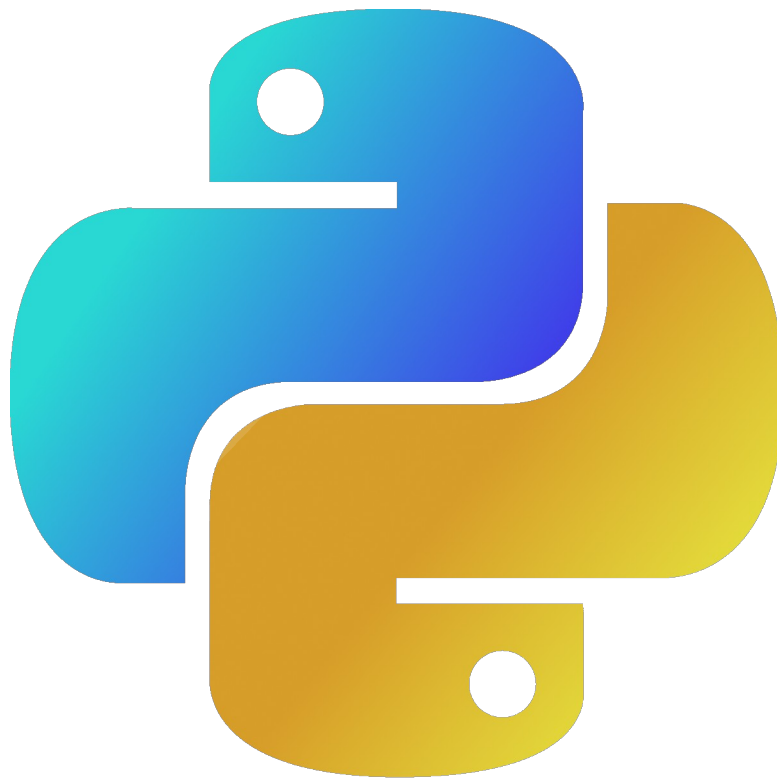


# Python für Einsteiger



Autor: Holger Junge  
Skript: Version 1.1  
Lizenz: [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/)

## Inhaltsverzeichnis

1	Einleitung.....	5
1.1	Python-Versionen.....	6
1.2	Python-Modus.....	6
2	Variablen.....	7
2.1	Strings.....	7
2.2	Integer.....	7
2.3	Long Integer.....	7
2.4	Float.....	7
2.5	Boolean.....	8
2.6	Globale und lokale Variablen.....	8
3	Operatoren.....	9
3.1	Operatoren für einfache mathematische Berechnungen.....	9
3.2	Logische Operatoren.....	9
3.3	Vergleichsoperatoren.....	9
3.4	String-Operationen.....	10
4	Kommentare.....	11
5	Schleifen.....	12
5.1	For.....	12
5.2	While.....	12
5.3	Until.....	13
5.4	Break.....	14
6	Funktionen.....	15
6.1	Typenkonvertierung.....	16
6.2	Mathematische Funktionen.....	16
6.3	Parameter und Argumente.....	17
6.4	Funktions-Variablen und -Parameter.....	17
6.5	Gründe für Funktionen.....	18
6.6	Modul-Import.....	18
7	Bedingungen.....	19
8	Ein- und Ausgabe.....	21
8.1	Eingaben per Tastatur.....	21
8.2	Lesen aus und Schreiben in Dateien.....	21
8.2.1	Lesen aus Dateien.....	21
8.2.2	Parameter und Argumente, die 2.....	23
8.2.3	Schreiben in Dateien.....	23
8.3	Formatierte Ausgabe.....	25
8.4	Dateien und Pfade.....	26
8.4.1	Aktuellen Pfad ausgeben.....	26
8.4.2	Absoluten Pfad einer Datei erhalten.....	26
8.4.3	Datei oder Verzeichnis auf Existenz überprüfen.....	27
9	Datenbanken.....	28
9.1	Das MySQL-Modul.....	28
9.2	Einrichten einer Test-Datenbank.....	29
9.3	Erstellen einer Tabelle.....	30
9.4	Datensätze hinzufügen.....	31
9.5	Lese-Operation.....	33

---

9.6 Update-Operation.....	34
9.7 Lösch-Operation.....	35
10 Systemprogrammierung.....	36
10.1 Ein Shell-Kommando ausführen.....	36
10.2 Die Benutzerumgebung abfragen.....	37
10.3 Ausgabe des aktuellen Verzeichnisses.....	37
10.4 Ausgabe der Gruppen-ID.....	37
10.5 Ausgabe der Benutzer-ID.....	38
10.6 Ausgabe der Prozess-ID.....	38
10.7 Abfrage des Betriebssystem-Kernels.....	38
10.8 Change-Root-Operation.....	39
10.9 Auflistung des Verzeichnisinhaltes.....	39
10.10 Erstellen eines Verzeichnisses.....	40
10.10.1 „Einfaches“ Erstellen von Verzeichnissen.....	40
10.10.2 „Rekursives“ Erstellen von Verzeichnissen.....	40
10.11 Löschen von Dateien.....	41
10.12 Löschen von Verzeichnissen.....	41
10.12.1 „Einfaches“ Löschen von Verzeichnissen.....	41
10.12.2 Rekursives Löschen von Verzeichnissen.....	41
10.13 Umbenennung von Verzeichnissen und Dateien.....	42
11 Strings, die 2.....	43
11.1 Len.....	43
11.2 String-Teile.....	44
11.3 String-Methoden.....	44
11.3.1 Umwandlung in Groß- und Kleinschreibung.....	45
11.3.2 Pattern-Matching.....	46
11.3.3 Substitution.....	46
12 Listen.....	47
12.1 Einzelelemente einer Liste abrufen.....	47
12.2 Einzelne Elemente ändern.....	48
12.3 Methoden und Operationen.....	48
12.3.1 Neue Elemente an eine Liste anhängen.....	48
12.3.2 Listen an Listen anhängen.....	49
12.3.3 Elemente sortieren.....	49
12.3.4 Letztes Element von der Liste entfernen.....	50
12.4 Elemente löschen.....	50
12.5 Listen und Strings.....	51
12.5.1 Funktion list().....	51
12.5.2 Methode split().....	52
12.5.3 Methode join().....	52
13 Tupel.....	53
14 Web-Programmierung mit Python.....	55
14.1 Grundlagen.....	55
14.1.1 Grundlagen HTML.....	55
14.1.2 Grundlagen Python-CGI.....	56
14.2 Formular-Elemente.....	58
14.2.1 Das Textfeld.....	58
14.2.2 Radio-Buttons.....	60

---

14.2.3 Auswahllisten.....	62
14.2.4 Buttons – Schaltflächen.....	64

# 1 Einleitung

Python ist eine interpretierte Programmiersprache. Man unterscheidet in der Informatik zwischen kompilierten und interpretierten Programmiersprachen. Kompilierte Sprachen wie C oder C++ benötigen einen Compiler, der die C- bzw. C++-Quellen in ausführbaren Code übersetzt. Diese daraus entstehenden Exe-Files (unter Linux/Unix auch Binaries genannt) sind nach dieser Übersetzung direkt ausführbar (lauffähig). Bei Python und anderen Interpreter Sprachen sieht der Fall anders aus. Der Python-Code benötigt keine vorangehende Kompilierung, sondern wird direkt über den dazugehörigen Python-Interpreter ausgeführt. Damit Python-Programme jedoch ausführbar sind, ist im Betriebssystem des Computers immer das Vorhandensein dieses speziellen Interpreters erforderlich. Weitere Interpretersprachen sind Perl und PHP. Abbildung 1 zeigt einen Code-Auszug aus dem Modul *turtle.py*.

```
class Shape(object):
    """Data structure modeling shapes.
    attribute _type is one of "polygon", "image", "compound"
    attribute _data is - depending on _type a polygon-tuple,
    an image or a list constructed using the addcomponent method.
    """
    def __init__(self, type_, data=None):
        self._type = type_
        if type_ == "polygon":
            if isinstance(data, list):
                data = tuple(data)
        elif type_ == "image":
            if isinstance(data, str):
                if data.lower().endswith(".gif") and.isfile(data):
                    data = TurtleScreen._image(data)
                # else data assumed to be Photoimage
        elif type_ == "compound":
            data = []
        else:
```

Abbildung 1

Python ist eine leicht zu erlernende Programmiersprache, was ihrer einfachen Syntax zu verdanken ist. Sie ist vielseitig einsetzbar. Man kann mit ihr Programme mit grafischer Oberfläche erstellen, Web-Applikationen entwickeln oder das Betriebssystem steuern und automatisieren. Auch Datenbank-Programmierung ist mit Python möglich.

Wegen seiner leichten Erlernbarkeit ist Python eine der besten Lehrsprachen der Welt und hat bereits seit einigen Jahren Java zum Einstieg in die Programmierung an amerikanischen Universitäten verdrängt.

Python ist frei erhältlich und kann für die Plattformen MS Windows, Mac OS, Linux und weitere unter <https://www.python.org/downloads/> herunter geladen werden.

## 1.1 Python-Versionen

Python-Skripte sind in zwei verschiedenen Python-Versionen im Umlauf. Version 2.X und Version 3.X. Beide wurden auch noch bis Oktober 2019 fortführend weiter entwickelt. Die letzte 2er-Version erschien jedoch am 19.10.2019 in Version 2.7.17. Version 2 wird seit dem nicht weiterentwickelt. Python 3 wird weiter entwickelt. Die 3er-Version ist nicht kompatibel mit der 2er-Version. Ein nicht unerheblich großer Anteil an Python-Programmen wurde in den letzten 20 Jahren entwickelt. Wer die Sprache jedoch neu erlernen möchte, sollte auf die Version 3 zurückgreifen.

## 1.2 Python-Modus

Python kann auf zwei verschiedene Arten ausgeführt werden. Interaktiv in der **IDLE**-Shell oder im **Script-Modus**. Letzterer wird verwendet, wenn ich ein Python-Programm fertig implementiert habe und es als py-Datei abgespeichert habe („file.py“). Wird diese Datei, dieses Script nun in der Kommandozeile aufgerufen oder per Doppelklick in einem Dateimanager gestartet, springt der mit auf dem System installierte Python-Interpreter an und führt das Programm für mich aus. IDLE hingegen kann u.a. über das Python-Menü aufgerufen werden. In ihm kann ich dann einzelne Befehlszeilen von Python testen. Wie ist ihre Wirkungsweise? Habe ich Syntaxfehler? Im Kapitel 2 wurde die maximale Größe einer Integer-Variable über IDLE ermittelt. Hierzu ist in IDLE folgendes einzugeben:

```
>>> import sys
>>> print(sys.maxsize)
```

Und auf die gleiche Art und Weise wurde auch die maximal größte Fließkommazahl in Python ermittelt:

```
>>> sys.float_info.max
```

## 2 Variablen

### Variablen-Typen

Ohne Variablen würde keine Programmiersprache funktionieren. Variablen werden Werte zugewiesen. Im Programmverlauf ändern sich jedoch diese Werte. Der Inhalt ist also „variabel“ (veränderbar).

In der Regel unterscheidet man zwischen Variablenarten, die Zeichen, Zeichenketten beinhalten und welche, die reine Zahlenwerten haben. Zusätzlich gibt es noch die booleschen Variablen. Sie kennen nur zwei Werte (bzw. Zustände): wahr und falsch.

