozent = "Smits"
hs = "Mannheim"

print(f"{dozent} lehrt an der Hochschule {hs}")
```

Ausgabe

```
Smits lehrt an der Hochschule Mannheim
```

Die Variablen können von einem beliebigen Typ sein, sie werden über die `str()`-Funktion umgewandelt:

```
pi = 3.1
print(f"Hier ist pi einfach {pi}")
# -> Hier ist pi einfach 3.1
```

6.5 Weitere Operationen für Strings [113]

Es gibt noch eine Reihe von Methoden, die man auf einem String aufrufen kann, um ihn zu verändern.

- `upper()` – in Großbuchstaben umwandeln
- `lower()` – in Kleinbuchstaben umwandeln
- `strip()` – Leerzeichen entfernen
- `count(x)` – Anzahl der Zeichen x
- `replace(a, b)` – ersetze `a` durch `b`
- und viele mehr…

```
'Hallo'.upper() # -> 'HALLO'
'Hallo'.lower() # -> 'hallo'
' Hallo '.strip() # - > 'Hallo'
'Hallo'.count('l') # -> 2
'Hallo'.replace('o', '0') # -> 'Hall0'
```

Neben den Methoden gibt es noch Funktionen und Operatoren, die für Strings eingesetzt werden können.

- `len(s)` – Länge des Strings

- `s1 == s2` – Testet, ob die Strings gleich sind
- `s1 in s2` – Testet, ob `s1` in `s2` vorkommt

```
s = "Hallo"

len(s) # -> 5

"ll" in s # -> true
"a" in s # -> true
"x" in s # -> false

"Hallo" == s # -> true
" Hallo " == s # -> false
```

Index

Aktualparameter, 35
Algorithmus, 1
Anweisung, 10
Argument, 32
Argumentliste, 32
Arrays, 48
assoziatives Array, 65
Aufrufoperator, 41
Ausdruck, 11
auszukommentieren, 9

Bedingungen, 20
Begrenzern, 73
Benannte Argumente, 39

Collections, 64

Dictionaries, 64
Docstring, 46
dynamische Typisierung, 5

Editor, 8
else-Zweig, 23
Endlosschleife, 25
Escape Sequenzen, 77
Escape-Zeichen, 77

Fließkommazahlen, 13
For-Schleife, 26
Formalparameter, 35
Format-Strings, 78
Funktion, 31
Funktionen, 31
fußgesteuerte Schleife, 25

Ganzzahlen, 11
globale Variable, 42
Gültigkeitsbereich, 42

Hash, 65

If-Else-Statement, 22, 23
If-Statement, 21
Indices, 68
Interpreter, 5
iterieren, 57

Key, 65
Key/Value-Paare, 65
Kontrollstrukturen, 20
kopfgesteuerten Schleife, 25

Lambda, 45
Laufvariablen, 57
List Comprehensions, 62
Listen, 48
Literal, 65
Logische Operatoren, 18
lokale Variablen, 42

Map, 65
map, 60
Map/Reduce, 61
Mehrfachzuweisung, 34
Methode, 56
Multiparadigmatisch, 5

Open Source, 6
Operand, 11
Operator, 11
operator overloading, 70
Ordnung, 68

packing, 55, 63
Parameter, 32
plattformunabhängig, 6
programmieren, 3
Programmierung, 3
Prozess, 2

reduce, 61
Referenz, 15
Referenzdatentypen, 15
Referenzübergabe, 37
REPL, 6

Schleifen, 20
Schleifenrumpf, 25
Schlüssel, 65
Schlüsselwörter, 16
Set Comprehensions, 70
Sets, 68
Slices, 52
Slicing, 51
Standardwerten, 40
statischen Typisierung, 5
Strings, 72

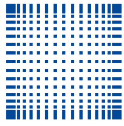
ternäre If, 24
ternäres If, 24
Tuples, 62
Typ, 9

unpacking, 55, 63

Vararg-Funktion, 38
Variablen, 14
Vergleichsoperatoren, 17

Werte, 9
Wertübergabe, 37
While-Schleife, 25
Whitespaces, 76

Zuweisung, 14
Zuweisungsoperator, 14



Python Programmierung (PYP)
Vorlesung - Hochschule Mannheim

Fortgeschrittene Konzepte

Prof. Thomas Smits

Sommersemester 2021

21. März 2022

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Python Programmierung (PYTHON) an der Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Inhaltsverzeichnis

1 Module 1
1.1 Importieren von Modulen [5] . . . 1
1.2 Eigene Module [8] . . . 2

2 Input/Output 4
2.1 Dateizugriff [11] . . . 4
2.2 Open [12] . . . 4
2.3 With [14] . . . 6
2.4 Binärdaten in Python [15] . . . 6
2.5 Lesen aus Datei [17] . . . 7
2.6 Schreiben in Datei [18] . . . 8
2.7 Lesen von der Console [19] . . . 9
2.8 Lesen von Kommandozeilenargumenten [20] . . . 9

3 Klassen 11
3.1 Motivation [22] . . . 11
3.2 Objekt und Klasse [23] . . . 12
3.3 Constructor [27] . . . 15
3.4 self [28] . . . 17
3.5 Klassenattribute [30] . . . 18
3.6 Vererbung [31] . . . 19
3.7 Standardmethoden [33] . . . 20
3.8 Überschreiben [34] . . . 21
3.9 Operatoren überladen [35] . . . 22

4 Rekursion 24
4.1 Idee der Rekursion [38] . . . 24
4.2 Aufbau rekursiver Funktionen [40] . . . 25

5 Ausnahmen 28
5.1 Motivation [44] . . . 28
5.2 try ... except [45] . . . 29
5.3 try ... except [46] . . . 30

5.4 try … except … else [47] 31
5.5 finally [48] 31
5.6 Ausnahme selbst werfen [50] 32

6 Unit-Test **34**
6.1 Definition [52] 34
6.2 Test-Case [53] 34
6.3 Assert [55] 35
6.4 Beispiel [56] 35

Index **iii**

Kapitel 1

Module

1.1 Importieren von Modulen [5]

Python ist eine mächtige Sprache und bietet eine große Menge von vorgefertigten Funktionen an, z. B. für mathematische Berechnungen, Darstellung von Grafik etc. Da aber ein Entwickler nicht immer alle diese Funktionalitäten benötigt und damit es übersichtlich bleibt, sind sie in abgeschlossenen Einheiten, sog. *Modulen* organisiert, die man in das Programm laden muss, um sie zu benutzen.

Um Funktionalitäten aus einem Modul nutzen zu können, muss man das jeweilige Modul in das eigene Programm *importieren*. Hierzu dient das Schlüsselwort `import`.

- Zusätzliche Funktionen sind in *Modulen* organisiert
- Module müssen für die Verwendung mit `import` geladen werden

```
# Modul math laden
import math

# Funktion sqrt aus math verwenden
s = math.sqrt(2)
print(s) # -> 1.4142135623730951
```

Um Namenskonflikte zu vermeiden, muss man nach einem `import` den Namen des Moduls vor den Namen der Funktion setzen, die man aufrufen möchte. Im Beispiel sieht man das an der Funktion `sqrt` für das Wurzelziehen, die aus dem Modul `math` stammt und deswegen mit `math.sqrt()` aufgerufen werden muss.

Es kommt vor, dass man ein Modul importiert, die Aufrufe der Funktionen aber mit einem anderen Namen (*Alias*) versehen möchte. Dies ist möglich, indem man nach dem `import mod` einen neuen Namen mit `as alias` angibt.

- Man kann das Modul beim Importieren umbenennen (*Alias*)

- Syntax: `import x as y`
- Modul ist nur noch unter dem Alias ansprechbar

```
# Modul math mit dem Namen m laden
import math as m

# Funktion sqrt aus math verwenden
s = m.sqrt(2)
print(s) # -> 1.4142135623730951
```

Der Alias ersetzt den Namen des Moduls im aktuellen Programm, sodass man den ursprünglichen Namen nicht mehr verwenden kann.

```
import math as m
s = math.sqrt(2) # Fehler
# NameError: name 'math' is not defined
```

Will man Funktionen `f` aus einem Modul `m` ohne Prefix verwenden, kann man sie mit `from x import f` importieren. Die Funktion ist dann ohne vorangestellten Namen direkt nutzbar.

- `from mod import func` erlaubt es, eine einzelne Funktion `func` aus dem Modul `mod` zu importieren
- Funktion ist dann ohne Prefix nutzbar

```
# Funktion sqrt aus dem Modul math holen
from math import sqrt

# Funktion sqrt wie eine lokale Funktion verwenden
s = sqrt(2)
print(s) # -> 1.4142135623730951
```

1.2 Eigene Module [8]

Ein Python-Modul ist nichts meta-magisches, sondern kann von jeder Entwicklerin selbst erstellt werden. Ein Modul ist einfach nur eine Datei mit dem Modulnamen und der Endung `.py`

- Ein Python-Modul ist eine Datei, die durch `import` geladen wird
- Name der Datei entspricht dem Namen des Moduls + `.py`

Datei: fibo.py

```
def fib(n):
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()
```

Datei: main.py

```
import fibo

fibo.fib(10)
# -> 1 1 2 3 5 8
```

Die Datei, die ein Modul enthält, ist ein gültiges Python-Skript. Sie kann also neben Funktionen auch Befehle enthalten, die direkt ausgeführt werden können. Beim Laden eines Moduls werden diese Befehle ebenfalls ausgeführt und können dazu dienen das Modul zu initialisieren.

In manchen Fällen enthält ein Modul aber Code, der nur laufen soll, wenn das Modul als Skript ausgeführt wird und der gerade nicht beim Laden zuschlagen soll.

Diesen Code kann man durch ein `if` bedingen, dass den Wert der Variable `__name__` prüft. Diese Variable wird von Python selbst gesetzt (das wird durch die Underscores `__` angezeigt) und hat

- in einem Modul den Namen des Moduls als Wert und
- in einem Skript den Wert `"__main__"`.

- Code im Modul wird beim Laden des Moduls ausgeführt
- `if __name__ == "__main__"`:-Konstruktion verhindert dies

```
def fib(n):
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

if __name__ == "__main__":
    print("Wird beim import nicht ausgeführt")
```

Kapitel 2

Input/Output

2.1 Dateizugriff [11]

Daten sind in den allermeisten Fällen nicht im Programm selbst gespeichert, sondern werden auf externen Speichern vorgehalten. Dateien sind hier der häufigste Fall für die Datenspeicherung, sodass wir uns diese zuerst ansehen.

Ein Programm braucht eine Möglichkeit, mit Dateien zu interagieren. Hierzu verwendet Python das Konzept von *Dateiobjekten*, d. h. Objekten, die im Programm benutzt werden können, deren Methoden aber Daten mit der Datei austauschen. Das Dateiobjekt ist somit ein Stellvertreter der Datei im Programm.

- Lesen und Schreiben in Dateien wird über *Dateiobjekte* (*file objects*) realisiert
- Sie werden über `open()` oder `file()` angelegt
- Dateien, müssen mit `close()` wieder geschlossen werden

Das *Öffnen* einer Datei, d. h. die Vorbereitung für die Interaktion, reserviert im Betriebssystem Ressourcen und Speicher. Je nach Betriebssystem wird die Datei auch für Zugriffe durch Andere gesperrt, solange sie geöffnet ist. Um die Ressourcen und Sperren wieder freizugeben, muss man eine Datei nach der Benutzung *schließen*.

Wird eine Datei nicht geschlossen, kann es zu Engpässen beim Speicher kommen und im schlimmsten Fall gehen Daten verloren.

2.2 Open [12]

Eine Datei wird mit der `open()`-Funktion geöffnet. Dieser Funktion muss man zum einen den Dateinamen übergeben und zum anderen anzeigen, was man mit der Datei machen möchte (`options`). Die Syntax mit den Buchstaben und dem Pluszeichen für die Angabe der Optionen stammt aus der Urzeit des Computers und wurde in den 1970er Jahren bei der Programmiersprache C so definiert, ist also historisch gewachsen.

```
f = open(filename, options)
```

- `filename`: Pfad zu der Datei
- `options`: Optionen

Option	Bedeutung
'r'	Datei lesen
'w'	Datei schreiben (wird vorher gelöscht)
'x'	Datei anlegen und zum Schreiben öffnen
'a'	An Ende der Datei anhängen
'r+'	Datei lesen und schreiben
'w+'	Datei lesen und schreiben (wird vorher gelöscht)
'a+'	Lesen und Schreiben am Ende der Datei
'b'	Binärmodus
't'	Textmodus (Standard)

Wenn man eine Datei mit `w` öffnet und es gibt die Datei bereits, dann wird der Inhalt der Datei gelöscht und man fängt wieder am Anfang der Datei an. Andernfalls wird die Datei angelegt.

Sollen alten Daten erhalten bleiben und die neuen ans Ende angehängt werden, muss man die Option `a` verwenden.

Wenn Sie die Option `x` verwenden, dann bricht `open()` mit einem Fehler ab, falls die Datei bereits existiert. Es stellt also sicher, dass Sie nicht aus Versehen eine Datei überschreiben, die bereits existiert.

Etwas trickreich sind die Optionen `b` und `t`, die mit den anderen kombiniert werden. Eine Datei kann von Python entweder im *Textmodus* (Option `t`) oder im *Binärmodus* (Option `b`) geöffnet werden. Im Textmodus macht Python automatische Konvertierungen zwischen Zeichensätzen und andere Anpassungen, die für Textdateien sinnvoll sind. Bearbeitet man allerdings im Textmodus eine Binärdatei (z. B. ein Bild), dann gehen die Daten mit großer Wahrscheinlichkeit kaputt. Deswegen muss man Dateien, die keine Textdateien sind, *explizit* mit der Option `b` öffnen, da `t` der Standardwert ist, wenn keine der beiden Optionen angegeben wird.

