

## Einführung in Python

Monty die Kleine Schlange  
ein Bilderbuch von Dungeon Keeper

## Table of Contents

1	Python.....	6
1.1	Funktionsweise.....	6
1.2	Python Bytecode.....	7
1.3	Erster Schritt im Python.....	8
1.4	Python Programme ausführen.....	9
1.4.1	Kommandozeile.....	10
1.4.2	Replit.....	11
1.4.3	Online GDB.....	12
1.4.4	Jupyter Notebook.....	13
1.4.5	Visual Studio Code.....	13
2	Programmieren in Python.....	16
2.1	Daten auf der Konsole ausgeben.....	16
2.1.1	Separatoren.....	17
2.1.2	Ausgabe ohne Zeilenumbruch.....	17
2.1.3	Ausgabe umlenken.....	17
2.2	Datentypen.....	18
2.2.1	Objekttypen in Python.....	18
2.2.2	Attribute eines Objekts.....	18
2.2.3	Verwaltung von Variablen in Python.....	19
2.2.4	int, float, boolean, complex.....	20
2.2.5	Wie groß ist ein Int ?.....	21
2.2.6	Größe eines Objektes im Speicher ermitteln.....	22
2.2.7	Konvertierung von Datentypen.....	23
2.2.8	Booleans.....	23
2.2.8.1	Vergleichsoperatoren Auswerten.....	23
2.2.8.2	Numerische Ausdrücke Auswerten.....	24
2.2.8.3	Zeichenketten auswerten.....	24
2.2.8.4	Boolesche Ausdrücke verrechnen.....	24
2.2.8.5	Boolesche Ausdrücke & boolesche Algebra.....	25
2.2.9	Zeichenketten.....	26
2.2.9.1	Zeichenketten definieren.....	26
2.2.9.2	Zeichenketten mit mehreren Zeilen.....	27
2.2.9.3	Zeichenketten <i>als Felder</i> .....	28
2.2.9.4	Zeichenketten iterieren.....	29
2.2.9.5	Zeichenketten zusammensetzen.....	29
2.2.9.6	Zeichenketten zusammensetzen mit join().....	30
2.3	Collectibles.....	31
2.3.1.1	Größe einer <i>Kollektion</i> bestimmen.....	31
2.3.1.2	<i>Kollektionen</i> umwandeln.....	31
2.3.1.3	Elemente einer <i>Kollektionen</i> iterieren.....	32
2.3.1.4	Zugehörigkeit <i>prüfen</i> .....	32
2.3.1.5	Zugriffe über Indexe.....	32
2.3.1.6	Negative Indexierung.....	33
2.3.1.7	<i>Kollektion</i> vereinigen.....	33
2.3.1.8	<i>Kollektion</i> löschen.....	34

2.3.2	Kollektionen sortieren.....	34
2.3.3	Reihenfolge der Elemente mit reversed() umdrehen.....	35
2.3.4	Sum().....	35
2.3.5	min() und max().....	36
2.3.6	all() und any().....	36
2.3.7	Sets.....	37
2.3.7.1	Sets Definieren.....	37
2.3.7.2	Elemente hinzufügen.....	37
2.3.8	Einzelne Elemente entfernen.....	38
2.3.9	Letztes Element entfernen.....	38
2.3.9.1	Alle Elemente eines Sets <i>entfernen</i> .....	38
2.3.9.2	Schnittmenge zweier Sets.....	39
2.3.9.3	Differenz zweier Sets Bilden.....	39
2.3.9.4	Symmetrische Differenz zweier Sets Bilden.....	40
2.3.9.5	Auf Untermenge prüfen.....	40
2.3.9.6	Auf Obermenge prüfen.....	41
2.3.9.7	Prüfung ob zwei Sets disjunkt sind.....	41
2.3.9.8	Sets Kopieren.....	42
2.3.9.9	Sets vereinigen.....	42
2.3.10	Tupel.....	43
2.3.10.1	Anzahl der Vorkommnisse eines Elements ermitteln.....	43
2.3.10.2	Index eines Elements im Tupel <i>ermitteln</i> .....	43
2.3.10.3	Tupel bearbeiten.....	44
2.3.11	Listen.....	45
2.3.11.1	Länge einer Liste ermitteln.....	45
2.3.11.2	Häufigkeit eines Elements zählen.....	45
2.3.11.3	Erstes Vorkommen in Liste ermitteln.....	46
2.3.11.4	Reihenfolge in der Liste umkehren.....	46
2.3.11.5	Liste sortieren.....	47
2.3.11.6	Liste leeren.....	47
2.3.11.7	Elemente hinzufügen.....	48
2.3.11.8	Elemente entfernen.....	49
2.3.11.9	Liste kopieren.....	49
2.3.12	Dictionaries.....	50
2.3.12.1	Dictionaries definieren.....	50
2.3.12.2	Werte auslesen.....	51
2.3.12.3	Wert eines Schlüssels ändern.....	51
2.3.12.4	Dictionary leeren.....	52
2.3.12.5	Dictionary kopieren.....	52
2.3.12.6	Auflistung aller Schlüssel <i>erstellen</i> .....	53
2.3.12.7	Auflistung aller Werte <i>erstellen</i> .....	53
2.3.13	Auflistung alle Wertepaare erstellen.....	54
2.3.13.1	Views dict_keys, dict_values und dict_items.....	54
2.3.13.2	Wert Auslesen.....	54
2.3.13.3	Wert holen.....	54
2.3.13.4	Letztes Wertepaar holen.....	55
2.3.13.5	Default Wert setzen, Wert auslesen.....	55
2.4	Views.....	56

2.5	IF...elif...else.....	56
2.5.1	Die if-Anweisung.....	56
2.5.2	Else.....	57
2.5.3	Die elif-Anweisung.....	57
2.5.4	Verschachtelung.....	57
2.6	Schleifen.....	58
2.6.1	Die For-Schleife.....	58
2.6.1.1	Else.....	58
2.6.1.2	Break.....	58
2.6.1.3	Continue.....	59
2.6.1.4	Pass.....	59
2.6.1.5	Range.....	59
2.6.1.6	Bereiche durchgehen oder for (i = 0; i<x; i+=y) in Python.....	60
2.6.1.7	Strings Iterieren.....	61
2.6.2	while Schleife.....	62
2.6.2.1	else.....	63
2.6.2.2	While...break.....	63
2.6.2.3	While...continue.....	64
2.6.2.4	While...pass.....	64
2.7	Funktionen.....	65
2.7.1	Funktionen dokumentieren.....	65
2.7.2	Parameter einer Funktion.....	65
2.7.2.1	Immutable Parameter.....	66
2.7.2.2	Mutable Parameter.....	67
2.7.3	Parameter mit optionaler Vorgabe.....	68
2.7.4	Funktionen mit optionalen Argumenten.....	69
2.7.5	Funktionen mit Schlüsselwörtern.....	69
2.7.6	Funktionen mit Schlüsselwörtern und variablen Parametern.....	69
2.7.7	Rückgabeparameter.....	70
2.7.8	Funktionen mit mehreren Rückgabeparametern.....	70
2.7.9	Templates und generische Funktionen.....	71
2.7.10	Funktionen mit Funktionen als Parameter.....	72
2.7.11	Funktionen Kopieren, löschen und erweitern.....	73
2.8	Iteratoren.....	74
2.8.1	iter().....	75
2.8.2	next().....	75
2.8.3	Iteratoren mit for auslesen.....	76
2.8.4	zip.....	76
2.8.5	Enumerator.....	77
2.8.6	map.....	77
2.8.7	Maps direkt in Tupel Umwandeln.....	78
2.8.8	Yield.....	79
2.9	Lambdas.....	80
2.9.1	Direkte Lambdas.....	80
2.9.2	Lambdas mit mehreren Parametern.....	80
2.9.3	Lambdas mit Auswertungslogik.....	81
2.10	Filter.....	81
2.11	List Comprehension.....	82

2.12	Objektorientierte Programmierung in Python.....	83
2.12.1	Klassen in Python.....	85
2.12.2	Öffentliche, geschützte und private Variablen.....	85
2.12.3	Statische Variablen.....	87
2.12.4	Methoden.....	88
2.12.4.1	Klassen um Methoden erweitern.....	90
2.12.4.2	Private und Geschützte Methoden.....	90
2.12.5	Klassen Ableiten.....	91
2.12.6	Der Nullpointer.....	92
2.12.7	Klassentypen vergleichen.....	93
2.13	Ausnahmen (Exceptions).....	94
2.13.1	Ausnahmen vs. dedizierte Fehlerabfrage.....	95
2.13.2	Ausnahmen Auslösen.....	96
2.13.3	Ausnahmen abfangen.....	97
2.13.4	Eigene Exceptionen definieren.....	97
2.13.5	Ende mit schrecken, oder die finally Anweisung.....	98
2.13.6	Ausnahmen über mehrere Ebenen.....	98
2.14	Context Manager.....	100
2.14.1	Anweisung With.....	100
2.14.2	Context Manager.....	101
2.14.2.1	Die Klasse Zeichenkette.....	101
2.14.2.2	Der passende Context Manager.....	102
2.14.2.3	Das Zusammenspiel.....	103
3	Numpy.....	105
3.1	Datentypen.....	105
3.1.1	Ganzzahlige Datentypen.....	105
3.1.2	Fließkommazahlen.....	106
3.1.3	Komplexe Zahlen.....	107
3.2	Arange.....	107
4	MathPlotLib.....	109
	Farben.....	109
	Folgende Grundfarbtöne sind im plotlib als Einzeichenparameter vordefiniert:.....	109
	‘b’ - blau.....	109
	‘g’ – blau.....	109
	‘r’ – rot.....	109
	‘c’ – cyan.....	109
	‘m’ – magenta.....	109
	‘y’ – gelb.....	109
	‘k’ – schwarz.....	109
	‘w’ – weis.....	109
	Außer den Grundfarbtönen lassen sich auch weitere Farben verwenden.....	109
	//ToDo wie ?.....	109
	4.1.1.1 Punkte Zeichnen.....	110
	4.1.1.2 Punktgröße ändern.....	111
	4.1.2 Linien Zeichnen.....	111
	4.1.2.1 Linienbreite.....	112
	4.1.3 Boxplot.....	113
4.2	Figure.....	113

# Vorwort

## 1 Python

Wozu brauche ich eine weitere Programmiersprache? Ich kann doch alles mit <Lieblingsprogrammiersprache einfügen> erledigen. Das stimmt schon, die Frage ist nur wie effizient dies mit der Lieblingsprogrammiersprache erfolgen kann. Ist es möglich bestimmte Aufgaben mit den „on Board“ mitteln zu erledigen oder braucht man dazu eine oder mehrere Bibliotheken, die vielleicht auch noch nicht zueinander passen? Wie sieht es mit der Effizienz aus? Kann ich ein Problem mit der Sprache A in einer oder wenigen Zeilen lösen oder muss ich dazu in der Sprache B 1000 Codezeilen schreiben – testen – debuggen?

Man kann auch mit einem Hammer Schrauben in die Wand „integrieren“. Die mit dem Hammer wieder herausnehmen wird schon schwieriger, aber nicht unmöglich, man kann ja die Wand um die Schraube mit dem Hammer weg klopfen.

### 1.1 Funktionsweise

Python ist ein sogenannter **byte code Interpreter**. Das zu einem Bytecode übersetztes Programm wird anschließend in einer virtuellen Maschine ausgeführt. Dabei wird die Quelltextdatei erst vor dem Ausführen automatisch in den entsprechenden Binärcode umgesetzt. Die meisten Bibliotheken sind bereits vorkompiliert und liegen bereits im Binärcode vor (Abbildung 1). Die Python VM kommuniziert mittels eines Laufzeitsystems, dessen Teile wiederum in Python Bibliotheken implementiert sind, mit der Außenwelt. Bei Bedarf kann Python aber auch mit Codeteilen (Bibliotheken) die in anderen Programmiersprachen implementiert wurden, interagieren, was aber in diesem Tutorial nicht behandelt wird weil es normalerweise auch nicht notwendig ist.

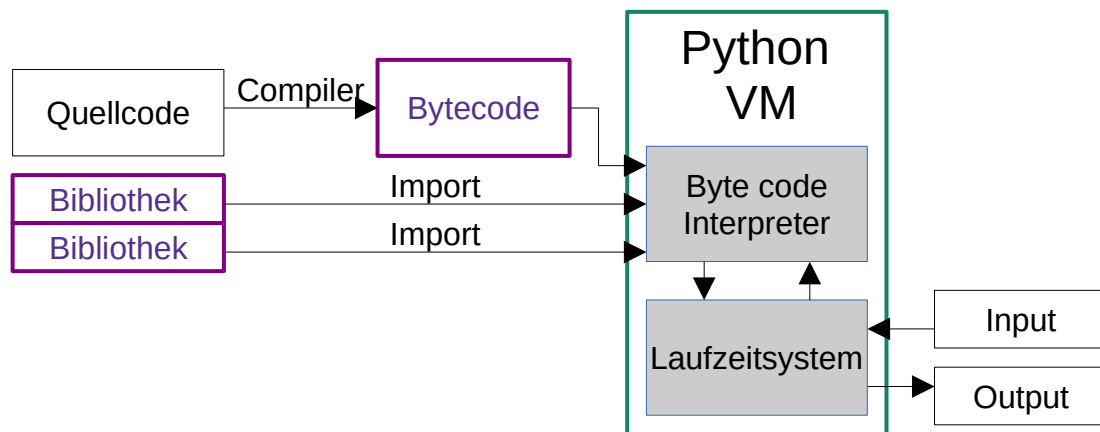


Abbildung 1: Funktionsweise von Python

## 1.2 Python Bytecode

Nur um die Neugierde einiger Leser zu decken schauen wir uns mal kurz dem Bytecode an, oder besser gesagt die Möglichkeit sich diesen anzuzeigen. Dafür ist zuerst ein Paket namens **dis** notwendig, dass mittels **pip** installiert werden kann (`python -m pip install dis`). Der Aufruf von der Methode `dis`, mit der Angabe des Objekts, dessen Bytecode angezeigt werden sollte, listet den Bytecode dieses Objektes auf (Abbildung 2). In der linken Spalte wird die entsprechende Codezeile angezeigt, gefolgt von dem Byte Offset, hier nimmt wohl jede Anweisung 2 Byte Platz. Danach folgt der Name der Anweisung selbst und ganz rechts die von dieser Anweisung verwendeten Objekte.

```
def function(value):
    value = value + 4
    print("value = ",value)
    return value
```

```
import dis
dis.dis(function)

function(2)
```

2	0 LOAD_FAST	0 (value)
	2 LOAD_CONST	1 (4)
	4 BINARY_ADD	
	6 STORE_FAST	0 (value)
3	8 LOAD_GLOBAL	0 (print)
	10 LOAD_CONST	2 ('value = ')
	12 LOAD_FAST	0 (value)
	14 CALL_FUNCTION	2
	16 POP_TOP	
4	18 LOAD_FAST	0 (value)
	20 RETURN_VALUE	

value = 6

6

Abbildung 2: Python Funktion als Bytecode

## 1.3 Erster Schritt im Python

Wie es sich gehört, fangen wir mal mit dem einfachsten Programm der (unter-) Welt in Python an (Abbildung 3). So weit so gut, bis auf dem fehlenden **main**, Klammern und Semikolon sieht es eigentlich fast wie ein C/Java Programm, der die gleiche Aufgabe löst, aus. Die Gründe für das fehlende Zubehör? - Python wird oft und gerne in einer

Kommandozeile ausgeführt. (Abbildung 4) Das Ergebnis der Eingabe wird dann bei einfachen Anweisungen zeilenweise, oder bei den Kontrollstrukturen blockweise (beachte die „...“ nach der **if** Anweisung) ausgegeben. Somit lassen sich bei Python schnell kleine Algorithmen ausprobieren was in Verbindung mit den mächtigen Bibliotheksfunktionen aus dem Bereich Data Science, Statistik und ML es für diese Zielgruppe als einen perfekten und einfach zu bedienenden „Taschenrechner“ ergibt. Bei einer solchen Bedienweise wären Klammern eher störend, stattdessen verlässt man sich hier auf die Formatierung der Zeilen. So sind hier die Befehle, die der vorangegangenen Anweisung untergeordnet sind, im Quelltext nach rechts verschoben (beachte die zweite und die dritte **print** Anweisung). Der positive Aspekt ist – der Coder kann nicht mehr einfach

My python code nr 1

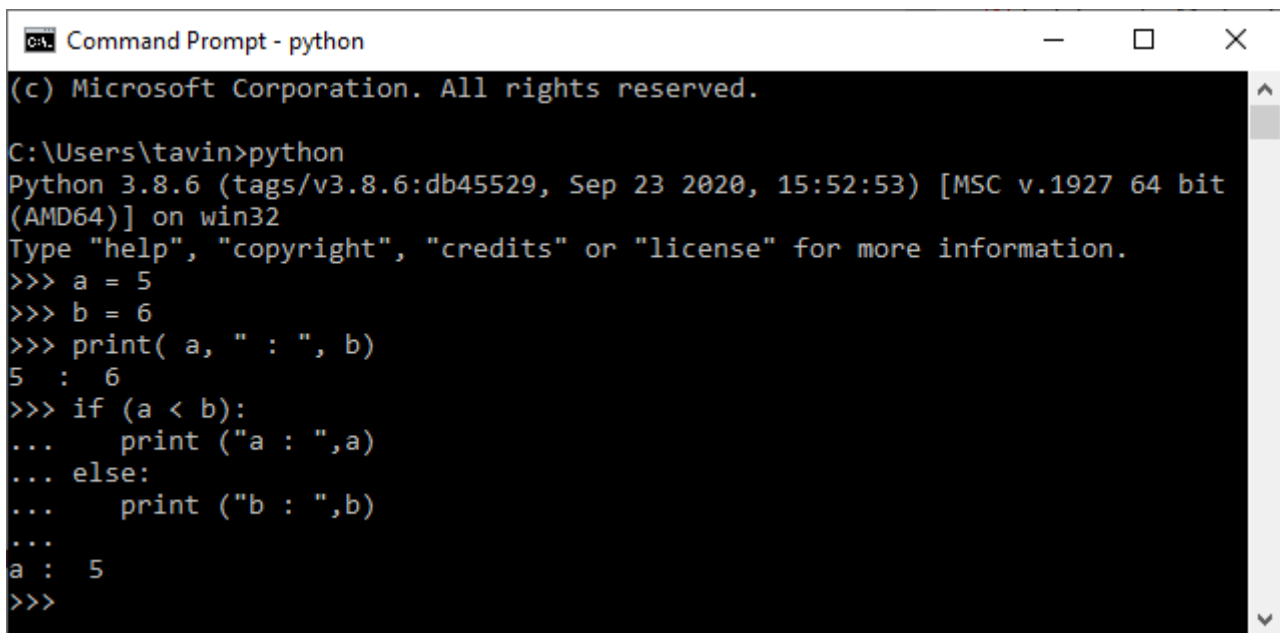
```
In [1]: print ("hell o underworld")

        hell o underworld
```

Abbildung 3: Erster Schritt in Python



das Program unlesbar und ohne jegliche Struktur „runterwürgen“, da der Compiler es nicht mehr akzeptiert (Danke Compiler!). Allerdings ist dafür ein anderes Problem latent – die Verwendung von Tabulatoren kann zu bösen Überraschungen führen da diese je nach Editor unterschiedlich „interpretiert“ und in unterschiedlich große sichtbare Abstände auf dem Bildschirm umwandeln. So ist eine Mischung aus Tabulatoren und echten „Leerzeichen“ in Python Quelltexten nicht empfehlenswert, da dies seitens des Compilers zu Fehlinterpretation der Hierarchie der Anweisungen führen kann.



```
Command Prompt - python
(c) Microsoft Corporation. All rights reserved.

C:\Users\tavin>python
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> b = 6
>>> print( a, " : ", b)
5 : 6
>>> if (a < b):
...     print ("a : ",a)
... else:
...     print ("b : ",b)
...
a : 5
>>>
```

Abbildung 4 Python in der Kommandozeile

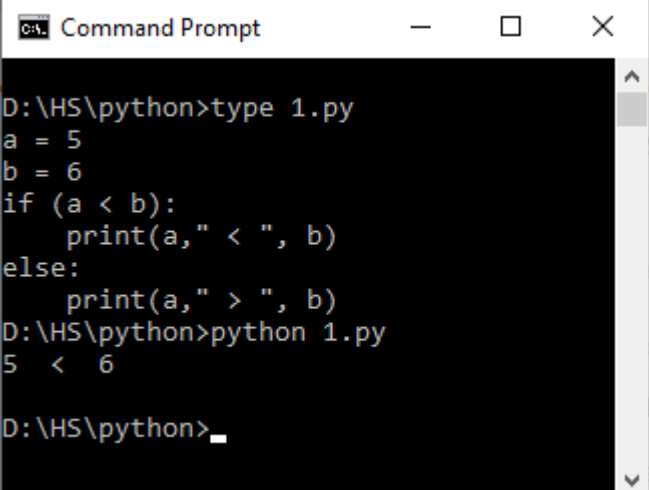
## 1.4 Python Programme ausführen

Es gibt inzwischen vielfältige Möglichkeiten Python-Programme auszuführen. Welche dieser Werkzeuge man persönlich nutzt hängt stark von persönlichen Vorlieben und dem bevorzugtem Workflow ab. Schauen wir uns mal kurz ein Paar der Möglichkeiten an. Außer den hier vorgestellten Werkzeugen gibt es inzwischen eine Fülle weiterer Möglichkeiten Python zu verwenden. Diese sind nicht unbedingt schlechter als die hier vorgestellten, so sind die unten angeführten Werkzeuge nur als eine kleine Typenübersicht zu verstehen.

### 1.4.1 Kommandozeile

Die ursprüngliche Möglichkeit Python auszuführen war der Kommandozeilenbefehl „python“:

Damit lassen sich einerseits vorgefertigte Programme aus Quelltextdateien (Abbildung 5) wie auch, in dem interaktivem Modus, einzelne Python Befehle ausführen (Abbildung 6). Für die Ausführung einer Quelltextdatei muss deren Name beim Aufruf des „python“ Befehls als Parameter angegeben werden. Wird keine Datei als Parameter übergeben, so startet die Anwendung in den interaktiven Modus, sodass jetzt die einzelnen Codezeilen eingegeben werden können. Die eingegebenen Zeilen werden dann automatisch ausgeführt, ausgenommen, es handelt es sich um Funktionen oder Teile einer Kontrollstruktur wie **if..else**, **for** usw. Diese werden erst dann ausgeführt, wenn die Kontrollstruktur komplett eingegeben wurde (dies erkennt der Compiler durch die Formatierung des Quelltextes!) oder im Falle einer Funktion diese aufgerufen wird.

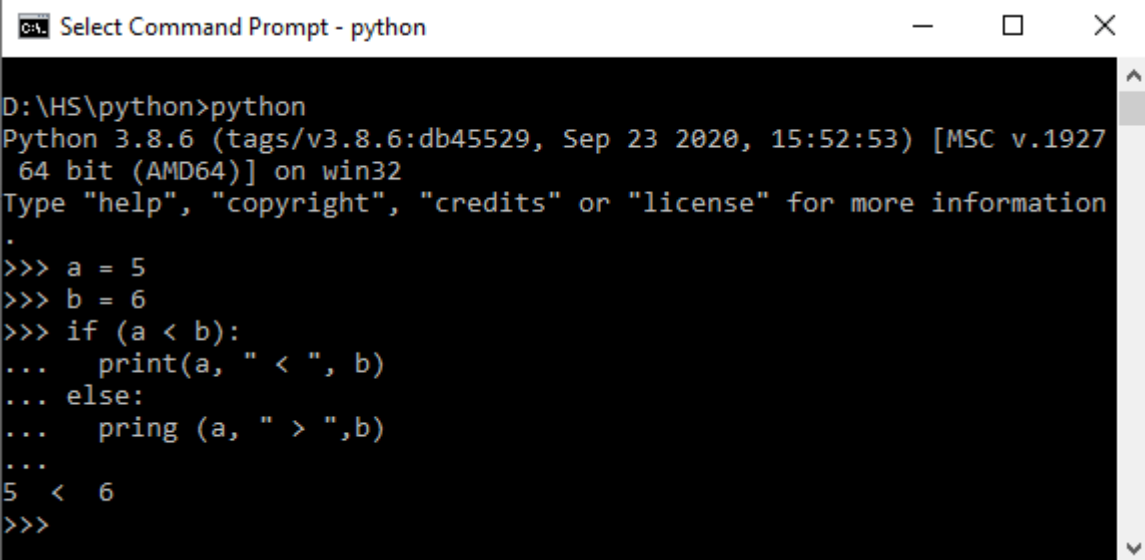


```

D:\HS\python>type 1.py
a = 5
b = 6
if (a < b):
    print(a, " < ", b)
else:
    print(a, " > ", b)
D:\HS\python>python 1.py
5 < 6
D:\HS\python>

```

Abbildung 5 Python-Quelltextdatei ausführen



```

D:\HS\python>python
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information
.
>>> a = 5
>>> b = 6
>>> if (a < b):
...     print(a, " < ", b)
... else:
...     print (a, " > ",b)
...
5 < 6
>>>

```

Abbildung 6 Python-Kommandozeilenanweisung im interaktiven Modus

Die Konsolenanwendung muss allerdings erst auf dem Rechner installiert werden. Im Gegensatz zu den Online Tools können hier aber dafür beliebige externe Bibliotheken in eigene Projekte eingebunden werden.

## 1.4.2 Replit

Als wohl beliebtestes Online-Tool für Python gilt der Replit. Als Online-Tool muss nichts auf dem lokalen Rechner installiert werden. Es kann sowohl in Dateimodus (Abbildung 7) sowie in dem interaktiven Modus (Abbildung 8) betrieben werden. Zu beachten ist, dass im Gegensatz zu dem interaktiven Modus Ergebnisse einfacher Rechnungen (2+4) nicht auf der Konsole ausgegeben werden.

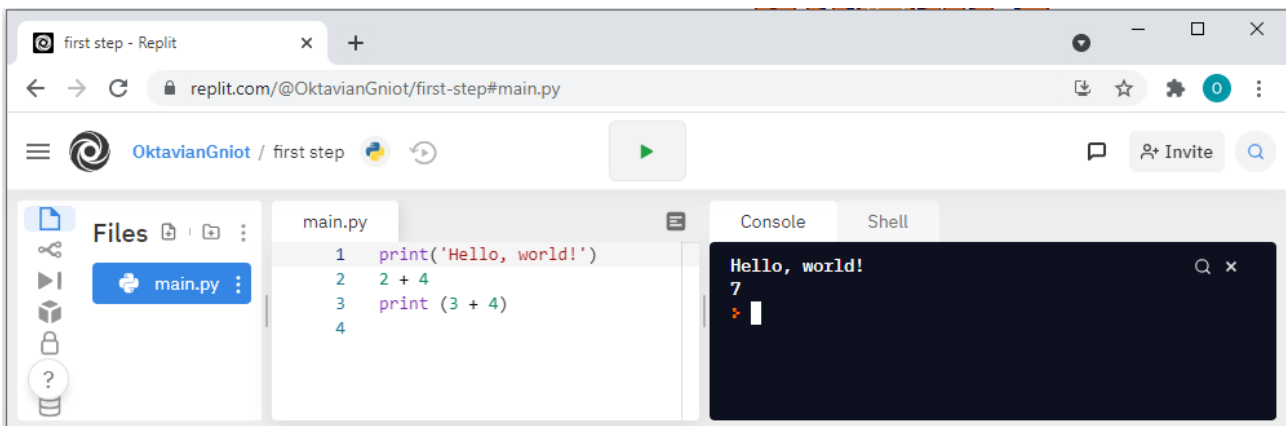


Abbildung 7 Replit in Dateimodus

Für die interaktive Verwendung können (bzw. müssen) die entsprechenden Codezeilen direkt in die Konsole eingegeben werden (Abbildung 8). In diesem Modus werden auch Ergebnisse einfacher Rechnungen direkt auf der Konsole ausgegeben.

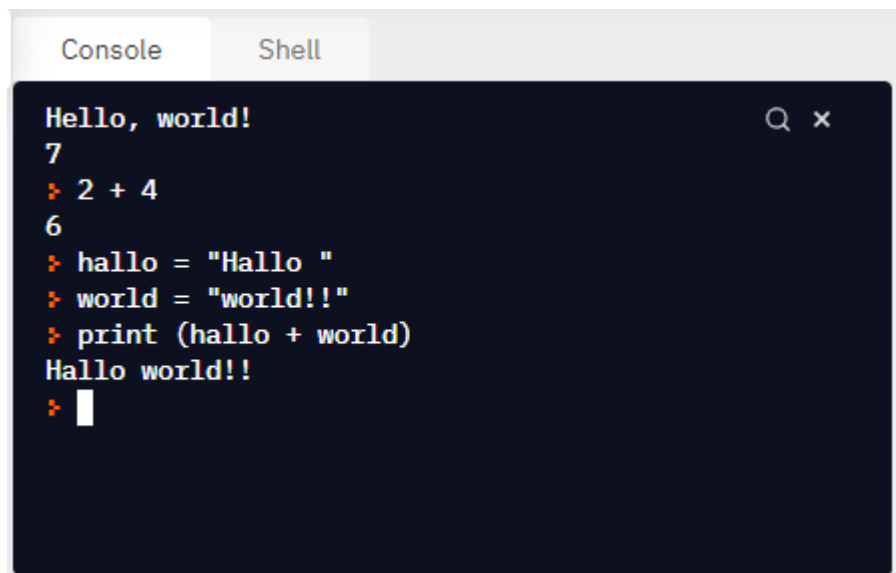


Abbildung 8: Replit im interaktiven Modus

### 1.4.3 Online GDB

Auch Online GDB lässt sich dazu überreden, Python-Programme auszuführen. Allerdings kennt Online GDB bis jetzt nur den Dateimodus. Interaktive Eingaben sind im Online GDB nicht möglich.

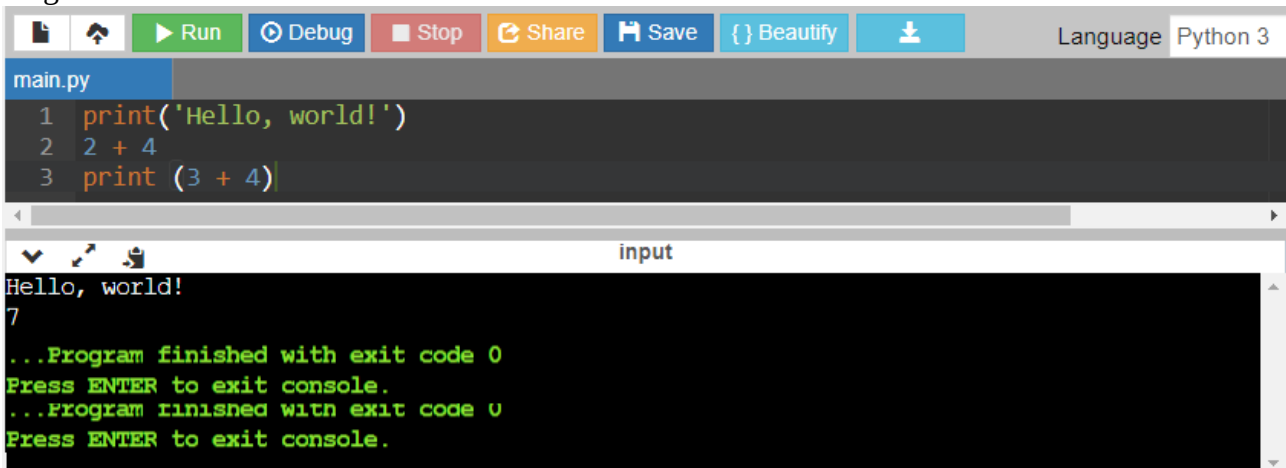


Abbildung 9 Python programmieren mit Online GDB

Dafür lassen sich in Online GDB Python-Anwendungen mit dem integrierten Debugger auf Fehler untersuchen (Abbildung 10).