### 2.1 Strings

Ein String ist eine Zeichenkette, die ein bis beliebig viele Zeichen beinhalten kann. Dies können Buchstaben, Sonderzeichen aber auch Zahlen sein. Sollte hierbei beachten, das auf Zahlen, wenn sie als String deklariert sind, keine mathematischen Operationen möglich sind. Die gängigste Verwendung von Strings sind Wörter, Namen und Texte.

### 2.2 Integer

Integer sind Variablen mit numerischen Inhalt. Integer-Zahlen sind ausnahmslos ganze Zahlen (sie haben also keine Nachkommastellen). Die maximale Größe von „int“ (Integer) ist die 19-stellige Zahl 9223372036854775807. Das sind also 9 Trillionen und ein paar „Zerquetschte“.

### 2.3 Long Integer

Möchte ich ganze Zahlen verwenden, die größer sind als 9223372036854775807, ist der Long Integer („long“) zu verwenden.

### 2.4 Float

Bei Zahlen mit Nachkommastellen wird der Variablen-Typ Float („float“) benötigt. Die maximale Größe beträgt  $1.7976931348623157e+308$ . Das ist eine Zahl mit 309 Stellen.  $1 \cdot 10^{300}$  wird Quinquagintillion genannt.

## 2.5 Boolean

Wie oben bereits erwähnt: Kann eine Variable (aus welchen Gründen auch immer) nur zwei Werte annehmen, können Variablen des Typs Boolean („bool“) verwendet werden: wahr und falsch.

Beispiel: „Ist das Licht an oder aus?“ Natürlich kann man hier auch stur Integer („0“, „1“) oder Strings („ja/nein“) verwenden ;-).

## 2.6 Globale und lokale Variablen

Werden Variablen im Hauptprogramm und somit außerhalb von Funktionen deklariert, nennt man sie **globale Variablen**. Dies bedeutet, dass sie überall im Programm gültig sind. Werden sie jedoch innerhalb einer Funktion deklariert, dann handelt es sich um **lokale Variablen**. Diese haben ihre Gültigkeit nur innerhalb der Funktion. Bei Funktionen handelt es sich um eine Art Unterprogramm (ein Programm innerhalb eines Programms). Hierzu kommen wir aber erst später im Kapitel 4.



## 3 Operatoren

Mittels Operatoren werden mathematische Berechnungen durchgeführt. Mit dem „+“ beispielsweise können zwei Operanden (Zahlen/Werte) mit einander addiert werden  $\rightarrow 1 + 1 = 2$ .

### 3.1 Operatoren für einfache mathematische Berechnungen

In Python sehen Operatoren für einfache mathematische Berechnungen wie folgt aus:

Operator	Beschreibung	Beispiel
+	Addition	$2 + 2 = 4$
-	Subtraktion	$6 - 4 = 2$
*	Multiplikation	$3 * 3 = 9$
/	Division	$4 / 2 = 2$
%	Rest	$25 \% 6 = 1$
+x	Zahl x mit positivem Vorzeichen (Default)	+10
-x	Zahl mit negativem Vorzeichen	-10
**	Potenzierung	$3 ** 2 = 3 * 3 = 9$

### 3.2 Logische Operatoren

Operator	Beschreibung	Beispiel
and	Logisches und	x and y
or	Logisches oder	x or y
not	Logisches nicht	(1==1) and not (2==3)

### 3.3 Vergleichsoperatoren

Operator	Beschreibung	Beispiel
<	Kleiner als	$1 > 2$
<=	Kleiner-gleich	$x <= 2$
>	Größer	$2 > 1$
>=	Größer-gleich	$y >= 3$
!=	ungleich	$2 != 3$
==	gleich	$(1+1) == (4/2)$

### 3.4 String-Operationen

Mit String-Variablen sind keine mathematische Operationen möglich. Man kann zwar einem String einen Wert ,1‘ oder ,2‘ zuweisen. Das ändert aber nichts an den Umstand, dass diese beiden Zahlen dann immer noch „Zeichen“ sind. Aber die Operatoren ,+‘ und ,\*‘ sind auch für Zeichenketten anwendbar, jedoch gilt dieses dann als String-Operationen und ihr Ergebnis fällt ein wenig anders aus.

Mit dem Pluszeichen ,+‘ können Strings mit einander verkettet werden.

Beispiel:

```
string1 = 'Mit'
string2 = 'fahr'
string3 = 'gelegenheit'
print(string1 + string2 + string3)
```

Die Ausgabe sieht dann wie folgt aus:

```
Mitfahrgelegenheit
```

Bleibt also noch zu klären, was ,\*‘ für eine Wirkung hat. Der folgende Beispielcode ...

```
string = 'Spass' * 3
print(string)
```

... ergibt das Ergebnis:

```
SpassSpassSpass
```

Die Zeichenkette wird also x mal wiederholt.

## 4 Kommentare

Je nach Programmierstil kann der geschriebene Programmcode für einen selbst und ganz besonders für andere mehr oder weniger verständlich sein. Jedoch je komplexer und komplizierter die Algorithmen werden, desto schwieriger kann es selbst für den Urheber des Codes sein, diesen nach einer gewissen Zeit noch nachvollziehen zu können. Hilfreich sind in diesem Fall Kommentare! Ist eine ‚#‘ vorangestellt, werden alle dahinter folgenden Zeichen bis zum Zeilenende vom Python-Interpreter ignoriert. Auf diese Weise ist das Deaktivieren von Programmteilen möglich. Hinter der ‚#‘ können jedoch auch Beschreibungen, Hinweise und Erklärungen schriftlich hinterlegt werden. Programmabschnitte können mit Überschriften betitelt und erklärt werden.

Ganz besonders bei Programmier-Projekten im Team kommt man eigentlich nur noch schwerlich an einer guten Kommentierung des Codes vorbei.

## 5 Schleifen

Kommt es zur Notwendigkeit, dass Teile des Programm-Codes mehrmals hintereinander ausgeführt werden müssen, wird in der Programmierung auf die Kontrollstruktur der Schleife zurückgegriffen. Hierbei unterscheidet man zwischen verschiedenen Typen.

### 5.1 For

Die For-Schleife. Sie ist der gängigste Typ in den meisten Programmiersprachen, um Programmteile mehrfach hintereinander laufen zu lassen. Ihre Syntax sieht folgendermaßen aus:

```
for i in range(3):  
    print('Hallo Welt!')
```

Die daraus resultierende Ausgabe hat folgendes Aussehen:

```
Hallo Welt!  
Hallo Welt!  
Hallo Welt!
```

Ins Deutsche mehr oder weniger wortwörtlich übersetzt macht der Code folgendes: Für *i* im Bereich 3 führe folgende Print-Anweisung aus. Hierbei sind *i* und 3 Integer (ganze Zahlen). Aber das klingt für den wenig Programmier-Erfahrenen immer noch sehr kryptisch. Also nehmen wir noch einen zweiten Übersetzungsversuch vom englisch angehauchten „pythonisch“ in die Deutsche Sprache der Menschen vor:

Die Integer-Variable „*i*“ ist meine Zählvariable, bei wievielten Durchlauf der For-Schleife ich mich befinde. Hierbei wird beim Zählen mit „1“ begonnen. „**in range(3)**“ ist dann genau genommen nichts weiter, als die Festlegung, wie oft der Inhalt der For-Schleife wiederholt wird: „Wiederhole dieses 3 Mal!“.

### 5.2 While

Ein naher Verwandter der For-Schleife ist die While-Schleife. Erklären wir ihre Funktionsweise an folgendem Code-Beispiel:

```
n = 10  
while n > 0:  
    print(n)  
    n = n-1  
print('lift-off!')
```

Als Output erhalten wir nun folgendes:

```
10
9
8
7
6
5
4
3
2
1
lift-off!
```

Bei dem Beispiel mit der For-Schleife war es noch nicht so offensichtlich, da das For-Schleifen-Beispiel nur aus einem einzeiligen Schleifeninhalt bestand. Das sieht bei der While-Schleife bereits anders aus. Hier haben wir im Code-Beispiel mehrere Zeilen. Falls der eine oder andere bereits erste Erfahrungen mit anderen Programmiersprachen gemacht hat und es sich hierbei eventuell um C oder C++ handelte, wird sich nun eventuell fragen: „Woher weiß Python eigentlich, wann seine Schleifen und anderen Strukturen aufhören? Nehmen wir zum Beispiel ein sehr nahen Verwandten von Python: Die Interpreter-Sprache Perl! Sie handhabt Schleifen und andere Strukturen der Sprache C sehr ähnlich. Das gleiche Beispiel hätte hier wie folgt ausgesehen:

```
$n = 10;
while($n > 0){
    print "$n\n";
    $n = $n - 1;
}
print "lift-off!\n";
```

Wir wollen an dieser Stelle gar nicht im Detail die Syntax-Unterschiede zwischen Python und Perl erörtern. Was jedoch sofort auffällt: Schleifen-Konstrukte werden in Perl, C und C++ (und vielen anderen Sprachen) mit Klammern geöffnet und auch wieder geschlossen. Python scheint dieses anders zu regeln. Aber wie? Es sind die Einrückungen! Alles was sich auf der gleichen Einrückungs-Ebene befindet, gehört dem gleichen logischen Level an (zum Beispiel der Inhalt einer While-Schleife). Also: Vorsichtig bei der Verwendung des Tabulators. Er kann darüber entscheiden, ob eine Kontrollstruktur funktioniert oder nicht!

## 5.3 Until

Für diejenigen die bereits andere Programmiersprachen kennengelernt haben, mag sich eventuell die Frage stellen, ob es auch Until-Schleifen in Python gibt. Die klare Antwort hierzu lautet: Nein! Die Funktionalität dieser Schleifenart wird hier durch die While-Schleifen bewerkstelligt.

## 5.4 Break

Es können Situationen entstehen, wo es manchmal erforderlich ist, den normalen Programmverlauf vorzeitig zu beenden. Wir nehmen als Beispiel mal den folgenden Fall einer While-Schleife. Hier wird „n“ mit jedem Durchlauf erhöht und soll dann mit sich selbst Multipliziert werden. Wenn jedoch das Ergebnis beginnt zweistellig zu werden, soll die While-Schleife abgebrochen werden. Hierfür verwenden wir den Befehl „**break**“.

```
n = 0
while n < 100:
    n = n + 1
    erg = n * n
    print(erg)
    if erg > 10:
        break
```

Es kommt durch den Code zu folgender Ausgabe:

```
1
4
9
16
```

An sich hätte die While-Schleife auch weiter gemacht, bis „n“ den Wert 100 erreicht hätte. Wir lassen das Ergebnis der Multiplikation in der Schleife noch ausgeben und fragen erst danach die Abbruchsbedingung ab. Aus diesem Grund wird das Rechenresultat auch noch ausgegeben, obwohl das Produkt bereits größer 10 ist.

## 6 Funktionen

Wie an anderer Stelle bereits erwähnt wurde, handelt es sich bei einer Funktion um eine Art von Unterprogramm – ein Programm innerhalb des Programms. Damit eine Funktion funktioniert muss sie an einer Stelle im Code definiert werden. Erst wenn dies geschehe ist, kann sie dann namentlich aufgerufen und somit verwendet werden.