```
f = open('myfile.txt', 'rt')

# Datei verarbeiten

f.close()
```

Das Beispiel zeigt, wie man eine Datei zum Lesen öffnet. Die Verarbeitung der Daten ist hier ausgelassen und wird weiter unten gezeigt.

2.3 With [14]

Das Schließen einer geöffneten Datei ist eine wichtige Operation, die man nicht vergessen sollte. Damit auch in komplexen Programmen das `close` zum `open` nicht vergessen wird, gibt es in Python die Möglichkeit mit dem `with`-Statement eine Reihe von Statements zu definieren, nach deren Abschluss die geöffnete Ressource (hier also Datei) wieder geschlossen wird. Allgemein ist die Syntax von `with`:

```
with EXPR as VAR:
    STMTS
    ...
```

Hierbei wird der Ausdruck `EXPR` ausgeführt und das Ergebnis wird der Variable `VAR` zugewiesen. Wenn die Statements `STMTS` abgearbeitet sind, wird automatisch eine Methode ausgeführt, welche die Ressourcen schließt, die `EXPR` geöffnet hat. Wie das genau funktioniert, würde hier zu weit führen – aber es funktioniert.

- `with`-Statement stellt sicher, dass die Datei korrekt wieder geschlossen wird
- kein `close` mehr nötig

```
with open('myfile.txt', 'r') as f:
    # Datei verarbeiten

# f wird automatisch geschlossen, wenn with verlassen wird
```

Durch die Verwendung von `with` in dem Code-Beispiel muss kein `close()` mehr für die Datei `f` aufgerufen werden. Dies passiert automatisch.

2.4 Binärdaten in Python [15]

Da Dateien auch Binärdaten enthalten können, stellt sich die Frage, wie man diese in Python darstellen kann. Hierzu dienen die beiden Klassen `bytes` und `bytearray`.

Binärdaten

- werden durch `bytes`- und `bytearray`-Objekte repräsentiert
- `bytes`
    - können als Literal mit `b""` oder `b''` angegeben werden
      z. B. `bs = b"binaerdaten"`
    - Literale dürfen nur ASCII-Zeichen enthalten
    - sind unveränderlich

- `bytarray`
  - sind veränderlich
  - werden über `bytearray()` erzeugt

Beispiel: Verwalten von Binärdaten

```
b = b"binaerdaten"
ba = bytearray(b)
print(b.hex()) # -> 62696e616572646174656e
print(ba.hex()) # -> 62696e616572646174656e

print(str(b)) # -> "b'binaerdaten'"
print(str(ba)) # -> "bytearray(b'binaerdaten')"
```

Ein Byte entspricht einer Zahl im Wertebereich 0–255. Damit lässt sich jedes Byte durch eine zweistellige Hexadezimalzahl (Basis 16) darstellen. Die `hex()`-Methode von `byte` und `bytearray` macht genau das und gibt den Inhalt als Folge von Hexadezimalziffern aus. Hierbei entsprechen immer zwei Ziffern einem Byte.

2.5 Lesen aus Datei [17]

Nachdem eine Datei geöffnet ist, muss man mit den Daten in der Datei interagieren. Hierzu bietet Python eine ganze Reihe von Methoden an, die man auf dem Dateiobjekt (Rückgabewert von `open()`) aufrufen kann.

Da man häufig Textdateien verarbeitet, bietet es sich an, diese Dateien zeilenweise zu verarbeiten, sodass hierfür Methoden vorhanden sind.

- `read(n)` – Liest n Bytes/Zeichen ein und gibt String zurück
- `readline()` – Liest pro Aufruf eine Zeile
- `readlines()` – Liest die ganze Datei in eine Liste von Zeilen
- `for line in f:` – Liest ebenfalls zeilenweise als Schleife

```
with open('myfile.txt', 'r') as f:
    for line in f:
        print(line)
```

Wenn die Datei in einem Binärformat ist, dann können Sie nur die `read()`-Methode benutzen, da die Methoden für das zeilenweise Lesen keinen Sinn ergeben.

Wenn man `read()` ohne Parameter aufruft, dann wird die gesamte Datei eingelesen und zurückgegeben. Dasselbe passiert bei `read(-1)`.

Denken Sie daran, dass der Speicher des Computers endlich ist und eine Datei durchaus größer sein kann. Deshalb muss man die Methoden `read()` und `readlines()` bei großen Dateien mit Vorsicht verwenden.

2.6 Schreiben in Datei [18]

Analog zu den Methoden zum Lesen gibt es auch Methoden zum Schreiben in eine Datei. Die wichtigste ist die `write()`-Methode, die einen String nimmt und in die Datei schreibt.

- `write(s)` schreibt den String `s` in eine Datei
- `truncate` leert die Datei
- `seek(pos)` springt an eine Position in der Datei
- `tell()` liefert die aktuelle Position

```
name = "Thomas"
with open('hello.txt', 'w') as f:
    f.truncate()
    print(f.tell()) # -> 0
    f.write("Hello, {}!\n".format(name))
    f.write("Genug der Tollerei")
    print(f.tell()) # -> 33
```

Das Beispielprogramm erzeugt folgende Datei:

hello.txt

```
Hello, Thomas!
Genug der Tollerei
```

Die Zeilenenden muss man selbst ausgeben (`\n`), da die `write()`-Aufurufe einfach aufeinanderfolgend in die Datei schreiben.

Bei der Arbeit mit Binärdateien kommt bei Python kein normaler String zum Einsatz, sondern ein `bytes`-Objekt.

```
with open('hello.txt', 'wb') as f:
    f.write(b"Hello, Thomas!\n")
    f.write(b"Genug der Tollerei")
```

Das Ergebnis ist in diesem Fall identisch mit dem vorhergehenden Beispiel, weil der Text nur ASCII-Zeichen enthält. Das ändert sich, wenn man Zeichen außerhalb des ASCII verwendet.

```
with open('non-ascii.txt', 'w') as f:
    f.write("Sonderzeichen äöüß\n")

with open('non-ascii.txt', 'rb') as f:
    b = f.read()
    print(b) # -> b'Sonderzeichen \xc3\xa4\xc3\xb6\xc3\xbc\xc3\x9f\n'
```

Man sieht, dass die Sonderzeichen im UTF-8-Format codiert werden und jedes Zeichen durch zwei Bytes repräsentiert wird.

2.7 Lesen von der Console [19]

Die Funktion `input(prompt)` erlaubt es, Eingaben von der Konsole zu Lesen

```
name = input("Wie heisst Du?: ")
print("Hallo {}".format(name))
```

Ausgabe

```
Wie heisst Du?: Thomas
Hallo Thomas
```

2.8 Lesen von Kommandozeilenargumenten [20]

Man kann Programmen bei deren Aufruf auf der Kommandozeile sogenannte *Argumente* bzw. *Kommandozeilenargumente* mitgeben. Wenn Sie z. B. Ihr Python-Programm mit `python myprog.py` starten, dann übergeben Sie dem Programm `python` das Kommandozeilenargument `myprog.py`.

Wenn man das Modul `sys` importiert, dann finden sich in der Liste `sys.argv` die Kommandozeilenargumente, die dem Skript übergeben wurden.

- Kommandozeile ist weitere Quelle von Eingaben
- Liste `sys.argv` (Modul `sys`) liefert die Argumente

arg_reader.py

```
import sys
a0 = sys.argv[0] # Name des Skripts
a1 = sys.argv[1] # 1. Argument
a2 = sys.argv[2] # 2. Argument
a3 = sys.argv[3] # 3. Argument

print(a0, a1, a2, a3)
```

Ausgabe

```
$ python3 arg_reader.py Argument1 Argument2 Argument3

arg_reader.py Argument1 Argument2 Argument3
```

Per Konvention steht im ersten Element (Index 0) der Argumentliste bei allen Programmiersprachen der Name des Programms, das aufgerufen wurde. Die Kommandozeilenargumente folgen dann in den folgenden Listenelementen.

Kapitel 3

Klassen

3.1 Motivation [22]

Listen und Dictionaries sind flexible und mächtige Datenstrukturen mit denen man schon relativ weit kommen kann. Trotzdem möchte man manchmal Daten so gruppieren, dass man zusammengehörige Daten auch an derselben Stelle hat.

Stellen Sie sich vor, Sie vollen die Daten von verschiedenen Autos verwalten und Sie interessieren sich aktuell für die Leistung in kW und die Höchstgeschwindigkeit in km/h. Dann müssten Sie mit Ihrem aktuellen Wissen wie folgt vorgehen:

```
porsche_leistung = 427 # kW
porsche_vmax = 320 # km/h

panda_leistung = 51 # kW
panda_vmax = 164 # km/h
```

Sie könnten die Daten auch in einem Dictionary speichern:

```
autos = { "porsche": { "leistung": 427, "vmax": 320},
         "panda": { "leistung": 51, "vmax": 164} }
```

Keine der Lösungen ist aber besonders schön. Die Verwendung von Dictionaries hat den Nachteil, dass hier mit Strings gearbeitet wird und schon ein kleiner Tippfehler zu Problemen führt: `autos["porsche"]["Leistung"]` → „KeyError: 'Leistung'“.

- Bisher haben wir eine Reihe von Objekten in Python benutzt
    - Integer
    - Strings
    - Listen
    - Dictionaries

    - etc.
- Man will aber im allgemeinen komplexere Strukturen aufbauen können

Was wir wollen, sind komplexere Strukturen, die alle Daten zusammenhalten. Wir hätten gerne etwas, mit dem wir Autos verwalten können, die eine Leistung und eine Höchstgeschwindigkeit haben.

Der Traum wäre:

```
porsche = Auto(leistung = 427, vmax = 320)
print(porsche.leistung) # -> 427
print(porsche.vmax) # -> 320
```

Im Folgenden wollen wir sehen, wie man solche Datenstrukturen bauen kann.

3.2 Objekt und Klasse [23]

Für die weitere Diskussion sollen zwei Begriffe eingeführt werden: *Objekt* und *Klasse*.

- Ein *Objekt* *(object)* repräsentiert ein einzelnes, konkretes Exemplar
    - Auto mit dem Kennzeichen UN-IX 1970
    - Brad Pitt
    - BluRay mit dem Film „Pulp Fiction“ in meinem Regal
    - Mein Laptop mit der MAC-Adresse A0:FF:E3:09:6E:34
- Objekte lassen sich Klassen zuordnen
    - Fahrzeuge der Marke Renault, Modell Clio
    - Schauspieler
    - BluRays des Film „Pulp Fiction“
    - MacBook Pro

Für das Verständnis der objektorientierten Programmierung ist der Unterschied zwischen *Objekt* und *Klasse* entscheidend. Das Objekt repräsentiert ein einzelnes Exemplar einer bestimmten Art von Objekten. Die Klasse hingegen fasst die Gemeinsamkeiten aller Objekte zusammen, die sich unter einem Oberbegriff zusammenfassen lassen.

So ist z. B. Brad Pitt ein konkretes Objekt, denn es gibt – soweit wir wissen – nur einen einzigen Menschen mit diesem Namen, der als Schauspieler weltbekannt ist und so aussieht wie Tyler Durden aus dem Film „Fight Club“. Es mag zwar noch andere Menschen geben, die „Brad Pitt“ heißen, aber es handelt sich dann um andere Personen und nicht den Schauspieler, den wir normalerweise mit dem Namen verbinden. Der Name ist also hier nicht eindeutig, das Objekt (die Person) aber schon.

Neben Brad Pitt gibt es noch eine Reihe von weiteren Schauspielern mit Weltruhm, z. B. George Clooney. Wir wissen sicher, dass George Clooney und Brad Pitt unterschiedliche Menschen sind und beide teilweise in denselben Filmen auftreten, z. B. in „Ocean's Eleven". Insofern sind Brad Pitt und George Clooney unterschiedliche Objekte, obwohl sie viele Gemeinsamkeiten haben. Trotz ihrer Unterschiede haben sie aber so viele Gemeinsamkeiten, dass wir uns erlauben sie beide mit Begriffen wie „Mensch", „Schauspieler", „Superstar" etc. zu bezeichnen.

Die Programmierung wäre langweilig, wenn man nur mit Objekten hantieren würde, sondern die Gemeinsamkeiten von Objekten sollen sich in einer Weise darstellen lassen, dass wir Objekte derselben Kategorie auf dieselbe Art und Weise behandeln können. Wir nennen diese Kategorien, in die sich Objekte einordnen lassen *Klassen*.

- *Klasse* *(class)* beschreibt den Typ von Objekten (führen neuen *Datentyp* ein)
- Bauplan (Schablone) für gleichartige Objekte
  - Konstruktionszeichnung für einen Auto
  - Tierrasse in der Biologie
- Fasst damit gleichartige Objekte zusammen
  - gleichen Eigenschaften (Attributen)
  - gleichem Verhalten (Methoden)
- Attribute und Methoden bilden eine Einheit

Eine Klasse fasst nun die Gemeinsamkeiten von Objekten zusammen. So wie wir Brad Pitt und George Clooney mit dem Begriff „Schauspieler" bezeichnen können, fasst die Klasse `Schauspieler` die gemeinsamen Eigenschaften von beiden (und allen anderen Schauspielern) zusammen. Ein Schauspieler zeichnet sich durch einen Künstlernamen, eine Liste von Filmen in denen er mitgespielt hat etc. aus.

In der Programmierung dient eine Klasse dazu, Objekte über deren Gemeinsamkeiten zu beschreiben. Anstatt also jedes Objekt einzeln detailliert zu definieren, erzeugt man mit der Klasse eine Schablone, mit der neue Objekte erzeugt werden können.

Man kann sich eine Klasse wie einen Stempel vorstellen und die Objekte als die Stempelabdrücke. Es gibt nur einen Stempel (die Klasse) aber beliebig viele Abdrücke (die Objekte).

In der Programmierung werden in einer Klasse nicht nur die Eigenschaften beschrieben, die wir *Attribute* nennen, sondern auch das Verhalten der Objekte. Unter Verhalten verstehen wir die Aktionen, die Objekte ausführen können und die wir mit den Objekten durchführen können. Das Verhalten manifestiert sich in Form von *Methoden*, die das Objekt trägt und die auf den Attributen des Objekts operieren.

Eine Klasse als konkreter Gegenstand der Programmierung hat verschiedene Bestandteile:

- *Name*, z. B. `Auto`

    - immer im Singular
    - großer Anfangsbuchstabe
- Eigenschaften (*Attribute*), z. B. `vmax`
    - Daten, die alle Objekte der Klasse tragen sollen
    - kleiner Anfangsbuchstabe
- Konstruktoren (werden später erst eingeführt)
- Verhalten (*Methoden*), z. B. `beschleunigen`
    - typische Verhaltensweisen für alle Objekte der Klasse
    - kleiner Anfangsbuchstabe

Die Nutzung der Klasse als reiner Datenspeicher ist nur ein erster Schritt. Viel wichtiger ist die Tatsache, dass die Klassen neben den eigentlichen Daten (*Attribute*) auch das Verhalten der Objekte beschreiben, also *Methoden* enthalten. Moment, wieso beschreiben sie das Verhalten der Objekte? Eine Klasse ist nur die Schablone, in freier Wildbahn treffen Sie auf die aus der Klasse erzeugten Stempelabdrücke (die Objekte). Insofern beschreiben wir in der Klasse, wie sich die Objekte verhalten.

Später werden noch die *Konstruktoren* behandelt, die bei der Erzeugung von Objekten einer Klasse eine entscheidende Rolle spielen.

Als unsere erste Klasse legen wir eine leere Klasse für Autos an. Diese nennen wir, naheliegenderweise, `Auto`.