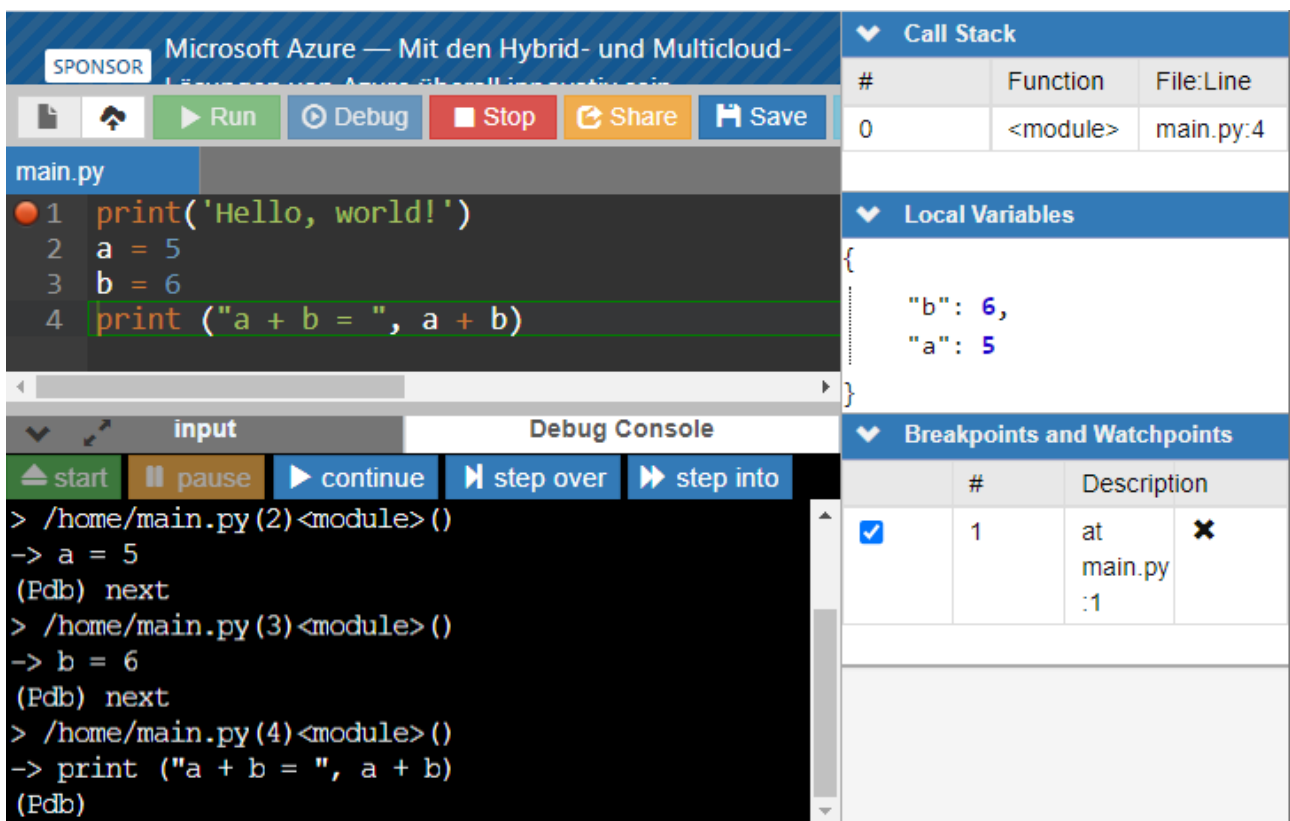


Abbildung 10 Python-Anwendungen debuggen mit online GDB

### 1.4.4 Jupyter Notebook

Ein anderes Werkzeug, das gerne für Python Programme verwendet wird, ist Jupyter Notebook (Abbildung 11). Es wird zwar als Applikation in einem Webbrowser ausgeführt, benötigt aber eine Installation auf dem lokalen Rechner. Es ist ein Mix aus Dateien und interaktiven Modus. Der Quelltext wird in einer Art Zellen eingegeben die dann einzeln oder alle nacheinander ausgeführt werden können. Diese Zellen sind zu jeder Zeit manipulierbar bzw. veränderbar und können zu jeder Zeit ausgeführt werden. Die Reihenfolge der stattgefundenen Ausführung wird über die Bezeichner **In[]** sowie **Out[]** angegeben. Beachte das die Ergebnisse von Einzelrechnungen (2+4) nur dann ausgegeben werden, wenn sich diese alleine oder als Letzte in einer Zeile befinden. Die Ausgabe erfolgt in diesem Fall in der Ausgabezeile Out[. Der Grund dafür ist, das das Ergebnis der letzten Zeile als das Gesamtergebnis der Zelle angesehen wird. Diese Ausgabe lässt sich durch ein abschließendes Semikolon unterdrücken (Abbildung 12).

```
In [1]: print('Hello, world!')
2 + 4
print (3 + 4)

Hello, world!
7
```

```
In [2]: print('Hello, world!')

Hello, world!
```

```
In [3]: 2 + 4
```

```
Out[3]: 6
```

```
In [4]: print (3 + 4)

7
```

Abbildung 11 Python in Jupyter

```
In [8]: 2 + 4;
```

```
In [9]: 2 + 4
```

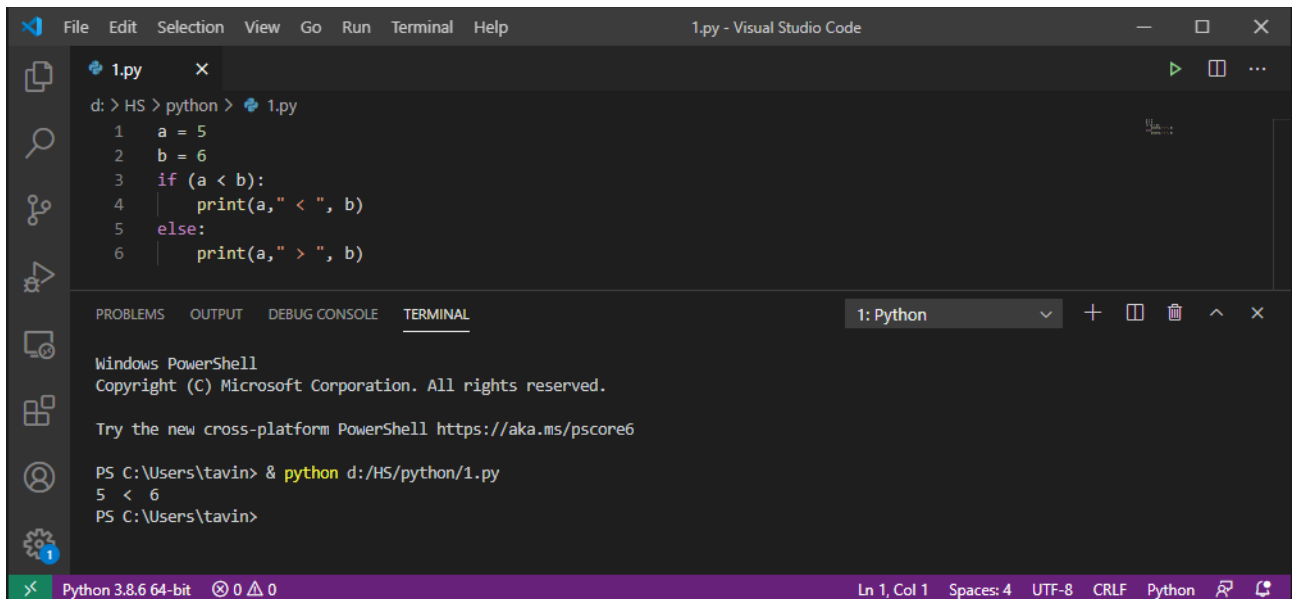
```
Out[9]: 6
```

Abbildung 12  
Jupyter Ausgabe  
unterdrücken

### 1.4.5 Visual Studio Code

Auch Visual Studio Code kann zur Programmierung in Python verwendet werden. Auch hier ist es möglich Quelltextdateien auszuführen oder interaktiv zu arbeiten. Über dem „get Started“ Karteireiter kann der entsprechende Modus ausgewählt werden (Abbildung 13).

## Einführung in Python



The screenshot shows the Visual Studio Code interface. The editor window displays a file named `1.py` with the following Python code:

```
d: > HS > python > 1.py
1 a = 5
2 b = 6
3 if (a < b):
4     print(a, " < ", b)
5 else:
6     print(a, " > ", b)
```

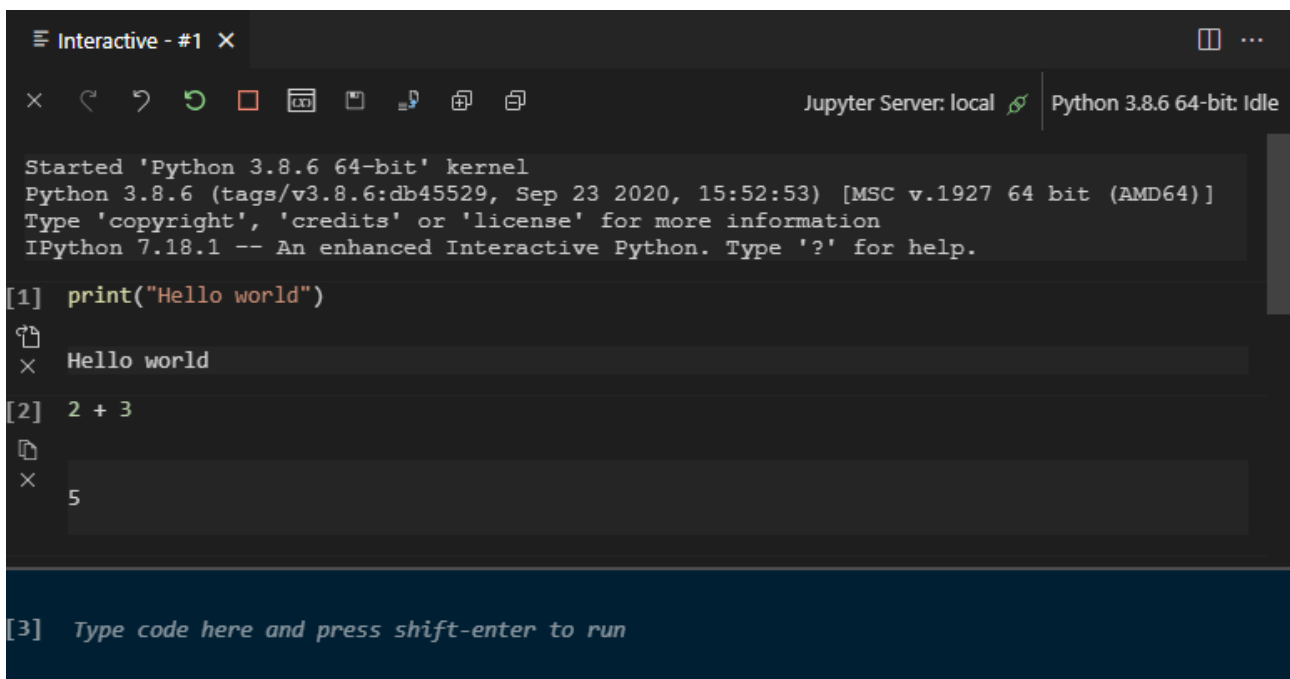
The bottom panel shows the `TERMINAL` tab with the following output:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\tavin> & python d:/HS/python/1.py
5 < 6
PS C:\Users\tavin>
```

The status bar at the bottom indicates `Python 3.8.6 64-bit` and shows the current cursor position as `Ln 1, Col 1`.



The screenshot shows the Jupyter Notebook interface within Visual Studio Code. The window title is `Interactive - #1`. The top bar indicates `Jupyter Server: local` and `Python 3.8.6 64-bit: Idle`. The notebook content shows the following:

```
Started 'Python 3.8.6 64-bit' kernel
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.18.1 -- An enhanced Interactive Python. Type '?' for help.
```

The first cell contains the code `print("Hello world")` and the output `Hello world`. The second cell contains the code `2 + 3` and the output `5`. The third cell is empty and contains the prompt `Type code here and press shift-enter to run`.

Abbildung 14 VS Code Python interactive window

Bei näherem Hinsehen stellt man fest, dass es sich hier dabei eigentlich um ein Jupyter Notebook handelt, das hier einem als die „interaktive“ Eingabe verkauft wird. Der Positive Aspekt davon ist, dass auch Jupyter Notebooks in VS Code verwendet werden können.

## Einführung in Python

Hinweis. Bei arbeiten mit VS Code & Python kommt es öfters dazu, das VS Code in den Dateinamen Steuerzeichen (er-),„findet“ sodass diese Dateien nicht ohne weiteres geladen werden können. Dies erkennt man daran, dass Teile des Pfades auf einmal in einer anderen Farbe angezeigt werden

## 2 Programmieren in Python

### 2.1 Daten auf der Konsole ausgeben.

Die Ausgabe der Daten auf der Konsole erfolgt über dem allseits bekannten **print()** Befehl. Diese Funktion gibt alles, was sich zwischen den Klammern befindet, auf der Konsole aus. Die einzige Einschränkung ist, dass die als Parameter angegebenen Ausdrücke als eine

Zeichenkette interpretiert bzw. automatisch umwandelt werden können (Abbildung 15). Neben den primitiven Datentypen lassen sich auch Listen, solange diese wandelbare Ausdrücke enthalten, problemlos mit

**print()** ausgeben (Abbildung 16). Ist das nicht der Fall, wird eine entsprechende Fehlermeldung mit dem Hinweis auf das erste nicht als Zeichenkette darstellbare Element ausgegeben (Abbildung 17)

```
print ("Romanes", "eunt", "domus")

integer = 8
float = 6.6
print (integer, " * ", float, ' = \n', integer * float)

Romanes eunt domus
8 * 6.6 =
52.8
```

Abbildung 15 Einfache Ausgaben mit **print()**

```
liste = ("Drache", "Schatz", "Ring", 5, -.67)
print(liste)

('Drache', 'Schatz', 'Ring', 5, -0.67)
```

Abbildung 16: Ausgabe einer Liste

```
liste = ("Drache", kriegt, den, Ring, nicht)
print(liste)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-82-8ff560e39b69> in <module>
----> 1 liste = ("Drache", kriegt, den, Ring, nicht)
      2 print(liste)

NameError: name 'kriegt' is not defined
```

Abbildung 17 Nicht als Zeichenkette darstellbare Eingaben



### 2.1.1 Separatoren

Bei der Ausgabe mit **print()** lässt sich bestimmen, wie die einzelnen Parameter der Funktion bei der Ausgabe voneinander getrennt werden sollten (Abbildung 18). In der Standardeinstellung, also bei keiner Angabe oder „sep = None“, wird ein Leerzeichen zwischen die Parameter eingefügt. Sollte kein Separator verwendet werden, so ist also die Angabe sep="" notwendig. Abgesehen davon kann jede beliebige Zeichenkette als Separator verwendet werden.

```
apfel = "Apfel"
print(apfel, "Kuchen")
print(apfel, "Kuchen", sep=None)
print(apfel, "Kuchen", sep='')
print(apfel, "Kuchen", sep='-im-')
```

```
Apfel Kuchen
Apfel Kuchen
ApfelKuchen
Apfel-im-Kuchen
```

Abbildung 18: Separatoren bei Ausgabe

### 2.1.2 Ausgabe ohne Zeilenumbruch

Die Ausgaben, die mit dem **print()** Befehl getätigt werden, enden in der Regel mit einem Zeilenumbruch. Ist dies nicht erwünscht, kann dieses Verhalten durch den **end** Parameter gesteuert werden (Abbildung 19). Statt des Zeilenumbruchs kann bei diesem Parameter eine beliebige Zeichenkette angegeben werden, die dann am Ende der **print()** Ausgabe als Letzte in der Konsole ausgegeben wird.

```
print("ML", end = " ist ")
print("Statistik", end = ".")
```

```
ML ist Statistik.
```

Abbildung 19 Zeilenumbruch bei print()

### 2.1.3 Ausgabe umlenken

Die Ausgabe des **print()** Befehls lässt sich auf andere Streams umlenken. Wie Sie in Abbildung 20 sehen können, wurde dort die Ausgabe des **print()** Befehls in eine Datei

```
with open('datei.txt', mode='w') as datei_objekt:
    print('hallo SSD', file=datei_objekt)
```

Abbildung 20 **print()** Ausgabe umlenken

geschrieben. Die Datei muss vor dem Schreiben mit Schreibrechten geöffnet werden. Das Objekt (Stream) auf dem es geschrieben werden sollte, wird über dem Parameter **file** angegeben. Standardmäßig wird die Ausgabe auf „**file = sys.stdout**“ (Konsole) getätigt.

## 2.2 Datentypen

### 2.2.1 Objekttypen in Python

Bevor wir uns mit den Datentypen in Python beschäftigen können, müssen wir uns mit der Verwaltung der Objekte in Python auseinandersetzen. Der Grund dafür ist, dass im Unterschied zu C/C++ oder Java es in Python gar keine primitiven Datentypen wie **char** oder **int**, gibt. Alle Daten in Python werden nämlich immer in Objekten gespeichert. Dabei gibt es 2 Typen von Objekten, einerseits sind es die sogenannten **immutable** Objekte. Einmal angelegt, kann ihr Wert oder ihre Werte nicht mehr geändert werden. Die anderen sind **mutable** Objekte, die Ihren Wert jederzeit ändern können.

### 2.2.2 Attribute eines Objekts

fff

```
a = 0.6
print(dir(a))
```

```
['__abs__', '__add__', '__bool__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floordiv__', '__format__', '__ge__', '__getattr__', '__getformat__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__int__', '__le__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__pos__', '__pow__', '__radd__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__', '__round__', '__rpow__', '__rsub__', '__rtruediv__', '__set_format__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', 'as_integer_ratio', 'conjugate', 'fromhex', 'hex', 'imag', 'is_integer', 'real']
```

Abbildung 21

```
def add2(x,y):
    return (x + y)

print(dir(add2))

['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattr__', '__globals__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Abbildung 22

### 2.2.3 Verwaltung von Variablen in Python

Da die primitiven Typen fehlen, sind alle Variablen, die deren Funktion in Python annehmen vom Typ **immutable**, also nicht veränderlich...was ? In der Tat, einmal angelegt behält eine solche Variable ihrem Wert für ihre gesamte Existenz. Wird die Variable in Ihrem Wert verändert, dann erzeugt Python einfach eine neue Variable und weist dem Bezeichner eine neue Speicher Id (Adresse) zu (Abbildung 23). Wie Sie in der Abbildung sehen können hat sich nach der Zuweisung des neuen Wertes die Adresse von der Variable a geändert. Andererseits hat die Variable a nach der Zuweisung von a = b die Adresse der Variable b übernommen. Interessanterweise, wenn die Variable c mit dem Wert 6 erzeugt wird, der dem zweiten Wert der Variable a entspricht, bekommt diese genau die gleiche Adresse zugewiesen wie es die Variable a gehabt hat, als sie diesem Wert beinhaltet hat. Somit ist es auch ersichtlich, das für die „primitiven“ Typen in Python ein „Call by Reference“ Zugriff nicht möglich ist.

```
a = 5
print("Speicherstelle von a =", a, "- " , id(a))

a = 6
print("Speicherstelle von a =", a, "- " , id(a))

b = 7
print("Speicherstelle von b =", b, "- " , id(b))

a = b
print("Speicherstelle von a =", a, "- " , id(a))

c = 6
print("Speicherstelle von c =", c, "- " , id(c))
```

Speicherstelle von a = 5 - 140716344018720  
 Speicherstelle von a = 6 - 140716344018752  
 Speicherstelle von b = 7 - 140716344018784  
 Speicherstelle von a = 7 - 140716344018784  
 Speicherstelle von c = 6 - 140716344018752

Abbildung 23: Variablen Objekte und Ihre Adressen

## 2.2.4 int, float, boolean, complex

Schauen wir uns zuerst mal die „einstelligen“ Datentypen in Python. Dazu gehören die neben der bekannten Ganzzahl (**int**), Gleitkommazahl (**float**) sowie Binärzahl (**bool**) auch der Datentyp einer komplexen Zahl (Abbildung 24). Das die Komplexe Zahl in Python einem eigenen eingebauten Datentyp aufweist, sollte keine Überraschung sein, da Python für wissenschaftliche Arbeiten konzipiert war. Des Weiteren können Sie in der Abbildung erkennen, dass es sich wie bereits erwähnt bei diesen Datentypen immer um Instanzen der entsprechenden Klassen handelt. Abgesehen davon sind alle diese Objekte **immutable**, sodass sie ihren Wert nicht mehr ändern können, wenn sie einmal angelegt wurden. Somit werden bei einem schreibenden Zugriff, wie bereits im Kapitel 2.2.3 Verwaltung von Variablen in Python einfach neue Objekte angelegt, die den neuen Wert aufnehmen. Werden diese Datentypen miteinander verrechnet so nimmt das Ergebnis

immer den Typ des „größeren“ Datentyps ein (Abbildung 25). Somit wird eine Variable des Typs Ganzzahl nach einer Multiplikation mit einer Fließkommazahl selbst zu einer Fließkommazahl. Bei solchen Operationen wird einer booleschen Variable mit der Wert **True** als 1 interpretiert somit ist **True + True = 3**.

```
ganzzahl = 6;
print(type(ganzzahl))

gleitkommazahl = 6.9;
print(type(gleitkommazahl))

komplex = 2.0+4.9j
print(type(komplex))

boolean = True;
print(type(boolean))

<class 'int'>
<class 'float'>
<class 'complex'>
<class 'bool'>
```

Abbildung 24 Primitive Datentypen

```
integer = 7
flieskomma = 7.9
print("integer",type(integer), integer)
integer = integer * flieskomma
print("integer",type(integer), integer,"\n")

print(type(8 + True), 8 + True + True)
print(type(8 + 7.9), 8 + 7.9)
print(type(8 + 7.9+3j), 8 + 7.9)
```

```
integer <class 'int'> 7
integer <class 'float'> 55.300000000000004

<class 'int'> 10
<class 'float'> 15.9
<class 'complex'> 15.9
```

Abbildung 25: Automatische Datentypkonvertierung

## 2.2.5 Wie groß ist ein Int ?

Im Gegensatz zu anderen Programmiersprachen ist der Datentyp **Int** nur durch die Größe des Speichers der Maschine, auf der die Python Laufzeitumgebung ausgeführt wird, eingeschränkt. Der Datentyp wächst sozusagen mit seiner Aufgabe, bzw. mit der darin gespeicherten Zahl mit. Somit ist ein Überlauf einer Ganzzahl in Python nahezu unmöglich. Wie sie in der Abbildung 26 sehen können „verbraucht“ der Datentyp **Int** deutlich mehr Speicher als dieser zuerst eigentlich sollte. Bei 32 Bit sind es 4 „Nutzbytes“ + 28 Bytes „Verwaltungsdaten“. Bei 300 Bit sind es dann wiederum bereits ca. 30 Bytes an Verwaltungsdaten vorhanden - diese werden wohl für zusätzliche „Zeiger“ bzw. Referenzen auf die dazugekommenen höherwertigen Stellen benötigt. Somit gleicht der Int Datentyp intern wohl eher einer Liste aus den einzelnen Stellen (die mehrere hexadezimale Stellen groß sind), welche bei Bedarf vergrößert bzw. verkleinert wird, als einem „durchgehenden“ Datentyp, wie man ihn von C oder Java kennt.

```
bytes = [32,64,128,256,300,512]

for i in bytes:
    zahl = 1 << i
    print ("2 ^ ", i, " = ", zahl)
    size = sys.getsizeof(zahl)
    print ("size in bytes - ", size, ": Header - ", size - (i / 8))

2 ^ 32 = 4294967296
size in bytes - 32 : Header - 28.0
2 ^ 64 = 18446744073709551616
size in bytes - 36 : Header - 28.0
2 ^ 128 = 340282366920938463463374607431768211456
size in bytes - 44 : Header - 28.0
2 ^ 256 = 115792089237316195423570985008687907853269984665640564039457584007
913129639936
size in bytes - 60 : Header - 28.0
2 ^ 300 = 203703597633448608626844568840937816105146839366593625063614044935
4381299763336706183397376
size in bytes - 68 : Header - 30.5
2 ^ 512 = 134078079299425970995740249982058461274793658205923933777235614437
2176403007354697680187429816690342769003185818648605085375388281194656994643364
9006084096
size in bytes - 96 : Header - 32.0
```

Abbildung 26: Größe von Int in Python

## 2.2.6 Größe eines Objektes im Speicher ermitteln

Mit der Methode ***getsizeof*** lässt sich die Größe eines Objekts in Bytes ermitteln. Dabei ist zu beachten, dass diese Methode nur die Größe des tatsächlich von dem eigentlichen Objekt belegten Speichers zurückgibt und nicht den Gesamtspeicher, den die Datenstruktur belegt. So wird z.B. bei Listen (Abbildung 27) der Speicher, den die in der Liste verlinkten Daten belegen, in der Größenangabe nicht berücksichtigt. Wie in der Abbildung zu sehen ist, ist die Größe konstant bei 64 Bytes solange die Liste nur ein Element beinhaltet. Wichtig hierbei ist auch, dass die einzelnen Elemente der Liste noch zusätzlich ihren eigenen Speicher belegen, wie es dem Test in der Abbildung 28 zu entnehmen ist. Sollte also der Speicherverbrauch einer gesamten Datenstruktur ermittelt werden, so ist dies entsprechend zu berücksichtigen und alle damit verbundenen Speicherbereiche müssen rekursiv abgefragt werden. Dies betrifft nicht nur Listen, sondern auch alle anderen Objekte in Python.

```
import sys

list = [1]
print(sys.getsizeof(list))

list = [1,2,3,4,5]
print(sys.getsizeof(list))

list = ["1 2 3 4 5"]
print(sys.getsizeof(list))

list = ["1 2 3 4 5 7 8 9"]
print(sys.getsizeof(list[0]))
```

64  
96  
64  
64

Abbildung 27 Größe eines Objekts in Bytes ermitteln

```
list = ["1", "1 2 3 ", "1 2 3 4 5 7 8 9 1 2 3 4 5 7 8 9 1 2 3 4 5 7 8 9"]
print("Liste    = ",sys.getsizeof(list))
print("Element 0 = ",sys.getsizeof(list[0]))
print("Element 1 = ",sys.getsizeof(list[1]))
print("Element 2 = ",sys.getsizeof(list[2]))
```

```
Liste    = 80
Element 0 = 50
Element 1 = 55
Element 2 = 96
```

Abbildung 28 Speicherbelegung einer Datenstruktur am Beispiel einer Liste

## 2.2.7 Konvertierung von Datentypen

Bei Bedarf lassen sich die 4 Typen ohne Probleme konvertieren, wobei bei einer solchen Konvertierung die Genauigkeit der Ursprungszahl, je nachdem in welche die Zielzahl bzw. deren Typ gewandelt wird, verringert wird (Abbildung 29). So wird aus der Fließkommazahl 9.8 nur 9 bei einer Wandlung in eine Integer-Zahl.

```
print("to int      :",int(9.8))
print("to float    :",float(True))
print("to complex  :",complex(9.8))
print("to bool     :",bool(4+3j))
```

```
to int      : 9
to float    : 1.0
to complex  : (9.8+0j)
to bool     : True
```

Abbildung 29 Datentypen Konvertieren

## 2.2.8 Booleans

Da Booleans sehr wichtig für die Auswertung der **if** sowie **while** Anweisungen sind, schauen wir uns mal die Funktionsweise der Booleans in Einzelnen etwas genauer an.

### 2.2.8.1 Vergleichsoperatoren Auswerten

Da die mathematischen Vergleichsoperatoren bereits boolesche Ausdrücke als Ergebnis zurückliefern ist an dieser Stelle eine Konvertierung nicht notwendig. Die Ergebnisse können direkt weiterverwendet werden (Abbildung 30).

```
a = 8
b = 9
print(" a < b :", a < b)
print(" a > b :", a > b)
print("a == b :", a == b)
print("a != b :", a != b)
```

```
a < b : True
a > b : False
a == b : False
a != b : True
```

Abbildung 30 Vergleichsoperatoren Auswerten

### 2.2.8.2 Numerische Ausdrücke auswerten

Bei der booleschen Auswertung von Numerischen Ausdrücken in Python müssen diese zuerst mit der Funktion **bool()** in einem booleschen Ausdruck umgewandelt werden (Abbildung 31). Ist der Numerische Ausdruck von 0 verschieden, wobei es keine Rolle spielt ob dieser Ausdruck positiv oder negativ ist, dann wird dieser Ausdruck als Wahr (*True*) Ausgewertet. Ist der Wert des Ausdrucks 0 dann wird dieser als Falsch (*False*) ausgewertet.

```
a = 8
b = 0
print(" 8 ist - ",bool(a))
print("-8 ist - ",bool(-a))
print(" 0 ist - ",bool(b))
print(" 8 * 8 - ",bool(a * a))
print(" 8 - 8 - ",bool(a - a))
```

```
8 ist - True
-8 ist - True
0 ist - False
8 * 8 - True
8 - 8 - False
```

Abbildung 31 Numerische Ausdrücke boolesch auswerten

### 2.2.8.3 Zeichenketten auswerten

Auch Zeichenketten können boolesch über die Funktion **bool()** direkt ausgewertet werden (Abbildung 32). In diesem Fall ist der Rückgabewert *True*, wenn die Zeichenkette nicht leer ist bzw. *False* wenn diese keine Zeichen enthält. Genauso können auch Methoden der Klasse String aus Grundlage für boolesche Entscheidungen fungieren (auch Abbildung 32).

```
zeichenkette = "Hallo"
zeichen = "H"
print("'Hallo' ist nicht leer ",bool(zeichenkette))
print("' ' ist nicht leer - ",bool(""))
print(" h kommt vor - ",bool(zeichenkette.count("h")))
print(" H kommt vor - ",bool("Hallo".count(zeichen)))
```

```
'Hallo' ist nicht leer True
' ' ist nicht leer - False
h kommt vor - False
H kommt vor - True
```

Abbildung 32: Zeichenketten boolesch auswerten

### 2.2.8.4 Boolesche Ausdrücke verrechnen

Wie bereits erwähnt, lassen sich in Python boolesche Ausdrücke miteinander „mathematisch“ verrechnen. Das Ergebnis einer solchen Operation ist allerdings kein boolescher Ausdruck mehr sondern zuerst ein mathematischer (Abbildung 33). Sollte das Ergebnis boolesch ausgewertet werden, muss dieser, solange keine automatische Umwandlung seitens der Laufzeitumgebung



stattfindet, wieder mit der Funktion **bool()** umgewandelt werden. Im Grunde genommen wird hier nur ermittelt wie oft der Wert „Wahr“ („True“) in dem Ergebnis der Rechnung vorkommt (Abbildung 34).

```
a = True
b = False
print("True + False =", a + a + a, " =", bool(a + a + a))
print("True - False =", a - b, " =", bool(a - b))
print("True - True =", a - a, " =", bool(a - a))
print("True * False =", a * b, " =", bool(a * b))
print("True * True =", a * a, " =", bool(a * a))
print("True * 0.4 =", a * 0.4, " =", bool(a * 0.4))
```

```
True + False = 3    = True
True - False = 1    = True
True - True = 0     = False
True * False = 0     = False
True * True = 1     = True
True * 0.4 = 0.4    = True
```

Abbildung 33: Boolesche Ausdrücke verrechnen

```
a = True
b = False
print("True + True + True =", a + a + a, " =", bool(a + a + a))
print("True + False + True =", a + b + a, " =", bool(a + b + a))
```

```
True + True + True = 3    = True
True + False + True = 2   = True
```

Abbildung 34 Ergebnis einer Addition von booleschen Ausdrücken

### 2.2.8.5 Boolesche Ausdrücke & boolesche Algebra

Bei der Verwendung von boolescher Ausdrücke mit boolescher Algebra ist das Ergebnis der Operation, wie erwartet, immer ein boolescher Ausdruck (Abbildung 35).

```
a = True
b = False
print("True and False =", a and b)
print("True or False =", a or b)
print("True or False or True =", a or b or a)
print("not True =", not a)
```

```
True and False = False
True or False = True
True or False or True = True
not True = False
```

Abbildung 35 Boolesche Ausdrücke & boolesche Algebra

## 2.2.9 Zeichenketten

Zeichenkette im Python sind **immutable** Objekte und somit nicht veränderbar. Wird eine Zeichenkette modifiziert, so wird von der Python-Laufzeitumgebung dafür extra ein neues Objekt erstellt

(Abbildung 36). Andererseits haben zwei unterschiedliche Variablen, die identische Zeichenketten beinhalten, dieselbe Id und somit auch die gleiche Speicheradresse (Abbildung 36 und 37), zumindest solange bis eine der beiden Zeichenketten modifiziert wird.

```
string = "Hallo"
string2 = "Hallo"
print (id(string))
print (id(string2))

2395376942640
2395376942640
```

Abbildung 37  
Speicherung von  
Zeichenketten in Python

```
string = "Hallo"
print (id(string))
string += "World"
print (id(string))
string2 = "Hallo"
print (id(string2))
```

```
2395376942640
2395376964848
2395376942640
```

Abbildung 36  
Zeichenketten in Python

### 2.2.9.1 Zeichenketten definieren

Im Unterschied zu C/C++ ist es in Python unerheblich, ob für Literale einfache (') oder doppelte (") Hochkommas bei der Definition von Zeichenketten verwendet werden. Beides wird als eine Begrenzung einer Zeichenkette erkannt. Dies kann allerdings auch produktiv verwendet werden, falls Hochkommas in der Ausgabe selbst enthalten werden sollten. Dann wird einfach der gewünschte Typ des Hochkommas, also der, der in der Ausgabe erscheinen sollte, von dem anderen umgeben (Abbildung 38). Alternativ können aber auch Steuerzeichen (Escape-Zeichen) verwendet werden um Hochkommas auszugeben (auch Abbildung 38, letztes Beispiel).

```
print('Ich sehe "doppelt" ')
print("ich sehe 'einfach' ")
print("Ich sehe \"doppelt anders\" ")
```

```
Ich sehe "doppelt"
ich sehe 'einfach'
Ich sehe "doppelt anders"
```

Abbildung 38: Hochkommas ausgeben

### 2.2.9.2 Zeichenketten mit mehreren Zeilen

Im Unterschied zu C oder Java können Zeichenketten auch ohne zusätzliche Formatierungszeichen aus mehreren Zeilen bestehen. Zur Definition dieser Zeichenketten wird einfach das Anführungszeichen am Anfang sowie am Ende der Zeichenkette dreifach angegeben (Abbildung 39) Somit wird die eigentliche Zeichenkette nicht durch Steuerzeichen sichtbar „verunstaltet“, was die Lesbarkeit dieser in dem Quellcode deutlich verbessert.

```
Zeichenkette = """ Erste Zeile  
Zweite Zeile  
Dritte Zeile"""  
print(Zeichenkette)
```

```
Erste Zeile  
Zweite Zeile  
Dritte Zeile
```

Abbildung 39: Mehrzeilen  
Zeichenkette

### 2.2.9.3 Zeichenketten als Felder

Eine Zeichenkette ist in Python als ein Feld von Buchstaben implementiert. Im Unterschied zu anderen Programmiersprachen sind Zeichenketten, wie auch andere Felder, nicht nur positiv sondern auch negativ indiziert

(Abbildung 40). Abgesehen davon gibt es bei Python Zeichenketten auch keine „nervigen Extras“ wie den Nullterminator bei C/C++. Bei Zugriffen auf die Felder einer Zeichenkette ist es nicht

nur möglich einzelne Zeichen zu adressieren, sondern auch ganze Bereiche, die aus mehreren Zeichen bestehen (Abbildung 41 & 42). Zu beachten ist, dass bei den Bereichsangaben der Anfang inklusive, aber das Ende exklusive angegeben wird.