Wichtig bei den Funktionen ist, dass diese am Anfang des Python-Scriptes definiert werden. Hier ein Beispiel:

```
#!/usr/bin/python3
ausgabe()
def ausgabe():
    print('Heute ist ein schöner Tag!')
```

**Coding 1**

Hier haben wir es also genau falsch gemacht. Zuerst wird die Funktion aufgerufen und dann erst wird diese definiert. Es kommt daher zu folgender Fehlerausgabe:

```
Traceback (most recent call last):
  File "./funktion.py", line 3, in <module>
    ausgabe()
NameError: name 'ausgabe' is not defined
```

Hier also die Reihenfolge, wie sie korrekt ist:

```
#!/usr/bin/python3
def ausgabe():
    print('Heute ist ein schöner Tag!')
ausgabe()
```

**Coding 2**

Die Ausgabe sieht im Übrigen folgendermaßen aus:

```
Heute ist ein schöner Tag!
```

Nebenbei erwähnt: Die erste Zeile in diesem Beispiel-Script ist eine sogenannte „Shebang-Zeile“. Man kann sie auch unter Windows einsetzen. Dort jedoch ist nach einer Python-Installation meistens schon automatisch gesetzt, dass Dateien mit der Endung „.py“ mit dem Python-Interpreter auszuführen sind. Mac OS arbeiten sehr oft auch ohne Dateiendung. Eine ausführbare Datei namens „**script**“ könnte daher alles mögliche sein. Das Resultat einer kompilierten C-Datei, ein Python- oder ein Perl-Skript. In den letzten beiden Fällen wird dann als erstes eine sogenannte Shebang-

Zeile hinterlegt, die zum einem Auskunft darüber gibt, wie das Skript / das Programm auszuführen ist und wo der hierfür erforderliche Interpreter liegt. Wie bereits erwähnt: Unter Windows wird dieses nicht oder selten praktiziert. Unter MS Windows wird in erster Linie mit den Dateieindungen gearbeitet.

Funktionen können einen Rückgabewert haben. Dies gilt besonders bei Funktionen mit mathematischen Berechnungen. Hier ein einfaches Beispiel:

```
#!/usr/bin/python3

def zwei_zum_quadrat():
    ergebnis = 2**2
    return ergebnis

ergebnis2 = zwei_zum_quadrat()
print(ergebnis2)
```

**Coding 3**

Als Ausgabe erhalten wir den Wert 4.

Mit der Integer-Variable `ergebnis` berechnen wir in der Funktion `zwei_zum_quadrat()` den Wert von  $2^2$ . Mit dem Befehl `return` wird diese als Berechnungsergebnis der Funktion an das Hauptprogramm zurück gegeben. Das Hauptprogramm wiederum ruft die Funktion auf, indem es die Funktion (und damit deren Rückgabewert) der Integer-Variable `ergebnis2` zuordnet. Und diese wird dann mittels Print-Funktion ausgegeben.

## 6.1 Typenkonvertierung

Variablen können zu anderen Typen konvertiert werden. So kann mit `int(3.2)` eine Kommazahl in einen Integer umgewandelt werden. Hierbei gehen leider jedoch die Nachkommastellen verloren. Umgekehrt ist es auch möglich, einen Integer in eine Float-Variable umzuwandeln: `float(32)`. Das ganze geht auch in Richtung zu String-Variablen: `str(32)`, `str(32.0)`.

## 6.2 Mathematische Funktionen

Während Addition, Subtraktion, Multiplikation, Division und das Potenzieren von Zahlen noch ohne Import von Modulen auskommt, sieht es bereits anders aus, wenn ich die Quadratwurzel aus Zahlen berechnen möchte. Ab hier ist es erforderlich das Mathematik-Modul von Python zu laden:

```
import math
```



Hier ein kurzes Beispiel, wie auf Grund dessen der Code zur Quadratwurzelberechnung zu gestalten ist:

```
#!/usr/bin/python3

import math

ergebnis = math.sqrt(2)
print(ergebnis)
```

Coding 4

## 6.3 Parameter und Argumente

Am letzten Beispiel in 6.2 konnte man es schon ganz gut beobachten. Die mathematische Funktion `math.sqrt()` erwartet die Übergabe eines Zahlenwerts. Aber auch selbst geschriebene Funktionen können derart geschrieben werden, dass man ihnen Parameter oder Argumente übergeben kann. Hierzu bauen wir das Beispiel Nr. 4 aus dem Kapitel 6 folgendermaßen um:

```
#!/usr/bin/python3

def zwei_zum_quadrat(zahl):
    ergebnis = zahl**zahl
    return ergebnis

ergebnis2 = zwei_zum_quadrat(2)
print(ergebnis2)
```

Coding 5

Der Zahlenwert 2 ist nun nicht mehr direkt in die Funktion einkodiert, sondern wird vom Hauptprogramm übergeben.

## 6.4 Funktions-Variablen und -Parameter

Das Beispiel aus 6.3 als auch sein Vorgänger aus 6.2 beinhaltet noch ein weitere wichtige Eigenart von Programmiersprachen. Die Integer-Variable `ergebnis` wird in der Funktion `zwei_zum_quadrat` definiert und gesetzt. Da dies innerhalb der Funktion geschieht, kennt das Hauptprogramm diese Variable nicht – sie ist eine **lokale Variable** und somit nur der Funktion bekannt, in welcher sie definiert wurde. Die Variable `ergebnis2` hingegen wurde in der Main-Funktion gesetzt. Sie ist dadurch auch Funktionen in diesem Programm bekannt. Man nennt sie daher **globale Variable**.

## 6.5 Gründe für Funktionen

Sicherlich ist es möglich, auf die Verwendung von Funktionen zu verzichten und alle erforderlichen Code-Anteile im Hauptprogramm unterzubringen. Aber es gibt ein paar Vorteile von Funktionen, so dass man ihre Verwendung nicht gleich wieder verwerfen sollte.

Besondere Programmteile, können mit ihnen abgetrennt vom Hauptprogramm gesondert behandelt oder abgelegt werden. Wiederholen sich inhaltlich bestimmte Programmteile, ist es möglich, sie ebenfalls gut in Funktionen auszulagern. Der Code-Umfang kann hierdurch erheblich reduziert werden.

Sind bestimmte Anweisungen in einer Funktion zusammen gefasst, kann dieses die Lesbarkeit des Python-Codes erhöhen.

Die Funktionen können im Einzelnen untersucht, analysiert und debuggt werden, welches die Fehlersuche bei sehr komplexen Programmen vereinfacht.

Wird der Code für Funktionen sehr nachhaltig und wiederverwendbar geschrieben, kann er ausgelagert werden und als Import auch in anderen Python-Projekten verwendet werden. Gewisse Aufgaben können so für verschiedenste Python-Programme verwendet werden: Prinzip „Only Write Once!“

Das Rad muss auf diese Weise nicht dutzende Male neu erfunden werden.

## 6.6 Modul-Import

Wie im Beispiel von 6.2 bereits angewandt, können wir mit `import` Python-Module in unsere Skripte einbinden und darin gespeicherte Funktionen oder Konstanten abrufen. Es gibt jedoch auch die Möglichkeit z.B. einzelne Konstanten direkt in Modulen wie `math` anzusprechen:

```
from ... import
```

Dazu folgendes Beispiel mit der Eulersche Zahl:

```
from math import e
print(e)
```

Mit `from math import *` wird im Übrigen alles aus diesem Modul importiert.

## 7 Bedingungen

Es kann zu Situationen kommen, wo es erforderlich ist, dass ein bestimmter Teil des Programmcodes nur ausgeführt wird, wenn eine bestimmte Bedingung eintritt. Man spricht dann auch von einer bedingten Ausführung. Umgesetzt wird derartiges wie folgt:

```
#!/usr/bin/python3

zahl1 = 3
zahl2 = 2

if zahl1 < zahl2:
    print('Zahl1 ist kleiner Zahl2!')
```

**Coding 6**

Wir verwenden also das Statement `if` und lassen dahinter eine logische Operation folgen. Trifft diese zu (ergibt die logische Operation also den Wert wahr), dann wird die darunter folgende Print-Anweisung ausgeführt. Wenn nicht, wird der Programm-Part eingeschlossen von der sogenannten `if-Clause` nicht durchgeführt. Hierbei hätten `zahl1` und `zahl2` zwei Integer-Werte sein können, die auch durch Eingabe von Daten oder das Eintreten bestimmter Konstellationen in einem komplexeren Python-Programm im Vorfeld ergeben können. If-Clauses können jedoch auch aneinandergereiht werden. Sie ergeben somit eine Art Fall Unterscheidung. Im folgenden Code wurde das vorige Beispiel um das Statement `elif` erweitert. Es ist die Abkürzung vom englischen `elseif`. Man könnte es mit „wenn aber“ übersetzen.

```
#!/usr/bin/python3

zahl1 = 3
zahl2 = 2

if zahl1 < zahl2:
    print('Zahl1 ist kleiner Zahl2!')
elif zahl1 > zahl2:
    print('Zahl1 ist größer Zahl2!')
```

**Coding 7**

Zu den „Bedingungs-Statements“ gibt es noch einen dritten Mitspieler: `else` (sondern). Unser Code-Beispiel wird abermals angepasst und erweitert:

```
#!/usr/bin/python3

zahl1 = 3
zahl2 = 3

if zahl1 < zahl2:
    print('Zahl1 ist kleiner Zahl2!')
elif zahl1 > zahl2:
    print('Zahl1 ist größer Zahl2!')
else:
    print('Zahl1 und Zahl2 sind gleichgroß!')
```

**Coding 8**

Tritt also keine der ersten beiden Bedingungen ein, dann soll die *Else-Clause* gelten. Zugegeben, das sind jetzt nur sehr einfache Beispiele, aber mit ihnen ist eigentlich schon alles wesentlich behandelt, was Kontrollstrukturen mit Bedingungen betrifft.

## 8 Ein- und Ausgabe

### 8.1 Eingaben per Tastatur

Um Eingaben über die Tastatur in Python einzulesen, benötigen wir die Funktion `input()`. Sie wird einer String-Variable zugewiesen, mit welcher wir anschließend arbeiten können (z.B. Ausgabe des Inputs). Im Folgenden ein einfaches Beispiel:

```
#!/usr/bin/python3

eingabe = input()
print('Du hast eingegeben: ')
print(eingabe)
```

**Coding 9**

### 8.2 Lesen aus und Schreiben in Dateien

#### 8.2.1 Lesen aus Dateien

Neben der Eingabe per Tastatur können Informationen für unser Python-Programm aus Dateien ausgelesen werden.

Ganz besonders im Unix- und Linux-Umfeld (Mac OS gehört auch dazu) wird viel mit Konfigurations-Informationen in Text-Dateien gearbeitet. Unter Windows finden sie als Ini- und Sys-Dateien bedingt auch ihren Einsatz, doch seit Einführung der Registry-Datenbank von Windows NT ist dieses hier stark rückläufig. Hochkonjunktur hatten diese auch ASCII-Dateien genannten Files eher in Zeiten von MS DOS und MS Windows bis zur Version MS Windows ME<sup>1</sup>.

Zum Auslesen des Inhalts einer Textdatei bedienen wir uns der folgenden Code-Zeilen:

```
#!/usr/bin/python3

textdatei = open('textdatei.txt', 'r')
print(textdatei)
print()
print(textdatei.read())
```

**Coding 10**

---

<sup>1</sup> ME steht hierbei für MS Windows Millennium Edition. Es war die letzte dos-basierende Version von MS-Windows. Danach (begonnen mit Windows XP gab es nur noch Windows-Versionen

Die Skriptausführung hat diese Ausgabe zur Folge:

```
<_io.TextIOWrapper name='textdatei.txt' mode='r' encoding='UTF-8'>  
Das ist eine Textdatei.  
Sie wird auch ASCII-Datei genannt.  
Der in ihr enthaltene Text hat keine  
Formatierung.
```

Vorher wurde natürlich eine entsprechende Textdatei erstellt. Aber gehen wir die Programmzeilen einmal im Einzelnen durch.