```
class Auto:
    pass

mein_auto = Auto()

print(mein_auto)
# <__main__.Auto object at 0x1021acb70>
```

Den `pass`-Befehl haben Sie bereits kennengelernt: Er hat keine Funktion, befriedigt aber die Syntax von Python, die für eine Klasse immer mindestens einen Befehl fordert.

Der Ausdruck `Auto()` erzeugt ein neues Objekt von der Klasse `Auto`. Eine Referenz auf dieses Objekt wird in der Referenzvariable `mein_auto` gespeichert. Wir müssen streng trennen zwischen dem Objekt und der Variable, die auf es *zeigt*. Auf ein Objekt können beliebig viele Referenzvariablen zeigen, mindestens aber eine, sonst wird das Objekt automatisch gelöscht.

```
auto1 = Auto() # Eine Variable zeigt auf das Objekt
auto2 = auto1 # Zwei Variablen zeigen auf das Objekt

auto1 = None # Eine Variable zeigt auf das Objekt
auto2 = None # Keine Variable zeigt auf das Objekt -> wird gelöscht
```

Wenn man das Objekt ausgibt, d. h. der `print()`-Funktion übergibt, dann wird die Klasse `__main__.Auto` und die aktuelle Speicherstelle (`0x1021acb70`) angezeigt. Das `__main__` vor dem Klassennamen bedeutet, dass die Klasse keinen Modul zugeordnet wurde, sodass sie automatisch im Hauptmodul namens `__main__` gelandet ist.

Klassen sind die Schablonen für Objekte, d. h. wir können von der Klasse Auto beliebig viele Objekte anlegen, obwohl das aktuell ziemlich sinnlos ist.

Anlegen mehrerer Autos

```
class Auto:
    pass

fuhrpark = [ Auto(), Auto(), Auto() ]

print(fuhrpark)
# [<__main__.Auto object at 0x7ff8ac1f8340>,
# <__main__.Auto object at 0x7ff8ac1777f0>,
# <__main__.Auto object at 0x7ff8ab881e50>]
```

3.3 Constructor [27]

Unser Auto aus den vorherigen Beispielen ist ausgesprochen langweilig. Wir können Objekte davon anlegen, diese tragen aber überhaupt keine sinnvollen Daten. Wir müssen also irgendwie die für ein Auto relevanten Informationen in die Objekte bekommen. Dies geschieht über den *Konstruktor*, der bestimmt, welche Daten wir bei der Erzeugung in das Objekt speichern können.

- Der *Konstruktor* (*Constructor*) ist eine spezielle Methode, die das Objekt bei der Erzeugung *initialisiert*
- Die Variablen des Objektes (*Attribute*) entstehen automatisch bei der ersten Zuweisung

```
class Auto:
    def __init__(self, vmax):
        self.vmax = vmax

porsche = Auto(250)
ente = Auto(100)
```

```
print(porsche.vmax) # -> 250
print(ente.vmax) # -> 100
```

In Python hat der Konstruktor den seltsamen Namen `__init__` und bekommt immer mindestens einen Parameter, namens `self`. Dieser Parameter ist eine Referenz auf das aktuell erzeugte Objekt und kann deswegen dazu benutzt werden, die Attribute des Objektes zu initialisieren.

self zeigt auf das aktuelle Objekt

```
class Selfie:
    def __init__(self):
        print(self)

a = Selfie()
print(a)
```

Ausgabe

```
<__main__.Selfie object at 0x7ff8ab1c2ac0>
<__main__.Selfie object at 0x7ff8ab1c2ac0>
```

Man sieht, dass beide Ausgaben dasselbe Objekt zeigen.

Ein Konstruktor kann zusätzlich zu `self` weitere Parameter haben, die dann dazu benutzt werden können, Daten im Objekt abzulegen. Natürlich müssen diese nicht immer eins-zu-eins gespeichert werden, sondern können auch im Konstruktor beliebig verarbeitet werden, bevor sie im Objekt gespeichert werden. Weiterhin kann der Konstruktor auch Attribute anlegen, zu denen es keine Parameter gibt. Die Beziehung zwischen Parametern des Konstruktors und Attributen ist daher keine feste, sondern hängt von der Klasse ab.

Attribute und Konstruktor-Parameter

```
class Auto:
    def __init__(self, ps):
        self.leistung = ps * 0.73549875
        if self.leistung > 100:
            self.fahrspass = "Hoch"
        else:
            self.fahrspass = "Niedrig"

a1 = Auto(100)
a2 = Auto(200)

print(a1.leistung, a1.fahrspass) # -> 73.549875 Niedrig
print(a2.leistung, a2.fahrspass) # -> 147.09975 Hoch
```

Das Beispiel zeigt, wie ein Attribut (`leistung`) aus einem Parameter berechnet wird und ein anderes Attribut (`fahrspass`) in Abhängigkeit davon gesetzt wird.

Der Konstruktor (`__init__`) wird von Python immer automatisch gerufen, wenn ein Objekt erzeugt wird.

3.4 self [28]

Python hat eine Besonderheit, die es von vielen anderen objektorientierten Programmiersprachen unterscheidet: Bei der Definition von Methoden muss man den Parameter `self`, der auf das aktuelle Objekt zeigt, explizit mit angeben. Andere Programmiersprachen machen das nicht so, sondern erzeugen den Parameter hinter den Kulissen und stellen ihn dann magisch in der Methode zur Verfügung.

In Ruby zum Beispiel sieht eine Methodendefinition wie folgt aus:

Klasse mit self in Ruby

```ruby
class Auto
  def methode()
    print(self)
  end
end

a = Auto.new()
a.methode() # -> #<Auto:0x0000559d8e6846e8>
```

In Python muss man den `self`-Parameter selbst angeben:

Klasse mit self in Python

```python
class Auto:
    def methode(self):
      print(self)

a = Auto()
a.methode() # -> <__main__.Auto object at 0x7f3d6c7bf880>
```

In beiden Programmiersprachen muss man beim Aufruf den Parameter für das aktuelle Objekt nicht angeben, dieser wird automatisch gesetzt. Bei der Definition der Methode muss er aber in Python aufgeführt werden.

- In jeder Methode gibt es eine spezielle Referenzvariable, die auf das eigene Objekt zeigt, die *self-Referenz*
- Über die *self-Referenz* greift die Methode auf Attribute des Objektes zu
- Sie verhält sich ähnlich einer normalen Referenzvariable (allerdings keine Zuweisung möglich)
- Anders als in anderen Programmiersprachen, muss `self` explizit in den Methoden deklariert werden

Das folgende Beispiel zeigt die Verwendung von `self` im Konstruktor und in einer Methode `beschleunigen`.

```
class Auto:
    def __init__(self, vmax):
        self.vmax = vmax
        self.v = 0

    def beschleunigen(self):
        self.v = self.v + 10

        if (self.v > self.vmax):
            self.v = self.vmax

ente = Auto(100)
ente.beschleunigen()
print(ente.v) # -> 10

ente.beschleunigen()
print(ente.v) # -> 20
```

3.5 Klassenattribute [30]

Die Attribute, die im Konstruktor erzeugt werden, sind für jedes Objekt individuell, d. h. jedes Objekt kann einen anderen Wert in diesen Variablen haben. Oben wurde das z. B. mit Autos gezeigt, die unterschiedlichen Höchstgeschwindigkeiten haben können.

In manchen Fällen möchte man Attribute nicht pro Objekt speichern, sondern für alle Objekte nur einen einzigen Wert verwalten. Hierzu benutzt man *Klassenattribute*.

Klassenattribute *(class attributes)* werden über Instanzen hinweg geteilt

```
class Auto:
    zaehler = 1 # Klassenattribut

    def __init__(self):
        self.serien_nummer = Auto.zaehler
        Auto.zaehler += 1

ente = Auto()
print(ente.serien_nummer) # -> 1

porsche = Auto()
print(porsche.serien_nummer) # -> 2

print(Auto.zaehler) # -> 3
```

Man sieht in dem Beispiel, dass der Wert von Autozähler über die Objekte hinweg existiert, weil er auf der Ebene der Klasse und nicht der Objekte definiert ist.

3.6 Vererbung [31]

Objektorientierte Programmierung wäre nicht so erfolgreich, wenn es die *Vererbung* nicht gäbe: Man kann neue Klassen von bereits existierenden Klassen erben lassen und damit die Eigenschaften und Methoden der *Elternklasse* auf die *Kindklasse* übertragen. Die Kindklasse hat dann die Möglichkeit, die Attribute und Methoden der Elternklasse zu erweitern und zu verfeinern.

- Klassen können Eigenschaften (Methoden und Attribute) an andere Klassen weitergeben (*vererben*)
- Man hat dann *Kindklassen* und *Elternklassen*

```
class Tier:
    def __init__(self, beine):
        self.beine = beine

class Hund(Tier): # Hund erbt von Tier
    def __init__(self, name):
        super().__init__(4)
        self.name = name # Hunde haben Namen

    def beissen(self):
        print(self.name, 'hat dich gebissen')
```

Die Kindklasse (*Subklasse*) gibt die Elternklasse (*Superklasse*) in Klammern hinter ihrem Namen an `Hund(Tier)`. Sie kann dann in ihrem Konstruktor den Konstruktor der Elternklasse rufen, damit deren Attribute korrekt initialisiert und angelegt werden. Dies geschieht über das Konstrukt `super().__init__(...)`.

Man sieht im Beispiel, dass die Kindklasse `Hund` ein weiteres Attribut `name` einführt und eine Methode `beissen()` hinzufügt. Das Attribut `beine` erbt sie von der Elternklasse.

```
h = Hund("Hasso")

print(h.beine) # -> 4
print(h.name) # -> Hasso
h.beissen() # -> Hasso hat dich gebissen
```

```
print("Hund")
h = Hund('Hasso')
h.beissen()
print(h.beine)
print("-------")
s = Tier(8)
print("Spinne")
print(s.beine)
```

Ausgabe

```
Hund
Hasso hat dich gebissen
4
-------
Spinne
8
```

3.7 Standardmethoden [33]

Jede Klasse in Python erbt automatisch eine Reihe von Standardmethoden. Diese Methoden können *überschrieben* werden (siehe unten), um das Verhalten an die jeweilige Klasse anzupassen.

Python hat die Konvention, dass Methoden, die von magisch von Python eingeführt werden mit zwei Underscores (`__`) beginnen und enden. Als Programmierer sollte man auf keinen Fall neue Methoden einführen, die diese Konvention benutzen.

Jede Klasse erbt eine Reihe von Standardmethoden

- `__init__`: Konstruktor
- `__repr__`: String-Repräsentation des Objekts (für die Maschine)
- `__str__`: String-Repräsentation des Objekts (für Menschen)
- `__eq__`: Vergleicht den Inhalt (für ==-Operator)
- ...

Die `__init__`-Methode haben wir im Zusammenhang mit dem Konstruktor bereits kennen gelernt.

Für die Umwandlung eines Objektes in einen String gibt es in Python zwei verschiedene Methoden: `__repr__` und `__str__`. Der Unterschied besteht in der Zielgruppe der Methode:

- `__str__` wendet sich an den Benutzer uns gibt den Inhalt des Objektes in einer Form aus, die für einen Menschen gut verständlich ist. Wie diese Format genau aussieht, kann die Entwicklerin selbst entscheiden. Diese Methode wird gerufen, wenn man das Objekt an `str()` übergibt.

- `__repr__` liefert eine String-Repräsentation des Objektes, die für eine Maschine verständlich ist. Was ist damit gemeint? Die Ausgabe von `__repr__` sollte, wenn man sie dem Python-Interpreter verfüttert, wieder das Objekt erzeugen können. D. h. sie liefert einen gültigen Python-Ausdruck als String, der das Objekt erzeugen kann. Diese Methode wird gerufen, wenn man `repr()` mit dem Objekt aufruft.

Beispiel: __repr__ und __str__

```
class Auto:
    def __init__(self, name, vmax):
        self.name = name
        self.vmax = vmax

    def __str__(self):
        return f"{self.name}: vmax={self.vmax}"

    def __repr__(self):
        return f"Auto('{self.name}', {self.vmax})"

p = Auto("Porsche 911", 427)
print(str(p)) # -> Porsche 911: vmax=427
print(repr(p)) # -> Auto('Porsche 911', 427)

# Die Ausgabe von repr(...) kann man an eval() übergeben
# und bekommt wieder das Objekt heraus
p2 = eval(repr(p))
print(p2) # -> Porsche 911: vmax=427
```

Die Methode *eval* kann einen beliebigen String als Python-Code interpretieren. Hierbei ist zu beachten, dass

1. auf keinen Fall Strings von außen in den Code einfließen sollten
2. die Ausführung deutlich langsamer ist
3. der Code nur sehr schwer zu verstehen ist.

Besonders Punkt 1 kann sehr viel Kopfzerbrechen verursachen: Jede Möglichkeit Daten, die von außen kommen, als Code zu interpretieren, öffnet Tür und Tor für sogenannte *Code Execution* oder noch schlimmer *Remote Code Execution* Schwachstellen. Ein Angreifer kann dem Programm durch geeignete Eingaben seinen Schadcode unterschieben und diesen dann ausführen lassen.

3.8 Überschreiben [34]

Die Methoden, die von der Elternklasse geerbt werden, passen nicht immer für die Kindklasse. Insbesondere die bereits erläuterten Methoden `__str__` etc. müssen von der Kindklasse ersetzt werden können, weil die Standardimplementierung keinen Sinn ergibt.

- Nicht immer passen die Methoden der Elternklasse für die Kinder

- Subklassen können Methoden ihrer Eltern *überschreiben* (*overwrite*)
- Die neue Methode ersetzt die alte

```
class A:
    def __str__(self):
        return "Ich bin ein A!"

a = A()
print(a) # -> Ich bin ein A
```

3.9 Operatoren überladen [35]

Klassen können die vorhandenen Operatoren von Python für ihre Zwecke anpassen, indem sie sie überladen. Wir haben das bereits bei den Listen gesehen, die + und * mit einer neuen Bedeutung überlegen.

Jede Klasse kann Operatoren für ihre Zwecke umdefinieren.

Operatoren überladen (*operator overloading*)

- Operatoren (+, -, *, /) werden für eihene Klassen umdefiniert
- Zu jedem Operator gehört eine Methode
  - `__add__` für +
  - `__sub__` für -
  - ...

Es gibt in Python eine eingebaute Unterstützung für komplexe Zahlen. Trotzdem werden diese im folgenden Beispiel genutzt, um das Überladen von Operatoren zu zeigen.