Positiver Index	0	1	2	3	4	5	6
Zeichen	H	A	L	L	O		!
Negativer Index	-7	-6	-5	-4	-3	-2	-1

Abbildung 40: Zeichenkette als Feld

```
string = "Hallo !"
str = string[-4:]
print(">"+str+"")

lo !

str = string[1:-3]
print(">"+str+"")

'all'

str = string[-3:-1]
print(">"+str+"")

'o '
```

Abbildung 42  
Zugriffe auf Zeichen eines Strings

[-4:]	0	1	2	-4	-3	-2	-1
Zeichen	H	A	L	L	O		!
[1:-3]	0	1	2	3	-3	-2	-1
Zeichen	H	A	L	L	O		!
[-3:-1]	0	1	2	3	-3	-2	-1
Zeichen	H	A	L	L	O		!

Abbildung 41: Zugriffe auf Zeichen eines Strings

### 2.2.9.4 Zeichenketten iterieren

Da die Zeichenketten als Felder in Python implementiert sind, ist es ohne Weiteres möglich über einzelne Zeichen eines Strings zu iterieren.

```
string = "Hallo"
for char in string:
    print(char)
```

```
H
a
l
l
o
```

Abbildung 43

### 2.2.9.5 Zeichenketten zusammensetzen

Mit dem „+“-Operator lassen sich Zeichenketten zusammensetzen. Dabei können nicht nur Zeichenketten die in Variablen definiert sind verwendet werden, sondern auch „ad hoc“ definierte Zeichenketten dem Endergebnis zugemischt werden (Abbildung 44). Falls numerische Ausdrücke verwendet werden, müssen diese zuerst in Zeichenketten mit der `str()` Funktion umgewandelt werden (Abbildung 45). Zu beachten ist, dass so entstandenen Ausdrücke statischer Natur sind, deswegen muss bei der Änderung eines Wertes die Zeichenkette komplett neu gebildet werden.

```
anfang = "Am Anfang"
mitte = "war das"
ende = "Ende"
alles = anfang + " " + mitte + " " + ende
print(alles)
```

```
Am Anfang war das Ende
```

Abbildung 44: Zeichenketten Zusammensetzen

```
menge = [3, 2, 8, 10, 6]
maximum = " Größter Wert ist " + str(max(menge))
print(maximum)
```

```
menge.append(11.5)
print(maximum)
```

```
maximum = " Größter Wert ist " + str(max(menge))
print(maximum)
```

```
Größter Wert ist 10
Größter Wert ist 10
Größter Wert ist 11.5
```

Abbildung 45: Zeichenketten sind Statisch

### 2.2.9.6 Zeichenketten zusammensetzen mit *join()*

Sollten Zeichenketten aus Listen oder anderen Iterable-Objekten zusammengesetzt werden, kann dies mit der Funktion ***join*** erfolgen. Zu beachten ist, dass diese Funktion, neben der Liste auch noch einen Separator benötigt. Der Separator selbst, wird vor dem Funktionsaufruf angegeben (Abbildung 46). Als Ergebnis der Operation wird eine Zeichenkette erstellt, die anschließend ausgegeben oder einer Variable zugewiesen werden kann.

```
wochentage = ("Montag","Dienstag","Mittwoch")
print(wochentage)

tage = "Tage = "
tage.join(wochentage)
print(tage)

tage += ";".join(wochentage)
print(tage)
```

```
('Montag', 'Dienstag', 'Mittwoch')
Tage = 
Tage = Montag;Dienstag;Mittwoch
```

Abbildung 46 Listen mit *join* umwandeln

Format:

*Zeichenkette* = <Separator>.***join***(iterable)

## 2.3 Collectibles

Die Programmiersprache Python kennt vier verschiedene Elementen Kollektionen.

### 2.3.1.1 Größe einer Kollektion bestimmen

Die Größe einer Kollektion kann mit der Funktion **len()** ermittelt werden (Abbildung 47). Die zu untersuchende Kollektion wird der Funktion als Parameter übergeben. Eine Klassen-Funktion zur Bestimmung der Länge einer Kollektion gibt es in Python nicht.

```
print(bauwerk)
print(len(bauwerk))

('Haus', 'Haus', 'Haus', 'Garage', 'Monument')
5
```

Abbildung 47 Größe einer Kollektion bestimmen

### 2.3.1.2 Kollektionen umwandeln

Die Kollektionstypen Set, Tuple und List lassen sich problemlos untereinander umwandeln. (Abbildung 51). Es ist allerdings dabei zu beachten, dass dabei Eigenschaften (Sortierung, Duplikate) verloren gehen können wenn z.b. eine Liste zuerst zu einem Set und danach wieder zurück zu einer Liste umgewandelt wird.

```
bauwerk = ["Haus", "Monument", "Tempel", "Haus"]

bauwerkTuple = tuple(bauwerk)
bauwerkSet = set(bauwerk)
bauwerkListe = list(bauwerkSet)

print("Tuple : ",bauwerk)
print("Tuple : ",bauwerkTuple)
print("Set   : ",bauwerkSet)
print("Liste : ",bauwerkListe)

Tuple :  ['Haus', 'Monument', 'Tempel', 'Haus']
Tuple :  ('Haus', 'Monument', 'Tempel', 'Haus')
Set   :  {'Tempel', 'Haus', 'Monument'}
Liste :  ['Tempel', 'Haus', 'Monument']
```

Abbildung 48 Kollektionen umwandeln

### 2.3.1.3 Elemente einer Kollektionen iterieren

Über den Inhalt bzw. den einzelnen Elementen einer Kollektion lässt sich z.B. mit einer **for** Schleife iterieren (Abbildung 49)

```
bauwerk = ["Haus", "Monument", "Tempel", "Haus"]  
for objekt in bauwerk:  
    print("Bauwerk :", objekt)
```

```
Bauwerk : Haus  
Bauwerk : Monument  
Bauwerk : Tempel  
Bauwerk : Haus
```

Abbildung 49 Elemente einer Kollektion iterieren

### 2.3.1.4 Zugehörigkeit prüfen

Ob ein bestimmtes Element in einer Kollektion vorhanden ist, kann einfach mit der **if..in** Funktion abgeprüft werden (Abbildung 50). Ist es erwünscht festzustellen, ob ein bestimmtes Element nicht in der Kollektion vorhanden ist, so kann hingegen **if..not in** verwendet werden.

```
bauwerk = ["Haus", "Monument", "Tempel", "Haus"]  
if "Haus" in bauwerk:  
    print("Haus ist da")
```

```
if "Garage" not in bauwerk:  
    print("Garage ist nicht da")
```

```
Haus ist da  
Garage ist nicht da
```

Abbildung 50 Zugehörigkeit eines Elements prüfen

### 2.3.1.5 Zugriffe über Indexe

Auf die Elemente einer Kollektion können auch über den direkten Index in die Kollektion zugegriffen werden (Abbildung 51) Wobei neben den Zugriff auf einzelne Elemente hier auch Bereiche [von:bis] angegeben werden können. Fehlt die Angabe des Anfangsbereichs, dann gilt der Anfangsindex Index = 0. Ist

```
bauwerk = ["Haus", "Monument", "Tempel", "Haus", "Garten"]  
print(bauwerk)  
print(bauwerk[1])  
print(bauwerk[:2])  
print(bauwerk[2:4])  
print(bauwerk[2:])
```

```
['Haus', 'Monument', 'Tempel', 'Haus', 'Garten']  
Monument  
['Haus', 'Monument']  
['Tempel', 'Haus']  
['Tempel', 'Haus', 'Garten']
```

Abbildung 51 Zugriffe über Indexe



hingegen der Endindex nicht angegeben, dann gilt der Index des letzten Element als der letzte Index. In übrigen ist zu beachten, dass der Endindex exklusiv interpretiert wird, sodass die Angabe [2:4] nur die Elemente 2 & 3 indiziert nicht aber den Element mit dem Index 4.

### 2.3.1.6 Negative Indexierung

Die Elemente einer Kollektion können auch von der letzten Position ausgehend rückwärts indiziert werden. Zu diesem Zweck können negative Indizes verwendet (Abbildung 52) werden. Dabei hat das letzte Element in der Kollektion den Index -1, das vorletzte -2 usw.

```
bauwerk = ["Haus", "Monument", "Tempel", "Garage"]
print(bauwerk)
print(bauwerk[-1], ", ", bauwerk[-3])
```

['Haus', 'Monument', 'Tempel', 'Garage']  
Garage , Monument

Abbildung 52: Negative Indexierung

### 2.3.1.7 Kollektion vereinigen

Zwei oder mehrere Kollektionen des gleichen Typs lassen sich mit dem + Operator zu einer neuen Kollektion vereinigen. Ist ein Element in beiden Kollektionen mehrfach vorhanden, dann wird dieser auch mehrfach (außer im Set und Dictionary) in der neu erstellten Kollektion übernommen (Abbildung 53) und zwar genauso oft wie er in beiden Kollektionen

```
profan = ["Haus", "Garage", "Monument"]
sakral = ["Tempel", "Monument"]
print(profan)
print(sakral)

bauwerk = profan + sakral
print(bauwerk)
```

['Haus', 'Garage', 'Monument']  
['Tempel', 'Monument']  
['Haus', 'Garage', 'Monument', 'Tempel', 'Monument']

Abbildung 53 Kollektion vereinigen.

zusammen auftritt. Die beiden Quellkollektionen werden bei dieser Operation nicht verändert.

Format:

Kollektion = Quellkollektion 1 + Quellkollektion 2

### 2.3.1.8 Kollektion löschen

Eine Kollektion kann mit der Funktion **del** gelöscht werden (Abbildung 54). Danach ist der Bezeichner bis zu einer Neudefinition nicht mehr gültig. Ein Zugriff auf die entsprechende Variable verursacht einem Zugriffsfehler.

```
bauwerk = {"Haus", "Monument", "Tempel", "Haus"}
print(bauwerk)
del bauwerk
print(bauwerk)

{'Tempel', 'Haus', 'Monument'}
```

---

```
NameError                                Traceback (most recent call last)
<ipython-input-85-42b4e7d9568f> in <module>
      2 print(bauwerk)
      3 del bauwerk
----> 4 print(bauwerk)

NameError: name 'bauwerk' is not defined
```

Abbildung 54 Container löschen

### 2.3.2 Kollektionen sortieren

Die Iteratoren, bzw. dessen Elemente, lassen sich, der Größe ihrer Werte nach, sortieren (Abbildung 55). Dies gilt gleichermaßen für numerische wie auch für alphanumerische Elemente. Die alphanumerische Sortierung erfolgt nach der Position der Zeichen in der ASCII- bzw. UTF-Tabelle. Somit sind Zahlen kleiner als Großbuchstaben, welche wiederum kleiner als Kleinbuchstaben sind. Es folgen die Sonderzeichen und Zeichen der nicht lateinischen Alphabete.

Beachten Sie, dass in dem Beispiel das sumerische Zeichen ✱ („*diĝir*“ - „Gottheit“) als letzte in der Ausgabeliste einsortiert wurde.

```
liste = [1,2.1,3,4.99,0]
print("original - ",liste);
print("sortiert - ",sorted(liste))

liste = ["u","g","a","Ä","✱","U","1"]
print("\n", liste)
print("",sorted(liste))

original -  [1, 2.1, 3, 4.99, 0]
sortiert -  [0, 1, 2.1, 3, 4.99]

['u', 'g', 'a', 'Ä', '✱', 'U', '1']
['1', 'U', 'a', 'g', 'u', 'Ä', '✱']
```

Abbildung 55 Kollektionen sortieren

### 2.3.3 Reihenfolge der Elemente mit `reversed()` umdrehen

Benötigt man eine umgekehrte Reihenfolge der Elemente einer Kollektion, so lässt sich diese mit der **`reversed()`** Methode erstellen. (Abbildung 57). Zu bedenken ist, dass diese Methode keine Liste zurückgibt, diese muss erst explizit mit der **`list()`** Funktion aus dem Zurückgegebenen Objekt erstellt werden.

```
liste = [1,2.1,3,4.99,0]
print("original - ",liste);
print("umgedreht - ",list(reversed(liste)))

liste = ["u","g","a","Ä","a","U","1"]
print("\noriginal - ",liste);
print("umgedreht - ", list(reversed(liste)))
print("umgedreht - ", reversed(liste))

original - [1, 2.1, 3, 4.99, 0]
umgedreht - [0, 4.99, 3, 2.1, 1]

original - ['u', 'g', 'a', 'Ä', 'a', 'U', '1']
umgedreht - ['1', 'U', 'a', 'Ä', 'a', 'g', 'u']
umgedreht - <list_reverseiterator object at 0x0000026110042880>
```

Abbildung 56: Elementen folge einer Kollektion umkehren

### 2.3.4 Sum()

Die Werte der Elemente einer Kollektion lassen sich mit **`sum()`** aufaddieren, solange es sich dabei um numerisch auswertbare Werte handelt (Abbildung 57). Alphanumerische Werte lassen sich, im Gegensatz zu der gleichnamigen Methode in Pandas, nicht mit der **`sum()`** Funktion aggregieren.

```
liste = [1,2.1,3,4.99,-1]
print("Werte - ",liste);
print("summe - ",sum(liste));

Werte - [1, 2.1, 3, 4.99, -1]
summe - 10.09
```

Abbildung 57 Elemente summieren

### 2.3.5 min() und max()

Der Wert des Elements mit dem kleinsten bzw. größten Wert lässt sich über die beiden Funktionen **min()** bzw. **max()** ermitteln. Bei numerischen Werten wird der nominelle Wert verwendet, um das Maximum bzw. Minimum zu bestimmen. Bei Buchstaben wird die Position der Zeichen in der ASCII (bzw. UTF) Tabelle zur Bestimmung der „Mächtigkeit“ der Werte herangezogen (Abbildung 58). Das erklärt auch, warum „U1“ kleiner als „Ue“ ist: Da das Zeichen „1“ (49) in der ASCII Tabelle vor dem Zeichen „e“ (101) positioniert ist, wird „U1“ als kleiner als „Ue“ gewertet.

```
liste = [-1, 2.1, 4.999, 4.99, 0]
print("minimum ist : ", min(liste))
print("maximum ist : ", max(liste))

liste = ["aa", "ab", "U1", "Ue"]
print("\nminimum ist : ", min(liste))
print("maximum ist : ", max(liste))

minimum ist : -1
maximum ist : 4.999

minimum ist : U1
maximum ist : ab
```

Abbildung 58 max() und min() Funktionen

### 2.3.6 all() und any()

Die beiden Funktionen werten eine Kollektion aus und prüfen, ob alle (**all()**) oder mindestens ein (**any()**) Element der Kollektion Wahr (True) ist. Bei numerischen Werten folgt die Auswertung den in Kapitel 2.2.8.2 *Numerische Ausdrücke Auswerten* erörterten Regeln. In Abbildung 59 sehen Sie ein Beispiel hierzu. Den Auswertungsregeln folgend wird hier untersucht, ob alle oder mindestens ein Wert in der Liste  $> 0$  ist. Wir erinnern uns – ein Wert von 0 wird als Falsch ausgewertet. Ein Wert ungleich 0 hingegen ist stets Wahr. So kommt es, dass die Liste ohne den letzten Wert ([0:4]) ein Wahr bei der Auswertung mit **all** liefert, da in dieser Liste der letzte Element mit dem Wert 0 nicht mehr enthalten ist.

```
liste = [1,2,3,4,0]
print("Werte = ", liste)
print("alle Werte > 0 ? ", all(liste))
print(" ein Wert > 0 ? ", any(liste))

print("\nWerte = ", liste[0:4])
print("alle Werte > 0 ? ", all(liste[0:4]))

Werte = [1, 2, 3, 4, 0]
alle Werte > 0 ? False
ein Wert > 0 ? True

Werte = [1, 2, 3, 4]
alle Werte > 0 ? True
```

Abbildung 59 Funktionsweise von all() und any()

## 2.3.7 Sets

Bei den Sets handelt es sich um **mutable** nicht geordnete und nicht indizierte Kollektionen. Sie können keine Duplikate enthalten. Da sie nicht geordnet sind, führen Zugriffe über Indizes zu unbestimmbaren Ergebnissen.

### 2.3.7.1 Sets Definieren

Ein Set wird durch geschweifte Klammern definiert, die Reihenfolge der Elemente in einem Set ist nach dessen Definition nicht änderbar, es können aber weitere Elemente zu einem Set hinzugefügt werden, bzw. bereits in dem Set vorhandene Elemente aus diesem Set gelöscht werden. Da Sets keine Duplikate enthalten können, werden die bei der Definition vorhandenen Duplikate ignoriert (Abbildung 60). Die Sets sind nicht geordnet, also kommen die Elemente bei Zugriffen auf das Set nicht in gleicher Reihenfolge wie bei Definition des Sets - beachte die geänderte Reihenfolge der Elemente „Monument“ und „Tempel“ in der Abbildung 60. Ein Set wird mit einer geschweiften Klammer definiert.

```
gebeude = {"Haus", "Monument", "Tempel", "Haus"}
print(gebeude)

{'Haus', 'Tempel', 'Monument'}
```

Abbildung 60 Definition eines Sets in Python

### 2.3.7.2 Elemente hinzufügen

Um weitere Elemente zu einem Set hinzufügen ist die Methode **.add()** zuständig (Abbildung 61). Da Sets nicht geordnet sind wird das neue Element an irgendeiner Stelle in das Set eingefügt. Des Weiteren lassen sich mit der Methode **.update()** ganze Sets oder z.B. auch Listen in dem aktuellen Set einfügen (Abbildung 62).

```
print(bauwerk)
bauwerk.add("Garage")
print(bauwerk)

{'Haus', 'Tempel', 'Monument'}
{'Garage', 'Haus', 'Tempel', 'Monument'}
```

Abbildung 61 Einzelne Elemente einfügen

```
print(bauwerk)
bauwerk.update({"Dom", "Schuppen"})
print(bauwerk)

{'Garage', 'Haus', 'Tempel', 'Monument'}
{'Schuppen', 'Dom', 'Garage', 'Haus', 'Tempel', 'Monument'}
```

Abbildung 62 Sets einfügen mit update

### 2.3.8 Einzelne Elemente entfernen

Zuerst besteht die Möglichkeit, einzelne Elemente eines Sets mit der Methode **.remove()** zu löschen (Abbildung 63). Die andere Möglichkeit ein bestimmtes Element eines Sets zu entfernen ist die Methode **.discard()**. Der Unterschied zwischen den beiden Methoden ist, dass **.remove()** einen Fehler meldet, falls ein entsprechendes Element nicht gefunden werden kann.

```
bauwerk = {"Haus", "Monument", "Tempel", "Haus"}
print(bauwerk)
bauwerk.remove("Haus")
print(bauwerk)

{'Tempel', 'Haus', 'Monument'}
{'Tempel', 'Monument'}
```

Abbildung 63 Einzelne Elemente eines Sets entfernen

### 2.3.9 Letztes Element entfernen

Mit der Methode **.pop()** kann das letzte Element eines Sets entfernt werden. Da die Sets ungeordnet sind kann nicht von vornherein vorausgesagt werden, welches der Elemente dann auch tatsächlich entfernt wird (Abbildung 64).

```
bauwerk = {"Haus", "Monument", "Tempel", "Haus"}
print(bauwerk)
bauwerk.pop()
print(bauwerk)

{'Tempel', 'Haus', 'Monument'}
{'Haus', 'Monument'}
```

Abbildung 64 Letztes Element eines Sets entfernen

#### 2.3.9.1 Alle Elemente eines Sets entfernen

Sollten alle Elemente eines Sets entfernt werden, so ist die Methode **.clear()** zu verwenden (Abbildung 65). Nach deren Anwendung bleibt immer nur ein leeres Set übrig.

```
bauwerk = {"Haus", "Monument", "Tempel", "Haus"}
print(bauwerk)
bauwerk.clear()
print(bauwerk)

{'Tempel', 'Haus', 'Monument'}
set()
```

Abbildung 65 Alle Elemente eines Sets entfernen

### 2.3.9.2 Schnittmenge zweier Sets

Es gibt zwei Möglichkeiten Schnittmengen mit Sets zu bilden. Einerseits kann das aktuelle Set mit der Methode ***.intersection\_update()*** mit einem zweitem Set eine Schnittmenge bilden. In diesem Fall bleibt in dem aktuellen Set nur die Schnittmenge der beiden Sets übrig. Andererseits kann mit der Methode ***.intersection()*** aus den beiden Sets ein neues gebildet werden. In diesem Fall bleiben die beiden ursprünglichen Sets unverändert und die Schnittmenge ist nur in dem neu erstellten Set enthalten (Abbildung 66).

```
profan = {"Haus", "Garage", "Monument"}
sakral = {"Tempel", "Monument"}
print(profan)
print(sakral)

bauwerk = profan.intersection(sakral)
print(bauwerk)

profan.intersection_update(sakral)
print(profan)
```

```
{'Haus', 'Garage', 'Monument'}
{'Tempel', 'Monument'}
{'Monument'}
{'Monument'}
```

Abbildung 66 Schnittmengen Bilden

### 2.3.9.3 Differenz zweier Sets Bilden

Wie bei der Schnittmenge, lässt sich die Differenzmenge zweier Sets auf zwei verschiedene Arten bilden. Hierbei erhält man als Ergebnis alle Elemente, die nur in dem aktuellen Set vorhanden sind und nicht gleichzeitig auch noch in dem vermengten Set existieren. Die entsprechenden Methoden heißen ***.difference()*** sowie ***.difference\_update()*** und funktionieren genauso wie die Methoden zu Schnittmengenerstellung (Abbildung 67).

```
profan = {"Haus", "Garage", "Monument"}
sakral = {"Tempel", "Monument"}
print(profan)
print(sakral)

bauwerk = profan.difference(sakral)
print(bauwerk)

profan.difference_update(sakral)
print(profan)
```

```
{'Haus', 'Garage', 'Monument'}
{'Tempel', 'Monument'}
{'Haus', 'Garage'}
{'Haus', 'Garage'}
```

Abbildung 67: Differenz Bilden

### 2.3.9.4 Symmetrische Differenz zweier Sets Bilden

Wie bei der Schnittmenge lässt sich die symmetrische Differenzmenge zweier Sets auf zwei verschiedene Arten bilden. Hierbei erhält man als Ergebnis alle Elemente, die in einem der beiden Sets existieren, aber nicht gleichzeitig in dem anderen Set vorhanden sind. Die beiden Methoden heißen `.symmetric_difference()` sowie `.symmetric_difference_update()` und funktionieren genauso wie die Methoden zur Schnittmengengenerierung (Abbildung 68).

```
profan = {"Haus", "Garage", "Monument"}
sakral = {"Tempel", "Monument"}
print(profan)
print(sakral)

bauwerk = profan.symmetric_difference(sakral)
print(bauwerk)

profan.symmetric_difference_update(sakral)
print(profan)
```

```
{'Haus', 'Garage', 'Monument'}
{'Tempel', 'Monument'}
{'Haus', 'Tempel', 'Garage'}
{'Tempel', 'Haus', 'Garage'}
```

Abbildung 68 Symmetrische Differenz zweier Sets bilden

### 2.3.9.5 Auf Untermenge prüfen

Die Methode `.issubset()` führt eine Prüfung aus, ob das aktuelle Set eine Untermenge eines zweiten Sets darstellt. Ist das der Fall, so ist die boolesche Antwort dieser Methode `True`, sonst ist der Rückgabewert `False` (Abbildung 69).

```
profan = {"Haus", "Garage", "Monument"}
sakral = {"Monument"}
print(profan)
print(sakral)

bauwerk = sakral.issubset(profan)
print(bauwerk)

bauwerk = profan.issubset(sakral)
print(bauwerk)
```

```
{'Haus', 'Garage', 'Monument'}
{'Monument'}
True
False
```

Abbildung 69 Auf Untermenge prüfen



### 2.3.9.6 Auf Obermenge prüfen

Die Methode `.issuperset()` führt hingegen die gegensätzliche Prüfung aus: Sie stellt fest, ob das aktuelle Set eine Obermenge des übergebenen Sets darstellt, also ob der übergebene Set in dem aktuellem Set vollständig enthalten ist. Diese Methode liefert eine boolesche Antwort *True* oder *False* (Abbildung 70).

```

profan = {"Haus", "Garage", "Monument"}
sakral = {"Monument"}
print(profan)
print(sakral)

bauwerk = sakral.issuperset(profan)
print(bauwerk)

bauwerk = profan.issuperset(sakral)
print(bauwerk)

{'Haus', 'Garage', 'Monument'}
{'Monument'}
False
True

```

Abbildung 70 Auf Obermenge prüfen

### 2.3.9.7 Prüfung ob zwei Sets disjunkt sind

Eine Prüfung, ob zwei Sets disjunkt sind, also keine gemeinsamen Elemente enthalten, ist mit der Methode `.isdisjoint()` möglich. Diese Methode liefert eine boolesche Antwort *True*, wenn die beiden Sets keine gleichen Elemente aufweisen, bzw. *False* falls beide Sets mindestens ein gemeinsames Element beinhalten (Abbildung 71).

```

profan = {"Haus", "Garage", "Monument"}
sakral = {"Monument", "Tempel"}
print(profan)
print(sakral)

print(sakral.isdisjoint(profan))
sakral.remove("Monument")
print(sakral.isdisjoint(profan))

{'Haus', 'Garage', 'Monument'}
{'Tempel', 'Monument'}
False
True

```

Abbildung 71 Prüfung ob zwei Sets disjunkt sind

### 2.3.9.8 Sets Kopieren

Von einem bereits vorhanden Set kann mit der Methode **.copy()** eine 1 zu 1 Kopie erstellt werden. (Abbildung 72).

```
profan = {"Haus", "Garage", "Monument"}
print(profan)

bauwerk = profan.copy()
print(bauwerk)

{'Haus', 'Garage', 'Monument'}
{'Haus', 'Garage', 'Monument'}
```

Abbildung 72: Sets Kopieren

### 2.3.9.9 Sets vereinigen

Zwei, oder mehrere Sets, können zu einem neuem Set vereinigt werden. Die Methode **.union()** nimmt als Parameter die Sets, die mit dem aktuellem Set vereinigt werden sollten (Abbildung 73). Der neue Set beinhaltet alle Elemente aus den vermengten Sets, da ein Set keine Dubletten beinhalten kann, werden diese bei der Vereinigung automatisch entfernt.

```
profan = {"Haus", "Garage"}
sakral = {"Tempel"}
antik = {"Monument"}

print(profan)
print(sakral)
print(antik)

bauwerk = profan.union(sakral, antik)

print(bauwerk)

{'Haus', 'Garage'}
{'Tempel'}
{'Monument'}
{'Tempel', 'Haus', 'Garage', 'Monument'}
```

Abbildung 73: Sets vereinigen

## 2.3.10 Tupel

Die Tupel sind im Unterschied zu den Sets sortierte Container aus Elementen, die aber dafür unveränderbar sind. Ist ein Tupel definiert, lassen sich weder weitere Elemente hinzufügen, noch die in dem Tupel vorhandenen Elemente löschen. Ein Tupel darf, auch hier im Unterschied zu einem Set, Duplikate der Elemente beinhalten. Die Tupel werden durch runde Klammern definiert. Zu beachten ist, dass bei Tupeln, die nur ein Element beinhalten, ein Komma nach dem ersten (und einzigen) Element folgt, sonst wird es nicht als Tupel erkannt (Abbildung 74).

```
bauwerk = ("Haus")
print(type(bauwerk),bauwerk)

bauwerk = ("Haus",)
print(type(bauwerk),bauwerk)

bauwerk = ("Haus","Haus","Haus","Garage","Monument")
print(bauwerk)

<class 'str'> Haus
<class 'tuple'> ('Haus',)
('Haus', 'Haus', 'Haus', 'Garage', 'Monument')
```

Abbildung 74: Tupel definieren

### 2.3.10.1 Anzahl der Vorkommnisse eines Elements ermitteln

Die Anzahl, in der ein Element in einem Tupel vorkommt, kann mit der Methode `.count()` ermittelt werden.

```
print(bauwerk)
print(bauwerk.count("Haus"))

('Haus', 'Haus', 'Haus', 'Garage', 'Monument')
3
```

Abbildung 75 Anzahl der Vorkommnisse ermitteln

### 2.3.10.2 Index eines Elements im Tupel ermitteln

Mit der Methode `.index()` lässt sich der Index des ersten Auftretens eines Elements in einem Tupel ermitteln (Abbildung 76).

```
print(bauwerk)
print(bauwerk.index("Garage"))

('Haus', 'Haus', 'Haus', 'Garage', 'Monument')
3
```

Abbildung 76 Index eines Elements im Tupel ermitteln

### 2.3.10.3 *Tupel bearbeiten*

Auch wenn Tupel an sich nicht veränderbar sind, lassen sich über einen Trick deren Eigenschaften ändern. Dazu wird der Tupel zuerst mit der ***list()*** Funktion in eine Liste gewandelt. Die neu erstellte Liste kann dann problemlos bearbeitet werden, indem z.B. Elemente hinzugefügt oder verändert werden. Die veränderte Liste wird schließlich mit der Funktion ***tuple()*** zurück in ein Tupel umgewandelt (Abbildung 77).

```
bauwerk = ("Haus", "Garage", "Monument")  
print(bauwerk)
```

```
tmp = list(bauwerk)  
tmp.append("Schuppen")  
tmp[2] = "Tempel";
```

```
bauwerk = tuple(tmp)  
print(bauwerk)
```

```
('Haus', 'Garage', 'Monument')
```

```
('Haus', 'Garage', 'Tempel', 'Schuppen')
```

Abbildung 77 Eigenschaften eines Tupels ändern

## 2.3.11 Listen

Im Unterschied zu einem Set können Listen verändert werden. Es ist möglich weitere Elemente hinzufügen oder bereits vorhandene Elemente aus einer Liste zu löschen. Des Weiteren kann eine Liste auch Duplikate enthalten (Abbildung 78). Eine Liste wird durch eckige Klammern definiert.

```
gebäude = ["Haus", "Monument", "Tempel", "Haus"]  
print(gebäude)
```

```
['Haus', 'Monument', 'Tempel', 'Haus']
```

Abbildung 78: Definition einer Liste in Python

### 2.3.11.1 Länge einer Liste ermitteln

Die aktuelle Länge einer Liste lässt sich mit der Funktion **len()** ermitteln. (Abbildung 79).

```
print(bauwerk)  
print(len(bauwerk))
```

```
['Haus', 'Monument', 'Tempel', 'Haus']  
4
```

Abbildung 79 Länge einer Liste ermitteln

### 2.3.11.2 Häufigkeit eines Elements zählen

Die Häufigkeit mit der ein Element in einer Liste vorkommt, kann mithilfe der **.count()** Methode festgestellt werden. Als einziger Parameter, den diese Funktion benötigt, wird der Name des gewünschten Elements angegeben (Abbildung 80).