Als erstes wird mit dem open-Befehl ein Datei-Handle mit dem Namen „textdatei“ eröffnet. Die zwei wichtigen Parameter des Open-Statements: Der Dateiname und der Modus. Befindet sich die Datei nicht in der gleichen Verzeichnisebene wie das Python-Programm, dann muss zusätzlich zum Dateinamen auch der Verzeichnispfad angegeben werden. Es gibt folgende Datei-Modi:

Modus	Beschreibung
r	Datei wird zum Lesen geöffnet
r+	Die Datei wird gelesen und kann auch geschrieben werden
w	Datei wird zum Schreiben geöffnet – bereits vorhandene Inhalte werden überschrieben und gehen somit verloren
a	Die Datei wird zum Schreiben geöffnet, jedoch werden nun alle neuen Zeile am Ende angehängt
b	Ein zusätzlicher Modus zu den bisher erwähnten, der benötigt wird, wenn Binär-Code gelesen oder geschrieben werden soll

Mit der Programmzeile `print(textdatei)` wird nicht etwa der Inhalt dieser Textdatei ausgegeben, sondern eine Informationszeile, die den Dateinamen, den Datei-Modus und die Datei-Kodierung. Bei der Zeichenkodierung von Computern werden intern die Zeichen mit einem Zahlenwert hinterlegt. Dieses lässt sich dann auch als Spannungs-Signal oder Faxton übertragen. Welcher Zahlenwert dann zum Beispiel für den Buchstaben A hinterlegt wird, legt die Zeichenkodierung fest (A in UTF8: 41). UTF8 ist weltweit die am meisten verwendete Zeichenkodierung. Weitere Zeichenkodierungen sind z.B. Latin1 oder Latin2. Mit Latin1 können in der Regel alle Zeichen aus dem Westeuropäischen Raum dargestellt werden. Wer mehr über Zeichenkodierungen wissen möchte, kann hier unter diesem Link nachschauen:

<https://im-coder.com/was-ist-der-unterschied-zwischen-utf-8-und-iso-8859-1.html>

### 8.2.2 Parameter und Argumente, die 2.

Vielleicht an dieser Stelle ein wenig unpassend (zumindest was das Thema des Hauptkapitels betrifft), aber wo wir schon bei den Übergabemöglichkeiten von Informationen an Programme sind, hier der dritte Weg, dieses zu tun: Informationen werden in Form von Argumenten oder Parametern beim Programmaufruf mit zu übergeben:

Wir stellen den folgenden Programm-Code zusammen und benennen ihn `args.py`.

```
#!/usr/bin/python3

import sys

print ('Anzahl der Argumente:', len(sys.argv), 'Argumente.')
print ('Argumenten-Liste:', str(sys.argv))
```

**Coding 11**

Auf der Kommandozeile rufen wir das Script folgend auf:

```
./args.py Arg1 Arg2 Arg3
```

Als Output kommt hierbei dieses heraus:

```
Anzahl der Argumente: 4 Argumente.
Argumenten-Liste: ['./args.py', 'Arg1', 'Arg2', 'Arg3']
```

Analysieren wir den Code. Mit dem Import des Moduls `sys` können dann mittels `sys.argv` die nach dem Script-Namen nachfolgend eingetippten Argumente ausgelesen werden. Es werden alle in diesem Kontext eingegebenen „Wörter“ gezählt. Inklusive dem Scriptnamen selbst kommen wir in diesem Beispiel dann auf vier Argumente, welche das Python-Programm danach auch noch in einer Liste aufzählt. Auf diese Weise kann das Verhalten von Programmen durch die Eingabe an Optionen oder Parametern in der Kommandozeile beeinflusst werden. Dieses ist dann auch automatisiert aus anderen Diensten oder per Aufruf durch weitere externer Programme möglich.

### 8.2.3 Schreiben in Dateien

Mit diesem Code fragt uns das Script einen Text-String von der Tastatur ab (bzw. fordert uns auf diesen per Keyboard einzugeben) und diese Zeichenkette wird dann in eine Ausgabedatei geschrieben:

```
#!/usr/bin/python3

print("Bitte geb hier jetzt Deinen Text ein:\n")
eingabe = input()
ausgabedatei = open("ausgabe.txt", "w")
ausgabedatei.write(eingabe)
ausgabedatei.write("\n")
ausgabedatei.close()
```

**Coding 12**

Gehen wir die Programmzeile kurz einmal durch. Mit `print()` wird die Aufforderung zur Texteingabe auf den Bildschirm ausgegeben. Den darauf von uns erwarteten Tastatur-Input weisen wir darauf hin der Variable `eingabe` zu. Danach wird das Datei-Handle eröffnet – unsere Ausgabedatei trägt den Namen `ausgabe.txt`. Damit wir schreiben können wir noch der Datei-Parameter „w“ (write) für Schreiben gesetzt. Mit der Methode `ausgabedatei.write` wird dann unsere vorgenommene Eingabe in die Datei geschrieben. Der Input-Zeile lassen wir noch eine Leerzeile folgen, zwecks besserer Anzeige in der Kommandozeile und so wie es die feine englische Art ist, Datei-Handles, die man eröffnet hat, die schließt man dann auch schön brav am Ende wieder.

Unter Windows kann man sich das ganze dann in der CMD z.B. mit dem Befehl `more ausgabedatei.txt` ansehen. Alternativ ginge hier dann auch ein Doppelklick im Dateimanager auf die Datei oder (falls Windows mit der Endung txt keine Applikation verknüpft hat) dann mit Rechtsklick „Öffnen mit“ im Kontextmenü wählen und hier den „Editor“ anklicken.

Unter Mac und Linux habe wir das Terminal. Hier können wir uns die Datei ausgeben lassen, indem wir

```
cat ausgabedatei.txt , more ausgabedatei.txt oder less ausgabedatei.txt
```

eingeben.

Haben wir jetzt zum Beispiel beim ersten Durchlauf des Programms den String „Eingabe“ eingegeben und rufen danach das Script ein zweites Mal auf, geben nun jedoch „Eingabe2“ ein, so werden wir bei dem Öffnen bzw. Ansehen des Dateiinhalts folgendes feststellen: Statt „Eingabe“ steht nun hier „Eingabe2“!

Der Inhalt wurde also überschrieben. Was ist aber, wenn ich möchte, dass neue Inhalte die alten nicht überschreiben, sondern die neue Zeichenkette am Ende der Datei angehängt wird?

Um das bewerkstelligen zu können, muss der Datei-Parameter „w“ (write – schreiben) durch den Parameter „a“ (append – anhängen) ausgetauscht werden. Die Zeile mit der Definition des Datei-Handles sieht dann so aus:

```
ausgabedatei = open("ausgabe.txt", "a")
```



Ändern wir also den Code unseres Scripts ab. Rufen wir es jetzt zum ersten Mal auf, geben „Eingabe1“ ein und rufen es danach ein zweites Mal auf und geben „Eingabe2“ ein, werden wir beim nachfolgenden Betrachten der Ausgabedatei diesen Inhalt hier feststellen:

Eingabe1

Eingabe2

Nun wurde unsere Textdatei nicht überschrieben, sondern unsere zweite Eingabe wurde unterhalb der ersten Eingabe angehängen.

## 8.3 Formatierte Ausgabe

Wir erstellen und das Script `form_ausgabe.py`:

```
#!/usr/bin/python3  
  
schafe = 100  
print('Ich habe %d Schafe gezählt.' % schafe)
```

**Coding 13**

`%d` steht hierbei für „decimal“ und weist die Print-Funktion an, an der Stelle `%d` im Satz die Zahl der Variable `schafe` einzufügen. Am Ende der eigentlichen Print-Ausgabe folgt noch die Zuordnung, welche Wert denn dieser Zahl zugeordnet werden soll (Variable `schafe`).

Manchmal besteht das Leben nicht nur aus ganzen Zahlen. Neben wir zum Beispiel die Tafel Schokolade aus dem Supermarkt, die möglicherweise 1,29 € kostet. Dies im Hinterkopf kommen wir noch einmal zurück auf unser Schäfchen-Zählen.

Habe ich mit Fließkommazahlen zu tun kann ich das ganze auch wie folgt angehen:

```
#!/usr/bin/python3  
  
schafe = 100  
print('Ich habe %6.2f Schafe gezählt.' % schafe)
```

**Coding 14**

Die hiermit einhergehende Ausgabe sieht dann so aus:

```
Ich habe 100.00 Schafe gezählt.
```

Das **f** steht hierbei für „float“ (Fließkommazahl), die 6 gibt die Anzahl der Gesamtzeichen an und die 2 steht für die Anzahl der Nachkommastellen. Erklären kann man die ganzen Zusammenhänge vielleicht ein wenig anschaulicher in folgender Grafik rechts:

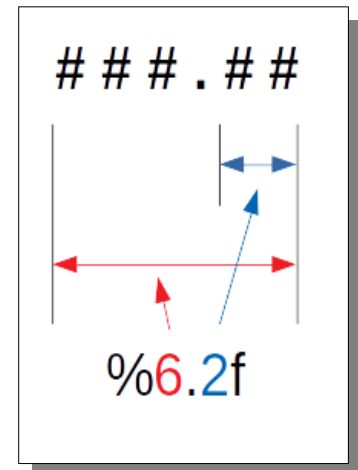


Abbildung 2

## 8.4 Dateien und Pfade

Manchmal kann es wichtig sein, zu erfahren, wo im Verzeichnisbaum man sich gerade befindet. Genauso kann es zu Fällen kommen, wo ich den absoluten Pfad einer Datei benötige. Oder auch der Inhalte des aktuellen Verzeichnisses kann von Interesse sein.

### 8.4.1 Aktuellen Pfad ausgeben

Die folgenden Codezeilen geben das aktuelle Verzeichnis aus, aus dem heraus das Script aufgerufen wurde:

```
#!/usr/bin/python3

import os
aav = os.getcwd()
print(aav)
```

Coding 15

Um derartige Verzeichnisabfragen machen zu können, benötigt man das Modul **os** und mit der Methode **os.getcwd()** kann einer Variable der Name des aktuellen Ordners übergeben werden. Die Abkürzung „cwd“ steht hierbei für „current working directory“. Übersetzt bedeutet dies „aktuelles Arbeits-Verzeichnis“. Dieses wiederum ist auch der Grund warum die Verzeichnis-Variable in dem Script mit „aav“ abgekürzt wurde.

### 8.4.2 Absoluten Pfad einer Datei erhalten

Ich kann das Abfragen von Verzeichnisinformationen auch mit einer andere Datei verbinden, welche sich am gleichen Ort (in der gleichen Verzeichnisebene) befindet:

```
#!/usr/bin/python3

import os
abspfad = os.path.abspath('ausgabe.txt')
print(abspfad)
```

**Coding 16**

Zeile 4 beherbergt des Rätsels Lösung, mit welchem Methoden-Aufruf ich das OS-Modul dazu verleiten kann, mir den absoluten Pfad einer Datei auszugeben, die sich ebenfalls im Script-Verzeichnis befindet: `os.path.abspath()`

### 8.4.3 Datei oder Verzeichnis auf Existenz überprüfen

Und auch das kann in der einen oder anderen Situation wichtig sein: Existiert eine bestimmte Datei oder ein bestimmter Ordner?

```
#!/usr/bin/python3

import os
datei_existiert = os.path.exists('ausgabe.txt')
print(datei_existiert)
```

**Coding 17**

Es ist wieder die Zeile 4, die verrät, welcher Methodenaufruf des OS-Moduls den Job macht:

```
os.path.exists()
```

Der Rückgabewert ist boolesch – bedeutet, das Ergebnis kann entweder `True` oder `False` sein (wahr oder falsch). To be or not be. Verzeichnis oder Datei existiert oder es/sie existiert nicht.