```
class Complex:
    def __init__(self, r, i):
        self.r = r
        self.i = i

    def __add__(self, o):
        return Complex(self.r + o.r, self.i + o.i)

    def __str__(self):
        return "Complex({}, {})".format(self.r, self.i)

c1 = Complex(1, 2)
c2 = Complex(4, 7)
```

```
print(c1 + c2) # -> Complex(5, 9)
```

Kapitel 4

Rekursion

4.1 Idee der Rekursion [38]

Ein Programm besteht aus Anweisungen, die man in Funktionen zusammenfassen kann. Funktionen rufen andere Funktionen auf, um ihre Arbeit zu verrichten. All das haben wir bisher bereits gesehen und angewendet.

Interessante Möglichkeiten ergeben sich, wenn man Funktionen sich selbst aufrufen lässt. Natürlich muss man etwas Gehirnschmalz in die Frage stecken, wie das genau erfolgen sollte, weil man sonst einfach eine Endlosschleife baut, die zu einem Programmabsturz führt:

```
def f():
    f()

f() # Fehler!
# RecursionError: maximum recursion depth exceeded
```

Wieso kommt es zu einem Absturz und nicht nur zu einer Endlosschleife? Der Python-Interpreter muss für jeden Funktionsaufruf Speicher auf dem sogenannten *Stack* reservieren. Wenn eine Funktion sich unkontrolliert selbst aufruft, wird dieser Stack-Speicher schnell erschöpft und das Programm stürzt ab.

Es gibt aber Fälle, in denen es die Programmierung vereinfacht, wenn Funktionen sich selbst aufrufen und um diese geht es, wenn man *Rekursion* einsetzt.

Um Rekursion zu verstehen, muss man zuerst Rekursion verstehen.

Rekursion *(recursion)* Algorithmus, bei dem sich eine Methode selbst aufruft

- Erlaubt elegante Lösung vieler Probleme, die sich mit Schleifen (*iterativ*) nur kompliziert lösen lassen
- Kann immer auch in einen iterativen Algorithmus umgewandelt werden

Rekursion haftet nichts Magisches an, da man jeden rekursiven Algorithmus auch in eine iterativen umwandeln kann und umgekehrt. Deswegen setzt man Rekursion dann ein, wenn es die Problemlösung eleganter, besser verständlich oder schneller macht.

Ein klassisches Beispiel für einen Algorithmus, den man gut rekursiv formulieren kann, ist die Berechnung der Fakultät.

Fakultät: Rekursiv

```
def fak(n):
    return n * fak(n - 1) if n > 1 else 1
```

Fakultät: Iterativ

```
def fak(n):
    ergebnis = 1

    for i in range(2, n + 1):
        ergebnis *= i;

    return ergebnis
```

Man sieht hier, dass die rekursive Variante deutlich einfacher aussieht und leichter zu verstehen ist. Schneller ist sie nicht, weil ein Funktionsaufruf immer etwas zusätzliche Zeit kostet.

4.2 Aufbau rekursiver Funktionen [40]

Damit rekursive Funktionen nicht in einer Endlosschleife enden, haben sie immer zwei Teile.

Rekursive Funktionen bestehen aus zwei Teilen

- *Rekursionsanfang*: Einfacher Fall, der am Ende der Rekursion erreicht wird (`n = 1: 1`)
- *Rekursionsschritt*: Bringt einen näher an den Rekursionsanfang und ruft die Funktion selbst `n > 1: f(n-1) + f(n-2)`

What? Der *Rekursionsanfang* wird am *Ende* der Rekursion erreicht, das ist doch total unlogisch. Nein ist es nicht, weil rekursive Funktionen das Pferd von hinten aufzäumen: Sie werden mit einem Wert aufgerufen, verarbeiten diesen und bewegen sich dann einen Schritt näher auf den Rekursionsanfang hin, bevor sie sich selbst aufrufen. Damit wird der Rekursionsanfang am Ende der Rekursion erreicht. Trotzdem ist er, wenn man die rekursive Funktion mathematisch formuliert, der Anfang der Rekursion.

Schauen wir uns das Ganze bei einer bekannten Formel an: der Berechnung einer Fibonacci-Zahl. Vielleicht kennen Sie diese Zahlen noch aus der Schule; es sind die mit den Kaninchen.

Mathematisch ist die Fibonacci-Zahl wie folgt definiert:

$$f(n) = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ f(n-1) + f(n-2) & \text{andernfalls} \end{cases}$$

In der Formel sehen wir schon die Elemente der Rekursion:

- *Rekursionsanfang*: wenn $n < 3$ gilt, dann ist das Ergebnis 1
- *Rekursionsschritt*: wenn $n > 2$ gilt, dann $f(n-1) + f(n-2)$

Diese Formel kann man sehr einfach in einer rekursiven Funktion darstellen:

```
def fib(n):
    """ Rekursive Berechnung der Fibonacci-Zahl zu n. """
    return fib(n - 1) + fib(n - 2) if n > 2 else 1
```

Im Beispiel wird ein ternäres `if` verwendet, um die Funktion möglichst kompakt darzustellen. Man könnte auch schreiben:

```
def fib(n):
    """ Rekursive Berechnung der Fibonacci-Zahl zu n. """
    if n <= 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Die Fibonacci-Funktion ist ein klassisches Beispiel für eine Funktion, die man ohne Rekursion nur sehr umständlich formulieren kann. Der Grund liegt darin, dass sie sich selbst sogar zweimal pro Iterationsschritt aufruft. In der iterativen Formulierung benötigt man deswegen zwei Zwischenvariablen:

```
def fib(n):
    """ Iterative Berechnung der Fibonacci-Zahl zu n. """

    n2 = 1
    n1 = 1
    n0 = 1

    for i in range(3, n + 1):
```

```
        n0 = n1 + n2
        n2 = n1
        n1 = n0

    return n0
```

Kapitel 5

Ausnahmen

5.1 Motivation [44]

Man programmiert nicht nur für die sonnigen Tage oder anders ausgedrückt, in einem Programm kann viel schiefgehen. Deswegen ist es in jeder Programmiersprache wichtig, sich Gedanke zu machen, wie man Fehler signalisiert und behandelt.

Bei einem Programm können viele Dinge schiefgehen

- Dateioperationen schlagen fehl
- Werte haben den falschen Datentyp
- Netzwerkverbindung wurde gekappt
- ...

Wie soll das Programm reagieren? Einfach abstürzen?

Die Programme, wie wir sie bisher geschrieben haben, stürzen bei einem Fehler einfach ab. Für einfache, kleine Berechnungen mag das ausreichen aber niemand möchte ein solches Programm als Benutzer einsetzen.

Schon eine falsche Benutzereingabe reicht für den Absturz, wenn man die Fehler nicht adäquat behandelt.

```
alter = float(input("Alter: "))
if alter >= 18:
    print("Du darfst Saw anschauen")
else:
    print("Leider nein, wie wäre es mit Frozen?")
```

Ausgabe

```
$ python3 kinokasse.py
Alter: Was geht dich das an?
```

```
Traceback (most recent call last):
  File "alter.py", line 1, in <module>
    alter = float(input("Alter: "))
ValueError: could not convert string to float: 'Was geht dich das an?'
$
```

5.2 try . . . except [45]

Python benutzt für die Fehlerbehandlung ein Konzept, das man so auch bei anderen Programmiersprachen findet: Durch die Schlüsselworte `try` und `except` werden die möglicherweise fehlerbehafteten Anweisungen und die Behandlung der Fehler auf unterschiedliche Abschnitte des Quelltextes aufgeteilt.

Fehlerbehandlung in Python erfolgt über *try* und *except*

- Die Befehle im `try` werden ausgeführt
- Tritt kein Fehler auf
  - werden die Befehle im `except` ignoriert
  - läuft das Programm nach dem try/except weiter
- Wenn ein Fehler auftritt
  - wird der Rest des `try` ignoriert
  - es wird nach einem `except`-Block gesucht, der zum Fehler passt
    → wird einer gefunden, wird er ausgeführt
    → wird keiner gefunden, bricht das Programm ab

Jeder Fehler (*Exception*) wird durch ein Objekt repräsentiert, dessen Klasse von `Exception` abgeleitet ist. Der Typ des Objektes liefert eine genaue Information, welche Art von Fehler aufgetreten ist.

Die Fehlerbehandlung wird von der normalen Programmlogik dadurch getrennt, dass man zwei unterschiedliche Blöcke verwendet. Der eine Block wird mit `try` eingeleitet und enthält die Anweisungen, die möglicherweise zu einem Fehler führen könnten, z. B. Zugriff auf eine Datei. Der andere Block wird mit `except` gekennzeichnet und enthält die Fehlerbehandlung. Er wird nur dann angesprungen, wenn im try-Block ein Fehler aufgetreten ist. Wenn ein solcher Fehler auftritt, wird der try-Block *abrupt* beendet, d. h. er wird nicht weiter ausgeführt, sondern die Programmausführung springt zu dem passenden `except`-Block. Man sagt auch, dass der `except`-Block die Ausnahme fängt (*Fangen einer Ausnahme*). Wenn kein Fehler auftritt, läuft der `try`-Block ganz normal zu Ende und der `except`-Block wird ignoriert. D. h. die Programmausführung geht nach dem letzen `except`-Block weiter.

5.3 try ... except [46]

Das folgende Beispiel zeigt die beiden Schlüsselworte in Aktion.

```
try:
    filename = sys.argv[1]
    k = int(sys.argv[2])
    print(handle(filename, k))
except IOError as ioe:
    print("File does not exist: {}".format(ioe))
except (ValueError, IndexError):
    print("Error in command line input")
    print("Run as: python wc.py <filename> <k>")
    print("where <k> is an integer")
```

Das Programm verarbeitet eine Datei, deren Name auf der Kommandozeile angegeben wurde. Weiterhin wird auf der Kommandozeile eine Ganzzahl angegeben, die bei der Verarbeitung der Datei verwendet wird.

Im Programm können nun verschiedene Fehler auftreten:

1. `IOError`: Die Datei kann nicht geöffnet werden, weil sie z. B. nicht existiert.
2. `IndexError`: Auf der Kommandozeilen könnten nicht die beiden benötigten Argumente angegeben worden sein.
3. `ValueError`: Der zweite Parameter könnte sich nicht in eine Ganzzahl (`int`) konvertieren lassen.

Das Objekt der Ausnahme kann man durch den Zusatz `as VARNAME` in einer Variable speichern und dann während der Fehlerbehandlung auswerten.

Diese drei Fehler-Fälle werden im Beispiel in zwei `except`-Blöcken verarbeitet. Man hätte auch drei Blöcke nehmen können oder nur einen. Will man die Fälle 2 und 3 unterscheiden können, verwendet man drei Blöcke:

```
try:
    filename = sys.argv[1]
    k = int(sys.argv[2])
    print(handle(filename, k))
except IOError as ioe:
    print("File does not exist: {}".format(ioe))
except (ValueError, IndexError):
    print("Too few arguments")
    print("Run as: python wc.py <filename> <k>")
except ValueError:
    print("Error in command line input")
    print("No integer given")
```

5.4 try … except … else [47]

Will man Operationen für den Fall durchführen, dass kein Fehler aufgetreten ist, dann kann man dies in einem `else`-Block durchführen.

Nach dem letzten `except` kann noch ein `else`-Block angegeben werden

- wird ausgeführt, wenn *keine* Ausnahme aufgetreten ist
- kann benutzt für Code werden, in dem Ausnahmen nicht gefangen werden sollen

Ein Beispiel für diesen Fall könnte z. B. das Speichern einer Datei sein.

```
try:
    f = open("data.txt", "w+")
    f.write("Data...")
except IOError:
    print("File does not exist")
else:
    f.close()
    print("File saved")
```

Warum schreibt man das `close()` nicht einfach in den `try`-Block? Dort würde es doch auch nicht ausgeführt, wenn ein Fehler auftritt, da der Block dann abrupt beendet wird.

Die Antwort liegt in der Frage, was passieren soll, wenn bei den Befehlen im `else` selbst wieder eine Ausnahme auftritt. Steht das `close()` im `try` dann würde bei einem Fehler während dieser Operation das `except IOError:` angesprungen. Schreibt man es aber in den `else`-Block, dann wird der Fehler nicht einfach behandelt, sondern sichtbar.

5.5 finally [48]

Als Gegenstück zu `else` gibt es mit `finally` die Möglichkeit, einen Block anzugeben, der in jedem Fall ausgeführt wird, sowohl im Fehler als auch im Nicht-Fehlerfall.