Format:

*Elementanzahl* = liste.**count**(Element)

```
print(bauwerk)
print("Haus ist",bauwerk.count("Haus"), "mal vorhanden")

['Haus', 'Monument', 'Tempel', 'Haus', 'Dom', 'Dom', 'Haus']
Haus ist 3 mal vorhanden
```

Abbildung 80 Häufigkeit eines Elements in einer Liste zählen

### 2.3.11.3 Erstes Vorkommen in Liste ermitteln

Mit der Methode **.index()** ist es möglich, den Index des ersten Vorkommens eines Elements zu ermitteln. Es wird nur der Index des ersten gefundenen Elementes zurückgegeben, alle anderen Elemente mit dem gleichen Namen werden ignoriert.

Format:

*Index* = liste.**index**(Element)

```
print(bauwerk)
print("der erste Index des Doms ist ",bauwerk.index("Dom"))

['Haus', 'Monument', 'Tempel', 'Haus', 'Dom', 'Dom', 'Haus']
der erste Index des Doms ist 4
```

Abbildung 81 Erstes Vorkommen eines Elements in Liste ermitteln

### 2.3.11.4 Reihenfolge in der Liste umkehren

Um die Reihenfolge der Elemente in einer Liste umzudrehen, wird die Methode **.reverse()** verwendet. Sie dreht die Reihenfolge der Elemente in der Liste um, sodass das letzte Element zum ersten wird und umgekehrt (Abbildung 82).

```
print(bauwerk)
bauwerk.reverse()
print(bauwerk)

['Haus', 'Monument', 'Tempel', 'Haus']
['Haus', 'Tempel', 'Monument', 'Haus']
```

Abbildung 82 Reihenfolge in der Liste umkehren

Format:

```
liste.reverse()
```

### 2.3.11.5 Liste sortieren

Die Listen lassen sich mit der `.sort()` Methode sortieren. Diese Methode kann über 2 Parameter konfiguriert werden. Der Parameter **reverse** erlaubt über die Steuerparameter **True** oder **False**, die Richtung der Sortierung einzustellen (Abbildung 83) .

Format:

```
liste.sort(reverse = True|False, key = keyFunction )
```

```
print(bauwerk)
bauwerk.sort(reverse = True)
print(bauwerk)

['Haus', 'Monument', 'Tempel', 'Haus', 'Dom', 'Dom', 'Haus']
['Tempel', 'Monument', 'Haus', 'Haus', 'Haus', 'Dom', 'Dom']
```

Abbildung 83 Elemente einer Liste sortieren

Der zweite Parameter **key**, erlaubt, über die Zuhilfenahme einer Schlüsselfunktion eine eigene Sortiervorschrift für die Daten zu bilden. In der Abbildung 84 wurden die Daten über eine eigene Funktion (**getLength**) nach der Menge der darin enthaltenen Buchstaben sortiert.

```
def getLength(e):
    return len(e)

bauwerk.sort(key=getLength)
print(bauwerk)

['Dom', 'Dom', 'Haus', 'Haus', 'Haus', 'Tempel', 'Monument']
```

Abbildung 84 Elemente einer Liste nach Schlüsseln sortieren

### 2.3.11.6 Liste leeren

Die Methode `.clear()` macht genau das, was ihr Name auch vermuten lässt - sie entfernt alle Elemente einer Liste, so das nur noch eine leere Liste übrig bleibt (Abbildung 85).

```
print(bauwerk)
bauwerk.clear()
print(bauwerk)

['Haus', 'Tempel', 'Monument', 'Haus']
[]
```

Abbildung 85 Liste leeren

Format:

`liste.clear()`

### 2.3.11.7 Elemente hinzufügen

Bei einer Liste stehen 3 Methoden zur Verfügung, um ein oder mehrere Elemente der Liste hinzufügen. Die Methode **.append()** fügt der liste exakt ein weiteres Element am Schluss der Liste (also als neues letztes Element) hinzu (Abbildung 86). Ähnlich funktioniert die Methode **.extend()**, nur in diesem Fall wird eine Liste an Elementen der ursprünglichen Liste an ihrem Ende hinzugefügt. Die letzte der Methoden, die Datensätze in die Listen einfügen, ist **.insert()**. Diese fügt ein Element an der als Parameter angegebenen Position in die Liste ein.

```
bauwerk = ["Haus"]
print(bauwerk)
bauwerk.append("Schule")
print(bauwerk)
bauwerk.extend(["Tempel", "Monument"])
print(bauwerk)
bauwerk.insert(1, "Garage")
print(bauwerk)
```

```
['Haus']
['Haus', 'Schule']
['Haus', 'Schule', 'Tempel', 'Monument']
['Haus', 'Garage', 'Schule', 'Tempel', 'Monument']
```

Abbildung 86 Elemente hinzufügen

Format:

`liste.append(Element)`

`liste.extend(liste)`

`liste.insert(Position, Element)`



### 2.3.11.8 Elemente entfernen

Listen können nicht nur wachsen, sondern auch schrumpfen. Für dieses Verhalten sind zwei weitere Listen Methoden zuständig. Die Methode **.pop()** löscht das Element mit dem angegebenen Index, die Methode **.remove()** hingegen das erste Vorkommen des angegebenen Elements. Ist das angegebene Element mehrfach in der Liste enthalten, bleiben nach dem **.remove()** alle anderen Instanzen des Elements, bis auf die erste, erhalten.

```
print(bauwerk)
bauwerk.pop(1)
print(bauwerk)
bauwerk.remove("Schule")
print(bauwerk)
```

```
['Haus', 'Garage', 'Schule', 'Tempel', 'Monument']
['Haus', 'Schule', 'Tempel', 'Monument']
['Haus', 'Tempel', 'Monument']
```

Abbildung 87 Elemente aus einer Liste entfernen

Format:

liste.**pop**(Position)

liste.**remove**(Element)

### 2.3.11.9 Liste kopieren

Die letzte der Listenmethoden **.copy()** erstellt eine vollständige 1 zu 1 Kopie der Ursprungsliste.

Format:

neueListe = liste.**copy**()

```
objekt=bauwerk.copy()
print(objekt)
```

```
['Haus', 'Schule', 'Tempel', 'Monument']
```

Abbildung 88: Liste kopieren

## 2.3.12 Dictionaries

Der hauptsächliche Unterschied zwischen den Dictionaries und den anderen Kollektionstypen ist, das Dictionaries immer zwei zusammenhängende Wertepaare speichern. Der eine Wert ist der „Schlüssel“ und der andere Wert ist der dazugehörige Wert.

### 2.3.12.1 Dictionaries definieren

Ein Dictionary wird als eine Folge von Schlüssel und Werten definiert (Abbildung 89). Der Schlüssel und Wert sind dabei durch einen Doppelpunkt getrennt.

```
frucht1 = {
    "Frucht" : "Orange",
    "Gewicht" : 160,
    "Geschmack": "süß"}

print(frucht1)
```

`{'Frucht': 'Orange', 'Gewicht': 160, 'Geschmack': 'Süß'}`

Abbildung 89 Ein Dictionary definieren

Eine weitere Möglichkeit ein Dictionary zu definieren stellt die Methode **.fromkeys()** zur Verfügung. Damit lassen sich Dictionaries mit vordefinierten Werten definieren (Abbildung 90)

```
keys = ('Frucht', 'Gewicht', 'Geschmack')
default = 42

frucht = dict.fromkeys(keys, default)
print(frucht)
```

`{'Frucht': 42, 'Gewicht': 42, 'Geschmack': 42}`

Abbildung 90 Dictionary aus Indexenliste und Defaultwert definieren

Eine weitere Möglichkeit, die nicht unerwähnt bleiben sollte, Dictionaries zu definieren ist es diese über die `zip` Funktion zu definieren (Abbildung 91). Dabei werden zwei Listen oder Tupels bzw. eine Kombination aus beiden mit der Klasse `zip` „gepackt“ und als Eingabe für die Erstellung eines Dictionaries mit der Funktion **dict()** verwendet.

```
index = ['Frucht', 'Gewicht', 'Geschmack']
values = ['Orange', 160, 'süß']
values2 = ['Zitrone', 120, 'sauer']

frucht1=dict(zip(index, values))
frucht2=dict(zip(index, values2))

print (frucht1)
print (frucht2)

{'Frucht': 'Orange', 'Gewicht': 160, 'Geschmack': 'süß'}
{'Frucht': 'Zitrone', 'Gewicht': 120, 'Geschmack': 'sauer'}
```

Abbildung 91: Dictionarys über die zip klasse definieren

### 2.3.12.2 Werte auslesen

Es gibt zwei Wege die Werte eines Dictionary auszulesen (Abbildung 92). Zuerst kann dies mit der Methode **.get()** mit Angabe des gewünschten Schlüssels als Parameter erfolgen. Die Methode liefert dann den entsprechenden Wert zurück. Die zweite Möglichkeit ist es direkt über den Schlüssel als Index auf den Wert des Dictionarys zuzugreifen.

```
print("Das Gewicht einer ", end="")
print(frucht1.get('Frucht'), end=" ist ")
print(frucht1['Gewicht'])
```

Das Gewicht einer Orange ist 160

Abbildung 92 Werte eines Dictionary auslesen

### 2.3.12.3 Wert eines Schlüssels ändern

Der Wert, der einem Schlüssel zugeordnet ist, lässt sich auf zwei verschiedene Arten ändern wie in der Abbildung 93 zu sehen ist. Einerseits lässt sich dieser über einen direkten Zugriff ändern, indem der Schlüssel als Index verwendet wird, bewerkstelligen. Andererseits kann der Wert auch über die Methode **.update()** unter der Angabe des Wertepaares als Parameter geändert werden. Ist der Schlüssel nicht vorhanden, wird in beiden Fällen ein neuer Schlüssel in dem Dictionary angelegt.

```
print (frucht1)
frucht1["Farbe"] = "Orange"
print (frucht1)

frucht1.update({"Gewicht" : 165})
print (frucht1)

{'Frucht': 'Orange', 'Geschmack': 'süß'}
{'Frucht': 'Orange', 'Geschmack': 'süß', 'Farbe': 'Orange'}
{'Frucht': 'Orange', 'Geschmack': 'süß', 'Farbe': 'Orange', 'Gewicht': 165}
```

Abbildung 93: Wert eines Schlüssels Ändern

### 2.3.12.4 Dictionary leeren

Mit der Methode `.clear()` können Sie alle Inhalte eines Dictionary auf einmal löschen (Abbildung 94). Es bleibt nur ein leeres Dictionary übrig.

```
print(frucht1)
frucht1.clear()
print(frucht1)

{'Frucht': 'Orange', 'Gewicht': 160, 'Geschmack': 'süß'}
{}
```

Abbildung 94: ein Dictionary Leeren

### 2.3.12.5 Dictionary kopieren

Mit der Methode `.copy()` kann eine Kopie eines Dictionarys erstellt werden.

```
frucht2 = frucht1.copy()
frucht2["Gewicht"] = 200
print (frucht1)
print (frucht2)

{'Frucht': 'Orange', 'Gewicht': 160, 'Geschmack': 'süß'}
{'Frucht': 'Orange', 'Gewicht': 200, 'Geschmack': 'süß'}
```

Abbildung 95 Ein Dictionary kopieren

Bitte beachten Sie, dass der Zuweisungsoperator (=) keine Kopie eines Dictionarys erstellt, sondern nur eine weitere Referenz auf dieses Dictionary erzeugt. So führt eine Änderung an einer der beiden Referenzen zwangsläufig zu Änderung der „beiden“ Dictionarys (Abbildung 96).

```
frucht2 = frucht1
print (frucht1)
frucht2["Gewicht"] = 200
print (frucht1)

{'Frucht': 'Orange', 'Gewicht': 160, 'Geschmack': 'süß'}
{'Frucht': 'Orange', 'Gewicht': 200, 'Geschmack': 'süß'}
```

Abbildung 96 Ein Dictionary mehrfach referenzieren

### 2.3.12.6 Auflistung aller Schlüssel erstellen

Eine Auflistung mit allen Schlüsseln in einem Dictionary kann über die Methode **.keys()** erstellt werden (Abbildung 97). Das erstellte Objekt ist keine Liste sondern nur ein View Objekt.

```
print(frucht1.keys())
dict_keys(['Frucht', 'Gewicht', 'Geschmack'])
```

Abbildung 97 Schlüsselliste erstellen

### 2.3.12.7 Auflistung aller Werte erstellen

Genauso wie die Auflistung aller Schlüssel lässt sich mit der Methode **.values()** eine Auflistung aller Werte in einem Dictionary erstellen (Abbildung 98). Das erstellte Objekt ist keine Liste sondern nur ein View Objekt.

```
print (frucht1.values())
dict_values(['Orange', 160, 'süß'])
```

Abbildung 98 Werteliste erstellen

### 2.3.13 Auflistung alle Wertepaare erstellen

Wird eine Auflistung aller Schlüssel mit den zugehörigen Werten benötigt, so lässt sich diese mit der Methode `.items()` erstellen (Abbildung 99). Auch hier handelt es sich bei dem erstellten Objekt nicht um eine Liste, sondern um ein View.

```
print (frucht1.items())
```

```
dict_items([('Frucht', 'Orange'), ('Gewicht', 160), ('Geschmack', 'süß')])
```

Abbildung 99 Auflistung aller Wertepaare in einem Dictionary

#### 2.3.13.1 Views `dict_keys`, `dict_values` und `dict_items`

Bei den Typen `dict_keys`, `dict_values` sowie `dict_items` handelt es sich streng genommen nicht um echte Listen, sondern um sogenannte Views. Für einen Zugriff auf die einzelnen Elemente müssen diese zuerst zu einer Liste umgewandelt werden (Abbildung 100).

```
items = frucht1.items()
print (frucht1.items())
```

```
liste = list(items)
print(liste[0][0],liste[0][1] )
```

```
dict_items([('Frucht', 'Orange'), ('Gewicht', 160), ('Geschmack', 'süß')])
Frucht Orange
```

Abbildung 100 `dict_items` in eine Liste umwandeln

#### 2.3.13.2 Wert Auslesen

Der Wert eines Schlüssels kann direkt über den Schlüssel als Index direkt ausgelesen werden (Abbildung 101).

```
print(frucht1["Gewicht"])
```

```
160
```

Abbildung 101 Wert auslesen

#### 2.3.13.3 Wert holen

Über die Methode `.pop()` lässt sich der Wert eines Schlüssels holen (Abbildung 102). Der Wert und dessen Schlüssel werden dabei aus dem Dictionary entfernt. Zusätzlich dazu lässt sich bei der `.pop()`

Methode ein default Wert bestimmen, der, falls ein entsprechender Schlüssel nicht gefunden wurde, dann zurückgegeben wird. Ist der entsprechende Schlüssel nicht vorhanden und ist durch die Pop() Methode kein default Wert definiert, meldet die Methode .pop() einen Fehler.

```
print (frucht1)
gewicht = frucht1.pop("Gewicht")
print (frucht1)
print(gewicht)
gewicht = frucht1.pop("Gewicht", 100)
print(gewicht)
```

{'Frucht': 'Orange', 'Gewicht': 160, 'Geschmack': 'süß'}  
{'Frucht': 'Orange', 'Geschmack': 'süß'}  
160  
100

Abbildung 102 Wert aus einem Dictionary holen

### 2.3.13.4 Letztes Wertepaar holen

Bei der Methode **.popitem()** wird das gesamte letzte Wertepaar, also der Schlüssel und der dazugehörige Wert geholt (Abbildung 103). Beides wird gleichzeitig auch aus dem Dictionary entfernt.

```
print(frucht1)
print(" geholt      :",frucht1.popitem())
print(" verblieben :", frucht1)
```

{'Frucht': 'Orange', 'Gewicht': 160, 'Geschmack': 'süß'}  
geholt : ('Geschmack', 'süß')  
verblieben : {'Frucht': 'Orange', 'Gewicht': 160}

Abbildung 103 Letztes Wertepaar holen

### 2.3.13.5 Default Wert setzen, Wert auslesen

Die Methode **.setdefault()** übernimmt zwei Aufgaben. Als erste Aufgabe wird der Wert des als Parameter übergebenen Schlüssels, falls dieser nicht gesetzt ist, auf dem als zweitem Parameter übergebenen default Wert gesetzt. Als zweite Aufgabe der Methode wird der aktuelle Wert des Schlüssels zurückgegeben (Abbildung 104).

```
print (frucht1)
print (frucht1.setdefault("Farbe" , "orange"))
print (frucht1)
print (frucht1.setdefault("Farbe" , "gelb"))
print (frucht1)

{'Frucht': 'Orange', 'Gewicht': 160, 'Geschmack': 'süß'}
orange
{'Frucht': 'Orange', 'Gewicht': 160, 'Geschmack': 'süß', 'Farbe': 'orange'}
orange
{'Frucht': 'Orange', 'Gewicht': 160, 'Geschmack': 'süß', 'Farbe': 'orange'}
```

Abbildung 104 Default Wert setzten und Wert auslesen

## 2.4 Views

## 2.5 IF...elif...else

Die Anweisung **if** neben den unterstützenden Anweisungen **else** sowie **elif** dient zur Implementierung bedingter Programmausführung.

### 2.5.1 Die if-Anweisung

Die **if**-Anweisung prüft, ob der rechts vor ihr stehende Ausdruck als Wahr gewertet werden kann (siehe Kapitel 2.2.8 Booleans). Ist es der Fall, so wird die folgende Codesequenz ausgeführt (Abbildung 105). Sollte der Ausdruck als Falsch gewertet werden, wird die folgende Codesequenz bis zum Ende des **if**-Blocks oder einer **else** bzw. **elif**-Anweisung übersprungen.

```
a =5
b =4

if (a>b):
    print ("a ist größer als b")

a ist größer als b
```

Abbildung 105 Die if Anweisung



## 2.5.2 Else

Wurde die Auswertung der **if**-Abfrage als Unwahr / Falsch gewertet, so kann dies mit der **else**-Anweisung abgefangen werden, um den Fall entsprechend weiter zu verarbeiten (Abbildung 106). Da **else** sonst keine Parameter hat, wird der auf **else** folgende Codeblock bei einer fehlgeschlagenen **if**-Abfrage auf jeden Fall ausgeführt.

```
a = 5
b = 5

if (a > b):
    print ("a ist größer als b")
else:
    print ("a ist nicht größer b")
```

a ist nicht größer b

Abbildung 106 Die else Anweisung

## 2.5.3 Die elif-Anweisung

Die Anweisung **elif** stellt eine Kombination aus einem **else** gefolgt von einem weiteren **if** dar. Sie wird ausgeführt wenn das Ergebnis der Bedingung der vorausgegangenen **if**-Abfrage als nicht wahr gewertet wurde, aber noch weitere Fälle abgefragt bzw. unterschieden werden sollten. Sie kann auch mehrfach hintereinander eingesetzt werden um weitere Fälle abdecken (Abbildung 107). Da Python keine switch..case Anweisungen kennt, ist das die einzige Möglichkeit Mehrfallabfragen zu implementieren.

```
a = 5
b = 6

if (a > b):
    print ("a ist größer als b")
elif (a < b):
    print ("a ist kleiner als b")
elif (a == b):
    print ("a ist gleich b")
```

a ist kleiner als b

Abbildung 107: Die elif Anweisung

## 2.5.4 Verschachtelung

Die if-Abfragen lassen sich verschachteln, um somit die abfragen „feiner“ zu machen. (Abbildung 108) Zu beachten ist, dass so etwas ab einer bestimmten Größe leicht unübersichtlich wird. So ist es zu empfehlen bei größeren Verschachtelungstiefe diese in Funktionen auszulagern.

```
a = 8
b = 5

if (a > b):
    if (a > (b+2)):
        print ("a ist viel größer als b")
    else:
        print ("a ist größer als b")
elif (a < b):
    print ("a ist kleiner als b")
else:
    print ("a ist gleich b")
```

a ist viel größer als b

Abbildung 108 Vierschachtelung bei if Abfragen

## 2.6 Schleifen

Python kennt grundsätzlich zwei Typen von Schleifen, im Python-Jargon auch Loops genannt. Eine **for**, sowie die **while**-Schleife. Sie dienen dazu eine Codesequenz wiederholt auszuführen.

### 2.6.1 Die For-Schleife

Die Implementierung der **for**-Schleife weicht erheblich von deren in anderen Programmiersprachen ab. In Python ist die Anweisung **for** eher ein Iterator, um einzelne Objekte einer Kollektion bearbeiten zu können. Die normale Funktionalität einer aus C/C++ oder Java bekannter for Schleife, ist aber auch über einem Umweg möglich.

#### 2.6.1.1 Else

Ja, tatsächlich lässt sich in Python eine **for**-Schleife mit einer **else**-Anweisung beenden. Diese wird, solange dies nicht durch eine aktive **break**-Anweisung verhindert wird, automatisch nach der letzten Iteration der **for**-Schleife ausgeführt.

```
for zahl in range(0, 10, 3):
    print(zahl, end = ' ')
else:
    print(';')
```

0 3 6 9 ;

Abbildung 109 if..else

#### 2.6.1.2 Break

Mit der Break-Anweisung lässt sich die Abarbeitung einer **for**-Schleife abbrechen (Abbildung 110). Wenn, wie in diesem Beispiel zu sehen ist, ein Objekt „Tempel“ in der Liste vorgefunden wird, wird die Verarbeitung der Liste sofort abgebrochen. Zu beachten ist, dass die Anweisung **else** in diesem Fall auch nicht mehr ausgeführt wird.

```
bauwerk = ["Haus", "Monument", "Tempel", "Haus"]
for objekt in bauwerk:
    print(objekt)
    if (objekt == "Tempel"):
        break
else:
    print("ende")
```

Haus  
Monument  
Tempel

Abbildung 110: for...break

### 2.6.1.3 Continue

Die **continue**-Anweisung in einer for-Schleife sorgt dafür, dass die Abarbeitung des aktuellen Schritts abgebrochen wird und stattdessen mit dem nächsten Schritt in der for-Schleife fortgefahren wird. In der Abbildung 111 wird die Ausgabe mit print() nur dann getätigt wenn das aktuelle Objekt nicht „Tempel“ ist, da bei allen anderen Objekten die Weiterverarbeitung der aktuellen Schleifeniteration durch die **continue**-Anweisung abgebrochen wird. **Continue** hat keine Auswirkung auf die Ausführung einer **else** Anweisung in einer **for** Schleife.

```
bauwerk = ["Haus", "Monument", "Tempel", "Haus"]
for objekt in bauwerk:
    if (objekt != "Tempel"):
        continue
    print(objekt)
```

Tempel

Abbildung 111 for..continue

### 2.6.1.4 Pass

Ein unglaublich wichtiger Befehl für die Schleifendurchführung ist die **pass**-Anweisung. Sie sorgt lediglich dafür dass eine **for**-Schleife ohne Rumpf definierbar ist (Abbildung 112). Es ist aber nicht so, dass Range hier nichts macht, es generiert seine Zahlen trotzdem weiter, es wird damit aber nichts gemacht. Es könnte aber trotzdem hier eine Methode verwendet werden, die ein gültiges Ergebnis für das **for** liefert und bereits bei Aufruf sinnvolle Arbeit verrichtet.

```
for value in range(0,3):
    pass

print("for hat nichts gemacht")
```

for hat nichts gemacht

Abbildung 112 for..pass

### 2.6.1.5 Range

Ein Objekt der Klasse **Range** ist der Umweg, mit dem es möglich ist, zumindest teilweise, die „normale“ Funktionsweise der **for**-Schleife in Python zu implementieren (Abbildung 113). Ein Objekt dieser Klasse ist ein einfacher Nummerngenerator für ganzzahlige Zahlen. Es hat genau 3 Eigenschaften:

- Start – ist die Anfangszahl
- Stop – ist die Endzahl (Exklusive)
- Step – legt die Schrittgröße mit der die Zahlen generiert werden.

Diese drei Attribute sind konstant, das heißt, nach dem ein Objekt der Klasse Range erstellt wurde lassen sich diese

```
menge = range(0,9,2)
print (type(menge))
print (menge)
print ("start : ",menge.start)
print ("stop : ",menge.stop)
print ("step : ",menge.step)
```

```
<class 'range'>
range(0, 9, 2)
start : 0
stop : 9
step : 2
```

Abbildung 113 Ein Range Objekt

Attribute nicht mehr ändern. Der Vorteil von **Range** gegenüber von Listen oder Sets ist, dass **Range** die Zahlen erst „bei Gebrauch“ generiert, was sich in einem deutlich geringeren Speicherverbrauch äußert.

Die Klasse **Range** ist nicht nur auf Generierung von steigenden positiven Zahlenfolgen eingeschränkt. Alle drei Attribute der Klasse können auch negative Werte aufnehmen (Abbildung 114). So kann z.B. die Zahlenfolge von Negativen Zahlen ausgehend gebildet werden, oder bei einem Negativen Schritt, kann die die Zahlenbildung rückwärts erfolgen. Der Wirkungsbereich der Klasse **Range** beschränkt sich nicht nur auf die Zusammenarbeit mit der Funktion **for**, da die Klasse **Range** auch verwendet werden kann um z.B. Listen, oder Sets zu definieren (Abbildung 115).

```
liste = list(range(1,5,1))
zahlen = set(range(1,5,1))
print("Liste = ",liste)
print("Set   = ",zahlen)
```

```
Liste = [1, 2, 3, 4]
Set   = {1, 2, 3, 4}
```

Abbildung 115 Weitere Anwendungen der Klasse Range

```
print ("steigend :",end = ' ')
for index in range(-5,4,1):
    print (index ,end = ' ')
else:
    print('')

print ("fallend :",end = ' ')
for index in range(14,2,-2):
    print (index ,end = ' ')
```

```
steigend : -5 -4 -3 -2 -1 0 1 2 3
fallend : 14 12 10 8 6 4
```

Abbildung 114 Negative Attribute in einem Range Objekt

### 2.6.1.6 Bereiche durchgehen oder **for (i = 0; i<x; i+=y) in Python**

Nun ist es auch möglich mit der **for**-Schleife auch indiziert zu Arbeiten. Zu diesem Zweck wird mit dem Objekt Range ein Iterator gebaut so das **for** Anweisung was zu iterieren hat. (Abbildung 116).

Die Angaben für das **Range**-Objekt müssen nicht „festkodiert“ sein, die können genauso gut in Variablen oder aus Rückgabewerten von Funktionen kommen (Abbildung 117). Zu bedenken ist, dass hier im unterschied zu C/C++ nur mit Ganzzahlen gearbeitet werden kann, da das **Range**-Objekt nur mit Ganzzahlen umgehen kann. Bedenken Sie das nach der Definition das **Range**-Objekt nicht mehr veränderbar ist.

```
for zahl in range(0, 10, 3):
    print(zahl)
else:
    print('ende')
```

```
0
3
6
9
ende
```

Abbildung 116 **for i =0; i< 10;i+=3**

```
start = 0
ende = 10
schritt = 4
for zahl in range(start, ende, schritt):
    print(zahl)
```

0  
4  
8

Abbildung 117 Range mit Variablen.