## 9 Datenbanken

Die im Web-Bereich am meisten verwendete Datenbank (und das seit bestimmt mindestens 20 Jahren) ist MySQL und ihr Fork<sup>2</sup> MariaDB. Es gibt dann noch das eine oder andere Projekt, wo bei großen Datenbanken extrem schnelle Zugriffszeiten notwendig sind. Hier findet dann nicht selten auch die extrem performante Datenbank PostgreSQL Verwendung. Hat man jedoch den Umgang mit MySQL/MariaDB verstanden, ist das Verstehen und Verwenden weiterer Datenbanksysteme keine große Hürde mehr.

### 9.1 Das MySQL-Modul

Damit MySQL/MariaDB von Python aus ansprechbar sind, ist der Import des Moduls `pymysql` notwendig. Bei MariaDB funktionieren in der Regel alle Befehle, wie sie auch im MySQL Verwendung finden.

Gebe ich auf der Kommandozeile den Befehl `python3` ein, gelange ich in den Interpretationsmodus. In diesem kann ich Befehlssequenzen von Python im Vorfeld testen und mich zum Beispiel so mit der Syntax von Methoden bei neuen, mir bis dahin noch unbekannten Modulen vertraut machen. Dies kann auch ein bequemer Weg sein, das Vorhandensein benötigter Module vorab zu überprüfen.

```
ben@huibuh:~$ python3
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Im Interpreter-Modus gebe ich den Lade-Befehl für die Modul-Import ein.

```
>>> import pymysql
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'pymysql'
>>>
```

Erhalte ich dann die Fehlermeldung „No module named ‘pymysql‘“, ist dies der Beweis dafür, dass das Modul auf dem System nicht installiert ist. Zumindest nicht in der Python-Umgebung, die ich

---

<sup>2</sup> Bei OpenSource-Projekten ist der Quellcode frei einsehbar und darf je nach Lizenz verwendet und weiterentwickelt werden. Somit ist es in der Regel möglich, jederzeit die Codebasis eines vorhandenen Projekts zu nehmen und diese mit eigenen Modifikationen unter neuem Namen weiter zu entwickeln. Bekannte Beispiele für derartige Forks: OpenOffice/LibreOffice, MySQL/MariaDB, XFree86/X.Org

gerade verwende.<sup>3</sup> Mit `pip3 install pymysql` kann ich dann das fehlende Modul in die verwendete Umgebung installieren. Im Übrigen wird der Interpretations-Modus durch `Strg+D` wieder verlassen.

## 9.2 Einrichten einer Test-Datenbank

Nun gibt es aber noch ein paar kleine Vorbereitungen, die getroffen werden müssen. In einem Terminal (unter Mac/Linux) und in der CMD (unter Windows) können wir uns mit dem lokalen Datenbankdienst unseres Rechners verbinden, indem wir folgendem Login-Befehl eingeben: „`sudo mysql -u root -p`“<sup>4</sup>. Unter Windows kann/muss das vorangestellte `sudo` entfallen. Setzt man ein Sudo-Kommando unter Linux ab, wird man in der Regel danach nach dem Passwort des Benutzer gefragt, mit dem man sich am System anmeldet. Nach erfolgreicher Anmeldung erhalten wir den Datenbank-Prompt von MySQL/MariaDB.

Mit dem folgenden SQL-Befehl erstellen wir uns eine Test-Datenbank:

```
MariaDB [(none)]> create database TESTDB;
```

Danach müssen wir noch in die frisch erstellte Datenbank wechseln.

```
USE TESTDB;
```

Als nächsten Schritt benötigen wir noch einen Test-Benutzer, in dessen Kontext wir das Python-DB-Script ausführen lassen wollen (bezieht sich auf die Datenbank-Operationen).

```
CREATE USER 'testuser'@'localhost' IDENTIFIED BY 'test123';
```

Jetzt benötigt dieser Test-User noch die vollen Zugriffsrechte auf die Test-Datenbank:

```
GRANT ALL PRIVILEGES ON TESTDB.* TO 'testuser'@'localhost';  
FLUSH PRIVILEGES;
```

- 
- 3 Auf einem klassischen Linux-System wird es meistens eine zentrale Python-Installation geben. Zusätzlich kann eine Benutzer diese Konfiguration in seinem User-Home um weitere Module erweitern. Diese stehen dann aber nicht systemweit zur Verfügung. Ein weiterer Weg ist, innerhalb im Kontext des Benutzers virtuelle Umgebungen einzurichten. Manche Module sind untereinander nicht kompatibel und so kann man sie in verschiedenen, voneinander abgegrenzten Environments einrichten. Diese virtuellen Umgebungen kann man dann z.B. verschiedenen Aufgabenbereichen zuordnen.
- 4 Falls bei der DB-Installation bereits ein Passwort für den DB-User „root“ gesetzt worden ist, so ist dieses direkt (und ohne Leerzeichen) hinter dem letzten Optionsparameter „-p“ einzusetzen

Der Grant-Befehl vergibt die Rechte. Mit „**FLUSH PRIVILEGES;**“ werden diese Berechtigungsveränderungen an die Datenbank weitergereicht. Wichtig ist hierbei, dass man darauf achtet, jede SQL-Befehlszeile mit einem Semikolon zu beenden. Haben wir diese drei grundlegenden Datenbankoperationen vollzogen können wie die „MySQL-Session“ mit einem „**exit**“ wieder verlassen.

## 9.3 Erstellen einer Tabelle

Nun kommen wir zu unserem ersten Python-Skript, welches eine Tabelle „Angestellte“ in unserer frisch angelegten Datenbank anlegen wird:

```
#!/usr/bin/python3

import pymysql

# Öffnen der Datenbank-Verbindung
db = pymysql.connect("localhost","testuser","test123","TESTDB")

# Einen sogenannten Cursor für die Ausführung vorbereiten
cursor = db.cursor()

# Erstellen der Tabelle "Angestellte"
sql = """CREATE TABLE ANGESTELLTE (
    VORNAME CHAR(20) NOT NULL,
    NACHNAME CHAR(20),
    ALTER INT,
    GESCHLECHT CHAR(1),
    EINKOMMEN FLOAT )"""

cursor.execute(sql)

# Die Datenbank-Verbindung wieder schließen
db.close()
```

**Coding 18**

Die Kommentare zu den meisten Codezeilen liefern an sich bereits eine ausreichende Erklärung über ihre Bedeutung mit. Leicht anders verhält es sich mit dem Create-Table-Statement, welches doch recht komplex ist. Wir haben kennengelernt, dass Programmiersprachen zwischen verschiedenen Variablen-Typen unterscheiden. So verhält es sich auch bei Datenbanken. So scheint in dieser Tabelle eine Datensatz aus folgenden Datenfeldern zu bestehen: Vorname, Nachname, Alter, Geschlecht und Einkommen. Vor- und Nachname dürfen aus eine jeweils 20 Zeichen langen Zeichenkette bestehen. Beim Datenfeld „Alter“ wird die Eingabe einer Ganzen Zahl erwartet. Das Geschlecht mit einem einzelnen Buchstaben gekennzeichnet (m, w, d) und als Gehalt soll und kann

eine Fließkommazahl eingegeben werden. Die darauffolgende Codezeile „`cursor.execute(sql)`“ führt das SQL-Statement aus.

## 9.4 Datensätze hinzufügen

Mit dem nächsten Skript werden wir diese Datenbank-Tabelle nun mit einem ersten Datensatz befüllen.

```
#!/usr/bin/python3

import pymysql

# Öffnen der Datenbankverbindung
db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# Vorbereitung des Ausführungs-Cursors
cursor = db.cursor()

# Zusammenstellung der SQL-Query
sql = """INSERT INTO ANGESTELLTE(VORNAME,
    NACHNAME, AGE, GESCHLECHT, EINKOMMEN)
    VALUES ('Peter', 'Müller', 33, 'm', 2300)"""
try:
    # Absetzen des SQL-Querys
    cursor.execute(sql)
    # "Schreib-Änderungen" benötigen ein Commit!
    db.commit()
except:
    # "Rollback" im Fehlerfall
    db.rollback()

# Trennen der Datenbankverbindung
db.close()
```

**Coding 19**

Auch hier stehen alle relevanten Erklärungen zum Code bereits als Kommentar im Script. Dennoch folgt an dieser Stelle noch eine kurze Erläuterung zu „Commit“ und „Rollback“. Wenn es schreibende Veränderungen an einer Datenbank gibt (hier in diesem Fall das Einfügen von einem neuen Datensatz in eine Tabelle), da muss diese Veränderung noch durch ein „Commit“ an die Datenbank durchgereicht werden. Vorher sind diese Veränderungen auch nicht für andere Benutzer an der Datenbank zu sehen. Manchmal kann es dazu kommen, dass aufgrund von datenbankinternen Problemen eine Veränderung nicht umsetzbar ist. Der neue Datensatz in der Tabelle ist also noch in einem „schwebenden Status“ („pending“). Durch das „Rollback-Kommando“ werden die Änderungen an der Tabelle wieder rückgängig gemacht. Besonders stark frequentierte Datenbanken bei Internetanwendungen können das Problem haben, dass gleich mehrere Prozesse gleichzeitig auf

ein und die selbe Tabelle schreibend zugreifen wollen. Es kommt dann häufig zu sogenannten „Deadlocks“. Ein Rollback würde diese Access-Konflikte wieder aufheben. Das vorangegangene Script würde dieses Bemerkten und selbst den Rollback auslösen. Zusätzlich könnte man es noch mit einer Fehlermeldung verbinden. Auf diese Weise würde der Anwender darüber in Kenntnis gesetzt werden, dass seine Operation nicht durchgeführt werden konnte und er es später noch einmal probieren muss.

Konnte der Datensatz erfolgreich in der Tabelle angelegt werden, so können wir uns das auf folgende Weise ansehen:

```
ben@huibuh:~$ mysql -u testuser -ptest123      → Anmeldung an der DB
MariaDB [(none)]> use TESTDB;                  → Wechsel zur Testdatenbank
MariaDB [TESTDB]> select * from ANGESTELLTE;    → Wechsel zur Test-Datenbank
```

Um die Zusammenhänge besser nachvollziehen zu können, sind die Kommando-Prompts vor den eigentlichen Befehlen im obigen Beispiel belassen worden. Sie gehören zu den eigentlichen Befehlen nicht dazu. Damit sie sich dennoch leicht von der eigentlichen Befehls-Syntax abheben, wurde für sie eine kleinere Schriftgröße gewählt. In der ersten Befehlszeile findet die Anmeldung an der Datenbank statt. Wir verwenden hierfür den gleichen Account, der auch im Python-Script Verwendung fand. Ab der zweiten Zeile: Befanden wir uns eben noch in der Kommandozeile des Betriebssystems, so sind wir nun innerhalb der Datenbanksitzung (der Kommando-Prompt hat sich verändert). Mit `use TESTDB;` wechseln in der DB-Session zu unserer Test-Datenbank. Man beachte den Prompt. Steht anfangs noch in eckigen Klammern „(none)“, so ändert sich dies nach dem Use-Kommando in „TESTDB“. Der SQL-Client zeigt uns also an, in welcher Datenbank wir uns gerade befinden. Mit der Select-Zeile teilen wir dem Datenbank-Dienst dann mit, dass er uns alle Einträge in der Tabelle „`ANGESTELLTE`“ zeigen soll.

Und hier ist zu sehen, was uns die Datenbank darauf hin antwortet:

```
+-----+-----+-----+-----+-----+
| VORNAME | NACHNAME | AGE  | GESCHLECHT | EINKOMMEN |
+-----+-----+-----+-----+-----+
| Peter   | Müller   | 33   | m           | 2300       |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Ein kleiner Hinweis noch: Dem aufmerksamen Leser mag an dieser Stelle aufgefallen sein, dass ich beim Datenfeld „AGE“ vom Muster deutscher Datenfeldnamen abgewichen bin. Das deutsche Wort für „Age“ ist „Alter“. „to alter“ bedeutet im Englischen jedoch auch „zu ändern“. Es ist ein für SQL reserviertes Befehls-Schlüsselwort und kann daher nicht als Name verwendet werden.