Es kommt sehr häufig vor, dass man bestimmte Aktionen vorbereitet (z. B. eine Datei öffnet), dann eine Aktion durchführt (in die Datei schreibt) und dann Aufräumarbeiten durchführen muss (die Datei schließen). Wenn jetzt während der Durchführung ein Fehler auftreten kann, wird man entsprechende try-catch-Block benutzen. Tritt ein Fehler auf, so muss man in den meisten Fällen trotzdem die Aufräumarbeiten durchführen. Würde man diese im try-Block machen, würden sie im Fehlerfall nicht ausgeführt. Macht man sie im catch-Block, werden sie nur im Fehlerfall ausgeführt. Es fehlt also die Möglichkeit, Aufräumarbeiten sowohl im Fehler- als auch im Nicht-Fehler-Fall durchzuführen.

Genau zu diesem Zweck dient der finally-Block, der durch das Schlüsselwort `finally` eingeleitet wird. Die Anweisungen in diesem Block werden auf jeden Fall ausgeführt, egal, ob ein Fehler auftritt oder nicht. Sogar wenn der try- oder catch-Block mit `return` verlassen werden sollte, wird vorher noch der finally-Block ausgeführt, bevor die Methode zurückkehrt.

Zusätzlich kann ein `finally`-Block angegeben werden

- wird immer ausgeführt
  - keine Exception geworfen
  - Exception geworfen aber gefangen
  - Exception geworfen aber nicht gefangen
- wird benutzt, um Ressourcen zu schließen

Ein künstliches Beispiel, das die Funktionsweise des `finally`-Blocks verdeutlicht, folgt hier.

Beispiel: finally

```
def div(x, y):
    try:
        return x / y
    except ZeroDivisionError:
        print('Division by zero!')
    finally:
        print('Finally clause')

div(3, 2)
div(3, 0)
```

Ausgabe

```
Finally clause
1.5

Division by zero!
Finally clause
```

5.6 Ausnahme selbst werfen [50]

Man kann nicht nur eigene Exceptions schreiben, man sollte es auch. Zu einem guten Programm gehören auch passende Ausnahmen. Allerdings sollte man dies nicht übertreiben und so häufig wie möglich auch vorhandene Exceptions von Python verwenden, da man hier dem Verwender des APIs bereits bekannte Ausnahmen anbieten kann.

Da Ausnahmen ganz normale Python-Objekte sind, muss man nur eine entsprechende Klasse schreiben, um eine eigene Art von Ausnahmen zu definieren. Die Ausnahme-Objekte werden, wie alle Objekte erzeugt.

Damit hat man aber erst ein Objekt erzeugt, es aber noch nicht auf den Weg gebracht. Hierzu muss man es mit dem Schlüsselwort *raise* werfen (*Werfen von Ausnahmen*).

- Mit `raise` kann eine Ausnahme selbst ausgelöst (*geworfen*) werden
- Man kann eigene Exception-Klassen schreiben

```
def div(x, y):
    if y == 0:
        raise ZeroDivisionError('So nicht mein Freund!')
    return x / y
```

REPL

```
>>> div(5, 2)
2.5

>>> div(2, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in div
ZeroDivisionError: So nicht mein Freund!
```

Eine eigene Ausnahme muss von `Exception` abgeleitet werden. Die Konvention fordert, dass der Name auf `Error` endet.

```
class MeinError(Exception):
    pass

def magic(n):
    if n == 42:
        raise MeinError()

magic(1)
magic(42) # Error
# __main__.MeinError
```

Man kann die eigenen Ausnahmen auch mit weiteren Daten versehen, da sie normale Klassen sind.

Kapitel 6

Unit-Test

6.1 Definition [52]

Programme können Fehler enthalten und diese möchte man entdecken bevor die Software ausgeliefert bzw. weitergegeben wird. Deswegen muss man eine Software *testen*. Die einfachsten und von der Entwicklerin normalerweise direkt mitproduzierten Tests bezeichnet man als *Unit-Tests*.

- Ein *Unit-Test* testet eine kleine selbständige Einheit (Unit) der Software, z. B. eine Klasse, ein API, ein Modul
- Der Test versucht die Unit möglichst *isoliert* zu testen
- Unit Test werden häufig *vom Entwickler* selbst *geschrieben*
- Unit Tests sind ein wichtiger Teil der *Testgetriebenen Entwicklung*
- Für Python verwendet man das Modul `unittest`

Eine detaillierte Einführung in das Schreiben von Unit-Tests kann hier aus Zeitgründen nicht gegeben werden. In der Praxis gibt es eine ganze Reihe von Hilfsmitteln, um die zu testende Einheit zu isolieren und dann einzeln zu testen. Hierzu gehören Mocks, Stubs, Test Harness etc.

Die *Testgetriebene Entwicklung* *(test-driven development)* (*TDD*) ist eine Methode, die häufig bei der agilen Entwicklung von Computerprogrammen eingesetzt wird. Hier erstellt der Software-Entwickler die Tests grundsätzlich *vor* den zu testenden Komponenten.

Die Menge der Tests, die zusammengehören, nennt man *Test-Suite*.

6.2 Test-Case [53]

Die Tests für ein Programm bestehen aus einer Reihe von *Testfällen*.

- Ein Test besteht aus *Testfällen* (*test case*)
- Ein *Testfall*
    - beantwortet eine einzelne Frage zum Code

    - sollte automatisch laufen (keine menschlichen Eingaben)
    - sollte selbst feststellen, ob er erfolgreich war
    - sollte unabhängig von den anderen Tests sein

Was sollte getestet werden?

- Bekannte Werte
- Randfälle
- Falsche Eingabewerte
    - Zu lange Eingaben
    - Negative Werte
    - String statt Integer
    - Integer statt String
- ...

6.3 Assert [55]

Es gibt eine Reihe von *Assert-Funktionen*, mit denen Ergebnisse getestet werden

- `assertEqual(a, b)`, `assertNotEqual(a, b)`
- `assertTrue(x)`, `assertFalse(x)`
- `assertIs(a, b)`, `assertIsNot(a, b)`
- `assertIsNone(x)`, `assertIsNotNone(x)`
- `assertIn(a, b)`, `assertNotIn(a, b)`
- `assertAlmostEqual(a, b)`, `assertNotAlmostEqual(a, b)`
- `assertGreater(a, b)`, `assertLess(a, b)`
- `assertItemsEqual(a, b)`

Eine *Zusicherung* oder *Assertion* (lateinisch/englisch für Aussage, Behauptung) ist eine Aussage über den Zustand eines Computerprogramms. Mithilfe von Zusicherungen können logische Fehler im Programm erkannt und das Programm kontrolliert beendet werden. Des Weiteren können Assertions Informationen über den Grad der Testabdeckung während der Verifikation liefern. Quelle: Wikipedia

6.4 Beispiel [56]

known_input.py

```
import unittest

from my_script import is_palindrome
```

```python
class KnownInput(unittest.TestCase):
    knownValues = (('lego', False), ('radar', True))

    def testKnownValues(self):
        for word, palin in self.knownValues:
            result = is_palindrome(word)
            self.assertEqual(result, palin)
```

Ausführen der Tests

```
$ python3 -m unittest discover .
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Index

Alias, 1
Argumente, 9
Assert-Funktionen, 35
Assertion, 35
Attribute, 13, 15

Binärdaten, 6
Binärmodus, 5

Code Execution, 21

Dateiobjekte, 4

Elternklassen, 19
eval, 21
except, 29
Exception, 29

Fangen einer Ausnahme, 29

importieren, 1
iterativ, 24

Kindklassen, 19
Klasse, 13
Klassenattribute, 18
Kommandozeilenargumente, 9
Konstruktor, 15

Methoden, 13
Modulen, 1

Objekt, 12
Operatoren überladen, 22

raise, 33
Rekursion, 24
Rekursionsanfang, 25
Rekursionsschritt, 25
Remote Code Execution, 21

self-Referenz, 17
Subklasse, 19
Superklasse, 19

TDD, 34
Test-Suite, 34
testen, 34
Testfällen, 34
Testgetriebene Entwicklung, 34
Textmodus, 5
try, 29

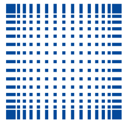
Unit-Test, 34

vererben, 19
Vererbung, 19

Werfen von Ausnahmen, 33

Zusicherung, 35

überschreiben, 22



hochschule mannheim

Python Programmierung (PYP)
Vorlesung - Hochschule Mannheim

NumPy und Matplotlib

Prof. Thomas Smits

Sommersemester 2021

21. März 2022

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Python Programmierung (PYTHON) an der Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Inhaltsverzeichnis

1 NumPy **1**

- 1.1 Was ist NumPy? [5] . . . 1
- 1.2 NumPy importieren [6] . . . 2
- 1.3 NumPy Beispiele [7] . . . 3
- 1.4 Matrizen sind veränderlich [9] . . . 4
- 1.5 Attribute von Matrizen [10] . . . 5
- 1.6 Operatoren für Arrays [12] . . . 6
- 1.7 Broadcasting [14] . . . 7
- 1.8 Vektor-Operationen [17] . . . 9
- 1.9 Slicing [18] . . . 9
- 1.10 Iterieren über Matrizen [19] . . . 10
- 1.11 Matrix aus Datei lesen [20] . . . 10
- 1.12 Lineare Regression [21] . . . 11
- 1.13 Regression höherer Ordnung [23] . . . 12

2 Lineare Algebra mit NumPy **14**

- 2.1 Matrix-Operationen [26] . . . 14
- 2.2 Lineare Algebra mit numpy [32] . . . 16
- 2.3 Zufallszahlen [33] . . . 17
- 2.4 Beispiel: Tragwerk [34] . . . 17

3 Matplotlib **22**

- 3.1 Was ist Matplotlib? [42] . . . 22
- 3.2 Beispiel [43] . . . 22
- 3.3 Schönere Graphen mit Seaborn [44] . . . 23
- 3.4 Plot als Grafik speichern [46] . . . 24
- 3.5 Achsenbeschriftung [47] . . . 25
- 3.6 Mehrere Plots [50] . . . 26
- 3.7 Histogramm [52] . . . 27
- 3.8 Box-Plots [54] . . . 28
- 3.9 Heatmaps [56] . . . 29

Index **ii**

Kapitel 1

NumPy

1.1 Was ist NumPy? [5]

Python ist in technisch-wissenschaftlichen Anwendungen eine der beliebtesten Programmiersprachen und wird von vielen Nicht-Informatikern eingesetzt. Diese Beliebtheit liegt neben der einfachen Sprachsyntax auch an den leistungsfähigen Bibliotheken, die es für mathematische Anwendungen in Python gibt.

Eine wichtige Bibliothek ist NumPy. Sie stellt viele Funktionen und Klassen bereit, die man für Berechnungen in den Natur- und Ingenieurswissenschaften benötigt.

NumPy ist Teil einer Sammlung von Python-Modulen für das wissenschaftliche Rechnen, die SciPy heißt.

Ein großer Vorteil von NumPy liegt darin, dass es die eigentlichen Berechnungen nicht in Python durchführt, sondern hinter den Kulissen auf sehr schnelle C/C++-Funktionen zurückgreift, die alle Beschleunigungsmöglichkeiten moderner Prozessoren voll ausnutzen. Deswegen laufen Berechnungen mit NumPy auch bei großen Datenmengen schnell ab.

NumPy

- Paket für wissenschaftliche Berechnungen mit Python
- n-dimensionale Arrays (Vektoren, Matrizen)
- Lineare Algebra
- Fourier Transformationen
- Zufallszahlen
- Statistik
- Open Source
- …

Sie lernen in dieser Vorlesung NumPy kennen, weil es Ihnen bei der täglichen Arbeit mit den mathematischen Herausforderungen einer Ingenieurswissenschaft helfen kann.

1.2 NumPy importieren [6]

Das zentrale Element von NumPy ist das *ndarray*, ein mehrdimensionales Array von Daten, das anders als die Liste in Python eine feste Größe hat und nur Daten eines Datentyps aufnimmt. Die Daten im Array können verändert werden, nicht aber die Größe und die Dimensionen.

Bevor man NumPy nutzen kann, muss man es installieren (siehe hierzu die Dokumentation) und dann in das Python-Programm importieren.

- NumPy muss installiert werden
- NumPy muss mit `import numpy` in eigene Skripte importiert werden

```
import numpy as np

# Array mit Ganzzahlen anlegen
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a)
# [[1 2 3]
# [4 5 6]]

# Array mit Fließkommazahlen anlegen
b = np.array([[1, 2, 3], [4, 5, 6]], float)
print(b)
# [[ 1. 2.  3.]
# [ 4. 5.  6.]]
```

NumPy mit dem Alias `np` zu installieren ist eine Konvention, die Sie häufig bei den Verwendern von NumPy sehen werden. Verpflichtend ist das natürlich nicht und Sie können jeden Alias verwenden, der Ihnen gefällt.

Ein Array in NumPy wird mit der Funktion `array` angelegt. Dieser können Sie normale Python-Listen übergeben, um diese in ein NumPy-Array umzuwandeln. Als optionalen Parameter können Sie den Datentyp des Array bestimmen; NumPy konvertiert dann alle Elemente in diesen Datentyp.

```
l = [ 1, 2.3, 3 ]
a = np.array(l, int)
print(a) # -> array([1, 2, 3])

b = np.array(l, float)
print(b) # -> array([1., 2.3, 3.])

c = np.array(l, str)
print(c) # -> array(['1', '2.3', '3'], dtype='<U1')
```

Das Array mit den `string`-Werten ist natürlich für Berechnungen nur begrenzt sinnvoll und soll hier nur den grundlegenden Mechanismus verdeutlichen.

1.3 NumPy Beispiele [7]

NumPy ist so umfangreich, dass es hier nicht komplett behandelt werden kann. Trotzdem sollen einige Beispiele die Bereite der Möglichkeiten zeigen.

In weiteren Verlauf dieser Vorlesung werden wir nur auf die Funktionen zur linearen Algebra in NumPy eingehen.