### 2.6.1.7 Strings Iterieren

Eine nette Eigenschaft von Python ist die Fähigkeit Zeichenketten als iterable Objekte zu betrachten. Somit lassen sich mit einer *for*-Anweisung Zeichenketten zeichenweise durchwandern (Abbildung 118). Eine Indizierung ist, zumindest wenn die gesamte Zeichenkette durchwandert werden muss, nicht notwendig. Sollte aber nur ein Teil der Zeichenkette bearbeitet werden, dann ist wiederum arbeiten mit Index notwendig (Abbildung 119). Alternativ könnte hier zuerst eine neue Zeichenkette generiert werden, die nur die gewünschten Zeichen enthält.

```
for zeichen in "Kette":
    print(zeichen)
```

K  
e  
t  
t  
e

Abbildung 118: Zeichenketten iterieren

```
kette = "Kette"
for zeichen in range(1,4):
    print(kette[zeichen])
```

e  
t  
t

Abbildung 119: Zeile einer Zeichenkette iterieren

## 2.6.2 while Schleife

Bei einer **while**-Schleife in Python wird die unterhalb der Schleife definierte Codesequenz so lange ausgeführt wie die boolesch ausgewertete Bedingung zutrifft. Wird die die Bedingung nicht mehr als wahr (*True*) ausgewertet, wird die Schleife verlassen (Abbildung 120). Bitte beachten Sie, dass wenn der Prüfausdruck eine Funktion enthält, diese auch jedes mal neu ausgewertet wird. Somit lassen sich, bei ungeschickter Programmierung, spielend leicht Endlosschleifen bauen (Abbildung 121). Wie Sie in der Abbildung sehen können hat das eine, einzige lausige **print()** Anweisung, die in einer, vermutlich ungewollten, Dauerschleife lief, den

```
boolean = True
value = 0

while boolean:
    if (value > 1):
        boolean = False
    print (value)
    value +=1
```

```
0
1
2
```

Ausgabepuffer des Jupyter Notebooks mit Ausgabedaten überflutet. Das Problem bei Implementierung der Schleife in der Abbildung 121 ist, dass die Länge der Liste bei jeder Abfrage immer um eins größer ist als der Inhalt des Variablen-Index.

```
zahlen = ['0']
index = 0
while index < len(zahlen):
    index+=1
    zahlen.append(index)
    print(zahlen)
```

```
IOPub data rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_data_rate_limit`.
```

```
Current values:
NotebookApp.iopub_data_rate_limit=1000000.0 (bytes/sec)
NotebookApp.rate_limit_window=3.0 (secs)
```

Abbildung 121: Endlose while schleife

### 2.6.2.1 *else*

Genauso wie bei der **for**-Schleife kann auch die **while**-Schleife eine **else**-Anweisung enthalten (Abbildung 122). Auch hier wird diese automatisch einmal ausgeführt, wenn die Bedingung nicht mehr zutrifft.

```
bauwerk = ["Haus", "Monument", "Tempel", "Haus"]
index = 0
while index < len(bauwerk):
    print(bauwerk[index], end ="; ")
    index +=1
else:
    print("\nDas waren alle Bauwerke")
```

```
Haus; Monument; Tempel; Haus;
Das waren alle Bauwerke
```

Abbildung 122 while ...else

### 2.6.2.2 *While...break*

Genauso wie bei der **for** - Anweisung gibt es auch bei der **while** - Schleife die Möglichkeit, die Bearbeitung der Schleife vorzeitig mit der **break** - Anweisung abubrechen (Abbildung 123). Sollte auf die **while** - Anweisung eine **break** - Anweisung folgen, so wird diese bei einem aktiviertem **break** nicht ausgeführt.

```
bauwerk = ["Haus", "Monument", "Tempel", "Haus"]
index = 0
while index < len(bauwerk):
    print(bauwerk[index], end ="; ")
    if bauwerk[index] == "Monument":
        break
    index +=1
else:
    print("\nalle Bauwerke bis zum Monument")
```

```
Haus; Monument;
```

Abbildung 123 break bei while

### 2.6.2.3 While...continue

Auch das **while...continue** - Konstrukt kann in Python verwendet werden (Abbildung 124). In diesem Beispiel werden unerwünschte Einträge aus einer Kopie einer Liste entfernt. Es sollte an dieser Stelle nicht unerwähnt bleiben, dass dieses Verfahren eher an C/C++ als an Python erinnert, da man es in Python eigentlich deutlich eleganter lösen kann (Abbildung 23).

```
bauwerk = ["Haus", "Monument", "Tempel", "Garage", "Haus"]
neu = []
key = "Haus"
index = 0
while index < len(bauwerk):
    index += 1
    if bauwerk[index-1] == key:
        continue
    neu.append(bauwerk[index-1])
else:
    print(neu, "\nalle Bauwerke außer", filter)
```

```
['Monument', 'Tempel', 'Garage']
alle Bauwerke außer Haus
```

Abbildung 124 filtern mit continue

```
bauwerk = ["Haus", "Monument", "Tempel", "Garage", "Haus"]
neu = bauwerk.copy()
key = "Haus"

while neu.count(key):
    neu.remove(key)

else:
    print(neu, "\nalle Bauwerke außer", key)
```

```
['Monument', 'Tempel', 'Garage']
alle Bauwerke außer Haus
```

Abbildung 125 schöner Filtern mit Python

### 2.6.2.4 While...pass

Zum Schluss lernen wir noch wie man mit **while...pass** eine perfekte Endlosschleife bauen kann (Abbildung 126). Diese Schleife läuft solange, bis sie jemand manuell mit <control>+<c> abbricht.

```
while True:
    pass
```

```
-----
KeyboardInterrupt                                Traceback (most recent call last)
<ipython-input-27-b7133701d76c> in <module>
      1 while True:
----> 2     pass
```

```
KeyboardInterrupt:
```

Abbildung 126: Die Mutter aller Endlosschleifen



## 2.7 Funktionen

### 2.7.1 Funktionen dokumentieren

Als erste „Operation“ in einer Funktion kann eine Zeichenkette definiert sein (Abbildung 127). Diese dient dann als Dokumentation der Funktion. Über die eingebaute Eigenschaft `__doc__`, die in jeder Funktion vorhanden ist, kann diese über den `print()` Befehl angezeigt werden. So auch die Standard bzw. Bibliotheksfunktionen (Abbildung 128).

```
def funktion(boolean):
    """ Funktion macht folgendes:
        noch nix """
    pass
```

```
print(funktion.__doc__)
```

```
Funktion macht folgendes:
    noch nix
```

Abbildung 127: Funktionen Dokumentieren

```
print(print.__doc__)
```

```
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.
```

Abbildung 128 Dokumentation der `print()`-Funktion

### 2.7.2 Parameter einer Funktion

Ob ein Parameter als *call by value* oder *call by reference* hängt ganz alleine von dessen Typ ab. Alle Datentypen die **immutable** sind werden als *call by value* Parameter verwendet, alle **mutable** Datentypen hingegen als *call by reference* verarbeitet. Somit ist vor allem bei den Mutable Objekten zu berücksichtigen das diese in der Funktion geändert werden können. Falls dies nicht ausdrücklich erwünscht ist, dann sollte eine Kopie an die Funktion übergeben werden.

### 2.7.2.1 *Immutable Parameter*

Bei Parametern die der Klasse Immutable gehören, behält der Parameter bis zu der ersten Änderung seine Adresse (Abbildung 129). Wird dieser in einer Funktion geändert, dann wird eine neue Instanz der entsprechenden Klasse erstellt, die dann den geänderten Wert aufnimmt. Bei der Rückgabe wird die neue Instanz, in der die geänderte Daten gespeichert sind, zurückgegeben. Bei der Zuweisung des Rückgabewertes wird die alte Instanz durch die neue Instanz ersetzt.

```
def funktion(zahl):  
    print("zahl in der Funktion      :", zahl, " ; id=", id(zahl))  
    zahl *= 5  
    print("zahl in der Funktion * 5 :", zahl, " ; id=", id(zahl))  
  
    return(zahl)  
  
zahl = 5  
print("zahl vor der Funktion      :", zahl, " ; id=", id(zahl))  
zahl = funktion(zahl)  
print("zahl nach der Funktion     :", zahl, " ; id=", id(zahl))  
  
zahl vor der Funktion      : 5 ; id= 140703308711712  
zahl in der Funktion       : 5 ; id= 140703308711712  
zahl in der Funktion * 5   : 25 ; id= 140703308712352  
zahl nach der Funktion     : 25 ; id= 140703308712352
```

Abbildung 129 Funktion mit einem immutable Parameter

### 2.7.2.2 Mutable Parameter

Ist der übergebene Parameter Mutable, dann wird dieser Parameter als **call by reference** betrachtet (Abbildung 130). In diesem Fall wird auf dem übergebenen Parameter gearbeitet und auch dieser

```
def funktion(liste):
    print("liste in der Funktion      :", liste, "      ; id=", id(liste))
    liste *= 2
    print("liste in der Funktion * 2 :", liste, "; id=", id(liste))

    return(liste)

liste = [1,2]
print("liste vor der Funktion      :", liste, " ;      id=", id(liste))
liste = funktion(liste)
print("liste nach der Funktion     :", liste, "; id=", id(liste))

liste vor der Funktion      : [1, 2] ;      id= 1539539337984
liste in der Funktion       : [1, 2] ; id= 1539539337984
liste in der Funktion * 2   : [1, 2, 1, 2] ; id= 1539539337984
liste nach der Funktion     : [1, 2, 1, 2] ; id= 1539539337984
```

Abbildung 130 Funktion mit einem mutable Parameter

wird dann zurückgegeben, sodass hier eine Rückgabe des Parameters an sich meistens nicht notwendig ist (Abbildung 131), da der Aufrufer bereits eine Referenz darauf besitzt.

```
def funktion(liste):
    print("liste in der Funktion      :", liste, "      ; id=", id(liste))
    liste *= 2
    print("liste in der Funktion * 2 :", liste, "; id=", id(liste))

liste = [1,2]
print("liste vor der Funktion      :", liste, " ;      id=", id(liste))
funktion(liste)
print("liste nach der Funktion     :", liste, "; id=", id(liste))

liste vor der Funktion      : [1, 2] ;      id= 1539538199744
liste in der Funktion       : [1, 2] ; id= 1539538199744
liste in der Funktion * 2   : [1, 2, 1, 2] ; id= 1539538199744
liste nach der Funktion     : [1, 2, 1, 2] ; id= 1539538199744
```

Abbildung 131 Funktion mit einem mutable Parameter ohne Rückgabewert

Sollte aber der Original-Parameter auch nach dem Funktionsaufruf unverändert bleiben, so ist in diesem Fall nicht das Originalobjekt, sondern eine Kopie davon zu übergeben (Abbildung 132). Dabei ist zu beachten, dass diese Kopie nach dem Rücksprung der Funktion auch „abgeholt“ und in einer (neuen) Variable gespeichert wird. Alternativ ist die Funktion so zu designen, dass diese den Originalparameter zuerst kopiert und so die Funktion nur die Kopie verändert.

```
def funktion(liste):
    print("liste in der Funktion      :", liste, "      ; id=", id(liste))
    liste *= 2
    print("liste in der Funktion * 2 :", liste, "; id=", id(liste))

    return(liste)

liste = [1,2]
print("liste vor der Funktion      :", liste, " ;      id=", id(liste))
liste2 = funktion(liste.copy())
print("liste nach der Funktion      :", liste, ";      id=", id(liste))
print("liste2 nach der Funktion      :", liste2, "; id=", id(liste2))

liste vor der Funktion      : [1, 2] ;      id= 1539539337728
liste in der Funktion      : [1, 2]      ; id= 1539538894592
liste in der Funktion * 2 : [1, 2, 1, 2] ; id= 1539538894592
liste nach der Funktion      : [1, 2] ;      id= 1539539337728
liste2 nach der Funktion      : [1, 2, 1, 2] ; id= 1539538894592
```

Abbildung 132 Funktion mit einem mutable Parameter mit neuem Rückgabewert

### 2.7.3 Parameter mit optionaler Vorgabe

Damit eine Funktion immer eine korrekte Arbeitsbasis aufweist, lassen sich Parameter vorgeben. Dieser vorgegebene Wert wird dann eingesetzt, falls der entsprechende Parameter bei der Übergabe fehlt (Abbildung 133).

```
def funktion(bauwerk = "Haus"):
    print("Bauwerk ist", bauwerk)

funktion()
funktion("Halle")
```

```
Bauwerk ist Haus
Bauwerk ist Halle
```

Abbildung 133 Default Parameter

## 2.7.4 Funktionen mit optionalen Argumenten

Eine Funktion kann auch statt einem eine unbestimmte Zahl an Parametern entgegennehmen. In diesem Fall ist die Variable Parameterliste mit einem Stern „\*“ gekennzeichnet (Abbildung 134). Intern wird aus den übergebenen Parametern ein Tupel erstellt, das dann auch wie jeder andere Tuple entweder iteriert oder über Indizes bearbeitet werden kann.

```
def function(*bauerke):
    print(type(bauerke))
    for bauerk in bauerke:
        print(bauerk)
    print(bauerke[1])

function("Haus",2,"Tempel","Haus")
```

```
<class 'tuple'>
Haus
2
Tempel
Haus
2
```

Abbildung 134 Optionale Parameter

## 2.7.5 Funktionen mit Schlüsselwörtern

Die Parameter können nicht nur „blind“ übergeben werden, im Vertrauen darauf, dass die Reihenfolge korrekt ist, sondern die Namen der Parameter können bei der Übergabe als Schlüsselwörter verwendet werden (Abbildung 135). Sollten keine Defaultparameter vorgegeben sein, müssen alle Parameter im Aufruf angegeben werden. Eine weitere Einschränkung ist, dass kein Parameter ohne Vorgabe einem mit Vorgabe folgen darf.

```
def function(tier1, tier2, tier3="Hörnchen"):
    print (tier3)

function(tier1="Katze",tier2="Maus",tier3="Affe")
function(tier1="Katze",tier2="Maus")

Affe
Hörnchen
```

Abbildung 135: Parameternamen als Schlüsselwörter

## 2.7.6 Funktionen mit Schlüsselwörtern und variablen Parametern

Eine weitere Möglichkeit Parameter über Schlüsselwörter zu übergeben, wenn die Anzahl der Parameter nicht bekannt ist, wird unter Zuhilfenahme der Dictionarys durchgeführt.

```
def function(**tier):
    print (type(tier))
    print (tier["gattung"])

function(familie="Katzen",gattung="Hauskatze")
```

```
<class 'dict'>
Hauskatze
```

Abbildung 136 Funktionen mit Schlüsselwörtern und variablen Parameter.

Diese wird der Laufzeitumgebung durch die Verwendung von doppelten Sternchen (\*\*) signalisiert (Abbildung 136). Bei dieser Vorgehensweise baut die Laufzeitumgebung aus den Parametern ein Dictionary, das an die Funktion übergeben wird. Innerhalb der Funktion kann dann auf die Daten wie auf jedes andere Dictionary zurückgegriffen werden.

### 2.7.7 Rückgabeparameter

Der Typ des Rückgabeparameter einer Funktion wird erst bei dem Rücksprung mit der **return()**-Anweisung bestimmt. Somit ist es möglich, dass eine Funktion Werte unterschiedlichen Typs zurückliefert (Abbildung 137), je nachdem welche der **return()**-Anweisungen aktiv wird. Es kann durchaus sein, dass in dem einen oder anderen Fall diese sehr praktisch und nützlich ist, andererseits kann das auch zu Verwirrung führen und die Verständlichkeit der Funktion verringern.

```
def funktion(boolean):  
    if (boolean):  
        return(8.9)  
    return(8)  
  
value = funktion(True)  
print(type(value))  
  
value = funktion(False)  
print(type(value))  
  
<class 'float'>  
<class 'int'>
```

Abbildung 137  
Rückgabeparameter

### 2.7.8 Funktionen mit mehreren Rückgabeparametern

Grundsätzlich kann eine Funktion nur einen einzigen Parameter zurückgeben. Will man mehr aus einer Funktion „rausholen“, dann kann dieser Parameter auch ein Tupel oder eine Liste sein, da beide mehrere Daten aufnehmen können und diese geordnet und indiziert sind. (Abbildung 138). Von dem Versuch hier mit einem Set zu arbeiten sollte abgesehen werden, da dieser undefinierte Ergebnisse liefert – wir erinnern uns: Sets können keine Dubletten enthalten, sind nicht geordnet und auch nicht indiziert.

```
def funktion():  
    zahl = 5.6  
    index = 7  
    return ("Maus", index, zahl)  
  
ergebnis = funktion()  
print("erster Parameter :", ergebnis[0])  
print("zweiter Parameter :", ergebnis[1])  
print("dritter Parameter :", ergebnis[2])  
  
erster Parameter : Maus  
zweiter Parameter : 7  
dritter Parameter : 5.6
```

Abbildung 138: Funktion mit mehreren Rückgabeparametern

## 2.7.9 Templates und generische Funktionen

Da Python alle Variablen als dynamische Typen behandelt, sind auch Parameter der Funktionen in Python zuerst typfrei. Solange sich diese Parameter mit der in der Funktion verwendeten Logik verarbeiten oder kombinieren lassen, kann die Funktion als generische Funktion verwendet werden (Abbildung 139). Der Versuch numerische Werte mit Zeichenketten zu verrechnen scheitert allerdings, wie in Abbildung 140 leicht zu erkennen ist.

```
def add(x, y):
    return(x+y)

liste1=["a","b"]
liste2=["a","c"]
print(add(1,2),type(add(1,2)))
print(add(1,2.2),type(add(1,2.2)))
print(add("a","b"),type(add("a","b")))
print(add(liste1,liste2),type(add(liste1,liste2)))

3 <class 'int'>
3.2 <class 'float'>
ab <class 'str'>
['a', 'b', 'a', 'c'] <class 'list'>
```

Abbildung 139 Generische Funktionen in Python

```
def add(x, y):
    return(x+y)

print(add(1,"a"),type(add(1,"a")))
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-13-ce72335f5c81> in <module>
      2     return(x+y)
      3
----> 4 print(add(1,"a"),type(add(1,"a")))
```

```
<ipython-input-13-ce72335f5c81> in add(x, y)
      1 def add(x, y):
----> 2     return(x+y)
      3
      4 print(add(1,"a"),type(add(1,"a")))
```

**TypeError:** unsupported operand type(s) for +: 'int' and 'str'

Abbildung 140: Grenzen der generischen Programmierung in Python

### 2.7.10 Funktionen mit Funktionen als Parameter

Da in Python alles, auch Funktionen, Objekte sind, lassen sich auch Funktionen problemlos als Parameter in anderen Funktionen verwenden. In dem Beispiel in der Abbildung 141 wird, je nach Bedarf, eine unterschiedliche Funktion bei der Ausgabe der Temperatur als Parameter angegeben. Diese Funktion übernimmt sowohl die Umrechnung der Einheit als auch deren Ausgabe. Diese Methode lässt sich weiter „verfeinern“, indem man Funktionen in einer Kollektion unterbringt (Abbildung 142). Zum Abschluss gönnen wir uns noch einen kleinen Befehlsinterpreter (Abbildung 143), indem wir Befehle und dazugehörige Parameter in einer, oder

```
def asCelsius(temperature):
    print(temperature - 273.15, "°Celsius")

def asKelvin(temperature):
    t = (((temperature - 273.15) * 9) / 5) + 32
    print(t, "°Kelvin")

def Temperature(unit, temperature):
    unit(temperature)

temperature = 285.15
Temperature(asCelsius, temperature)
Temperature(asKelvin, temperature)

12.0 °Celsius
53.6 °Kelvin
```

Abbildung 141: Funktionen als Parameter in Funktionen

mehreren, Liste(n) Speichern - et voilà: fertig ist eine kleine virtuelle Maschine. Mit solchen Techniken ist es möglich, ein enAlgorithmus zu dessen Laufzeit zu konfigurieren ohne das Programm dafür umzuschreiben bzw. spezifische Funktionen für jeden Fall schreiben zu müssen.

```
temperature = 310.15
funktionen = {asCelsius, asKelvin}

for x in funktionen:
    x(temperature)
```

```
98.6 °Kelvin
37.0 °Celsius
```

Abbildung 142: Funktionen als Parameter in einem Set

```
def add(x,y):
    print(x, " + ", y, end = " = ")
    return(x+y)

def sub(x,y):
    print(x, " - ", y, end = " = ")
    return(x-y)

a = 4
befehle = [(add,5),(sub,1),(add,100)]
for x in befehle:
    a =x[0](a,x[1])
    print(a)
```

```
4 + 5 = 9
9 - 1 = 8
8 + 100 = 108
```

Abbildung 143: Kleiner Befehlsinterpreter in Python



## 2.7.11 Funktionen Kopieren, löschen und erweitern

Funktionen lassen sich kopieren dann über den neuen Alias aufrufen. Eine eigene Funktion kann auch zu jeder Zeit wieder gelöscht werden (Abbildung 144).

```
liste = [1,2,3,4,5]
laenge = len

print (laenge(liste))
del(laenge)
print (laenge(liste))
```

5

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-1-612cdf73d7a> in <module>
      4 print (laenge(liste))
      5 del(laenge)
----> 6 print (laenge(liste))

NameError: name 'laenge' is not defined
```

Abbildung 144: Funktionen kopieren und löschen

Die Frage ist, wozu das gut sein sollte? Dadurch lassen sich z.B. bereits vorhandene Funktionen um eigene Funktionalitäten erweitern. In der Abbildung 145 sehen Sie eine (mehr oder weniger sinnvolle) Erweiterung der Funktion **len()**. Zuerst wird ein Duplikat (**laenge**) der Funktion **len()** erstellt. Danach wird eine neue **len()** Funktion definiert: Eine, die die alte **len()** (jetzt **laenge()**) Funktion aufruft und zusätzlich dazu noch eine Ausgabe auf der Konsole macht. Inwiefern man sich mit solchen Spielchen bei den Arbeitskolleg\*innen beliebt macht, steht allerdings auf einem anderen Blatt. Dies kann z.B. Für das Debuggen sehr praktisch sein, allerdings kann es auch sehr verwirrend sein. Sollte also mit bedacht verwendet werden wenn überhaupt.

```
liste = [1,2,3,4,5]
laenge = len

def len(object):
    print("Länge : ", laenge(object))
    return(laenge(object))

print (len(liste))

Länge : 5
5
```

Abbildung 145: Vorhandene Funktionen erweitern

## 2.8 Iteratoren

Iteratoren sind Klassen, die einen sequentiellen Zugriff auf die mit dem Iterator verbundenen Daten erlauben. Dadurch ist es möglich, auf die von dem Iterator verwalteten Daten sequentiell zuzugreifen ohne die Struktur der Daten zu berücksichtigen bzw. zu kennen. Des Weiteren muss der Zeiger auf die Stelle, an der sich das Aktuelle/Folgende Element befindet, nicht von dem Benutzer verwaltet werden, da dieser Zugriffszeiger von dem Iterator selbst verwaltet wird. Im Gegensatz zu einem wahlfreien Zugriff auf Elemente (Abbildung 146) kann bei einem Iterator immer nur einmalig auf das aktuelle Element zugegriffen werden. Danach wird der Lesezeiger automatisch auf das nächste Element verschoben (Abbildung 147). Ist das Ende des

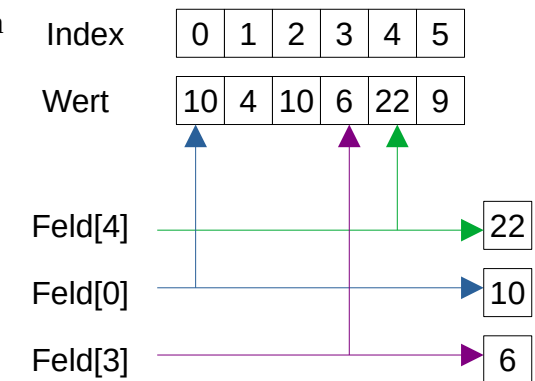


Abbildung 146: Wahlfreier Zugriff auf Elemente(ohne Iterator)

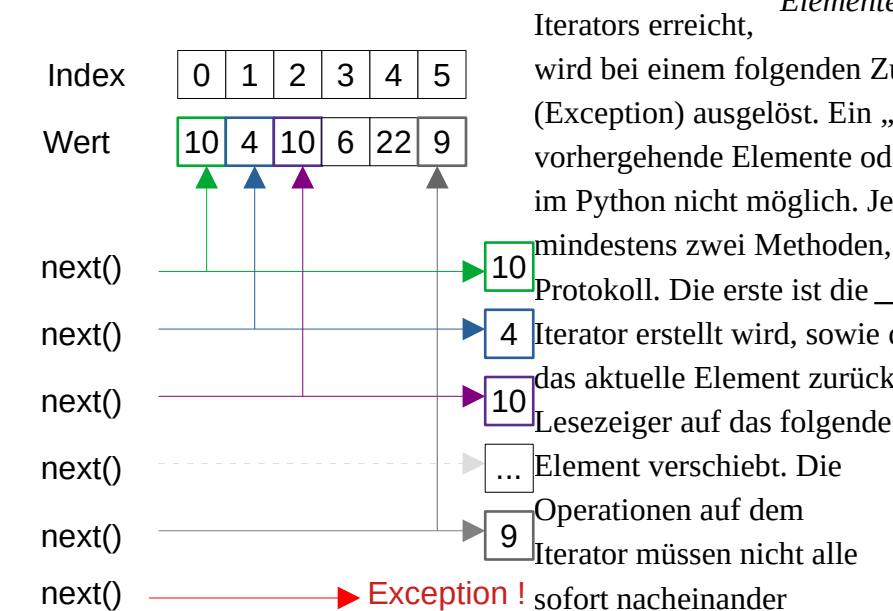


Abbildung 147: Zugriff auf Elemente mittels eines Iterators

Iterators erreicht, wird bei einem folgenden Zugriff eine Ausnahme (Exception) ausgelöst. Ein „zurückspulen“ des Zeigers auf vorhergehende Elemente oder dem Anfang des Iterators ist im Python nicht möglich. Jeder Iterator implementiert mindestens zwei Methoden, das sogenannte Iterator-Protokoll. Die erste ist die `__iter__()` Methode, mit der ein Iterator erstellt wird, sowie die `__next__()` Methode, die das aktuelle Element zurückgibt und gleichzeitig den Lesezeiger auf das folgende Element verschiebt. Die Operationen auf dem Iterator müssen nicht alle sofort nacheinander passieren (Abbildung 148), sondern können auch voneinander getrennt stattfinden. Es kann auch sein, dass z.B. die ersten zwei Elemente des Iterators vor dem Aufruf einer Funktion geholt werden, die restlichen (oder auch nur einige folgende) aber erst innerhalb der Funktion verarbeitet werden. Dadurch, dass der Iterator den Zugriffszeiger selbst verwaltet, stellt es kein Problem dar und stellt sicher, dass alle Elemente des Iterators bearbeitet werden können. Auch in der Abbildung 148 sehen Sie einen ähnlich

Iterators erreicht,

wird bei einem folgenden Zugriff eine Ausnahme (Exception) ausgelöst. Ein „zurückspulen“ des Zeigers auf vorhergehende Elemente oder dem Anfang des Iterators ist im Python nicht möglich. Jeder Iterator implementiert

mindestens zwei Methoden, das sogenannte Iterator-Protokoll. Die erste ist die `__iter__()` Methode, mit der ein Iterator erstellt wird, sowie die `__next__()` Methode, die das aktuelle Element zurückgibt und gleichzeitig den Lesezeiger auf das folgende

Element verschiebt. Die Operationen auf dem Iterator müssen nicht alle

sofort nacheinander passieren (Abbildung 148), sondern können auch

```
liste = [10,4,10,6,22,9]
iterator = iter(liste)
```

```
print(next(iterator))
print(next(iterator))
print(next(iterator))
```

```
10
4
10
```

```
for element in iterator:
    print(element)
```

```
6
22
9
```

Abbildung 148: Zugriffe auf einem Iterator

gestalteten Fall. Die ersten 3 Elemente des Iterators werden über die **next()** Methode geholt, die restlichen 3 in dem folgenden Jupyterblock innerhalb der **for** Schleife.

### 2.8.1 iter()

Mit der `iter()` Funktion kann aus einer Kollektion (Liste, Tupel,..) ein neuer Iterator erstellt werden. Der neu erstellte Iterator entspricht vom Typ einem speziell für diesen Operatortyp vorgesehenen Iterator (Abbildung 149). So wird aus einer Liste ein **list\_iterator** und aus einem Tuple ein **tuple\_iterator** Objekt erstellt. Dadurch, dass diese Untertypen das Iterator Protokoll beinhalten, können alle diese Iterator-Objekte auf gleiche Art und Weise verarbeitet werden.

```
liste = [10,4]
iterator = iter(liste)
print(type(iterator))

tupel = (10,4)
iterator = iter(tupel)
print(type(iterator))
```

```
<class 'list_iterator'>
<class 'tuple_iterator'>
```

Abbildung 149: Iteratoren erstellen

### 2.8.2 next()

Die Funktion **next()** liefert das nächste Element des Iterators zurück. Dabei wird der Lesezeiger auf das nächste Element verschoben. Gibt es keine weiteren lesbaren Elemente mehr, so wird eine

```
liste = [10,4]
iterator = iter(liste)

print(next(iterator))
print(next(iterator))
print(next(iterator))
```

```
10
4
```

```
-----
StopIteration                                Traceback (most recent call last)
<ipython-input-7-26d9fdb975e8> in <module>
      4 print(next(iterator))
      5 print(next(iterator))
----> 6 print(next(iterator))
```

**StopIteration:**

Abbildung 150: Iterieren mit der `next()` Anweisung.

„StopIteration“ Exception geworfen (Abbildung 150). Leider gibt es in Python keine Möglichkeit festzustellen, ob ein Iterator noch mit `next` lesbare Elemente beinhaltet.

### 2.8.3 Iteratoren mit for auslesen.

Wird ein Iterator mit einer **for** Schleife bearbeitet, so gibt es im derartigen Fall kein Problem mit der Erkennung, ob alle Elemente bereits abgearbeitet sind. Dies erledigt die **for** Schleife automatisch und hört mit dem letztem Element auf. Aus dem Grund wird hier auch keine Exception geworfen. Es ist allerdings nicht notwendig Kollektionen vor einer **for** Schleife explizit zu einem Iterator zu wandeln, da das von der **for** Schleife automatisch erledigt wird (Abbildung 151). Listen, Sets, Tupel und Dictionaries sind sogenannte Iterable Objekte, aus denen die Python Laufzeitumgebung bei Bedarf automatisch über die Funktion **iter()** einen Iterator erstellen kann.

```
liste =[1,2,3]
iterator = iter(liste)
for number in iterator:
    print (number)
```

1  
2  
3

```
liste =[1,2,3]
for number in liste:
    print (number)
```

1  
2  
3

Abbildung 151: einem Iterator mit for auslesen

### 2.8.4 zip

Die Funktion **.zip()** erstellt aus einem oder mehreren Objekten mit der **iterable** Eigenschaft einen gemeinsamen Iterator, der die Länge des kürzesten Eingangs-Iterators aufweist (Abbildung 152). Wenn z.B. der erste der beiden Eingabe-Iteratoren 5 Elemente und der zweite 10 Elemente aufweist, so ist der Ausgangsiterator 5 Elemente lang, wobei er alle Elemente des ersten und die 5 vordersten Elemente des zweiten Iterators aufnimmt. Mit zip lassen sich auch Kollektionen „entpacken“ (Abbildung 153). Dazu muss die zu entpackende Kollektion beim Aufruf vom **zip** mit einem „\*“ markiert werden. Das von der Funktion generierte zip Objekt kann dann wiederum zu Listen, Sets etc. entpackt werden. Die Zielobjekte werden, wie normale Variablen, mit Kommas getrennt linksseitig angegeben.

```
index = ['A', 'B', 'C', 'D']
values1 = {1,2,3}
values2 = (3,4,5,6)

a = zip(index, values1, values2)

for index in a:
    print(index)
```

('A', 1, 3)  
('B', 2, 4)  
('C', 3, 5)

Abbildung 152: Funktionsweise von zip

```
bzListe = [('A',1), ('B',2), ('C',33),('D',5)]
print(bzListe)
bListe, zListe = list(zip(*bzListe))
print("Buchstaben ",bListe)
print("Zahlen ",zListe)
```

[('A', 1), ('B', 2), ('C', 33), ('D', 5)]  
Buchstaben ('A', 'B', 'C', 'D')  
Zahlen (1, 2, 33, 5)

Abbildung 153: Entpacken mit zip

## 2.8.5 Enumerator

Mit der Enumerate Funktion lassen sich Elemente einer Kollektionen indexieren. Dabei wird jedem Element ein eigener Index zugewiesen (Abbildung 154). Wird ein Startpunkt als zweiter Parameter angegeben, so werden die Indizes ab diesem Startpunkt vergeben (Abbildung 154 unten).

```
liste = ["Apfel", "Banane", "Zitrone"]
enum = enumerate(liste)
print(list(enum))
print(list(enum))
```

```
[(0, 'Apfel'), (1, 'Banane'), (2, 'Zitrone')]
[]
```

```
liste = ["Apfel", "Banane", "Zitrone"]
enum = enumerate(liste, 30)
print(list(enum))
```

```
[(30, 'Apfel'), (31, 'Banane'), (32, 'Zitrone')]
```

Abbildung 154: Kollektionen indizieren

## 2.8.6 map

Diese Funktion verknüpft ein Iteratorobjekt mit einer Funktion. Beim Auswerten des **map** Objektes wird die verknüpfte Funktion auf jedes Element des Iteratorobjektes angewandt und das Ergebnis in ein Iterable Objekt zurückgeschrieben. Die Funktion sowie die Daten werden als Parameter an die **map** Funktion übergeben. Das Ergebnis der **map** Funktion ist zuerst „nur“ ein **map** Objekt, der die Verlinkung der Funktion mit den Daten speichert – bei der Erstellung einer map werden keine neuen Daten generiert. Diese entstehen erst, wenn das **map** Objekt selbst ausgelesen, bzw. zu einem Kollektionsobjekt umgewandelt wird. Dieses Verhalten können Sie auch in der Abbildung 156 beobachten. Als die Map erstellt wurde waren in der Liste „zahlen“ 4 Elemente, während die Ausgabe der map aber nur 3 Elemente aufweist. Da, wie bereits erwähnt, die in der map verknüpfte Funktion erst bei einem Lesezugriff auf die Daten erfolgt und der pop Befehl, der das zweite Element entfernt, vor dieser Umwandlung auf der Zahlenliste ausgeführt wurde, fehlt auch das alte Element mit dem Index 1 in der aus der map erstellten Liste. Des Weiteren ist in dieser Abbildung noch eine weitere Eigenschaft der map zu sehen – nach der Umwandlung ist die map bei einem zweitem Zugriff leer und generiert keine neuen Daten. Will man also wieder die gleiche Liste mit dieselben Funktion bearbeiten, dann muss die **map** und somit die Verknüpfung neu erstellt werden

```
def offset(a):
    return a + 4

zahlen = [0,4,5,9]
m = map(offset, zahlen)

print(m)
print("org -", zahlen)
print("+ 4 -", list(m))
```

```
<map object at 0x0000027E6278BD90>
org - [0, 4, 5, 9]
+ 4 - [4, 8, 9, 13]
```

Abbildung 155: die map Funktion

```
def offset(a):
    return a + 4

zahlen = [0,4,5,9]
m = map(offset, zahlen)
print("org -", zahlen)
zahlen.pop(1)
print("pop -", zahlen)
print("+ 4 -", list(m))
print("+ 4 -", list(m))
```

```
org - [0, 4, 5, 9]
pop - [0, 5, 9]
+ 4 - [4, 9, 13]
+ 4 - []
```

Abbildung 156:

Die Maps funktionieren genauso gut mit Zahlen wie auch mit Zeichenketten (Abbildung 157). So können auch nichtnumerische Ausdrücke durch die verknüpften Funktionen bearbeitet werden.