## 9.5 Lese-Operation

Oft werden wir bei DB-Anwendungen auch den Fall haben, dass wir Informationen aus einer Datenbank abfragen müssen. Ein einfaches Lesen einer Datenbank-Tabelle sähe als Python-Script wie folgt aus:

```
#!/usr/bin/python3

import pymysql

# Öffnen der Datenbankverbindung
db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# Deklaration des Datenbank-Cursors
cursor = db.cursor()

# Zusammenstellen der SQL-Query
#sql = "SELECT * FROM ANGESTELLTE \
#      WHERE EINKOMMEN > '%d'" % (1000)
sql = "SELECT * FROM ANGESTELLTE WHERE EINKOMMEN > 1000;"
try:
    # Ausführen des SQL-Statements
    cursor.execute(sql)
    # Alle Zeilen werden als Liste in einer Liste abgefangen
    ergebnisse = cursor.fetchall()
    for zeile in ergebnisse:
        vname = zeile[0]
        nname = zeile[1]
        age = zeile[2]
        geschlecht = zeile[3]
        einkommen = zeile[4]
        # Ausgabe der abgefangenen Ergebnisse
        print ("vname = %s,nname = %s,age = %d,geschlecht = %s,einkommen = %d" % \
              (vname, nname, age, geschlecht, einkommen ))
except:
    print("Fehler: Konnte die Daten nicht abfragen")

# DB-Verbindung trennen
db.close()
```

**Coding 20**

Interessant und wirklich neu ist hierbei eigentlich nur die Select-Zeile. Während wir bei unserer SQL-Client-Sitzung alle Datensätze der Tabelle abgerufen hatten, schränken wir hier durch die Where-Klausel die Abfrage ein: Es sollen nur die Gehälter aufgelistet werden, die über 1000 € liegen. Am meisten werden wir allerdings mit dem Abfrage-Typus „=" konfrontiert werden. Mit ihm werden Datensätze gesucht, wo ein bestimmtes Datenfeld einem bestimmten Wert entspricht.

## 9.6 Update-Operation

Eine Datenbank-Operation die einem App- und Web-Programmierer oft begegnet, ist das ändern eines vorher eingepflegten Datensatzes. Wie das im Prinzip geht, zeigt uns der nächste Code-Auszug.

```
#!/usr/bin/python3

import pymysql

# Öffnen der Datenbankverbindung
db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# Datenbank-Cursor wird definiert
cursor = db.cursor()

# Das notwendige SQL-Statement wird zusammengesetzt
sql = "UPDATE ANGESTELLTE SET AGE = 22 WHERE NACHNAME = 'Müller'"
try:
    # Ausführen des SQL-Kommandos
    cursor.execute(sql)
    # Commit-Anweisung an die Datenbank senden
    db.commit()
except:
    # Rollback im Fehlerfall
    db.rollback()

# Datenbankverbindung wieder trennen
db.close()
```

**Coding 21**

Wirklich interessant ist hierbei die Zeile mit dem SQL-Statement. In unserem Beispiel hatte es einen Fehler bei der Datenerfassung unseres Angestellten gegeben. Sein Alter muss von 33 Jahren auf 22 korrigiert werden. Dies wird mit dem SQL-Statement „**UPDATE ANGESTELLTE SET AGE = 22 WHERE NACHNAME = 'Müller';**“ bewerkstelligt. Diese Zeile einmal übersetzt, lautet die Anweisung im etwa folgend: „Aktualisiere (Tabelle) ANGESTELLTE und setze ‚Age‘ (Alter) auf 22, wo NACHNAME = ‚Müller‘ lautet“.

## 9.7 Lösch-Operation

Auch das Löschen von Datensätzen ist eine DB-Operation, die fast alle datenbankgestützten Applikationen beherrschen müssen.

```
#!/usr/bin/python3

import pymysql

# Datenbankverbindung öffnen
db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# Definition des DB-Cursors
cursor = db.cursor()

# Zusammenstellung des SQL-Statements
sql = "DELETE FROM ANGESTELLTE WHERE NACHNAME = 'Müller'"
try:
    # Ausführen des SQL-Kommandos
    cursor.execute(sql)
    # Commit an die DB senden
    db.commit()
except:
    # Rollback im Fehlerfall
    db.rollback()

# Datenbankverbindung wieder trennen
db.close()
```

**Coding 22**

Die SQL-Zeile „`DELETE FROM ANGESTELLTE WHERE NACHNAME = 'Müller'`“ löscht also alle Datensätze, wo der Nachname ‚Müller‘ lautet.

## 10 Systemprogrammierung

Pythons OS-Modul stellt eine Schnittstelle zum Betriebssystem dar. Hierbei funktioniert das Modul in allen Betriebssystemen, für die eine Python-Version bereitgestellt wird. Dieses ermöglicht es uns, dass Betriebssystem mit eigenen Tools oder Diensten zu erweitern. Auch Automatisierungsaufgaben sind damit möglich.

### 10.1 Ein Shell-Kommando ausführen

Mit `os.system()` lässt sich ein beliebiger Betriebssystembefehl der Kommandozeile ausführen.

```
#!/usr/bin/python3

import os

kommando = 'df -h'
os.system(kommando)
```

**Coding 23**

Um `os.system()` und auch alle folgenden Methoden des Moduls `os` ausführen zu können, muss es mittels `import` vorher im Python-Skript geladen werden. Mit der String-Variable `kommando` definieren wir im Vorfeld das Betriebssystem-Kommando. Der Befehl `df -h` listet unter Mac OS und Linux alle im System eingebundenen Partitionen bzw. Datenträger und deren Speicherbelegung auf. Der Output auf dem Rechner, auf dem dieses Vorlesungsskript geschrieben wurde, sieht folgendermaßen aus:

Dateisystem	Größe	Benutzt	Verf.	Verw%	Eingehängt auf
udev	7,8G	0	7,8G	0%	/dev
tmpfs	1,6G	1,7M	1,6G	1%	/run
/dev/sdc3	39G	13G	25G	34%	/
tmpfs	7,8G	225M	7,6G	3%	/dev/shm
tmpfs	5,0M	4,0K	5,0M	1%	/run/lock
tmpfs	7,8G	0	7,8G	0%	/sys/fs/cgroup
/dev/sda1	917G	703G	168G	81%	/data
tmpfs	1,6G	76K	1,6G	1%	/run/user/1000

## 10.2 Die Benutzerumgebung abfragen

Die Methode `os.environ()` zeigt die Benutzerumgebung des Script ausführenden Anwenders an. Gemeint sind hiermit alle System-Variablen, welche für den Benutzer in seiner Betriebssystem-Sitzung definiert und gesetzt sind.

```
#!/usr/bin/python3

import os
umgebung = os.environ
print(umgebung)
```

Coding 24

## 10.3 Ausgabe des aktuellen Verzeichnisses

Der Methodenaufruf `os.getcwd()` erlaubt die Ausgabe des aktuellen Arbeitsverzeichnisses: Current Working Directory (cwd).

```
#!/usr/bin/python3

import os
aav = os.getcwd()
print(aav)
```

Coding 25

## 10.4 Ausgabe der Gruppen-ID

Mittels `os.getgid()` kann die Gruppen-ID des Script ausführenden Benutzers ausgegeben werden.

```
#!/usr/bin/python3

import os
group_id = os.getgid()
print(group_id)
```

Coding 26

Unter Linux und Mac OS bekommen Benutzer und ihre Gruppen eine numerische Identifikationsnummer zugeordnet, anhand derer das System sie erkennt. Diese und die folgende Abfrage funktioniert nicht unter Windows (wohl aber unter Android und iOS).

## 10.5 Ausgabe der Benutzer-ID

Wie eben zuvor bereits erwähnt, werden auch Benutzern unter Mac OS und Linux Identifikationsnummern zugeordnet. Ausgegeben werden kann diese mittels `os.getuid()`.

```
#!/usr/bin/python3

import os
user_id = os.getuid()
print(user_id)
```

**Coding 27**

## 10.6 Ausgabe der Prozess-ID

Da die unixoiden Betriebssysteme scheinbar sehr gerne mit numerischen ID's arbeiten, wird einen eventuell bereits weniger verwundern, dass dieses bei Prozessen genauso gehandhabt wird. Mit `os.getpid()` wird die Prozess-ID ausgegeben.

```
#!/usr/bin/python3

import os
process_id = os.getpid()
print(process_id)
```

**Coding 28**

## 10.7 Abfrage des Betriebssystem-Kernels

Auch dieses Feature steht Windows-Benutzer nicht zur Verfügung. Für unix-artige Systeme hingegen kann mit dieser Methode (`os.uname()`) die Version des Betriebssystem-Kerns abgefragt werden.

```
#!/usr/bin/python3

import os
kernel = os.uname()
print(kernel)
```

**Coding 29**

Die Ausgabe auf einem Ubuntu-System sieht wir folgt aus:

```
posix.uname_result(sysname='Linux', nodename='huibuh', release='5.3.0-46-generic',
version='#38~18.04.1-Ubuntu SMP Tue Mar 31 04:17:56 UTC 2020', machine='x86_64')
```

Der Ausgabe können wir folgende Informationen entnehmen: Es handelt sich mit Ubuntu um ein Linux. Der Rechner trägt den Namen „huibuh“. Momentan läuft das System mit einem Kernel 5.3.0-46. Es handelt sich um die Ubuntu-Version 18.04 (LTS) und das ganze läuft auf einer 64-Bit-Hardware.

## 10.8 Change-Root-Operation

Eine weitere Errungenschaft, die dem Windows-Lager vorenthalten bleibt: **Change Root**. Ein wenig ist die Change-Root-Technik der unix-artigen Systeme mit Jails (Käfigen) zu vergleichen. Eine Technik, die bei Mac, BSD-Unix, Solaris und Linux verwendet wird, um z.B. bestimmte Anwendungen in einem eigenen abgesicherten Systembereich laufen zu lassen (so eine Art Betriebssystem innerhalb des Betriebssystems). Eine neue Change-Root-Umgebung kann mit Python mittels `os.chroot()` eingerichtet bzw. aktiviert werden. Wichtig ist dabei noch eine Pfad-Angabe, worin das ganze stattfinden soll. Dies ist als absoluter Pfad anzugeben. Das Verzeichnis dient dann als Container.