```
import numpy as np

# Matrix nur mit Nullen
a = np.zeros((2, 3))
# array([[ 0., 0., 0.],
#        [ 0., 0., 0.]])

# Form der Matrix
a.shape # -> (2, 3)

# Diagonal-Matrix
b = np.diag([1, 2, 3])
# array([[1, 0, 0],
#        [0, 2, 0],
#        [0, 0, 3]])
```

Warum steht die `(2, 3)` im Beispiel in Klammern? Die Funktion `zeros` erwartet als ersten Parameter die *Dimensionen des Arrays*. Da beliebig viele Dimensionen unterstützt werden, muss man diese in Form eines Tupels übergeben.

```
np.zeros((2, 2, 2))
# array([[[0., 0.],
#         [0., 0.]],
#
#        [[0., 0.],
#         [0., 0.]]])
```

Man kann in NumPy auch sehr einfach Zufallszahlen erzeugen oder Daten speichern und laden.

```
# Matrix mit Zufallszahlen
a = np.random.random((2,3))
# array([[ 0.1834851 , 0.49019724, 0.1798613 ],
#        [ 0.96952566, 0.884116 , 0.40070798]])
```

```
# Speichern in eine Datei
np.savetxt("a_out.txt", a)

# Laden aus Datei
b = np.loadtxt("a_out.txt")
```

1.4 Matrizen sind veränderlich [9]

Das `ndarray` ist die zentrale Datenstruktur von NumPy. Es hat nach der Erzeugung eine feste Größe, erlaubt aber die Veränderung der darin enthaltenen Daten. Damit steht es zwischen den Listen und Tupeln in Python. NumPy-Arrays stellen Matrizen im mathematischen Sinne dar.

Matrizen

- haben feste, unveränderliche Größe (→ Tupel)
- enthalten Daten, die geändert werden können (→ Liste)

```
import numpy as np

# 2x2-Matrix nur mit Nullen anlegen
a = np.zeros((2, 2))
print(a)
# [[ 0. 0.]
#  [ 0.  0.]]

# Element an der Position (0, 0) ändern
a[0, 0] = 7
print(a)
# [[ 7. 0.]
#  [ 0.  0.]]
```

Das Beispiel zeigt, wie man eine *Null-Matrix* anlegt, indem man die gewünschte Dimension als Tupel übergibt. Die Elemente der Matrix kann man dann über deren Index adressieren. Der Index hat genau so viele Dimensionen, wie die Matrix auch.

```
v = np.zeros(3)
print(v) # -> [0. 0. 0.]

v[2] = 42
print(v) # -> [ 0. 0. 42.]

v[0, 0] = 7.0 # Fehler
# IndexError: too many indices for array
```

1.5 Attribute von Matrizen [10]

Jedes `ndarray` in NumPy kann bezüglich der Eigenschaften der dargestellten Matrize befragen. Hierzu bietet es eine Reihe von Attributen an.

Attribute von Matrizen

- `ndim`: Dimensionen
- `shape`: Form
- `size`: Anzahl der Elemente
- `T`: Transponierte Form
- `dtype`: Datentyp

Beispiel: Attribute von Matrizen

```
a = np.array([[1, 2, 3], [4, 5, 6]])

print(a.ndim) # -> 2
print(a.shape) # -> (2, 3)
print(a.size) # -> 6
print(a.dtype) # -> int64
print(a.T)
# [[1 4]
# [2 5]
# [3 6]]
```

Die *transponierte Form* vertauscht Zeilen und Spalten:

Beispiel: Attribute von Matrizen

```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a)
# [[1 2 3]
#  [4 5 6]]

print(a.T)
# [[1 4]
#  [2 5]
#  [3 6]]
```

1.6 Operatoren für Arrays [12]

Der erste große Unterschied zwischen NumPy-Arrays und den Listen in Python ist, dass die arithmetischen Operatoren elementweise angewendet werden. Addiert man z. B. zwei Arrays, dann werden die Elemente der Arrays addiert. Multipliziert man ein Array mit einem Skalar, dann werden alle Elemente mit dem Skalar multipliziert.

Operatoren werden elementweise angewandt (+, -, *, /)

- Skalar: auf jedes Element des Arrays
- Array: auf die passenden Elemente der Arrays

```
a = np.ones((3, 3))

a + 1
# [[ 2., 2.,  2.],
#  [ 2., 2.,  2.],
#  [ 2., 2.,  2.]]

a * 3
# [ 3., 3.,  3.],
# [ 3., 3.,  3.],
# [ 3., 3.,  3.]]
```

Das Beispiel zeigt, wie die Skalare einfach auf alle Elemente der Matrize angewandt werden.

Sind beide Operanden NumPy-Arrays, erfolgt eine elementweise Anwendung der Operatoren. Die Elemente mit demselben Index in beiden Arrays werden verknüpft und das Ergebnis wieder an diesen Index im Ziel-Array geschrieben.

```
a = np.arange(4)
# array([0, 1, 2, 3])

b = np.array([2, 3, 2, 4])
# array([2, 3, 2, 4])

b * a # array([0, 3, 4, 12])
b - a # array([2, 2, 0, 1])

c = [2, 3, 4, 5]
a * c
# array([0, 3, 8,15])
```

In diesem Beispiel wird mit `arange(4)` ein Array `a` mit vier Elementen erzeugt, die den Bereich `0` bis `3` abdecken. Das zweite Array `b` wird über eine Python-Liste erzeugt.

Man sieht, wie die Operatoren auf die einzelnen Elemente der Arrays angewendet werden, um das Ergebnis-Array zu erzeugen.

Was aber passiert, wenn die Arrays unterschiedliche Größen haben? Dann fehlen ja bei einem Operanden die Elemente für die Anwendung des Operators? In diesem Fall kommt das *Broadcasting* zum Einsatz.

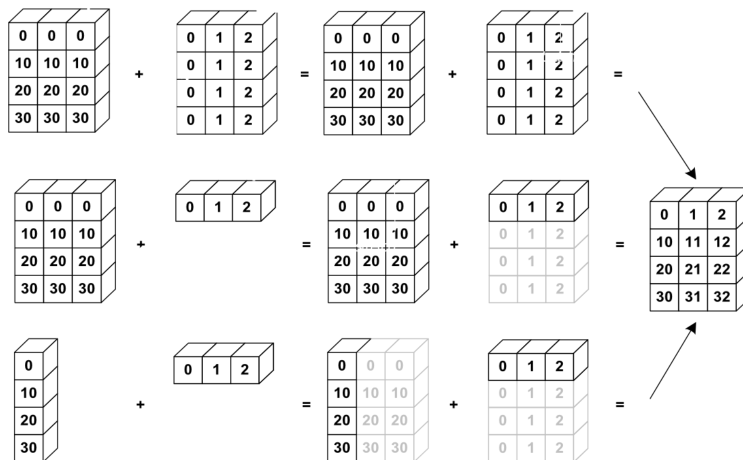
1.7 Broadcasting [14]

Das *Broadcasting* kümmert sich darum, wie die Operationen ablaufen sollen, wenn die Arrays *nicht* dieselbe Dimension haben.

- *Array broadcasting*: Wenn NumPy auf zwei Arrays arbeiten soll, werden die Dimensionen verglichen
- die Dimensionen sind *kompatibel*, wenn
  - sie dieselbe Größe haben *oder*
  - eine von ihnen 1 ist

```
a = np.arange(3)
b = np.arange(4)

a + b
# ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

In diesem Beispiel kann die Addition nicht durchgeführt werden, weil das Array `a` ein Element zu wenig hat. Ein Broadcasting ist nicht möglich, weil die Arrays nur eine Dimension haben und diese nicht identisch und auch nicht 1 ist.



```python
a = np.array([ 0, 1, 2 ])
b = np.array([3])

c = a + b
print(c) # -> [3 4 5]
```

In diesem Fall funktioniert das Broadcasting, weil das Array b die Dimension 1 hat und somit sein Element (3) mit allen Elementen des Arrays a verknüpft werden kann.

Broadcasting führt also keine „magischen“ Modifikationen an den Arrays durch, sorgt aber dafür, dass in bestimmten Fällen Python-Code vermieden wird.

Ohne Broadcasting müsste im Beispiel das Element aus dem Array b herausgeholt werden und als Skalar verwendet:

```python
a = np.array([ 0, 1, 2 ])
b = np.array([3])

c = a + b[0]
print(c) # -> [3 4 5]
```

Was hier bei einem eindimensionalen Array noch relativ wenig Aufwand macht, kann bei mehrdimensionalen Arrays erhebliche Schreibarbeit erfordern – diese wird durch das Broadcasting minimiert.

1.8 Vektor-Operationen [17]

NumPy bietet die üblichen Vektor-Operationen an. Diese sind allerdings nicht als Methoden auf dem NumPy-Array realisiert, sondern sind eigene Funktionen, denen man die Vektoren übergeben kann. Der Grund für diese Lösung liegt darin, dass Vektoren kein Sonderfall von Matrizen sind.

Vektor-Operatonen

- `inner`: Inneres Produkt
- `outer`: Äußeres Produkt
- `dot`: Matrix-Multiplikation
- `cross`: Kreuzprodukt (Vektorprodukt)

```
# Listen werden automatisch in Vektoren konvertiert
u = [1, 2, 3]
v = [1, 1, 1]

np.inner(u, v) # -> 6
np.dot(u, v) # -> 6
np.cross(u, v) # -> [-1, 2, -1]

np.outer(u, v) # -> [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

Man muss keine Python-Listen verwenden, sondern kann auch eindimensionale NumPy-Arrays verwenden. Die Möglichkeit, direkt Listen einzusetzen ist als Vereinfachung gedacht, um die Konvertierung in Arrays zu vermeiden.

```
# Listen werden automatisch in Vektoren konvertiert
u = np.array([1, 2, 3])
v = np.array([1, 1, 1])

np.inner(u, v) # -> 6
np.dot(u, v) # -> 6
np.cross(u, v) # -> [-1, 2, -1]
```

1.9 Slicing [18]

NumPy erweitert die Syntax für das *Slicen* von Liste auf die NumPy-Arrays, sodass man auch diese komfortable in Teilstücke zerlegen kann.

```
a = np.array([[1, 2, 3], [4, 5, 6]])

# 2 Zeile, alle Spalten
a[1, :] # -> [4, 5, 6]

# 1. und 2. Zeile, alle Spalten
a[0:2]
# [[1, 2, 3],
# [4, 5, 6]

# Alle Zeilen, Spalten 3 und 4
a[:, 2:4]
# [[3],
#  [6]]
```

1.10 Iterieren über Matrizen [19]

So wie man über Listen in Python *iterieren* kann – sich also alle Elemente nacheinander ansehen – kann man auch über die Arrays von NumPy laufen. Diese sind so konstruiert, dass man die normale `for`-Schleife aus Python für die Iteration einsetzen kann.

```
a = np.array([[1, 2, 3], [4, 5, 6]])

# Zeilen und Spaltenweise iterieren
for row in a:
    for column in row:
        print(column)

# Über alle Elemente iterieren
for element in a.flat:
    print(element)
```

In beiden Fällen ist die Ausgabe „1 2 3 4 5 6“. Allerdings weiß man im zweiten Fall nicht mehr, wann der Wechsel in die nächste Zeile erfolgt, da diese Informationen durch das `.flat` verloren geht.

1.11 Matrix aus Datei lesen [20]

Größere Matrizen möchte man möglicherweise nicht im Python-Programm selbst pflegen, sondern in einer externen Datei ablegen. Oder die Daten, die man auswerten möchte stammen aus einer anderen Software, z. B. aus einem Laborsystem, mit dem man Messungen aufgezeichnet hat.

Für diese Fälle bietet NumPy die Möglichkeit, die Daten eines Arrays aus einer externen Datei zu laden.

- `loadtxt` erlaubt es ein Array aus einer Datei zu lesen

matrix.txt

```
1 2 3.5
4 5 6
7.0 8 9.3
```

```
import numpy as np

m = np.loadtxt("matrix.txt")

print(m)
# [[ 1. , 2. , 3.5],
#  [ 4. , 5. , 6. ],
#  [ 7. , 8. , 9.3]])
```

Die Datei muss hierbei das Format haben, dass die Zeilen des Arrays auch Zeilen in der Datei sind. Jede Zeile wird durch ein Newline-Zeichen (`\n`) abgeschlossen.

1.12 Lineare Regression [21]

Die folgenden Beispiele zur linearen Regression verwenden die Zeichenfunktionen aus `matplotlib`, die erst im nächsten Kapitel besprochen werden. Da die *lineare Regression* aber so besser darzustellen ist, sei dieser Vorgriff entschuldigt.

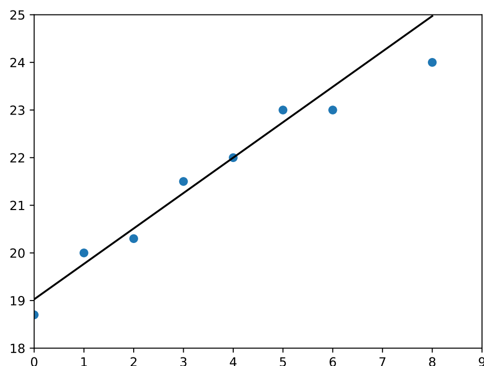
```
import numpy as np
import matplotlib.pyplot as plt

x = [ 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 ]
y = [ 18.7, 20.0, 20.3, 21.5, 22.0, 23.0, 23.0, 25.5, 24.0 ]

fit = np.polyfit(x, y, 1)
fit_fn = np.poly1d(fit)

print(fit_fn) # => 0.7433 x + 19.03

plt.plot(x, y, 'o', x, fit_fn(x), 'k')
plt.xlim(0, 9)
plt.ylim(18, 25)
plt.show()
```

Die Magie liegt hier in der Funktion `polyfit`, die ausgehend von den x- und y-Werten ein Polynom bestimmt, dass die Daten mit den geringsten Fehlerquadraten annähert. Der letzte Parameter von `polyfit` bestimmt den Grad des Polynoms, der hier 1 ist. Damit wird eine lineare Funktion bestimmt, welche die Bedingung erfüllt und somit eine lineare Regression durchgeführt.

Mit `np.poly1d(fit)` wird aus der Beschreibung des Polynoms eine Funktion erzeugt, die dann benutzter werden kann, um die Gerade der linearen Regression zu zeichnen.

Im folgenden Beispiel sieht man das deutlicher:

```
x = [ 0.0,  1.0,  2.0,  3.0,  4.0,  5.0,  6.0,  7.0,  8.0 ]
y = [ 18.7, 20.0, 20.3, 21.5, 22.0, 23.0, 23.0, 25.5, 24.0 ]

fit = np.polyfit(x, y, 1)
fit_fn = np.poly1d(fit)

print(fit_fn(0)) # -> 19.026666666666667
print(fit_fn(1)) # -> 19.77
print(fit_fn(2)) # -> 20.513333333333335
print(fit_fn(3)) # -> 21.256666666666668
print(fit_fn(8)) # -> 24.973333333333336
```

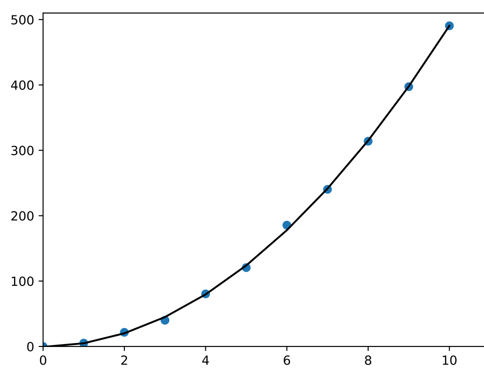


1.13 Regression höherer Ordnung [23]

Nachdem die Funktionsweise von `polyfit` klar geworden ist, kann man leicht auch *Regressionen höherer Ordnung* bestimmen. Hier im Beispiel ein Polynom mit Grad 2.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = [ 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0 ]
y = [ 0.0, 5.1, 21.6, 40.1, 80.4, 120.6, 185.5, 240.3, 313.9, 397.3, 490.5 ]

fit = np.polyfit(x, y, 2)
fit_fn = np.poly1d(fit)

print(fit_fn)

plt.plot(x, y, 'o', x, fit_fn(x), 'k')
plt.xlim(0, 11)
plt.ylim(0, 510)
plt.show()
```



Kapitel 2

Lineare Algebra mit NumPy

2.1 Matrix-Operationen [26]

NumPy bietet alle Operationen an, die man auf Matrizen anwenden kann. Diese werden im folgenden in ein paar Beispielen gezeigt, ohne groß erläutert zu werden. Die mathematische Bedeutung der einzelnen Operationen sei den entsprechenden Mathematikvorlesungen vorbehalten bzw. den einschlägigen Lehrbüchern.

- Definition einiger Matrizen mit dem Wert `1.0` an jeder Stelle
- *Transponieren* der Matrizen mit `.T`

```
a = np.ones((3, 2))
# [[ 1., 1.],
#  [ 1., 1.],
#  [ 1., 1.]]

a.T # Transponierte Matrix
# [[ 1., 1.,  1.],
#  [ 1., 1.,  1.]]

b = np.ones((2, 3))
# [[ 1., 1.,  1.],
#  [ 1., 1.,  1.]]
```

- *Matrix-Multiplikation* mit `dot`
- Dimensionen der Matrizen müssen passen

```
np.dot(a, b)
# [[ 2., 2.,  2.],
#  [ 2., 2.,  2.],
#  [ 2., 2.,  2.]]
```

```
np.dot(b, a)
# [[ 3., 3.],
#  [3., 3.]]

np.dot(A, B.T)
# ValueError: shapes (3,2) and (3,2) not aligned: ...
# ... 2 (dim 1) != 3 (dim 0)
```

Für die Multiplikation der Matrizen müssen die Dimensionen der beiden Operanden passen. Kann auch kein Broadcasting erfolgen (siehe oben), kommt es zu einem `ValueError`.

- *Summieren* aller Elemente oder entlang einer Achse mit `sum`
- *Kumulative Summe* mit `cumsum`

```
a = np.array([[1, 2, 3], [4, 5, 6]])

# Spalten und Zeilensumme
a.sum(axis=0) # -> [5, 7, 9]
a.sum(axis=1) # -> [ 6, 15]
a.sum() # -> 21

# Kummulative Summe
a.cumsum(axis=0) # -> [[1, 2, 3], [5, 7, 9]]

a.cumsum(axis=1) # -> [[ 1, 3, 6], [ 4, 9, 15]]
```

- *Maximum* und *Minimum* mit `max` und `min`
- Auch entlang der Achsen möglich

```
a = np.array([[1, 2, 3], [4, 5, 6]])

# Minumum und Maximum
a.min() # -> 1
a.max() # -> 6

# Maximum und Minimum entlang einer Achse
a.min(axis=1) # -> [1, 4]
a.max(axis=1) # -> [3, 6]
```

- `eye(n)`: Erzeugt die *Identitätsmatrix*
- `trace(a)`: *Spur* berechnen

```
np.eye(3) # 3x3 Idenititätsmatrix
# [[ 1., 0., 0.],
#  [ 0., 1., 0.],
#  [ 0., 0., 1.]]

# Spur berechnen
a = np.array([[1, 2, 3], [4, 5, 6], [7, 6, 8]])
np.trace(a) # -> 14
```

- `column_stack((a, b, ..))`: Erzeugt aus Vektoren eine Matrix (Spalten)
- `row_stack((a, b, ..))`: Erzeugt aus den Vektoren eine Matrix (Zeilen)

```
# Zwei Vektoren
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

np.column_stack([a, b])
# [[1, 4],
#  [2, 5],
#  [3, 6]]

np.row_stack((a, b))
# [[1, 2, 3],
# [4, 5, 6]]
```

2.2 Lineare Algebra mit numpy [32]

NumPy ist so umfangreich, dass die erweiterten Funktionen in Untermodulen untergebracht sind. Diese muss man mit einem weiteren `import` laden, z. B. `import numpy.linalg` für Funktoonen zur linearen Algebra. Verwendet man keinen Alias (siehe oben), dann sind die Funktionen mit dem Prefix `numpy.linalg.` ansprechbar, bzw. `np.linalg.`, wenn NumPy selbst mit dem Alias `np` geladen wurde.

Um sich die vielen Punkte im Aufruf zu sparen, wird man diese Zusatzmodule häufig mit einem neuen Alias laden, z. B. `import numpy.linalg as lin`.

```
import numpy as np
import numpy.linalg as lin

a = eye(3)
# [[1., 0., 0.],
#  [0., 1., 0.],
```

```
# [0., 0., 1.]]

n = lin.norm(a)
print(n) # -> 1.7320508075688772
```

Importieren der Funktionen für lineare Algebra mit `import numpy.linalg`

- `norm(A)`: Norm der Matrix (des Vektors) berechnen
- `inv(A)`: Inverse Matrix berechnen
- `solve(A, b)`: Gleichung `Ax = b` lösen
- `lstsq(A, b)`: Kleinste Fehlerquadrate berechnen
- `eig(A)`: Eigenwert-Vektor berechnen
- `eigvals(A)`: Eigenwert berechnen
- `pinv(A)`: Pseudo-inverse (Moore-Penrose) Matrix

2.3 Zufallszahlen [33]

Für Simulationen oder zum Test von Funktionen benötigt man Zufallswerte. Auch hier bietet NumPy eine Reihe von nützlichen Funktionen, um Matrizen mit entsprechenden, zufälligen Werten zu erzeugen.

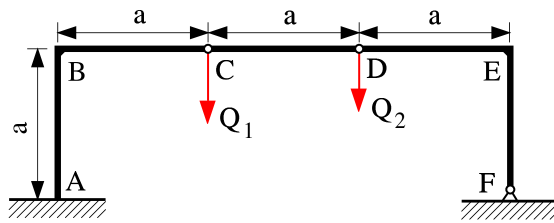
Paket `numpy.random`

- `rand(d0, d1, ..., dn)`: Matrix mit Zufallszahlen (Größe nach Dimensionen)
- `randn(d0, d1, ...,dn)`: Matrix mit normierten Zufallszahlen (Größe nach Dimensionen)
- `randint(lo, hi, size)`: Liste von `size` zufälligen Integers zwischen `lo` und `hi`
- `shuffle(a)`: Mischen der Matrix `a`. Matrix wird verändert
- `permutation(a)`: Mischen der Matrix `a` und zurückgeben einer neuen

2.4 Beispiel: Tragwerk [34]

In einem komplexeren Beispiel wollen wir zeigen, wie man mit NumPy eine statische Berechnung vereinfachen kann. Hierzu gehen wir von einem Tragwerk aus, das in folgender Zeichnung visualisiert ist.

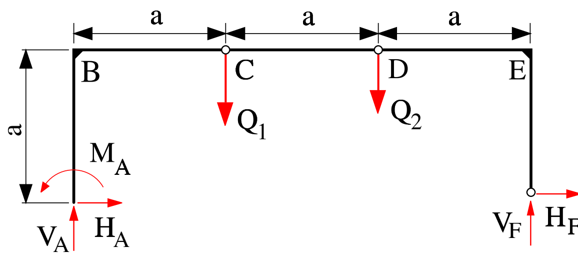
Am Punkt F ist das Tragwerk drehbar gelagert, am Punkt A ist es fest mit dem Untergrund verbunden.



Hierbei sind die Längen a und die Kräfte Q_1 und Q_2 dem Betrag nach gegeben.

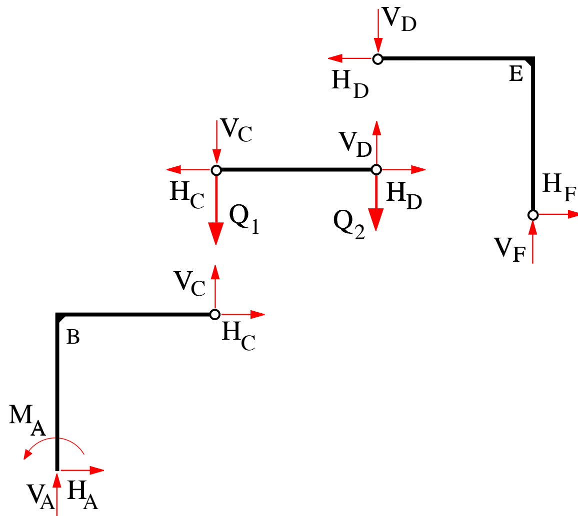
$$a = 3m, Q_1 = 1kN, Q_2 = 2kN$$

In einem ersten Schritt wird das Tragwerk von dem Fundament freigeschnitten und die entsprechenden Kräfte (und Drehmomente) werden notiert.



$$a = 3m, Q_1 = 1kN, Q_2 = 2kN$$

Im nächsten Schritt werden die einzelnen Elemente voneinander freigeschnitten und die Kräfte werden eingetragen.



Da das Tragwerk im Stillstand ist, müssen die Kräfte sich gegenseitig aufheben. Dies notieren wir in Form einer Reihe von Gleichungen.

$$
\begin{aligned}
H_A + H_C &= 0 \\
V_A + V_C &= 0 \\
M_A + a \cdot V_C - a \cdot H_C &= 0 \\
-H_C + H_D &= 0 \\
-V_C - Q_1 + V_D - Q_2 &= 0 \\
a \cdot V_D - a \cdot Q_2 &= 0 \\
-H_D + H_F &= 0 \\
-V_D + V_F &= 0 \\
a \cdot H_D + a \cdot V_D &= 0 \\
a = 3m, Q_1 = 1kN, Q_2 = 2kN &
\end{aligned}
$$

Wir setzen die bekannten Werte für a, Q_1 und Q_2 ein.

$$\begin{aligned} H_A + H_C &= 0 \\ V_A + V_C &= 0 \\ M_A + 3 \cdot V_C - 3 \cdot H_C &= 0 \\ -H_C + H_D &= 0 \\ -V_C - 1 + V_D - 2 &= 0 \\ 3 \cdot V_D - 3 \cdot 2 &= 0 \\ -H_D + H_F &= 0 \\ -V_D + V_F &= 0 \\ 3 \cdot H_D + 3 \cdot V_D &= 0 \end{aligned}$$

Das Gleichungssystem lässt sich als Produkt von Matrix und Vektor schreiben.

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 3 & 3 & 0 & 0 \end{bmatrix} \begin{bmatrix} H_A \\ V_A \\ M_A \\ H_C \\ V_C \\ H_D \\ V_D \\ H_F \\ V_F \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1+2 \\ 3 \cdot 2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Mit der Darstellung als Matrix haben wir alles, um das Problem von NumPy lösen zu lassen. D. h. wir formulieren die Matrix als NumPy-Array und lassen das System von NumPy über die `solve`-Funktion lösen.

```
import numpy as np
import numpy.linalg as lin

A = np.array(
    [[ 1, 0, 0, 1, 0, 0, 0, 0, 0 ],
     [ 0, 1, 0, 0, 1, 0, 0, 0, 0 ],
     [ 0, 0, 1, -3, 3, 0, 0, 0, 0 ],
     [ 0, 0, 0, -1, 0, 1, 0, 0, 0 ],
     [ 0, 0, 0, 0, -1, 0, 1, 0, 0 ],
     [ 0, 0, 0, 0, 0, 0, 3, 0, 0 ],
     [ 0, 0, 0, 0, 0, -1, 0, 1, 0 ],
     [ 0, 0, 0, 0, 0, 0, -1, 0, 1 ],
     [ 0, 0, 0, 0, 0, 3, 3, 0, 0 ]])
```

```
b = np.array([ 0, 0, 0, 0, 3, 6, 0, 0, 0])

x = lin.solve(A, b)

print(x) # => [ 2. 1. -3. -2. -1. -2. 2. -2. 2.]
```

Damit haben wir alle Kräfte aus dem Tragwerk berechnet.

- H_A = 2 kN
- V_A = 1 kN
- M_A = -3000 Nm
- H_C = -2 kN
- V_C = -1 kN
- H_D = -2 kN
- V_D = 2 kN
- H_F = -2 kN
- V_F = 2 kN

Kapitel 3

Matplotlib

3.1 Was ist Matplotlib? [42]

Viele Probleme lassen sich besser verstehen oder lösen, wenn man sie grafisch darstellt. Beispielsweise kann man bei einer linearen Regression gut sehen, ob die bestimmte Regressionsgrade wirklich mittig zwischen den Punkten verläuft, wenn man sie zeichnet. Genauso kann man die Verteilung von Werten grafisch sehr schnell überblicken.