```
def schwaebisch (a):  
    return (a + "le")  
  
a = ['Haus', "Auto", "Bombe"]  
print( list(map (schwaebisch,a)))  
  
['Hausle', 'Autole', 'Bombele']
```

Abbildung 157: map mit Zeichenketten

## 2.8.7 Maps direkt in Tupel Umwandeln

Mit dem Sternoperator (Abbildung 158) lässt sich eine **map** direkt bei deren Erstellung „entpacken“ (Abbildung 158). Dabei wird aus der map automatisch ein Tupel erstellt. Zu beachten ist, dass die gesamte Operation in Klammern gesetzt werden muss und dass vor der letzten abschließenden Klammer ein Komma unbedingt notwendig ist.

```
s = (*map (schwaebisch,a),)  
print(type(s))  
print(s)  
  
<class 'tuple'>  
('Hausle', 'Autole', 'Bombele')
```

Abbildung 158: automatische Umwandlung von Maps

## 2.8.8 Yield

Mit der Anweisung **yield** lässt sich ein Generator definieren. Dabei wirkt yield wie eine Art

„mehrfach Return“, der die mit Yield angegebenen Werte in einem Iterator „sammelt“ (Abbildung 159). Dieser Iterator ist dann auch der eigentliche Rückgabewert der Funktion. Ist in einer Funktion, die die **yield** Anweisung verwendet, auch eine **return** Anweisung vorhanden, so wird dann diese **return** Anweisung einfach ignoriert und es wird auf jedem Fall ein Generator zurückgegeben. Zu bedenken ist, dass ein Generator genauso wie ein Iterator nur einmal verwendet werden kann, da er nach dem ersten vollständigen Durchlauf leer ist und nicht zurückgespult werden kann (Abbildung 160). Die

Vorgehensweise von Python ist an der Stelle folgende: Die definierte Funktion **generate** wird zuerst nicht ausgeführt, sondern erst wenn die einzelne Elemente in einer **for** Schleife abgefragt werden. So wird die Funktion bis zu der nächsten yield Anweisung ausgeführt und aus den aktuellen Daten ein neues Element gebildet. Der Gedanke, der hinter dieser Vorgehensweise steckt, ist, die Daten erst dann zu generieren, wenn diese gebraucht werden und somit bei großen generierten Datenbeständen den Speicherplatz zu sparen. Selbstverständlich lässt sich ein Generator auch automatisch in ein Kollektionsobjekt umwandeln (Abbildung 161).

```
def generate():
    yield 2
    yield "text"
    yield 4
    return 5
```

```
values = generate()
print(values)
```

```
for value in values:
    print(value)
```

```
<generator object generate at 0x0000021A773BA740>
2
text
4
```

Abbildung 159: Anweisung yield

```
def generate(y):
    x = 0
    while x < 10:
        yield x*y
        x+=5
```

```
values = generate(4)
```

```
print("First time: ")
for value in values:
    print(value)
```

```
print("Second time: ")
for value in values:
    print(value)
print("end")
```

```
First time:
```

```
0
```

```
20
```

```
Second time:
```

```
end
```

Abbildung 160: Mehrfache Verwendung von Generatoren

```
def generate():
    yield "Nach"
    yield "Hause"
    yield "telefonieren"

list(generate())
```

```
['Nach', 'Hause', 'telefonieren']
```

Abbildung 161: Einem Generator in eine Kollektion umwandeln

## 2.9 Lambdas

Lambdas sind kleine anonyme Funktionen. Sie können zwar mehrere Parameter besitzen, aber sind nur auf einen Term beschränkt. Eine Lambda Funktion hat an sich keinen Namen, kann aber einem Ausdruck zugewiesen werden. Auf diese Weise kann eine Lambda Funktion auch außerhalb ihres Definitionsbereich verwendet werden.

```
In [28]: a = lambda x: 10
         print(a(4))
```

10

```
In [15]: a = lambda x: x + 10
         print(a(4))
```

14

Abbildung 162 einfache Lambdas

### 2.9.1 Direkte Lambdas

Eine Lambda Funktion kann auch direkt bei ihrer Definition aufgerufen werden (Abbildung 163). In diesem Fall muss sowohl die Lambda Funktion wie auch dessen Parameter jeweils mit einer eigener Klammer dekoriert werden. Da solche Lambdas keine Namen haben, können sie nur direkt bei der Definition verwendet werden.

```
(lambda x: x + 1)(2)
```

3

```
a = (lambda x: x + 1)(11)
print("x + 1 = ",a)
```

x + 1 = 12

Abbildung 163 Direkte Lambdas

### 2.9.2 Lambdas mit mehreren Parametern

Die Lambdas können auch mehrere Parameter aufnehmen und diese dann innerhalb der Funktion verarbeiten (Abbildung 164).

```
b = lambda x,y : x + y
print(b(2,3))
```

5

```
mul = lambda x,y : x * y
a1 = 2
a2 = 3
print(a1, " * ", a2, " = ", mul(a1,a2))
```

2 \* 3 = 6

Abbildung 164 Lambda mit mehreren Parametern



### 2.9.3 Lambdas mit Auswertungslogik

Nun es ist auch möglich Auswertungslogik in Lambdas zu benutzen (Abbildung 165). In dem Beispiel sehen Sie die Auswertung von zwei Listen. Es soll jedes mal bestimmt werden ob alle Elemente der Liste größer als 10 sind. Die verwendete Lambda Funktion überprüft, ob das gegebene Element tatsächlich größer als 10 ist und liefert dann über eine **if...else** Abfrage entweder den Wert Wahr oder Falsch zurück.

Die so aufgebaute **map** aus Wahr/Falsch Werten wird anschließend mit der **all()** Funktion ausgewertet. Als Endergebnis wird dann nur noch ein einzelner Wert ausgegeben, der vermittelt, ob alle Werte in der Liste größer als 10 sind oder nicht.

```
liste = [10, 12, 13]
x = all((*map(lambda x: True if x > 10 else False, liste),))
print(x)

liste = [11, 12, 13]
x = all((*map(lambda x: True if x > 10 else False, liste),))
print(x)

False
True
```

Abbildung 165: Lambdas mit Auswertungslogik

### 2.10 Filter

Wenn es darum geht, nur bestimmte Datensätze aus einer Kollektion „herauszuholen“, so ist das in Python der Aufgabenbereich eines Filters. Ein Filter generiert aus der übergebenen Kollektion eine neue Kollektion, die den im Filter vorgegebenen Kriterien entspricht. Als Filterfunktion kann entweder eine „normale“ Funktion, oder eine Lambda Funktion verwendet werden (Abbildung 166). So werden in dem nebenstehenden Beispiel alle Datensätze, dessen Wert größer als 10 aus der Quelle ist, in das neu erstellte Objekt des Typs **filter** übernommen. Der Filter kann aber mit dem Star-Operator direkt zu einem Tuple ausgewertet werden.

```
liste = [1,2,13,14]
a = filter(lambda x: x > 10, liste)
print(list(x))

a = (*filter(lambda x: x > 10, liste),)
print(x)

[13, 14]
(13, 14)
```

Abbildung 166: Filter in Python

## 2.11 List Comprehension

List Comprehension („Listen-Abstraktion“) ist ein sehr mächtiges Werkzeug, um Listen zu generieren. Die Listendaten können dabei entweder aus einer anderen Liste oder von einem Iterator stammen, wobei streng genommen Listen ja auch Iteratoren sind. Die Erzeugung aus einer Liste sehen sie in der Abbildung 167. Dabei können auch Bedingungen verwendet werden, sodass die neu erstellte Liste weniger Elemente als die ursprüngliche Liste haben kann. Wie in der Abbildung zu sehen ist, trifft die *if* Bedingung nicht zu, dann wird das entsprechende Element in die neue Liste nicht übernommen. Die Elemente müssen nicht so wie sie sind direkt übernommen werden, sondern können vorher noch bearbeitet werden, sodass die neu erstellten Elemente einen anderen, in der Regel in irgendeiner Form, abgeleiteten Wert besitzen (Abbildung 168). In Abbildung 168 sieht man, dass die zweite Liste aus einem Iterator (Range Objekt) direkt generiert wird. Der Wertebereich ist allerdings nicht nur auf numerische Werte eingeschränkt. Mit der List Comprehension lassen sich auch alphanumerische Werte bearbeiten (Abbildung 169). Zu guter Letzt können auch Funktionen bzw. Lambdas innerhalb der

```
liste = [0,1,2,3,4,5,6,7,8,9,10]
```

```
liste2 = [x for x in liste if x > 2 ]
liste2
```

```
[3, 4, 5, 6, 7, 8, 9, 10]
```

```
liste2 = [x for x in liste if x % 2 ]
liste2
```

```
[1, 3, 5, 7, 9]
```

Abbildung 167: Liste aus einer Liste erstellen.

```
liste3 = [x*x for x in liste if x % 2 ]
liste3
```

```
[1, 9, 25, 49, 81]
```

```
liste4 = [(x+1)*(x+1) for x in range(0,10,2) ]
liste4
```

```
[1, 9, 25, 49, 81]
```

Abbildung 168: Liste mit veränderten Elementen

```
fruechte = ["Apfel","Birne","Kiwi"]
liste5 = [x+"mus" for x in fruechte ]
liste5
```

```
['Apfelmus', 'Birnemus', 'Kiwimus']
```

Abbildung 169: List Comprehension mit Alphanumerischen Werten

```
def max(x,y):
    if x > y:
        return x
    return y

liste = [[0,1],[2,3],[4,1]]

liste2 = [max(x,y) for x,y in liste ]
liste2
```

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

Abbildung 170: List Comprehension mit Funktionen

List Comprehension zu Wertebildung verwendet werden. In der Abbildung Abbildung 170 wird so eine Liste mit den Maxima aller Wertepaare der Eingabeliste gebildet.

## 2.12 Objektorientierte Programmierung in Python

Der oder die eine oder andere kennt vielleicht dieses Konzept bereits von einer Programmiersprache. Bei Objektorientierter Programmierung werden zusammenhängende „Inhalte“ bzw. Kompetenzen in unterschiedlichen Klassen zusammengefasst. Dadurch soll eine inhaltliche Trennung von Inhalten erreicht werden und Software wie mit passenden LEGO Bausteinen. Somit behandelt eine Klasse nur einem bestimmten ganz kleinen Aspekt des Problems und erst durch das Zusammenspiel der einzelnen Klassen untereinander entsteht die eigentliche Anwendung. Nur die Klasse selbst weist am besten wie die in ihr enthaltenen Daten (auch Attribute genannt) am besten zu manipulieren sind und nur die sollte (in dem optimalen Fall) Veränderungen an diesen Attributen vornehmen. Als bestes Beispiel wären hier die sogenannten *setter* und *getter* Methoden zu nennen nur diese sollten wissen wie die Daten intern in der Klasse gespeichert werden so das bei Änderung des Internen Datenmodells wiederum im Optimalen Fall nur diese zwei Methoden angepasst werden müssen. Abgesehen davon können so lästige immer wieder vorkommende Operationen (z.B. Bereichsüberprüfung beim Speichern der Daten) an einer Stelle gebündelt werden, so das diese nicht wiederum an 1000 Stellen in der Anwendung implementiert werden müssen. Ändern sich die Eigenschaften eines Attributs – so muss nur noch die setter Methode angepasst werden – ein einziges einzelnes mal.

Dieses Bausteinsystem kann mittels Vererbung nochmals verfeinert werden (Abbildung 171), diesem Vorgang nennt man Spezialisierung (Abbildung 171). Dabei wird von der Basisklasse ein Tochterklasse erstellt die die Fähigkeiten der Basisklasse für bestimmte Teilaufgabe(n) erweitert.

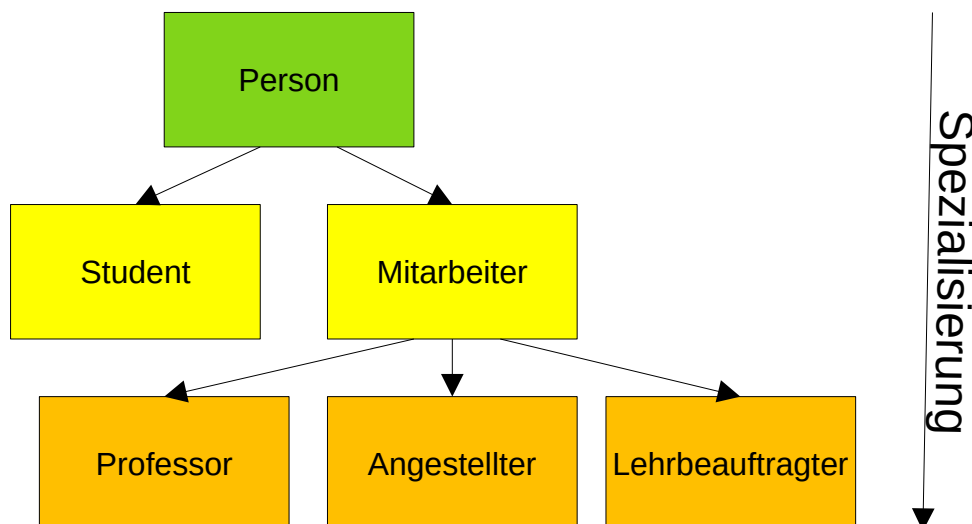


Abbildung 171: Spezialisierung der Klassen

Diese Klasse Tochterklasse kann dann wiederum ihrerseits durch folgende Tochterklassen erweitert werden so das dabei eine Baumartige Struktur entsteht. So auch in dem Beispiel – als Basisklasse dient hier die Klasse Person, diese speichert zuerst alle allgemeine Daten zu einer Person das wären dann zum Beispiel Name, Vorname (jede Person sollte mindestens ja eins von Beiden haben haben) etc. Dann kommt die „Erweiterung“ Student hier wird z.B. die Matrikelnummer (haben nur

Studenten) und die Fachrichtung gespeichert. Andererseits gibt es da Mitarbeiter die haben eine Personalnummer, die anders als Matrikelnummer Strukturiert ist sowie es gibt Mitarbeiter die nicht bei Fakultäten, sondern bei einer der weiteren Einrichtungen der Hochschule eingestellt sind z.b. Gebäudemanagement, Studentische Verwaltung. Da dies die Daten sind die bei Studenten nicht vorkommen können (Kein Student kann bei Gebäude Management studieren) wird hier in zwei unterschiedliche Richtungen spezialisiert. Damit aber nicht alles was bereits vorhanden ist nochmals implementiert und getestet wurde nochmal implementiert werden muss wird bei der Vererbung aka Ableitung aka Spezialisierung das bereits vorhandene in die neuen Klassen „eingebündelt“ und somit auch übernommen (Abbildung 172). Beachten Sie das die `__id` Variable aka Attribut nicht in die beiden Tochterklassen übernommen wurde – dieser Attribut ist eine private Eigenschaft der Klasse (im Python durch den Präfix „`__`“ erkennbar) und somit für die Tochterklassen nicht sichtbar. Bei Zugriffen auf die Eigenschaften und Methoden einer Abgeleiteten Klasse wird, soweit diese nicht in der aktuellen Klasse vorhanden sind, bei der Mutter Klasse(n) „nachgeschaut“ ob diese die entsprechende Kompetenz bearbeiten kann. Erst wenn dies nicht zutrifft wird ein Fehler erkannt.

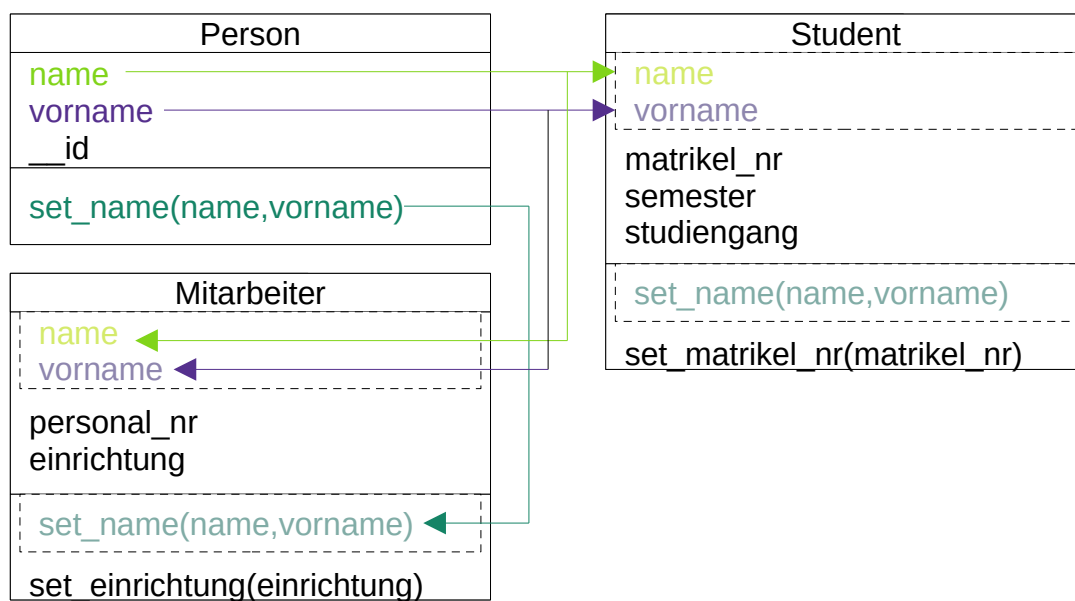


Abbildung 172: Spezialisierung und Übernahme der Eigenschaften

Ein weiterer nicht unwichtiger Vorteil von OOP ist, dass einmal geschriebene Klassen (bzw. die ganzen Klassenbäume) lassen sich ohne größere Probleme in anderen Anwendungen / Programmen verwenden. Diese müssen dann auch nicht mehr auf interne Abhängigkeiten / Probleme getestet werden sondern nur nach außen im Zusammenspiel mit der neuen äußeren Umgebung überprüft werden.

## 2.12.1 Klassen in Python

Zu beachten ist, das anders als bei Java oder C++, in Python bei dem Zugriff auf Variablen das Schlüsselwort **self** immer mitangegeben werden muss (Abbildung 173) da sonst die Variable nicht als klassenzugehörig erkannt wird und stattdessen eine neue lokale Variable erstellt wird (Abbildung 174 vorletzte Zeile). Ein Verhalten das eine Folge der schwachen Typisierung im Python ist.

```
class Person:
    vorname = "Micky"
    nachname = "Mouse"

    def __init__(self):
        pass

    def set_name(self, nachname, vorname):
        self.nachname = nachname
        vorname = vorname; #Fehler
```

Abbildung 173: Definition einer Klasse

Der um eine Klasse zu verwenden (Abbildung 174) muss zuerst eine Instanz bzw. ein Objekt mit dem entsprechenden Typ erstellt werden. Dies wird bewerkstelligt indem einer Variable die Klasse zugewiesen wird (Abbildung 174 die obersten zwei Zeilen). Danach können die Funktionen der Klasse auf die Instanz angewandt werden (Abbildung 174 Mitte). Solange es sich um öffentliche Variablen handelt, können diese direkt von jeder Stelle des Programms gelesen und beschrieben werden (Abbildung 174 Unten).

```
person_a = Person()
person_b = Person()

person_a.set_name( "Donald", "Duck")

print(person_a.vorname, person_a.nachname)
print(person_b.vorname, person_b.nachname)
```

```
Micky Donald
Micky Mouse
```

Abbildung 174: eine Klasse verwenden

## 2.12.2 Öffentliche, geschützte und private Variablen.

Wie auch andere Objektorientierte Sprachen kennt Python ein dreistufiges System um Zugriffsberechtigungen auf Lokale Variablen zu regeln:

- **public** – eine öffentliche Variable (*public*) ist für alle sichtbar kann von überall geschrieben und gelesen werden.
- **protected** – eine geschützte Variable (*protected*) kann nur von der Klasse selbst, sowie von den von ihr abgeleiteten Subklassen gesehen sowie gelesen und beschrieben werden. Diese Variablen werden mit dem Präfix „\_“ (einfacher Unterstrich) in Ihrem Name definiert.

- **private** – eine private Variable (*private*) ist nur für die Klasse selbst sichtbar und verwendbar. Alle anderen Klassen auch die Tochterklassen haben keinerlei Zugriff auf diese Variablen. Diese Variablen werden mit dem Präfix „**\_\_**“ (doppelter Unterstrich) in Ihrem Namen gekennzeichnet.

Eine Zugriffsversuch auf eine Variable ohne die entsprechenden Rechte führt zu einer Ausnahme (Abbildung 175). Da wir hier weder innerhalb der Klasse noch innerhalb einer Subklasse agieren, haben wir in dem Beispiel nur ein Zugriffsrecht auf öffentliche Variablen - bereits der Zugriff auf eine geschützte Variable scheitert aus diesem Grund – da wir nicht die erforderliche Rechtestufe besitzen ist die entsprechende Variable für uns unsichtbar.

```
class Person:
    _vorname = "Micky"    #protected
    __nachname = "Mouse" #private
    anrede = "Herr"      #public

person_a = Person()
print(person_a.anrede)
print(person_a.vorname)
print(person_a.nachname)
```

Herr

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-17-2ad8ba70f575> in <module>
      6 person_a = Person()
      7 print(person_a.anrede)
----> 8 print(person_a.vorname)
      9 print(person_a.nachname)
```

```
AttributeError: 'Person' object has no attribute 'vorname'
```

Abbildung 175: Zugriff auf Variablen in Klassen

Die Variablen einer Klasse müssen nicht direkt in der Klasse definiert sein, sie können auch einfach bei einem Schreibenden Zugriff in einer Methode über dem **self** Operator „entstehen“.

### 2.12.3 Statische Variablen

Statische Variablen sind Variablen die für alle Instanzen einer Klasse (und abgeleitete Klassen) gleich sind. Sie werden angelegt indem Sie in der Klassendefinition außerhalb einer Funktion definiert werden (Abbildung 176). Der Zugriff auf die statischen Variablen sollte sicherheitshalber über den Klassennamen erfolgen. Die statischen Variablen können zwar auch über Instanzen angesprochen werden (Abbildung 177), aber hier kommt das Problem auf, das man hier nicht immer wirklich die Statische Variable „erwischt“. Wurde zwischenzeitlich eine gleichnamige dynamische Variable mit dem gleichen Namen in einem Objekt erzeugt so wird diese bei einem Lesezugriff bevorzugt. Es können nämlich zwei Variablen mit dem gleichen Namen in einem Objekt erreichbar sein – eine Statische und eine dynamische. Existiert eine dynamische Variable mit den gleichen Namen wie eine statische, dann überblendet bei einem Zugriff über die Instanz diese die Statische. Gibt es keine dynamische Variable mit entsprechenden Namen, dann wird der Zugriff automatisch auf die Statische umgeleitet. Da eine Instanzvariable in Python durch einen einfachen Schreibzugriff erstellt wird ist bei statischen Variablen strengstens drauf zu achten diese auch über die Klassen und nicht deren Instanzen anzusprechen!

```
class Static:
    static = 4

    def __init__(self):
        self.dynamic = 10

print("Static.static = ", Static.static)
```

```
Static.static = 4
```

Abbildung 176: Statische Variablen

```
class Static:
    static = 4

    def __init__(self):
        pass

a = Static()
print("Static.static = ", Static.static)
print("a.static = ", a.static)

Static.static = 42
a.static = 21

print("Static.static = ", Static.static)
print("a.static = ", a.static)
```

```
Static.static = 4
a.static = 4
Static.static = 42
a.static = 21
```

Abbildung 177: Statische Variablen und Instanzen

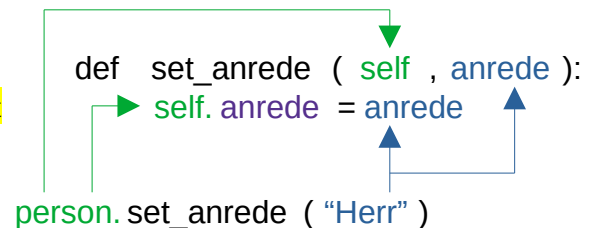
## 2.12.4 Methoden

Im Unterschied zu den was Sie vielleicht aus JAVA oder C++ kennen sind die Methoden einer Klassen nicht „fest verdrahtet“, sondern es handelt sich dabei, wie bei allem anderem im Python um selbständige Objekte. So ist es möglich im Laufe der Anwendung die Methoden einer Klasse zu löschen, neue Methoden hinzufügen, oder durch andere Methoden zu ersetzen. In wie fern das sinnvoll ist, das ist aber eine andere Frage. Im Entwicklungsprozess ist es auf jeden Fall allemal praktisch da man so die Klasse stückchenweise zusammenbauen kann.

Ein weiterer sehr wichtiger Unterschied zwischen Python und vielen anderen Objektorientierten Programmiersprachen ist, dass die Methoden in

Python immer den **self** Parameter benötigen, dieser aber wird allerdings bei Methoden die über ein Objekt bzw. eine Instanz einer Klasse aufgerufen werden nicht beim Aufruf angegeben. Dieser Parameter wird bei derartigen Aufrufen von der Laufzeitumgebung

automatisch gesetzt (Abbildung 178 & 179). Wird die Methode hingegen nicht über eine Instanz aufgerufen, wie und warum das geht sehen wir noch in einem der folgenden Kapiteln, dann muss in diesem Fall der Verweis auf die Instanz bei dem Aufruf angegeben werden (Abbildung 180 & 181).



```
def set_anrede ( self , anrede ):
    self.anrede = anrede

person.set_anrede ( "Herr" )
```

Abbildung 178: self Parameter bei einem Aufruf über eine Instanz

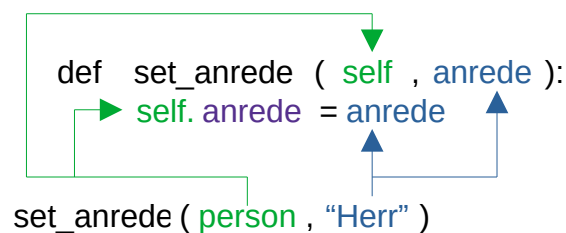
```
def set_anrede(self, anrede):
    self.anrede = anrede

person = Person()
set_anrede(person, "Herr")

print(person.anrede)
```

Herr

Abbildung 179: self Parameter bei einem Aufruf über eine Instanz



```
def set_anrede ( self , anrede ):
    self.anrede = anrede

set_anrede ( person , "Herr" )
```

Abbildung 180: Self Parameter bei einem direktem Aufruf

```
def set_anrede(self, anrede):
    self.anrede = anrede

set_anrede(person, "Dr.")
print(person.anrede)
```

Dr.

Abbildung 181: Self Parameter bei einem direktem Aufruf



### 2.12.4.1 Klassen um Methoden erweitern

Schauen wir uns an einem Beispiel (Abbildung 182), wie eine zusätzliche Methode in eine Klasse aufgenommen wird. Um eine weitere Methode zur einer Klasse hinzuzufügen, muss diese zuerst korrekt definiert werden und zwar inklusive der **self** Variable. Danach kann die Methode der Klasse zugewiesen werden die beiden Namen, also der Name unter dem die Funktion in der Klasse erreichbar ist sowie der eigentliche Name der Funktion sind nicht in irgendeiner Weise miteinander „verknüpft“ und müssen nicht übereinstimmen. So kann die Methode `set_anrede()` in der Klasse `Person` auch unter Bezeichnung `add_anrede()` erreichbar sein (Abbildung 183). Da die ursprüngliche Funktion immer noch existiert, kann diese auch weiterhin in der „rohen“ Form (d.h. ohne zu einer Klasse dazu zu gehören) verwendet werden (Abbildung 184). Allerdings muss hier jetzt der **self** Parameter mitangegeben werden, damit die Methode weißt welches Objekt sie zu bearbeiten hat.

```
def set_anrede(self, anrede):
    self.anrede = anrede

Person.set_anrede = set_anrede

person = Person()
person.set_anrede("Herr")

print(person.anrede)
print(person.vorname)
print(person.nachname)
```

Herr  
Micky  
Mouse

Abbildung 182: eine Methode zu einer Klasse hinzufügen

```
Person.add_anrede = set_anrede
person.set_anrede("Herr")
print(person.anrede)

person.add_anrede("Frau")
print(person.anrede)
```

Herr  
Frau

Abbildung 183 Methoden Aliase

```
set_anrede(person, "Dr.")
print(person.anrede)
```

Dr.

Abbildung 184: Verwendung der ursprünglichen Methode für Objekte

### 2.12.4.2 Private und Geschützte Methoden

Kurze Antwort – gibt es in Python nicht. Alle Methoden in Python sind öffentlich da davon ausgegangen wird das ein Entwickler „weiß was er Tut“.

## 2.12.5 Klassen Ableiten

Eine Klasse, die sogenannte abgeleitete Klasse, kann von einer anderen Klasse, der sogenannten Basisklasse „erben“. Dazu muss die Basisklasse allerdings nicht sterben, sondern die abgeleitete Klasse übernimmt mit diesem Vorgang alle nicht privaten Eigenschaften und Methoden der Basisklasse. Damit wird erreicht, dass eine Tochterklasse auf eine bestimmte (Teil-)Aufgabe spezialisiert wird. In dem Beispiel (Abbildung 185) wird aus der Klasse `Person` die Tochterklasse `Student` abgeleitet. Damit verfügt diese abgeleitete Klasse sofort über alle nicht privaten Eigenschaften, die auch die Klasse `Person` beinhaltet. Jetzt kann die Klasse `Student` um die Eigenschaften erweitert werden, die notwendig sind, um einen Studierenden zu beschreiben. Man könnte diese zwar auch in die Klasse `Person` übernehmen, aber da sich diese Eigenschaften speziell auf Studierende beziehen, sind diese für andere Personentypen irrelevant und belegen nur unnötig Speicher (Siehe Kapitel 2.12). Die Ableitung selbst wird erstellt, indem der Name der Klasse, von der abgeleitet werden sollte, nach dem Klassennamen in Klammern angegeben wird.

```
class Student(Person):
    def set_matrikel_nr(self, matrikel):
        self.matrikel_nr = matrikel

student = Student()
student.set_name("Keeper", "Dungeon")
student.set_matrikel_nr(2661)

print(student.matrikel_nr)
print(student.vorname)
print(student.nachname)
```

```
2661
Dungeon
Keeper
```

Abbildung 185: Klassen ableiten

## 2.12.6 Der Nullpointer

Ein waschechter Nullpointer existiert in Python nicht. Dessen Funktion übernimmt die fest eingebaute Klasse **None** (Abbildung 186). Diese Klasse wird z.b. verwendet wenn eine Funktion zwar ein Objekt zurückgeben muss, aber kein gültiges Ergebnis zurückliefern kann. So wird in diesem Fall, damit eine Rückgabe stattfinden kann ein Objekt vom Typ **None** zurückgegeben.