## 10.9 Auflistung des Verzeichnisinhaltes

Die folgenden sieben Operationen funktionieren alle auch unter Windows. Bei ihnen handelt es sich ausnahmslos um Befehle zum Bearbeiten von Dateien und Verzeichnissen.

An dieser Stelle noch eine kleine Anmerkung zur Syntax von Verzeichnispfaden. Unter Windows werden Pfade in folgender Schreibweise beschrieben: „C:\Dir1\Dir2\Dir3“. Unter Mac und Linux werden zum Trennen von Verzeichnis und Dateiname bzw. als Trennsymbol der Unterverzeichnisse nicht der Backslash, sondern der Slash verwendet: „/dir1/dir2/dir3“. Laufwerksbuchstaben (gefolgt von einem Doppelpunkt) gibt es hier nicht. Während unter Windows eine zweite Daten-Festplatte eventuell den Buchstaben D: und ein DVD-Laufwerk das E: erhalten hätten, so werden diese nach Unix-Philosophie z.B. so ins Datei-System eingebunden: Daten-Festplatte → „/data“, DVD-Laufwerk → „/media/cdrom“.

```
#!/usr/bin/python3

import os
pfad = '/var/log'
logdateien = os.listdir(pfad)
for zeile in logdateien:
    print(zeile)
```

Coding 30

Der hier zu sehende Beispiel-Code listet alle Dateien und Verzeichnisse auf, die sich unter „/var/log“ befinden. Auf einem Unix-System (und artverwandten) werden an dieser Stelle die Log-

Dateien des Systems vorgehalten. Mit der Variable `pfad` wird festgelegt, welcher Verzeichnisinhalt ausgegeben werden soll. Die Methode `os.listdir()` ermöglicht die Auflistung des Ordnerinhalts.

## 10.10 Erstellen eines Verzeichnisses

### 10.10.1 „Einfaches“ Erstellen von Verzeichnissen

Mit der Methode `os.mkdir()` können Verzeichnisse erstellt werden. Mit der Variable `pfad` (ich kann hier natürlich auch gleich direkt einen String eingeben) wird der absolute oder relative Pfad zu diesem neuen Verzeichnis angegeben.

```
#!/usr/bin/python3

import os
pfad = '/home/ben/test'
os.mkdir(pfad)
```

**Coding 31**

Obiger Programm-Code erstellt den Ordner „test“ in meinem User-Home-Verzeichnis.

### 10.10.2 „Rekursives“ Erstellen von Verzeichnissen

Möchte man mehrere Ordner z.B kaskadiert untereinander anlegen, ist jedoch dieser os-Modulaufruf zu verwenden: `os.makedirs()`

Mit dem Code

```
#!/usr/bin/python3

import os
pfad = '/home/ben/test1/test2/test3'
os.makedirs(pfad)
```

**Coding 32**

wird gleich eine ganze Verzeichnisstruktur angelegt. In meinem Heimatverzeichnis wird der Unterordner „test1“ kreiert. Gleichzeitig kommt es zur Erstellung des *Unterunterordners* „test2“, welcher wiederum nach ausführen des Scripts das weitere Unterverzeichnis „test3“ beherbergen wird.



## 10.11 Löschen von Dateien

Mit `os.remove()` kann der entsprechend in Klammern eingesetzte Datei-Pfad gelöscht werden.

```
#!/usr/bin/python3

import os
pfad = '/home/ben/test1/test2/test3/test4.txt'
os.remove(pfad)
```

**Coding 33**

Der Befehl funktioniert nur mit Dateien. Bei Verzeichnissen kommt es zu einer Fehlermeldung. Das Löschen von Verzeichnissen behandeln wir im nächsten Unterkapitel.

## 10.12 Löschen von Verzeichnissen

### 10.12.1 „Einfaches“ Löschen von Verzeichnissen

Mit der Methode `os.rmdir()` wird der Verzeichnispfad und somit der am Ende vom Pfad befindliche Ordner gelöscht.

```
#!/usr/bin/python3

import os
pfad = '/home/ben/test1/test2/test3'
os.rmdir(pfad)
```

**Coding 34**

Wichtig zu wissen, ist hierbei: Das Verzeichnis muss leer sein. Das gleiche gilt auch für das nun behandelte rekursive Löschen von Verzeichnissen.

### 10.12.2 Rekursives Löschen von Verzeichnissen

Zum einen bietet sich hier die Methode `os.removedirs()`, wobei dieser Weg aus Sicht des Autors mit Vorsicht zu genießen ist. Mittels es nachfolgenden Codes ließen sich die zuvor erstellen Verzeichnisse „/home/ben/test1“ und „/home/ben/test1/test2“ löschen.

```
#!/usr/bin/python3

import os
pfad = '/home/ben/test1/test2'
os.removedirs(pfad)
```

Coding 35

Belässt man die Pfadangabe auf „/home/ben/test1/“ kommt es zur Fehlermeldung, dass der Ordner „test1“ nicht leer ist. Nimmt man jedoch die Pfad-Definition, wie sie im Programm-Code zu sehen ist, wird das Verzeichnis „test1“ und sein Unterordner „test2“ gelöscht, nicht aber „/home“ und „/home/ben“.

Diese Syntax ist damit höchst unsicher. Besser ist für unixoide Systeme daher eher die Verwendung des folgenden Befehls:

```
os.system('rm -rf /home/ben/test1')
```

Man bedient sich also direkt dem Unix-Löschbefehls. Mit den oben zu sehenden Parametern löscht er ab dem Verzeichnis „test1“ rekursiv alle Dateien und Unterverzeichnisse (-rf ... recursive, force). Unter Windows ist hier der Löschbefehl `rmdir „c:\test1“ /s /q` zu verwenden.

## 10.13 Umbenennung von Verzeichnissen und Dateien

```
#!/usr/bin/python3

import os
name_alt = '/home/ben/test_alt'
name_neu = '/home/ben/test_neu'
os.rename(name_alt, name_neu)
```

Coding 36

Die Methode `os.rename(alterName, neuerName)` ermöglicht das Umbenennen von Dateien und Verzeichnissen. Im obigen Beispiel wurde das Verzeichnis „test\_alt“ in „test\_neu“ im Benutzerverzeichnis des Anwenders *Ben* umbenannt.

## 11 Strings, die 2.

Bei einem String handelt es sich um eine Zeichenkette. Er ist einer von drei sequentiellen Datentypen, die Python kennt. Die anderen beiden sind die Listen und Tupels. Anders als bei vielen anderen Programmiersprachen, lässt Python das Behandeln dieser Zeichenkette ähnlich einem Feld bzw. Arrays zu.

Hierbei wird also auch wie bei Arrays üblich von 0 an gezählt. Möchte ich den ersten Buchstaben einer Zeichenkette erhalte (nehmen wir zum Beispiel das Wort Apfel), muss ich wie im folgenden Script vorgehen:

```
#!/usr/bin/python3

zeichenkette = 'Apfel'
anfangsbuchstabe = zeichenkette[0];
print(anfangsbuchstabe)
```

**Coding 37**

Als Ausgabe erhalte ich „A“. Hätte ich auf das Element 1 dieser Zeichenkette zugegriffen, wäre nicht das A, sondern das p zurückgeliefert worden (wir beginnen von 0 an zu zählen, daher ist 1 bereits das zweite Element im String).

### 11.1 Len

Mit der integrierten Funktion `len()` kann die Länge einer Zeichenkette abgefragt werden bzw. aus wie vielen Zeichen sie besteht:

```
#!/usr/bin/python3

zeichenkette = 'Apfel'
str_laenge = len(zeichenkette)
print(str_laenge)
```

**Coding 38**

Als Resultat liefert uns dieses Script die Zahl 5 (die Anzahl Buchstaben, aus denen das Wort „Apfel“ besteht. Da wir ja auch hier wie bei den Elementen eines Arrays von Null an zählen, muss bei Abruf des letzten Elements der Zeichenkette folgende Beachtet werden:

```
#!/usr/bin/python3

zeichenkette = 'Apfel'
str_laenge = len(zeichenkette)
print(str_laenge)
letzter_buchstabe = zeichenkette[len(zeichenkette)-1]
print(letzter_buchstabe)
```

**Coding 39**

Die Funktion `len()` gibt zwar die Zahl 5 aus. Der letzte Buchstabe ist jedoch nach der Zählweise von Python der vierte (da wir ja bereits bei Null anfangen zu zählen). Also nur, wenn ich in diesem Fall `len()` um minus eins verringere bekomme ich den korrekten letzten Buchstaben des Wortes „Apfel“ ausgegeben.

## 11.2 String-Teile

Man kann Zeichenketten auch zerlegen und sich Teile des Strings ausgeben lassen. In Python spricht man hier von Slices (Scheiben). In anderen Programmiersprachen nennt man solche Scheiben auch Substrings.

```
#!/usr/bin/python3

zeichenkette = 'Vorgarten'
substring = zeichenkette[0:3]
print(substring)
```

**Coding 40**

Ich erhalte durch `substring = zeichenkette[0:3]` vom ersten Zeichen meines gewünschten Substrings bis zu seinem letzten (ich möchte die Teilzeichenkette „Vor“ aus dem Wort „Vorgarten“ herauslösen) den gewünschten String-Teil. Irritierend ist hierbei, dass ich für das Ende des Substrings die 3 eingeben muss. Nach Array-Zählweise wäre das doch die 2! Gebe ich bei den eckigen Klammern jedoch die 2 als zweite Zahl ein, erhalte ich lediglich „Vo“!

Das Ganze funktioniert ebenfalls mit `[ :3 ]`. Möchte ich hingegen den Teil hinter ‚Vor‘ und es soll bis zum letzten Zeichen des Strings gehen, funktioniert auch `[3: ]`.

## 11.3 String-Methoden

Methoden sind Funktionen sehr ähnlich. Während ich Funktionen in Strukturierter und in Funktionaler Programmierung verwenden kann, sind Methoden ein Teil der Objektorientierten

Programmierung (OOP). Eine Grunddefinition der OOP lautet: „Alles ist ein Objekt“. Im Programmcode erstellte Objekte, wie z.B. ein String, hat Eigenschaften (auch Attribute genannt). Und diese Eigenschaften kann ich mit Methoden verändern. Das ist ein zweiter Punkt, der OOP ausmacht. Wir wollen an dieser Stelle gar nicht tiefer in die Objektorientierte Programmierung eintauchen und auch nicht erörtern, was Programmierparadigmen sind und welche es gibt. Das würde den Inhalt dieses Einsteigerkurses sprengen. Im Folgenden werden wir jedoch die „Methoden“ behandeln, mit denen ich Zeichenketten verändern kann.

### 11.3.1 Umwandlung in Groß- und Kleinschreibung

Mit `strvar.upper()` werden die Zeichen meines Strings in Großbuchstaben umgewandelt. Die Methode `strvar.lower()` hingegen wandelt alle Zeichen des Strings in Kleinbuchstaben um. Hierbei ist „strvar“ lediglich eine beliebige String-Variable. Sie stellt somit das Objekt dar. Mit einem Punkt getrennt, wird dann die Methode, die wir anwenden wollen, angehängen.