Da die grafische Darstellung von Werten und Funktionen so wichtig ist, bietet Python mit *Matplotlib* eine Bibliothek an, mit der man sehr einfach und schnell Funktionen und Werte zeichnen kann.

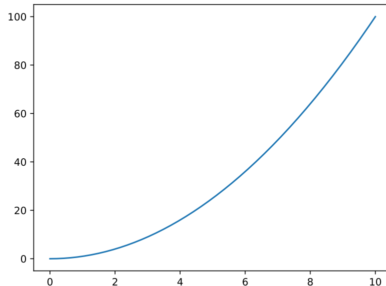
- Bibliothek zum Plotten von Graphen
- Harmoniert gut mit NumPy
- Syntax ähnlich zu Matlab

Matplotlib passt sehr gut zu NumPy, weil es direkt mit den NumPy-Arrays umgehen kann und deren Inhalt zeichnen. Bei der Syntax der Befehle lehnt sich Matplotlib an die entsprechenden Kommandos von Matlab an.

3.2 Beispiel [43]

Als Einstig in das Thema Matplotlib, soll eine Funktion gezeichnet werden. Hier einfach die Funktion $f(x) = x^2$.

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 1001) # Zwischen 0 und 10 1001 Werte erzeugen
y = np.power(x, 2) # Y-Werte berechnen
plt.plot(x, y)
plt.show()
```



Im Beispiel erzeugt der Befehl `np.linspace(0, 10, 1001)` ein Array mit 1001 Werten im Intervall 0 bis 10. Der Bereich wird also in Schritten a 0.01 durchlaufen. Dieses Array speichern wir in der Variable `x`.

Im nächsten Schritt wird ein zweites Array `y` berechnet, dass die Quadrate der Werte aus `x` enthält. Hier machen wir uns zu Nutze, dass NumPy die Operationen elementweise durchführt, wie also einfach jedes Element aus `x` quadriert in dem Array `y` wiederfinden.

Mit `plt.plot` weisen wir Matplotlib an, die beiden Array gegeneinander aufzutragen, wobei als erster Parameter die horizontale x-Achse und als zweiter Parameter die vertikale y-Achse angegeben wird.

Wir könnten jetzt noch weitere Graphen in das Diagramm zeichnen, indem wir weitere `plot`-Befehle absetzen. Hier soll aber die eine Kurve reichen.

Angezeigt wird der Plot durch den Befehl `show()`. Nach diesem geht ein neues Fenster auf, indem man die Kurve betrachten kann. Anstelle von `show` könnte man das Ergebnis auch mit `savefig()` auf der Festplatte speichern.

3.3 Schönere Graphen mit Seaborn [44]

Manchen erscheinen die Grafiken, die aus Matplotlib herausfallen als zu langweilig. In diesem Fall kann man mit dem Modul *Seaborn* stärkeren Einfluss auf die Ergebnisse und deren Aussehen nehmen.

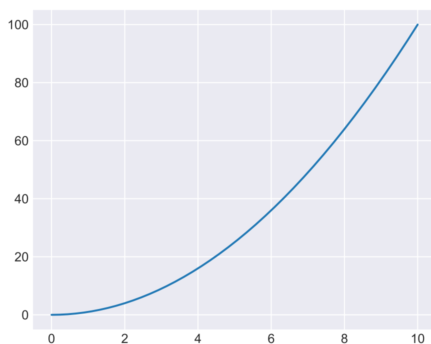
- Mit dem Modul *Seaborn* kann man die Graphen verschönern

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style("darkgrid") # Style auswählen
```

```
plt.tight_layout() # weniger Rand

x = np.linspace(0, 10, 1001) # Zwischen 0 und 10 1001 Werte erzeugen
y = np.power(x, 2) # Y-Werte berechnen

plt.plot(x, y)
plt.show()
```

Im Beispiel ist nur das Laden von Seaborn, das Auswählen eines Styles mit `set_style` und die Verkleinerung der Ränder mit `tight_layout()` hinzu gekommen.

Das Ergebnis sieht deutlich ansprechender aus, als ohne Seaborn.



3.4 Plot als Grafik speichern [46]

Den Plot anzuzeigen ist häufig der erste Schritt zur Analyse der Daten. Später möchte man die Grafik aber vielleicht in einer Hausarbeit verwenden oder weitergeben. Hierzu muss man nur das `show` durch das Kommando `savefig` ersetzen.

- Der Plot kann als Grafik über `pyplot.savefig(name)` gespeichert werden

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style("darkgrid") # Style auswählen

x = np.linspace(0, 10, 1000) # Zwischen 0 und 10 1000 Werte erzeugen
y = np.power(x, 2) # Y-Werte berechnen
```

```
plt.plot(x, y)

plt.savefig('square.pdf')
```

Die Art der Bilddatei bestimmt Matplotlib automatisch aus der Endung des angegebenen Dateinamens. Würde man im Beispiel oben `plt.savefig('square.png')` angeben, so würde der Plot als PNG-Bild abgelegt.

3.5 Achsenbeschriftung [47]

Beschriftung der Achsen

- `ax.set_xlabel(str)`: X-Achse beschriften
- `ax.set_ylabel(str)`: Y-Achse beschriften
- `ax.set_title(str)`: Titel des Diagramms

Wenn man die Achsen beschriften will, muss man sich von Matplotlib das Objekt holen, auf dem die Zeichenoperationen erfolgen. Dies geschieht über den Befehl `plt.subplots`. Dieser erlaubt es, mehrere Plots in einer Grafik zusammenzufassen und gibt dem Programmierer Zugriff auf die Objekte für die Achsen. Uns interessiert hier nur der zweite Teil, sodass wir die Funktion ohne Parameter aufrufen. Sie gibt zwei Werte zurück:

- das Objekt für die Figure (benötigen wir hier nicht) und
- das Objekt für die Achsen.

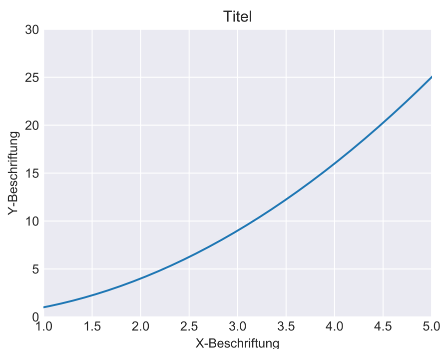
```
sns.set_style("darkgrid") # Style auswählen

figure, axis = plt.subplots()

x = np.linspace(0, 10, 1000)
y = np.power(x, 2)

axis.set_xlim((1, 5)) # X-Bereich
axis.set_ylim((0, 30)) # Y-Bereich

# Beschriftung der Achsen und Titel
axis.set_xlabel('X-Beschriftung')
axis.set_ylabel('Y-Beschriftung')
axis.set_title('Titel')
plt.tight_layout()

axis.plot(x, y)
plt.show()
```

Die Imports wurden ausgelassen

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Das Beispiel setzt die Wertebereiche für die Achsen mit `set_xlim` und `set_ylim` und fügt den Achsen mit `set_xlabel` bzw. `set_ylabel` Beschriftungen hinzu. Die gesamte Grafik bekommt noch einen Titel über die Methode `set_title`.

Wir müssen den Plot über `axis.plot` anstoßen und nicht `plt.plot`, weil Matplotlib sonst die Achsenbeschriftungen und Wertebereiche ignorieren würde und auf die Standardwerte zurückgehen.



Wir hätten uns den Weg über die Achsen-Objekte auch sparen können und stattdessen den Standardplot mit entsprechenden Beschriftungen versehen können, wie es im folgenden Beispiel gezeigt wird.

3.6 Mehrere Plots [50]

Man kann mit Matplotlib beliebig viele Kurven in eine Grafik zeichnen. Hierzu ruft man einfach mehrfach die `plot`-Funktion auf.

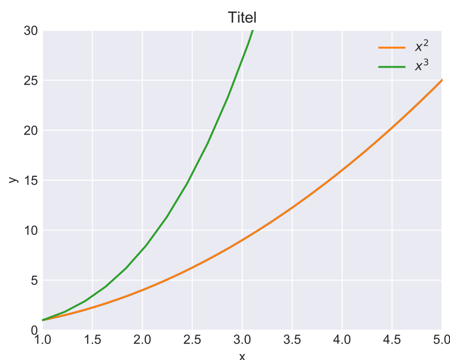
Matplotlib vergibt automatisch Farben für die verschiedenen Kurven.

```
x = np.linspace(0, 10, 50)
y1 = np.power(x, 2)
y2 = np.power(x, 3)

plt.plot(x, y1, label='$x^2$')
plt.plot(x, y2, label='$x^3$')
```

```
plt.xlim((1, 5))
plt.ylim((0, 30))
plt.tight_layout()

plt.xlabel('x')
plt.ylabel('y')
plt.title('Titel')
plt.legend()
plt.show()
```

Im Beispiel wird den Kurven eine Beschriftung mitgegebene, die als `label`-Parameter in der `plot`-Funktion angegeben wird. Das `$` zeigt an, dass das Format des Labels der LaTeX-Notation folgt.

Über `xlim` und `ylim` wid der Wertebereich eingegrenzt und die Achsen werden mit `xlabel` und `ylabel` beschriftet. Die Grafik erhält ihren Titel über `title`. Der Befehl `legend()` sorgt dafür, dass die Labels der Kurven angezeigt werden.



3.7 Histogramm [52]

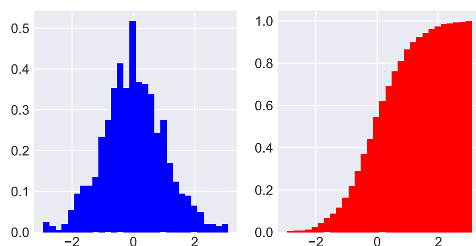
Das folgende Beispiel zeichnet ein Histogramm, also eine Kurve, welche die Verteilung von Werten anzeigt. Da wir zwei Histogramme in einer Grafik zeigen vollen, legen wir mit `plt.subplots(1, 2)` zwei Plots nebeneinander an.

Die darzustellenden Daten werden, der Einfachheit halber, als Zufallszahlen erzeugt.

Die Funktion `hist` sorgt dafür, dass die Daten als Histogramm dargestellt werden, hierbei legen wir über

- `bins` die Anzahl der Klassen
- `color` die Farbe und
- `cumulative` fest, ob die Daten aufsummiert werden sollen.

```
# Zufallsdaten erzeugen
data = np.random.randn(1000)

f, (axis1, axis2) = plt.subplots(1, 2)

# Histogram
axis1.hist(data, bins=30, color='b')

# Histogramm, kumuliert
axis2.hist(data, bins=30, color='r', cumulative=True)

plt.show()
```



3.8 Box-Plots [54]

Eine weitere wichtige Darstellung sind die *Boxplots* auch *Box-Whisker-Plot* genannt. Es sind Diagramme, zur grafischen Darstellung der Verteilung eines mindestens ordinalskalierten Merkmals.

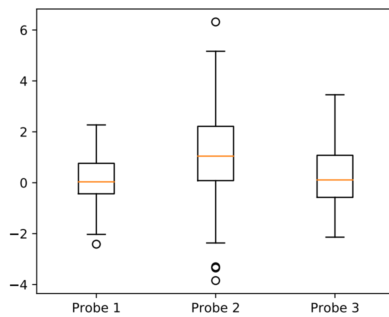
Ein Box-Blot fasst dabei verschiedene robuste Streuungs- und Lagemaße in einer Darstellung zusammen. Ein Box-Plot soll schnell einen Eindruck darüber vermitteln, in welchem Bereich die Daten liegen und wie sie sich über diesen Bereich verteilen. Deshalb werden alle Werte der sogenannten Fünf-Punkte-Zusammenfassung, also der

- Median,
- die zwei Quartile und
- die beiden Extremwerte,

dargestellt.

```
samp1 = np.random.normal(loc=0., scale=1., size=100)
samp2 = np.random.normal(loc=1., scale=2., size=100)
samp3 = np.random.normal(loc=0.3, scale=1.2, size=100)
```

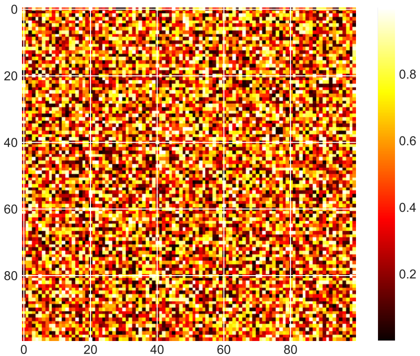
```
f, ax = plt.subplots(1, 1, figsize=(5,4))

ax.boxplot((samp1, samp2, samp3))
ax.set_xticklabels([ 'Probe 1', 'Probe 2', 'Probe 3' ])

plt.show()
```



3.9 Heatmaps [56]

Der letzte Diagrammtyp, den wir hier behandeln wollen, ist die *Heatmap*. Bei dieser werden Werte auf einer zweidimensionalen Fläche angeordnet und entsprechend ihres Wertes eingefärbt.

```
a = np.random.random((100, 100))

plt.imshow(a)
plt.hot()
plt.colorbar()

plt.show()
```


0
20
40
60
80
0
20
40
60
80
0.8
0.6
0.4
0.2

Index

Array broadcasting, 7

Beschriftung der Achsen, 25
Box-Whisker-Plot, 28
Boxplots, 28
Broadcasting, 7

Dimensionen des Arrays, 3

Heatmap, 29

Identitätsmatrix, 15
iterieren, 10

Kumulative Summe, 15

lineare Regression, 11

Matplotlib, 22
Matrix-Multiplikation, 14
Matrizen, 4
Maximum, 15
Minimum, 15

ndarray, 2
Null-Matrix, 4
NumPy, 1

Regressionen höherer Ordnung, 12

Seaborn, 23
Slicen, 9
Spur, 15
Summieren, 15

Transponieren, 14
transponierte Form, 5

Vektor-Operatonen, 9