Die Überprüfung ob ein Objekt gültig ist kann dann bei Bedarf mit den Schlüsselwörtern **is** oder **is not** durchgeführt werden (Abbildung 187).

```
def check(object):
    print(c, end = " - ")
    if object is not None:
        print("OK")
    if object is None:
        print("Error")
```

```
c = get_five(1)
check(c)

c = get_five(0.1)
check(c)
```

```
5 - OK
None - Error
```

Abbildung 187: Überprüfung ob eine Instanz „gültig“ ist.

```
def get_five(object):
    if isinstance(object,int):
        return int(5)
    elif isinstance(object,str):
        return str("5")
    else:
        return None
```

```
c = get_five(1)
print(c)

c = get_five('text')
print(c)

c = get_five(0.1)
print(c)
```

```
5
5
None
```

Abbildung 186: Die Klasse None

## 2.12.7 Klassentypen vergleichen

Manchmal ist es wichtig zu überprüfen ob es sich bei einer Instanz um eine Instanz einer Klasse handelt. Dies kann zwar auch mit der Anweisung `type()` erfolgen, allerdings kann mit dieser Anweisung nur eine direkte 1:1 Zugehörigkeit ermittelt werden. Somit lässt sich nicht überprüfen ob die geprüfte Instanz von einer von der gegebenen Klasse abgeleitete Klasse ist. Dieses Problemfeld deckt die **isinstance** Anweisung (Abbildung 188) Sie prüft neben der direkten Verwandtschaft auch die Abstammung des Objekts, da eine abgeleitete Klasse auch als ihre Basisklasse betrachtet werden kann. In dem Beispiel aus der Abbildung ist auch zu erkennen das ein Student nicht nur ein Student, sondern auch eine Person ist, aber eine Person nicht „per se“ ein Student ist und eine Zeichenkette wiederum ist weder das eine noch das andere. Da Python keine statische Typenprüfung hat ist dies an den Stellen wo mit Objekten einer unbekannter Abstammung gearbeitet wird besonders wichtig das es sich tatsächlich um Objekte handelt die der folgende Code auch bearbeiten kann. Eine andere Einsatzmöglichkeit ist es wenn es für verschiedene Tochterobjekte unterschiedliche Verarbeitungswege gibt so das diese Softwaretechnisch gesplittet werden müssen (Abbildung 189) Hier ist es allerdings sehr wichtig das bei der Abfrage die Hierarchie der Überprüfung auf jedem Fall bei den jüngsten Mitgliedern eines Abstammungsastes anfängt da dies ja sonst bereits bei ihren Vorgängern als klassenzugehörig erkannt werden.

```
def check_class(instance):
    class_name = type(instance).__name__
    if isinstance(instance, Person):
        print(class_name, "ist eine Person" )
    else:
        print(class_name, "ist keine Person" )

    if isinstance(instance, Student):
        print(class_name, "ist ein(e) Student*in" )
    else:
        print(class_name, "is kein(e) Student*in" )

check_class(Student())
check_class(Person())
check_class("He-Man")
```

```
Student ist eine Person
Student ist ein(e) Student*in
Person ist eine Person
Person is kein(e) Student*in
str ist keine Person
str is kein(e) Student*in
```

Abbildung 188: Abstammung einer Klasse testen

```
def check_class(instance):
    if isinstance(instance, Student):
        process_student(instance)
    elif isinstance(instance, Person):
        process_person(instance)
    else:
        print("wrong Data" )
```

Abbildung 189: Verarbeitungswege von Tochterinstanzen Trennen

## 2.13 Ausnahmen (Exceptions)

Eine Exception ist ein Hinweis der Laufzeitumgebung, dass bei der Verarbeitung der Daten etwas „schief gelaufen“ ist. Somit stellen Sie eine Möglichkeit dar, Fehler, die bei der Verarbeitung von Daten aufgetreten ist, z.B. weil ein Datensatz nicht vollständig oder korrupt ist, abzufangen. Das Schöne dabei ist, dass dieser Abfangmechanismus nicht unbedingt in der vorhergehenden Aufrufstufe verankert werden muss, sondern auch weit tiefer in der Aufrufkette

Der Vorteil einer Exception ist, dass es die Fehlerbehandlung innerhalb eines Programms deutlich vereinfachen kann. In der Abbildung 191 sehen Sie das Schema einer Fehlerbehandlung ohne Exceptions. Problem Nummer 1 ist – die Fehlerbehandlung muss in allen Aufrufinstanzen implementiert werden, auch wenn diese Zwischenschichten nichts zur Beseitigung des Problems beitragen können. Sie müssen trotzdem im Fehlerfall erkennen, dass Sie mit der weiteren Bearbeitung der aktuellen Daten aufhören müssen und dies auch noch Ihrem Aufrufer mitzuteilen. Das führt in der Regel zu Problem Nummer 2 – dass sich der entsprechende Code durch die vielen *if...else* Anweisungen einerseits aufbläht und dadurch sehr unübersichtlich wird. Für jeden möglichen Fehler muss ja eine Ausnahmebehandlung implementiert werden. Problem Nummer 3 – wird eine Fehlerquelle „unterwegs“ ignoriert, arbeitet das System unter Umständen dann mit falschen Daten, was wiederum an einer Stelle, die mit dem eigentlichen Fehler nichts zu tun hat, zu „merkwürdigen“ Ergebnissen führen kann. Das Finden solcher Fehler ist meistens nicht trivial und verlangt von dem,

```
def mult(x,y):
    z = x * y
    return(z)

print(mult(2,"a"))
```

aa

Abbildung 190: Zahlen Verrechnen

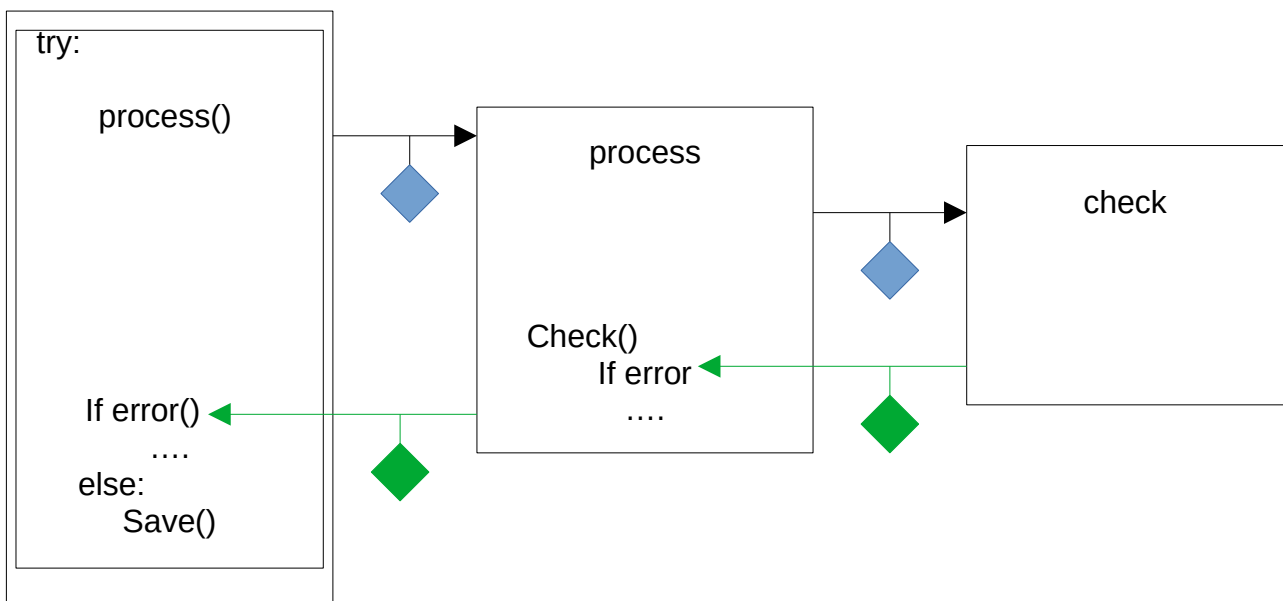


Abbildung 191: „Normale“ Fehlerbehandlung

der diesen Fehler beseitigen muss, meistens sehr viel Erfahrung und „debugging Skills“. Da die Fehlerübergabe über dem „Amtsweg“ über einem return Wert zurückgegeben werden muss, macht die Sache auch nicht gerade einfacher, da hier überlegt werden muss, wie man die Information über den Fehler „nach außen“ bekommt. Zu Verdeutlichung des Problems überlegen wir uns einen (sehr vereinfachten) Anwendungsfall: Eine Funktion soll zwei Zahlen miteinander verrechnen. Wir implementieren es so wie in der Abbildung 190 zu sehen ist. Bei der Eingabe der zweiten Zahl „passiert“ irgendwann mal ein Fehler, sodass statt einer tatsächlichen Zahl hier eine Zeichenkette

(in unserem Beispiel „a“ ) eingegeben wird. In diesem Fall ist das „berechnete“ Ergebnis für die weitere numerische Bearbeitung unbrauchbar. Also sollte in der Funktion irgendeine Art von Fehlererkennung stattfinden und diese dem Aufrufer dann informieren, dass hier was nicht stimmt. Da hier alle numerischen Werte als gültige Ausgaben gewertet werden können, fehlt uns die Möglichkeit, dem Aufrufer auf den Fehler hinzuweisen. Die Möglichkeit, es über einem „NaN“ Wert oder dem Typ zu machen, lassen wir erst mal außen vor. Jetzt modifizieren wir den Code, indem wir in der Funktion nach der Multiplikation den Typ des Ergebnisses anfragen (Abbildung 192). Ist das Ergebnis nicht vom Typ Integer, so werfen wir eine Ausnahme (Exception). In diesem Fall eine Standard Exception

```
def mult(x,y):
    z = x * y
    if type(z) != int:
        raise Exception("not a number")
    return(z)

try:
    print(mult(2,"a"))
except Exception as error:
    print("Fehler - ",error)
```

Fehler - not a number

Abbildung 192 Fehler Abfangen mit exceptions

### 2.13.1 Ausnahmen vs. dedizierte Fehlerabfrage

Betrachten wir mal wie sich das auswirkt, wenn mehrere nacheinander folgende Funktionsaufrufe ausgeführt werden. Hier können die entsprechende Funktionen einfach untereinander notiert werden. Sollte in irgendeiner der **mul()** Anweisungen eine Exception stattfinden, so springt die Fehlerbehandlung automatisch in dem **except** Bereich. Das führt allerdings auch dazu, dass das bis jetzt berechnete Zwischenergebnis in der Variable a gespeichert ist, aber es ist zunächst nicht bekannt, wie weit der Interpreter mit der Ausführung des Programms „gekommen ist“. Somit ist zwar die Anwendung zwar gerettet, aber das Ergebnis, oder ein Teilergebnis von diesem Codeblock ist auf jeden Fall Ungültig.

```
def mult(x,y):
    z = x * y
    if type(z) != int:
        raise Exception("not a number")
    return(z)

try:
    a = mult(2,3)
    a = mult(a,3)
    a = mult("a",3)
    print(a)
except Exception as error:
    print("Fehler - ",error)
```

Fehler - not a number

Abbildung 193 Lösung mit Ausnahmen

Betrachten Wir mal als Gegenbeispiel eine Lösung ohne eine Ausnahme (Abbildung 194). Es fällt auf, dass nach jedem Aufruf der Funktion auf Richtigkeit der Werte überprüft werden muss. Dadurch verkommt das Programm zu einer hässlichen **if..else** Orgie. Man könnte hier nuancieren, indem man die Ausgabe der Fehlermeldung nur einmal zum Schluss macht, aber das würde an dem grundlegenden Problem nichts ändern – das kleine Stückchen Code ist viel komplexer als die vorhergehende Lösung mit Exceptions.

```
def mult(x,y):  
    return(x * y)  
  
a = mult(2,3)  
if type(a) != int:  
    print("Fehler - not a number")  
else:  
    a = mult(a,3)  
    if type(a) != int:  
        print("Fehler - not a number")  
    else:  
        a = mult(a,3)  
        if type(a) != int:  
            print("Fehler - not a number")  
        else:  
            print(a)
```

Fehler - not a number

Abbildung 194 Lösung ohne Ausnahmen

### 2.13.2 Ausnahmen Auslösen

Um eine Exception manuell auszulösen, bedient man sich der **raise** Anweisung. Damit lässt sich sowohl eine Standard Exception oder eine davon abgeleitete spezifische Exception Klasse auslösen. Die so erzeugte Ausnahme kann keinem oder mehrere Parameter aufnehmen (Abbildung 195) die von der Anwendung ausgewertet werden können um so die Quelle der Ausnahme zu erkennen und gegebenenfalls Korrekturmaßnahmen einzuleiten. Wie bei Python üblich können die übertragenen Parameter von beliebigen Typ sein.

```
raise Exception("please write better code",-1)
```

```
-----  
Exception                                Traceback (most recent call last)  
<ipython-input-6-2b94644d2d7b> in <module>  
----> 1 raise Exception("please write better code",-1)
```

```
Exception: ('please write better code', -1)
```

Abbildung 195 eine Ausnahme auslösen

### 2.13.3 Ausnahmen abfangen

Nun ist es schön, wenn der Prozess bemerkt, dass mit den Daten „irgendetwas nicht stimmen kann“, aber andererseits sind nicht alle Fehler so kritisch, dass nach so einem Fehler die Anwendung sterben muss. Fehleingaben des Benutzers können nach deren Erkennung einfach wiederholt werden, oder „faule“ Datensätze ignoriert werden. Deswegen lassen sich Ausnahmen im Programm mit der Anweisung **except** „abfangen“ bzw. „akzeptieren“ (beachte das ist hier keine direkte Übersetzung des Wortes **except**!). Dabei kann auch angegeben werden, dass nur bestimmte Ausnahmen an dieser Stelle behandelt werden können, so dass Ausnahmen, die hier nicht erfolgreich behandelt werden können, an vorausgegangenen Stufen automatisch weitergeleitet werden (Abbildung 196). Sollten mehrere Ausnahmen an einer Stelle behandelt werden, so können mehrere **except** Anweisungen nacheinander implementiert werden.

```
try:
    int("a")
except ValueError as exception :
    print("ValueError")
except TypeError as exception :
    print("TypeError")
```

ValueError

Abbildung 196: Eine Ausnahme abfangen

```
try:
    int("a")
except:
    print("exception")
```

exception

Abbildung 197: Alle Ausnahmen akzeptieren

### 2.13.4 Eigene Exceptionen definieren

Über die Vererbung von der Basisklasse Exception lassen sich auch eigene Ausnahmen definieren. Wie auch bei „normalen“ Klassen ist es möglich dann innerhalb der selbstdefinierten Ausnahmen eigene Methoden zu definieren, die dann z.B. Zusatzinformationen über den Grund für diese Ausnahme aufnehmen und an die Stelle, an der die Ausnahme abgefangen wird, übermitteln können (Abbildung 198).

```
class fault(Exception):
    def cause(self):
        print("kludge")

def function():
    raise fault

try:
    function()
except fault as exception :
    print("exception cause is :")
    exception.cause()
```

exception cause is :  
kludge

Abbildung 198: Eigene Ausnahmen definieren



### 2.13.5 Ende mit schrecken, oder die finally Anweisung

Oft ist es notwendig, das nach einer Ausnahme in dem momentanen Kontext einiges noch „aufgeräumt“ sein müssen z.B die in einem **try** Block geöffnete Dateien, oder Datenbanktransaktionen geschlossen werden oder sonstige Ressourcen (Handels) wieder freigegeben. Der Schlüsselwort **finally** definiert deswegen einem Codebereich, der bei einer nicht in diesem Kontext abgefangenen Exception auf jedem Fall noch zuerst ausgeführt wird, bevor dieser Kontext verlassen wird (Abbildung 199). Dieser Mechanismus soll dafür sorgen das die aktuelle Funktion nach einer Ausnahme keine globalen Ressourcen blockiert da z.B. auf eine zum schreiben geöffnete Datei nur noch lesend zurückgegriffen werden kann. Wird dies nicht berücksichtigt kann erst nach einer Schließung und einem Neustart der Anwendung wieder auf diese Datei schreibend zugegriffen werden.

Wird die Exception hingegen abgefangen, oder es tritt gar keine Exception auf, so wird der Code der nach der **finally** Anweisung notiert ist auch innerhalb der normalen Programmausführung abgearbeitet (Abbildung 200). So wird sichergestellt, dass wichtige Operationen die eine Fortführung der Datenverarbeitung notwendig sind auf jedem Fall in der entsprechenden Funktion ausgeführt werden.

```
try:
    a = mult("2",3)
finally:
    print("Ende")
```

Ende

-----  
Exception  
<ipython-input-23-67e

Abbildung 199 finally

```
try:
    a = mult("a",3)
except Exception as error:
    print("Fehler - ",error)
finally:
    print("Ende")
```

Fehler - not a number  
Ende

```
try:
    a = mult(2,3)
except Exception as error:
    print("Fehler - ",error)
finally:
    print("Ende")
```

Ende

Abbildung 200 finally

### 2.13.6 Ausnahmen über mehrere Ebenen

Wie bereits aufgeführt, ist es mit Ausnahmen möglich Fehlerbehandlung über mehrere Ebenen zu implementieren. Der Mechanismus basiert drauf das bei einer Ausnahme solange dem Aufrufstack abbaut wird bis ein lokaler Kontext diese Ausnahme mit **except** „Abfängt“. Findet sich kein „Abnehmer“ für diese Ausnahme dann wird diese an die Laufzeitumgebung weitergereicht was einer Terminierung der Anwendung gleicht. Dadurch ist es möglich wie in der Abbildung 201 zu sehen ist das Fehlerbehandlung auf Saubere Art und Weise mehrere Aufrufstufen „überspringt“ und erst sich an der Stelle bemerkbar macht wo der Fehler auch tatsächlich behoben werden kann.

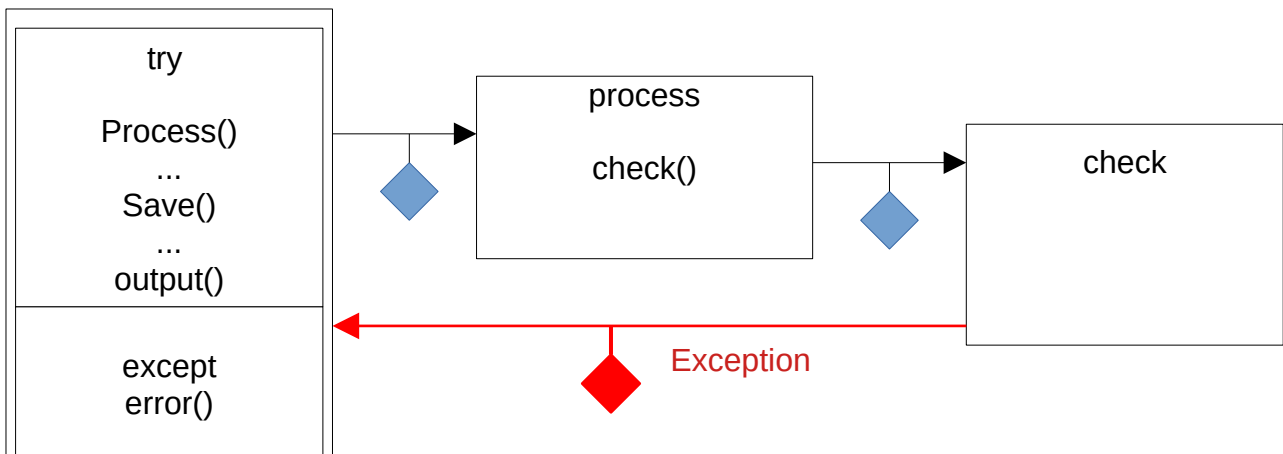


Abbildung 201: Fehlerbehandlung mittels Ausnahmen

Im Vergleich dazu (Abbildung 202) im gleichem Anwendungsfall ohne Ausnahmen sieht es deutlich komplizierter aus, da die Fehler unter Umständen über mehrere Stufen hinweg zu der Hauptruine abgefangen und durchgeschleift werden müssen. Das sorgt dafür das die Software einerseits durch die vielen **if** Abfragen komplexer wird, andererseits wächst die Gefahr, das hier Fehler von der Anwendung „verschluckt“ werden, falls vergessen wurde die entsprechenden Abfragen samt Weiterleitung zu implementieren.

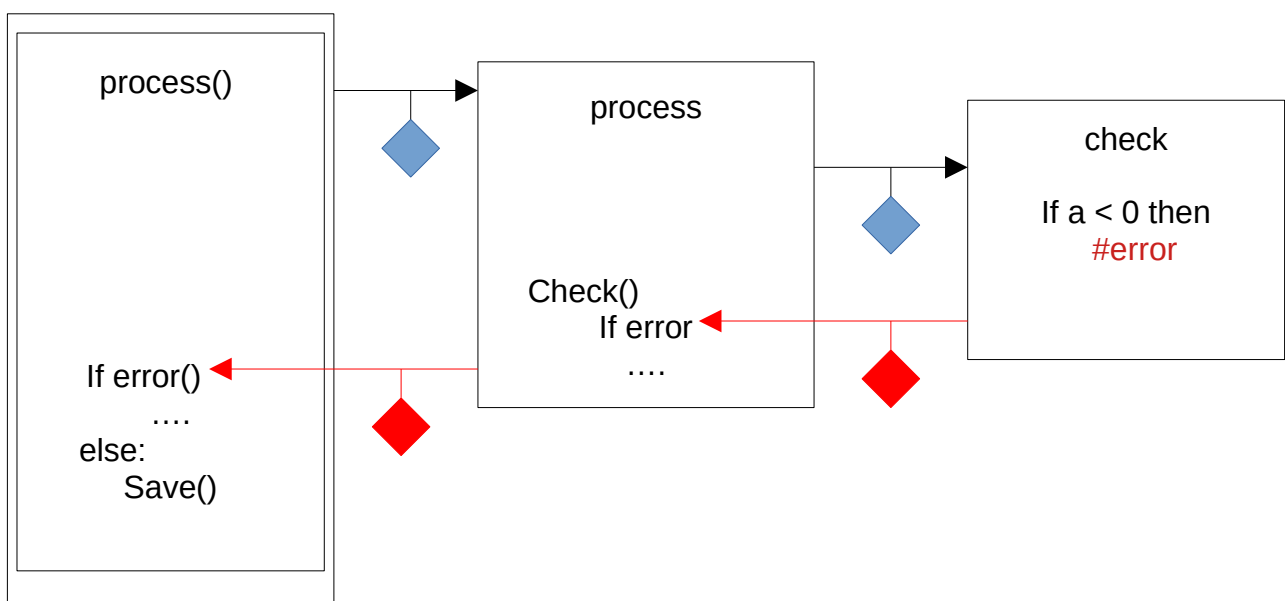


Abbildung 202: Fehlerbehandlung ohne Ausnahmen

## 2.14 Context Manager

Da das abfangen der Ausnahmen mit der **try..except...finally** Sequenz wenig Eleganz aufweist und dies auch noch bei jeder Verwendung des Codes, der mit Ausnahmen um sich herum werfen kann, separat implementiert werden muss, gibt es bei Python für solche Fälle eine deutlich schönere Lösung, die auch noch dazu wiederverwendet werden kann. Dies ist die Domäne des Context Manager. Dieser verkapselt die Problematik die Ausnahmen mit sich bringen, so dass diese nicht direkt an den Stellen, wo diese auftreten können behandelt werden müssen. Der Context Manager selbst wird von der **with** Anweisung gesteuert.

### 2.14.1 Anweisung With

Die Anweisung **with** eröffnet einen Anweisungsblock, in dem eine oder mehrere Objekte in einer Art „geschütztem Modus“ manipuliert werden. Die Ausführung von den einzelnen Anweisungen auf diese Objekte wird vom Context Manager überwacht und im Falle einer Ausnahme wird diese zuerst vom spezifischen Context Manager abgefangen. Für jedes der überwachten Objekte ist ein separater Context Manager zuständig. Ein typischer Beispiel für die Verwendung der **with** Anweisung ist das Arbeiten mit Dateien. Hier sollte, auch bei einer Ausnahme, die geöffnete Datei nach dem Zugriff auf jedem Fall wieder geschlossen werden, da sonst bei einer zum Schreiben geöffneten Datei kein weiterer Schreibzugriff mehr möglich ist (Abbildung 203). Damit dieser Fall hier nicht eintritt, sorgt der mit der Context Manager **open**, der die Datei im Falle einer Ausnahme automatisch schließt.

```
with open('test.txt', 'a') as file:  
    file.write("Hallo SSD")  
  
with open('test.txt', 'r') as file:  
    print(file.read())
```

Hallo SSD

Abbildung 203: With Anweisung

## 2.14.2 Context Manager

Um zu überhaupt angewandt zu werden benötigt die **with** Anweisung eine besonderen Klasse in der das abfangen der Fehler gehandhabt wird. Diese Klasse repräsentiert den sogenannten Context Manager. Im Grunde genommen besteht diese Klasse nur aus Zwei Methoden – der `__enter__()` Methode die automatisch bei initialisieren des Context Managers (am Anfang des with Blocks) Aufgerufen wird, sowie der `__exit__()` Methode die bei verlassen des **with** Blocks zu Ausführung kommt. Der With Block wird dabei entweder nach Beendigung der letzten Instruktion des Blocks oder im Fall einer Exception verlassen. In beiden fällen wird die Methode `__exit__()` automatisch aufgerufen. Sollte bei der Initialisierung des in dem Context Manager verwendeten Objekts einer, oder mehrere Parameter verwendet werden, so ist auch zusätzlich eine entsprechende `__init__()` Methode notwendig. Schauen Wir uns das am Beispiel eines Context Managers der eine Instanz der Klasse **Zeichenkette** verwaltet.

### 2.14.2.1 Die Klasse Zeichenkette

Damit unserer Context Manager was zu verwalten hat benötigt er ein natürlicherweise Objekt, diese funktion erfüllt in unserem Beispiel die Klasse Zeichenkette. An sich ist diese Klasse nichts besonderes, Sie beinhaltet ein Objekt der Klasse string zu dem, mit der Methode `append()`, ein weiterer string Objekt konkatenieren werden kann. Wichtig dabei ist, das im Fall dass das zu anhängende Objekt keine Objekt der Klasse string ist, eine Ausnahme ausgelöst wird. Zu Anschauungszwecken bauen wir in die `append()` funktion noch zwei Debugausgaben um zu sehen wann die funktion betätigt bzw. verlassen wird. Neben der `append()` funktion sind hier noch zwei weitere Funktionen implementiert - `clear()` um die Zeichenkette zu löschen sowie `print()` die die aktuelle Zeichenkette ausgibt.

```
class Zeichenkette:
    string = ""
    def append(self, string):
        print("append")
        if type(string) != type(""):
            raise Exception

        self.string += string
        print("append...done")

    def clear(self):
        self.string = ""

    def print(self):
        print(self.string)
```

Abbildung 204 Die Klasse Zeichenkette

### 2.14.2.2 Der passende Context Manager

Der eigentlich Context Manager (Abbildung 205) besteht wiederum aus drei Methoden. Da ist zuerst die Methode `__init__()`. Sie dient als der Konstruktor dieses Context Manager, diese ist in unserem Fall notwendig, da wir die Klasse **Zeichenkette** mit einem Parameter initialisieren können. Hat die zu verwaltende Klasse keinen Parameter im Konstruktor, kann die `__init__()` Methode entfallen. In dieser Methode wird der übergebene Parameter als die Anfangszeichenkette verwendet und über die Methode **append()** an die vorhandene (leere) Zeichenkette angehängt. Es folgt die Methode `__enter__()` sie besteht im Grunde genommen aus einer Debugausgabe sowie, sehr wichtig, einer `return` Anweisung die das verwaltete Objekt zurück gibt – das ist auch das Objekt was von der **with** Anweisung „überwacht“ wird.

```
class ZeichenkettenManager:

    zeichenkette = Zeichenkette()
    def __init__(self, string):
        self.zeichenkette.append(string)

    def __enter__(self):
        print("***entering***")
        return self.zeichenkette

    def __exit__(self, exc_type, exc_value, exc_tb):
        print("***exiting***")

        print(isinstance(exc_value, Exception))
        if isinstance(exc_value, Exception):
            print(exc_type, exc_value, exc_tb, sep=" : ")

        return True
```

Abbildung 205 Beispiel eines Context Managers

Als letzte Methode des Context Managers ist noch die Methode `__exit__()` implementiert. Diese prüft ob bei Operationen auf dem Objekt zu Ausnahmen gekommen ist und fängt diese automatisch ab. Im dem Fall das es bei der Manipulation des Objekts zu einer Ausnahme gekommen ist beinhaltet der Parameter **exc\_value** ein Objekt der Klasse exception, oder eine davon abgeleitete Klasse. Falls hier auch tatsächlich eine Ausnahme zu Beendigung des with Blocks geführt hat, werden die Parameter der Ausnahme auf der Konsole ausgegeben. Zu guter Letzt kommt noch die `return` Anweisung. Diese ist an sich nicht notwendig, aber – wird hier nichts zurückgegeben, dann wird eine etwaige Ausnahme einfach weitergeleitet und führt in dem with Block zu einer exception. Wird hingegen ein Wert **True** zurückgegeben, dann wird die Ausnahme in der `__exit__()` Methode „abgewürgt“ und der **with** Block merkt nichts von dieser Ausnahme. Aus diesem Grund kann der Context Manager die **try..except..finally** Anweisungsfolge auf sehr elegante

Art und weise ersetzen was der Lesbarkeit des Codes zu gute kommt da die Bearbeitung der Ausnahmen in dem Context Manager „versteckt“ wird.

### 2.14.2.3 Das Zusammenspiel

Nun wie funktioniert das in der Praxis ?in der Abbildung 206 Sehen sie zuerst eine fehlerfreie

Ausführung der **with** Anweisung

nach den beiden Konstruktoren (die ersten zwei Zeilen der Ausgabe)

wird die `__enter__()` Methode

aufgerufen , sichtbar gemacht durch

die Ausgabe in der 3ten Zeile.

Danach wird die Zeichenkette

„Welt“ angehängt und die neu

zusammengesetzte Zeichenkette

Ausgegeben. Da mit der `print()`

Methode die Letzte Anweisung des

**with** Block ausgeführt wurde kann jetzt die `__exit__()` Methode ihre Wirkung entfalten, so das der Context Manager damit auch verlassen wird.

```
with ZeichenKettenManager("Hallo") as zeichenkette:
    zeichenkette.append(" Welt")
    zeichenkette.print()
```

```
append
append...done
**entering**
append
append...done
Hallo Welt
**exiting**
```

Abbildung 206 Fehlerfreie Ausführung des Context Managers

Was Passiert nun in einem Fehlerfall? In der Abbildung 207 ist so ein Fall abgebildet. Der jetzt an

die Methode `append()` übergebene Parameter ist vom Typ `integer` so das die `append` Methode jetzt

eine Ausnahme „produziert“. Dies ist zu erkennen das jetzt das zweite „`append...done`“ nicht mehr

in der Ausgabe erscheint, das die entsprechende Codezeile nie erreicht wurde da vorher eine

exception ausgelöst wurde. Statt dessen wird jetzt an der Stelle die Ausgabe der Ausnahme sichtbar.

Da die `__exit__()` Methode mit einem Wert von `True` verlassen wurde ist jetzt keine „externe“

Ausgabe der Exception sichtbar. Es st auch nicht schwer zu erkennen das der eigentliche Code

deutlich schöner aussieht als wenn hier die die Ausnahme Manuell abgefangen wurde.

```
with ZeichenKettenManager("Hallo") as zeichenkette:
    zeichenkette.append(12345)
    zeichenkette.print()
```

```
append
append...done
**entering**
append
**exiting**
<class 'Exception'> : : <traceback object at 0x000002010983D9C0>
```

Abbildung 207 Fehlerhafte Ausführung des Context Managers

Eine generalisierte Zeitliche Abfolge der Vorgänge illustriert die Abbildung 208.

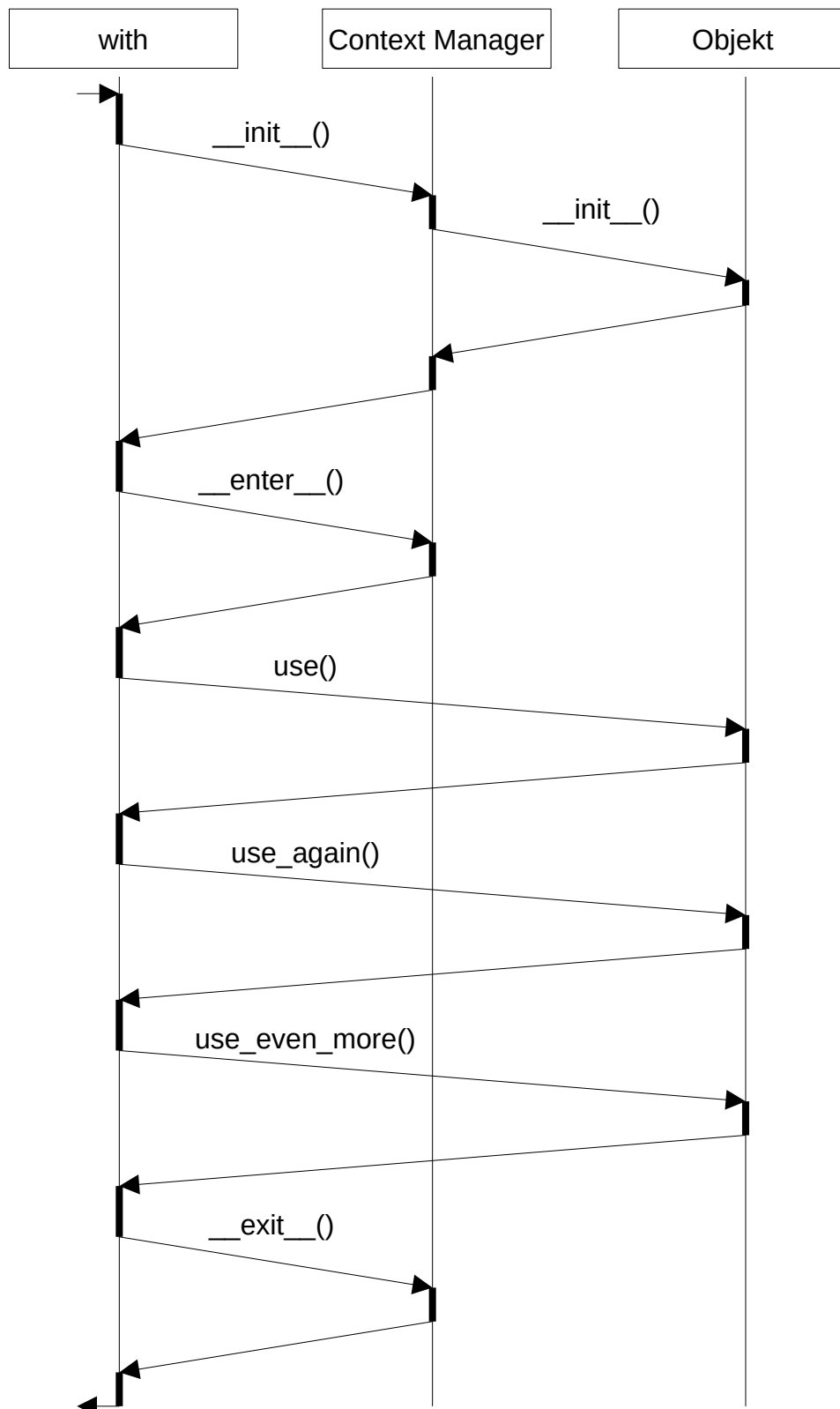


Abbildung 208: Zusammenspiel von der `with` Anweisung mit einem Context Manager

## 3 Numpy

### 3.1 Datentypen

#### 3.1.1 Ganzzahlige Datentypen

Numpy kennt folgende Ganzzahlige Datentypen:

- `int8` – 8 Bit vorzeichenbehaftet (-128 bis 127) oder ( $-2^7$  bis  $2^7-1$ )
- `uint8` – 8 Bit vorzeichenlos (0 bis 255) oder (0 bis  $2^8-1$ )
- `int16` – 16 Bit vorzeichenbehaftet (-32768 bis 32767) oder ( $-2^{15}$  bis  $2^{15}-1$ )
- `uint16` – 16 Bit vorzeichenbehaftet (0 bis 65535) oder (0 bis  $2^{16}-1$ )
- `int32` – 32Bit vorzeichenbehaftet (-viel bis viel-1) oder ( $-2^{31}$  bis  $2^{31}-1$ )
- `uint32` – 32 Bit vorzeichenbehaftet (0 bis viel\*2) oder (0 bis  $2^{32}-1$ )
- `int32` – 32Bit vorzeichenbehaftet (-viel bis viel-1) oder ( $-2^{31}$  bis  $2^{31}-1$ )
- `uint32` – 32 Bit vorzeichenbehaftet (0 bis viel\*2 -1) oder (0 bis  $2^{32}-1$ )
- `int64` – 64Bit vorzeichenbehaftet (-sehr viel bis sehr viel-1) oder ( $-2^{63}$  bis  $2^{63}-1$ )
- `uint64` – 64 Bit vorzeichenbehaftet (0 bis sehr viel\*2 -1 ) oder (0 bis  $2^{64}-1$ )

Die Datentypen selbst verhalten sich anders als bei Python und zwar so, wie man es aus der Assembler/C Welt erwarten würde. Es treten sowohl über- wie auch Unterläufe auf (Abbildung 209). So wird aus der Zahl 128 im vorzeichenbehafteten 8 Bittigen Zahlensystemsystem -128 das dies die nächste Zahl nach 127 ist (vergl. Zahlenkreis) auch die 65537 wird nachdem sie letzte Stelle Erreicht wurde zu 1 (65535 → 0 → 1) was auch über dem Zahlenkreis zu erklären ist. Die Namen

```
print(np.int8(128))
print(np.uint8(128))
print(np.int16(128))
print(np.int16(65537))
```

```
-128
128
128
1
```

Abbildung 209 :



dieser Datentypen lassen eine Abstammung von dem C99 Standard erkennen wo diese auch sehr ähnlich benannt sind (so `uint8` ↔ `uint8_t`).

Neben diesem Ganzzahltypen existieren noch weitere auf alten C Standard Typen basierende Datentypen. Die Tatsächliche Breite, und damit auch das „Fassungsvermögen“ dieser Datentypen ist von der verwendeten Plattform (wie im C) abhängig!

- **`byte`** – vorzeichenbehaftetes Charakter (*signed char*)
- **`ubyte`** – vorzeichenloses Charakter (*unsigned char*)
- **`short`** - vorzeichenbehafteter kleiner integer (*short*)
- **`ushort`** - vorzeichenloser kleiner integer (*unsigned short*)
- **`intc`** - vorzeichenbehafteter integer (*int*)
- **`uintc`** - vorzeichenbehafteter integer (*unsigned int*)
- **`int_`** - vorzeichenbehafteter langer integer (*long*)
- **`uint`** - vorzeichenloser integer (*unsigned long*)
- **`longlong`** - vorzeichenbehafteter sehr langer integer (*long long*)
- **`ulonglong`** - vorzeichenloser sehr integer (*unsigned long long*)

Vor allem bei umstieg vom 32 auf 64 Bit Systeme unterscheiden sich für gewöhnlich die Typen ab **`intc`** in der Breite ab. Für die Bytes (8 Bit) sowie short (16 Bit) kann man zumindest fast immer davon ausgehen das diese 8 bzw. 16 Bit breit sind.

### 3.1.2 Fließkommazahlen

Neben den Festkomma Datentypen Auch diese entsprechen den Typen aus der C Welt.

- **`single`** – 32 Bit (1Bit Vorzeichen, 8 Bit Exponent, 23 Bit Mantisse)
- **`double`** – 64 Bit (1Bit Vorzeichen, 11 Bit Exponent, 52 Bit Mantisse)
- **`longdouble`** – plattformabhängiger Fließkomatyp mit erweiterter Genauigkeit.

Bei der Darstellung werden

```
print(np.single(1.3))  
print(np.double(0.3))  
print(np.longdouble(-1))
```

```
1.3  
0.3  
-1.0
```

Abbildung 210 :

### 3.1.3 Komplexe Zahlen

Ähnlich sieht es bei den Komplexen Zahlen, auch hier gibt es wie bei den Fließkommazahlen drei auf den drei Fließkommazahlen basierte Datentypen:

- `csingle` – (single + single\*i)
- `cdouble` – (double + double\*i)
- `clongdouble` – (longdouble + longdouble\*i)

## 3.2 Arange

Mit der Anweisung **arange** lassen sich Numpy Felder erzeugen. Dabei kann der Datentyp der Feldellenente als Parameter mit angegeben werden (Abbildung 211). Zu beachten ist, dass hier im Unterschied zu der Abbildung 210 die Zahl 1.3 nicht mehr korrekt dargestellt wird. Das ist soweit korrekt, da diese Zahl auch tatsächlich keine entsprechende binäre Darstellung besitzt. Als Parameter können hier

Anfangsposition (**start**), Endposition (**stop**), Schrittgröße (**step**) sowie der Datentyp (**dtype**) angegeben. Bei Namenlosen Parametereingabe und einem Parameter wird dieser als die Endposition interpretiert die Startposition ist dann in diesem Fall 0. Bei zwei angegebenen Parameter sind das Anfangsposition sowie die Endposition (Abbildung 212). In beiden Fällen wird die Schrittlänge automatisch auf 1 gesetzt. Die Schrittweite selbst kann auch negative Werte aufnehmen, in diesem Fall wird das Feld von der größten bis zu kleinsten Zahl arrangiert.

```
x = np.arange(start = 1, stop = 10, step = 3)
y = np.arange(1, 1.4, 0.1, dtype=np.single)
z = np.arange(1, 300, 50, dtype=np.int8)
print(x)
print(y)
print(z)
```

```
[1 4 7]
[1. 1.1 1.2 1.3000001]
[ 1 51 101 -105 -55 -5]
```

Abbildung 211 : Felder generieren mit arange

```
x = np.arange(5)
y = np.arange(5, 1)
z = np.arange(5, 1, -1)
print(x)
print(y)
print(z)
```

```
[0 1 2 3 4]
[]
[5 4 3 2]
```

Abbildung 212 : Arange mit reduzierten Parametern oder negativen Schrittlänge

## 4 Mathplotlib

### Farben

Folgende Grundfarbtöne sind im plotlib als Einzeichenparameter vordefiniert:

- 'b' - blau
- 'g' – blau
- 'r' – rot
- 'c' – cyan
- 'm' – magenta
- 'y' – gelb
- 'k' – schwarz
- 'w' – weis

Außer den Grundfarbtönen lassen sich auch weitere Farben verwenden.

//ToDo wie ?





















## 4.1.1

### 4.1.1.1 Punkte Zeichnen

Darstellung der Punkte änderbar

punkte

Folgende Typen von Punkten sind möglich:

-  • `'.'` - runder Punkt
-  • `','` - runder kleiner Punkt
-  • `'o'` – runder ausgefüllter Kreis
-  • `'^'` – Dreieck nach oben gerichtet
-  • `'>'` – Dreieck nach rechts gerichtet
-  • `'v'` – Dreieck nach unten gerichtet
-  • `'<'` – Dreieck nach links gerichtet
-  • `'1'` – „Pfeil“ nach unten zeigend
-  • `'2'` – „Pfeil“ nach oben zeigend
-  • `'3'` – „Pfeil“ nach links zeigend
-  • `'4'` – „Pfeil“ nach rechts zeigend
-  • `'s'` – Rechteck
-  • `'p'` – Fünfeck
-  • `'*'` – Stern
-  • `'h'` – Hexagon 1
-  • `'H'` – Hexagon 2
-  • `'+'` – Kreuzformiger Punkt
-  • `'x'` – gekippter Kreuzformiger Punkt
-  • `'d'` – schmaler Diamant
-  • `'D'` – breiter Diamant

- '|' – vertikale Linie
- '\_' – horizontale Linie

#### 4.1.1.2 Punktgröße ändern

Über dem Parameter **markersize** lässt sich bestimmen wie groß die Punkte in dem Plot dargestellt werden sollten (Abbildung 213). Die Maßeinheit für den Parameter ist „Punkt“ was wohl einem Bildpunkt des Diagramms entspricht. So wirken die Punkte gleicher Größe auf dem Bildschirm unterschiedlich groß falls die für die Diagramme verwendete dpi Einstellungen nicht gleich ist.

```
plt.plot(1,1,'or',markersize=5)  
plt.plot(2,1,'or',markersize=10)  
plt.plot(3,1,'or',markersize=20)  
plt.show()
```

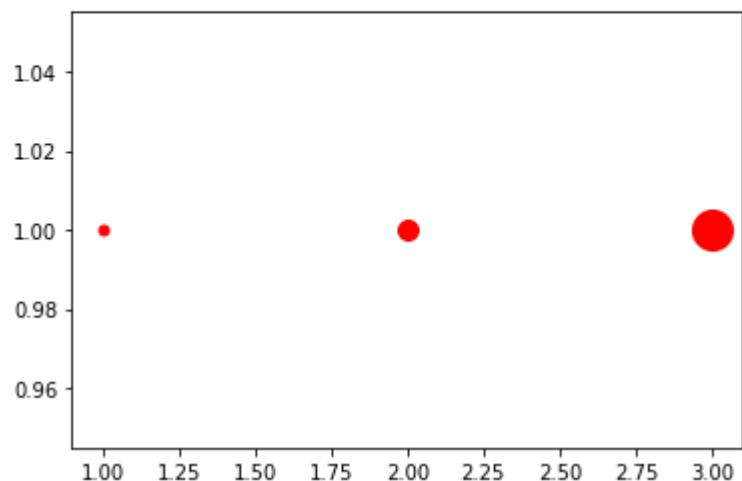


Abbildung 213 : Größe der dargestellten Punkte bestimmen

#### 4.1.2 Linien Zeichnen

Das Zeichnen der Linien erfolgt durch die gleiche Methode wie das Zeichnen der Punkte, allerdings werden bei zeichnen der Linien andere Parameter verwendet die dann die entsprechende Linienart

Spezifizieren (Abbildung 214). Im Unterschied zu den großen Diversität bei der Punktarten stehen bei den Linien nur vier verschiedene Typen zu Verfügung (Abbildung 214):

- '-' eine durchgezogene Linie
- '--' eine unterbrochene Linie
- '-.' eine „Stricht-Punkt“ Linie
- '..' eine gepunktete Linie

Werden verschiedene Linien in einem Plot gezeichnet, so werden diesen Linien automatisch unterschiedliche Farben zugewiesen. Andererseits lassen sich die Farben für eine Linie exakt spezifizieren, wie es für die erste Linie in dem nebenstehenden Beispiel mit '-k' geschehen ist. So bedeutet das '-' dass hier eine durchgezogene Linie in der Farbe Schwarz (Parameter 'k') gezeichnet werden sollte.

```
plt.plot ([1, 2],[1, 1], '-k')
plt.plot ([1, 2],[2, 2], '--')
plt.plot ([1, 2],[3, 3], '-.')
plt.plot ([1, 2],[4, 4], ':')
plt.show()
```

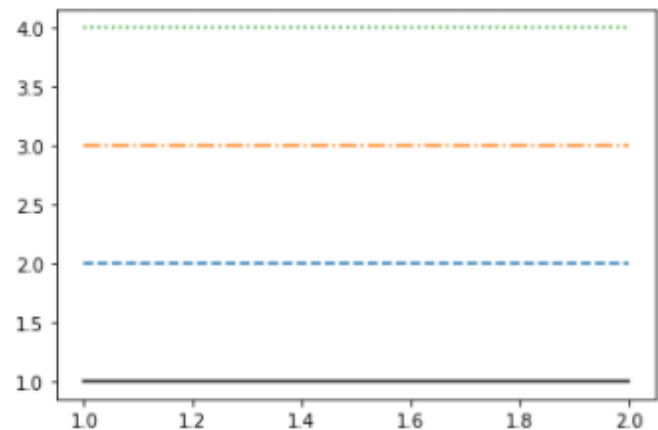


Abbildung 214 : Plottern von Linien

#### 4.1.2.1 Linienbreite

Für die Kontrolle der Linienbreite ist der Parameter **linewidth** zuständig. Auch hier analog zu der Angabe von Punktgrößen

```
plt.plot ([1, 2],[1, 1], linewidth = 1)
plt.plot ([1, 2],[2, 2], linewidth = 2 )
plt.plot ([1, 2],[3, 3], linewidth = 5)
plt.show()
```

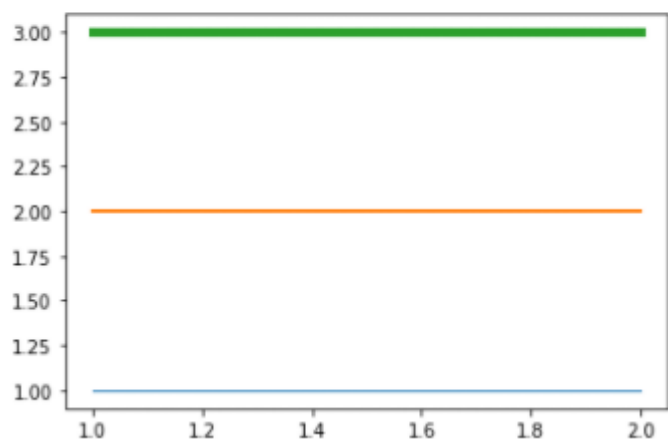


Abbildung 215

### 4.1.3 Boxplot

Ein Boxplot

## 4.2 Figure

Das eigentliche Zeichnen findet in der Klasse **figure** statt. Sie stellt dem Basiscontainer für die Zeichenoperationen eines Plots dar. Dazu gehören nicht nur die gezeichneten „Inhalte“ wie Linien und Punkte, sondern auch alle Rahmenelemente des eigentlichen Plots.

Nicht mehr benötigte Instanzen der Klasse **figure** sollten mit der funktion `close()` geschlossen werden.

```
figure = plt.figure()
plot = figure.add_subplot(1, 1, 1)
plt.show()
```

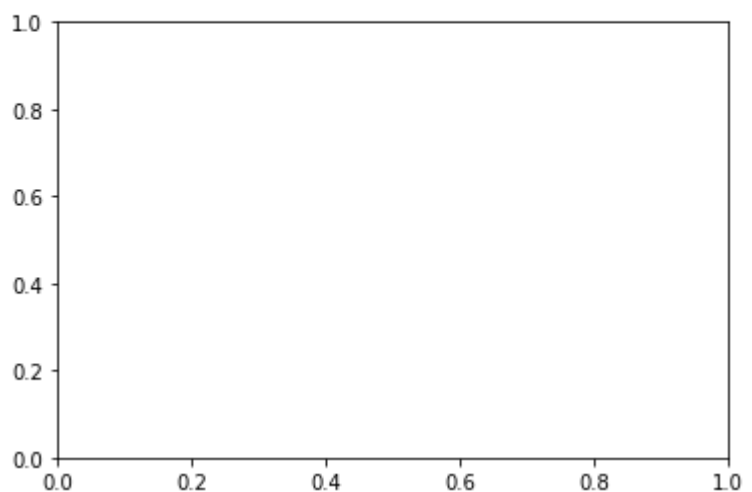


Abbildung 216 :



## Table of Figures

Abbildung 1: Funktionsweise von Python.....	7
Abbildung 2: Python Funktion als Bytecode.....	8
Abbildung 3: Erster Schritt in Python.....	8
Abbildung 4 Python in der Kommandozeile.....	9
Abbildung 5 Python-Quelltextdatei ausführen.....	10
Abbildung 6 Python-Kommandozeilenanweisung im interaktiven Modus.....	10
Abbildung 7 Replit in Dateimodus.....	11
Abbildung 8: Replit im interaktiven Modus.....	11
Abbildung 9 Python programmieren mit Online GDB.....	12
Abbildung 10 Python-Anwendungen debuggen mit online GDB.....	12
Abbildung 11 Python in Jupyter.....	13
Abbildung 12 Jupyter Ausgabe unterdrücken.....	13
Abbildung 13 VS Code Python „Get Started“ Karteireiter.....	14
Abbildung 14 VS Code Python interactive window.....	14
Abbildung 15 Einfache Ausgaben mit print().....	16
Abbildung 16: Ausgabe einer Liste.....	16
Abbildung 17 Nicht als Zeichenkette darstellbare Eingaben.....	16
Abbildung 18: Separatoren bei Ausgabe.....	17
Abbildung 19 Zeilenumbruch bei print().....	17
Abbildung 20 print() Ausgabe umlenken.....	17
Abbildung 21.....	18
Abbildung 22.....	19
Abbildung 23: Variablen Objekte und Ihre Adressen.....	19
Abbildung 24 Primitive Datentypen.....	20
Abbildung 25: Automatische Datentypkonvertierung.....	20
Abbildung 26: Größe von Int in Python.....	21
Abbildung 27 Größe eines Objekts in Bytes ermitteln.....	22
Abbildung 28 Speicherbelegung einer Datenstruktur am Beispiel einer Liste.....	22
Abbildung 29 Datentypen Konvertieren.....	23
Abbildung 30 Vergleichsoperatoren Auswerten.....	23
Abbildung 31 Numerische Ausdrücke cholerisch Auswerten.....	24
Abbildung 32: Zeichenketten boolesch auswerten.....	24
Abbildung 33: Boolesche Ausdrücke verrechnen.....	25
Abbildung 34 Ergebnis einer Addition von booleschen Ausdrücken.....	25
Abbildung 35 Boolesche Ausdrücke & boolesche Algebra.....	25
Abbildung 36 Zeichenketten in Python.....	26
Abbildung 37 Speicherung von Zeichenketten in Python.....	26
Abbildung 38: Hochkomas ausgeben.....	26
Abbildung 39: Mehrzeilen Zeichenkette.....	27
Abbildung 40: Zeichenkette als Feld.....	28
Abbildung 41: Zugriffe auf Zeichen eines Strings.....	28
Abbildung 42 Zugriffe auf Zeichen eines Strings.....	28
Abbildung 43.....	29
Abbildung 44: Zeichenketten Zusammensetzen.....	29

Abbildung 45: Zeichenketten sind Statisch.....	29
Abbildung 46 Listen mit join umwandeln.....	30
Abbildung 47 Größe einer Kollektion bestimmen.....	31
Abbildung 48 Kollektionen umwandeln.....	31
Abbildung 49 Elemente einer Kollektion iterieren.....	32
Abbildung 50 Zugehörigkeit eines Elements prüfen.....	32
Abbildung 51 Zugriffe über Indexe.....	32
Abbildung 52: Negative Indexierung.....	33
Abbildung 53 Kollektion vereinigen.....	33
Abbildung 54 Container löschen.....	34
Abbildung 55 Kollektionen sortieren.....	34
Abbildung 56: Elementen folge einer Kollektion umkehren.....	35
Abbildung 57 Elemente summieren.....	35
Abbildung 58 max() und min() Funktionen.....	36
Abbildung 59 Funktionsweise von all() und any().....	36
Abbildung 60 Definition eines Sets in Python.....	37
Abbildung 61 Einzelne Elemente einfügen.....	37
Abbildung 62 Sets einfügen mit update.....	37
Abbildung 63 Einzelne Elemente eines Sets entfernen.....	38
Abbildung 64 Letztes Element eines Sets entfernen.....	38
Abbildung 65 Alle Elemente eines Sets entfernen.....	38
Abbildung 66 Schnittmengen Bilden.....	39
Abbildung 67: Differenz Bilden.....	39
Abbildung 68 Symmetrische Differenz zweier Sets bilden.....	40
Abbildung 69 Auf Untermenge prüfen.....	40
Abbildung 70 Auf Obermenge prüfen.....	41
Abbildung 71 Prüfung ob zwei Sets disjunkt sind.....	41
Abbildung 72: Sets Kopieren.....	42
Abbildung 73: Sets vereinigen.....	42
Abbildung 74: Tupel definieren.....	43
Abbildung 75 Anzahl der Vorkommnisse ermitteln.....	43
Abbildung 76 Index eines Elements im Tupel ermitteln.....	43
Abbildung 77 Eigenschaften eines Tupels ändern.....	44
Abbildung 78: Definition einer Liste in Python.....	45
Abbildung 79 Länge einer Liste ermitteln.....	45
Abbildung 80 Häufigkeit eines Elements in einer Liste zählen.....	46
Abbildung 81 Erstes Vorkommen eines Elements in Liste ermitteln.....	46
Abbildung 82 Reihenfolge in der Liste umkehren.....	46
Abbildung 83 Elemente einer Liste sortieren.....	47
Abbildung 84 Elemente einer Liste nach Schlüsseln sortieren.....	47
Abbildung 85 Liste leeren.....	47
Abbildung 86 Elemente hinzufügen.....	48
Abbildung 87 Elemente aus einer Liste entfernen.....	49
Abbildung 88: Liste kopieren.....	49
Abbildung 89 Ein Dictionary definieren.....	50
Abbildung 90 Dictionary aus Indexenliste und Defaultwert definieren.....	50
Abbildung 91: Dictionarys über die zip klasse definieren.....	51
Abbildung 92 Werte eines Dictionary auslesen.....	51

## Einführung in Python

Abbildung 93: Wert eines Schlüssels Ändern.....	52
Abbildung 94: ein Dictionary Leeren.....	52
Abbildung 95 Ein Dictionary kopieren.....	52
Abbildung 96 Ein Dictionary mehrfach referenzieren.....	53
Abbildung 97 Schlüsselliste erstellen.....	53
Abbildung 98 Werteliste erstellen.....	53
Abbildung 99 Auflistung aller Wertepaare in einem Dictionary.....	54
Abbildung 100 dict_items in eine Liste umwandeln.....	54
Abbildung 101 Wert auslesen.....	54
Abbildung 102 Wert aus einem Dictionary holen.....	55
Abbildung 103 Letztes Wertepaar holen.....	55
Abbildung 104 Default Wert setzten und Wert auslesen.....	56
Abbildung 105 Die if Anweisung.....	56
Abbildung 106 Die else Anweisung.....	57
Abbildung 107: Die elif Anweisung.....	57
Abbildung 108 Vierschachtelung bei if Abfragen.....	57
Abbildung 109 if..else.....	58
Abbildung 110: for...break.....	58
Abbildung 111 for..continue.....	59
Abbildung 112 for..pass.....	59
Abbildung 113 Ein Range Objekt.....	59
Abbildung 114 Negative Attribute in einem Range Objekt.....	60
Abbildung 115 Weitere Anwendungen der Klasse Range.....	60
Abbildung 116 for i =0; i < 10;i +=3.....	60
Abbildung 117 Range mit Variablen.....	61
Abbildung 118: Zeichenketten iterieren.....	61
Abbildung 119: Zeile einer Zeichenkette iterieren.....	61
Abbildung 120 while Schleife.....	62
Abbildung 121: Endlose while schleife.....	62
Abbildung 122 while ...else.....	63
Abbildung 123 break bei while.....	63
Abbildung 124 filtern mit continue.....	64
Abbildung 125 schöner Filtern mit Python.....	64
Abbildung 126: Die Mutter aller Endlosschleifen.....	64
Abbildung 127: Funktionen Dokumentieren.....	65
Abbildung 128 Dokumentation der print()-Funktion.....	65
Abbildung 129 Funktion mit einem immutable Parameter.....	66
Abbildung 130 Funktion mit einem mutable Parameter.....	67
Abbildung 131 Funktion mit einem mutable Parameter ohne Rückgabewert.....	67
Abbildung 132 Funktion mit einem mutable Parameter mit neuem Rückgabewert.....	68
Abbildung 133 Default Parameter.....	68
Abbildung 134 Optionale Parameter.....	69
Abbildung 135: Parameternamen als Schlüsselwörter.....	69
Abbildung 136 Funktionen mit Schlüsselwörtern und variablen Parameter.....	69
Abbildung 137 Rückgabeparameter.....	70
Abbildung 138: Funktion mit mehreren Rückgabeparametern.....	70
Abbildung 139 Generische Funktionen in Python.....	71
Abbildung 140: Grenzen der generischen Programmierung in Python.....	71

## Einführung in Python

Abbildung 141: Funktionen als Parameter in Funktionen.....	72
Abbildung 142: Funktionen als Parameter in einem <i>Set</i> .....	72
Abbildung 143: Kleiner Befehlsinterpreter in Python.....	72
Abbildung 144: Funktionen kopieren und löschen.....	73
Abbildung 145: Vorhandene Funktionen erweitern.....	73
Abbildung 146: Wahlfreier Zugriff auf Elemente(ohne Iterator).....	74
Abbildung 147: Zugriff auf Elemente mittels eines Iterators.....	74
Abbildung 148: Zugriffe auf einem Iterator.....	74
Abbildung 149: Iteratoren erstellen.....	75
Abbildung 150: Iterieren mit der next() Anweisung.....	75
Abbildung 151: einem Iterator mit for auslesen.....	76
Abbildung 152: Funktionsweise von zip.....	76
Abbildung 153: Entpacken mit zip.....	76
Abbildung 154: Kollektionen indizieren.....	77
Abbildung 155: die map Funktion.....	77
Abbildung 156:.....	77
Abbildung 157: map mit Zeichenketten.....	78
Abbildung 158: automatische Umwandlung von Maps.....	78
Abbildung 159: Anweisung yield.....	79
Abbildung 160: Mehrfache Verwendung von Generatoren.....	79
Abbildung 161: Einem Generator in eine Kollektion umwandeln.....	79
Abbildung 162 einfache Lambdas.....	80
Abbildung 163 Direkte Lambdas.....	80
Abbildung 164 Lambda mit mehreren Parametern.....	80
Abbildung 165: Lambdas mit Auswertungslogik.....	81
Abbildung 166: Filter in Python.....	81
Abbildung 167: Liste aus einer Liste erstellen.....	82
Abbildung 168: Liste mit veränderten Elementen.....	82
Abbildung 169: List Comprehension mit Alphanumerischen Werten.....	82
Abbildung 170: List Comprehension mit Funktionen.....	82
Abbildung 171: Spezialisierung der Klassen.....	83
Abbildung 172: Spezialisierung und Übernahme der Eigenschaften.....	84
Abbildung 173: Definition einer Klasse.....	85
Abbildung 174: eine Klasse verwenden.....	85
Abbildung 175: Zugriff auf Variablen in Klassen.....	86
Abbildung 176: Statische Variablen.....	87
Abbildung 177: Statische Variablen und Instanzen.....	87
Abbildung 178: self Parameter bei einem Aufruf über eine Instanz.....	88
Abbildung 179: self Parameter bei einem Aufruf über eine Instanz.....	88
Abbildung 180: Self Parameter bei einem direktem Aufruf.....	88
Abbildung 181: Self Parameter bei einem direktem Aufruf.....	88
Abbildung 182: eine Methode zu einer Klasse hinzufügen.....	90
Abbildung 183 Methoden Aliase.....	90
Abbildung 184: Verwendung der ursprünglichen Methode für Objekte.....	90
Abbildung 185: Klassen ableiten.....	91
Abbildung 186: Die Klasse None.....	92
Abbildung 187: Überprüfung ob eine Instanz „gültig“ ist.....	92
Abbildung 188: Abstammung einer Klasse testen.....	93

## Einführung in Python

Abbildung 189: Verarbeitungswege von Tochterinstanzen Trennen.....	93
Abbildung 190: Zahlen Verrechnen.....	94
Abbildung 191: „Normale“ Fehlerbehandlung.....	94
Abbildung 192 Fehler Abfangen mit exceptions.....	95
Abbildung 193 Lösung mit <i>Ausnahmen</i> .....	95
Abbildung 194 Lösung ohne Ausnahmen.....	96
Abbildung 195 eine Ausnahme auslösen.....	96
Abbildung 196: Eine Ausnahme abfangen.....	97
Abbildung 197: Alle Ausnahmen akzeptieren.....	97
Abbildung 198: Eigene Ausnahmen definieren.....	97
Abbildung 199 finally.....	98
Abbildung 200 finally.....	98
Abbildung 201: Fehlerbehandlung mittels <i>Ausnahmen</i> .....	99
Abbildung 202: Fehlerbehandlung ohne Ausnahmen.....	99
Abbildung 203: With Anweisung.....	100
Abbildung 204 Die Klasse Zeichenkette.....	101
Abbildung 205 Beispiel eines Context Managers.....	102
Abbildung 206 Fehlerfreie Ausführung des Context Managers.....	103
Abbildung 207 Fehlerhafte Ausführung des Context Managers.....	103
Abbildung 208: Zusammenspiel von der with Anweisung mit einem Context Manager.....	104
Abbildung 209 :.....	105
Abbildung 210 :.....	107
Abbildung 211 : Felder generieren mit <i>arange</i> .....	108
Abbildung 212 : <i>Arange</i> mit reduzierten Parametern oder negativen Schrittlänge.....	108
Abbildung 213 : Größe der dargestellten Punkte bestimmen.....	111
Abbildung 214 : Plotten von Linien.....	112
Abbildung 215.....	112
Abbildung 216 :.....	113