Aber schauen wir uns das Ganze im Code an:

```
#!/usr/bin/python3

strvar = 'WoerterBuch'
gross = strvar.upper()
klein = strvar.lower()

print(gross)
print(klein)
```

**Coding 41**

Als Ergebnis kommen diese beide Zeilen heraus:

```
WOERTERBUCH
woerterbuch
```

### 11.3.2 Pattern-Matching

Taucht ein bestimmtes Muster in einer Zeichenkette auf? Dieses Pattern-Matching (Mustervergleich) löst Python durch den ***in-Operator***. Folgendes Coding zeigt, wie dieser zu verwenden ist:

```
#!/usr/bin/python3

begriff = 'Vorgarten'
if 'ten' in begriff:
    print('Der Begriff beinhaltet die Silbe \'ten\'')
```

Coding 42

Die entscheidende Zeile mit dem in-Operator: „`if 'ten' in begriff:`“. Kommt die Zeichenfolge „ten“ im Wort „Vorgarten“ vor? Das Ergebnis ist „wahr“!

### 11.3.3 Substitution

Teile in einem String auswechseln! Hierzu benötigen wir die Methode `strvar.replace()`.

```
#!/usr/bin/python3

begriff = 'Vorhof'
begriff = begriff.replace("Vor", "Hinter")
print(begriff)
```

Coding 43

Als Ergebnis wird die Zeile „Hinterhof“ ausgegeben. Ein Teil der Zeichenkette wurde durch einen anderen ersetzt.

## 12 Listen

### Zweiter Sequenz-Typ

Eine Liste wird in anderen Programmiersprachen auch Array oder Feld genannt. Die Inhalte einer Liste werden Elemente genannt. Ihr Typ kann beliebig sein (Integer, Fließkommazahl oder ein String).

Das nächste Script zeigt uns, wie man diese Listen erstellen kann und welche Inhalte sie exemplarisch beinhalten können.

<pre>#!/usr/bin/python3  feld1 = [2, 4, 5, 7] feld2 = ['Peter', 'Paul', 'Marry'] feld3 = ['Karl', 22, '34x@z', [3, 3]]  for element in feld1:     print(element)  print('----')  for element in feld2:     print(element)  print('----')  for element in feld3:     print(element)</pre>	<pre>2 4 5 7 ---- Peter Paul Marry ---- Karl 22 34x@z [3, 3]</pre>
--	--

**Coding 44**

Der Output dazu steht rechts in Grün neben dem Code. Code und Ausgabe zeigen, dass Elemente in Listen Zahlen, aber auch Zeichenketten sein können. Und nicht nur das. Array Nr. 3 beweist, dass auch eine Mischung verschiedener Typen möglich ist.

### 12.1 Einzelelemente einer Liste abrufen

Coding 45 wird uns zeigen, wie einzelne Elemente einer Liste gezielt abgerufen werden können.

```
#!/usr/bin/python3

feld1 = [2, 4, 5, 7]
print(feld1[2])
```

**Coding 45**

Es wird durch dieses Script die „5“ ausgegeben. Kleine Erinnerung: Bei den Listen-Elementen wird von der Null ab gezählt.

## 12.2 Einzelne Elemente ändern

Wenn ich über diesen Weg ein Listenelement gezielt aufrufen kann, dann sollte es doch sehr ähnlich möglich sein, dieses auch zu verändern? Der folgende Code schafft uns Gewissheit:

```
#!/usr/bin/python3
```

```
zahlen = [1, 2, 3]
print(zahlen)
zahlen[1] = 4
zahlen[2] = 9
print(zahlen)
```

Coding 46

```
[1, 2, 3]
[1, 4, 9]
```

Zuerst beinhaltet unser Array (Liste) „zahlen“ die ersten drei positiven Natürlichen Zahlen. In den Zeilen 5 und 6 ändern wir das erste und zweite Element ab (Index 1 und 2, denn wir zählen ja von Null an). Aus der „2“ wird ein „4“ und aus der „3“ eine „9“. Und schon ist die klassische Standard-Botschaft an intelligentes Leben in diesem Universum fertig: Die ersten drei Quadratzahlen<sup>5</sup>.

## 12.3 Methoden und Operationen

### 12.3.1 Neue Elemente an eine Liste anhängen

Auch für Listen gibt es Methoden, um diese zu bearbeiten. Mit der Methode `append()` werden neue Elemente an das Ende einer Liste angehängt:

```
#!/usr/bin/python3
```

```
liste = ['Katze', 'Hund', 'Vogel']
print(liste)
liste.append('Goldfisch')
print(liste)
```

Coding 47

```
['Katze', 'Hund', 'Vogel']
['Katze', 'Hund', 'Vogel', 'Goldfisch']
```

<sup>5</sup> Im 1968 verfilmten Sciencefiction „2001: Odyssee im Weltraum“ von Stanley Kubrick diente ein Monolith mit den exakten Seitenmaßen 1 x 4 x 9 und einer absolut ebenen Oberfläche als eine Art Botschaft von außerirdischen, hochentwickelten Mentoren an uns Menschen auf der Erde. Außerdem fungierte er in der Geschichte als „Intelligenz-Kathalsator“ für aussichtsreiche Lebensformen im Universum – nicht nur den Homo Sapiens.



### 12.3.2 Listen an Listen anhängen

Wenn es mehrere Elemente gibt, die an das Ende einer vorhandenen Liste gehangen werden sollen, hilft die Methode `extend()` weiter.

```
#!/usr/bin/python3
```

```
liste1 = ['Katze', 'Hund', 'Vogel']  
print(liste1)  
liste2 = ['Goldfisch', 'Vogelspinne', 'Alligator']  
liste1.extend(liste2)  
print(liste1)
```

**Coding 48**

```
['Katze', 'Hund', 'Vogel']  
['Katze', 'Hund', 'Vogel', 'Goldfisch', 'Vogelspinne', 'Alligator']
```

Code-Schnippse 48 in Blau zeigt uns, wie die `extend()`-Methode anzuwenden ist. Darunter in Grün ist der Output zu sehen. Die Liste „`liste1`“ wurde um das Feld „`liste2`“ erweitert.

### 12.3.3 Elemente sortieren

Möchte man die Elemente eines Arrays alphabetisch oder numerisch sortieren, dann ist die Methode `sort()` zu verwenden.

```
#!/usr/bin/python3
```

```
liste1 = ['Katze', 'Hund', 'Vogel']  
liste2 = [9, 23, 5, 4]  
print(liste1)  
print(liste2)  
liste1.sort()  
liste2.sort()  
print(liste1)  
print(liste2)
```

**Coding 49**

```
['Katze', 'Hund', 'Vogel']  
[9, 23, 5, 4]  
['Hund', 'Katze', 'Vogel']  
[4, 5, 9, 23]
```

Das Coding 49 und seine Ausgabe zeigt, `sort()` kann nicht nur Zahlen korrekt sortieren, sondern schafft dieses auch in richtiger alphabetischer Reihenfolge bei Zeichenketten.

### 12.3.4 Letztes Element von der Liste entfernen

Ist das letzte Element einer Liste zu entfernen, wird dieses mittels Methode `pop()` umgesetzt.

<pre>#!/usr/bin/python3  liste = ['Katze', 'Hund', 'Vogel'] print(liste) liste.pop() print(liste)</pre>	<pre>['Katze', 'Hund', 'Vogel'] ['Katze', 'Hund']</pre>
<b>Coding 50</b>	

Der Liste, bestehend aus den Elementen „Katze“, „Hund“ und „Vogel“, wurde der letzte Eintrag entnommen. Mit der `pop()`-Methode kann dieser letzte Wert, wenn nötig, zur Verarbeitung einer Variable zugeordnet werden.

## 12.4 Elemente löschen

Aber was ist, wenn wir einen Wert am Anfang oder mitten in der Liste entfernen möchten? Auch hier hilft uns die `pop()`-Methode weiter. In diesem Falle ist der entsprechende Index des Elements beim Aufruf der Methode in Klammern zu stellen. Lasse ich die Klammer leer, dann wird das letzte Element entfernt.

Im Folgenden zwei Beispiele, wie einmal das erste Element und zum anderen das mittlere entfernt werden.

<pre>#!/usr/bin/python3  liste = ['Katze', 'Hund', 'Vogel'] print(liste) liste.pop(0) print(liste)</pre>	<pre>#!/usr/bin/python3  liste = ['Katze', 'Hund', 'Vogel'] print(liste) liste.pop(1) print(liste)</pre>
<b>Coding 51</b>	<b>Coding 52</b>
<pre>['Katze', 'Hund', 'Vogel'] ['Hund', 'Vogel']</pre>	<pre>['Katze', 'Hund', 'Vogel'] ['Katze', 'Vogel']</pre>

Aber die `pop()`-Methode ist hierfür nicht die einzige Möglichkeit. Alternativ kann ich über den Operator `del()` ebenfalls Elemente einer Liste löschen, deren Index-Wert bekannt ist.

```
#!/usr/bin/python3

liste = ['Katze', 'Hund', 'Vogel']
print(liste)
del liste[1]
print(liste)
```

**Coding 53**

Aber was ist, wenn der Index-Wert unbekannt ist? In diesem Fall kommt die Methode „`remove()`“ zum Einsatz.

```
#!/usr/bin/python3

liste = ['Katze', 'Hund', 'Vogel']
print(liste)
liste.remove('Hund')
print(liste)
```

**Coding 54**

In Klammern muss hier jedoch dann der Wert bzw. der Inhalt des Elements genannt werden.

## 12.5 Listen und Strings

### 12.5.1 Funktion `list()`

Listen, als auch Strings sind Sequenzen. Eine Liste ist eine Sequenz von Werten, ein String eine Sequenz von Zeichen. Dennoch oder gerade aus diesem Grund ist ein String keine Liste. Möchte ich einen String in eine Liste überführen, kann dies mit der Funktion `list()` bewerkstelligt werden.

```
#!/usr/bin/python3

str = 'abc'
liste = list(str)
print(str)
print(liste)
```

**Coding 55**

```
abc
['a', 'b', 'c']
```

### 12.5.2 Methode split()

Wird man jedoch mit einer CSV-Datei oder Flat-DB konfrontiert, ist eine Zerlegung des Strings in Einzelzeichen wenig hilfreich. Entscheidend ist an dieser Stelle dann das Trennzeichen, welches die Datei verwendet. Hier kommt dann die Funktion `split()` ins Spiel. Diese Daten müssen nicht zwingend aus einer Datei stammen. Ich kann die Strings auch temporär zur Script-Laufzeit erstellen.

```
#!/usr/bin/python3
```

```
str = "101202;Montag;23:04;12.03.2019;Schule am Wassertor"  
liste = str.split(';')  
print(liste)
```

**Coding 56**

```
['101202', 'Montag', '23:04', '12.03.2019', 'Schule am Wassertor']
```

Wie man der Zeichenkette (definiert in Zeile 3 von Coding 56) entnehmen kann, wurde als Trennzeichen (*Delimiter*) das Semikolon verwendet. Dies ist sehr oft als Stand-Trenner in CSV-Dateien anzutreffen. Entsprechend wurde der `split-Methode()` dann auch das Semikolon als Delimiter mitgeteilt (siehe hierzu Zeile 4). Als Endresultat erhalten wir eine Liste, die als Datensatz weiterverwendet werden kann.

### 12.5.3 Methode join()

Habe wir den umgekehrten Fall, es liegt eine „Wörter-Liste“ vor, dann kann diese mittels der Methode `join()` in eine Zeichenkette umgewandelt werden.

```
#!/usr/bin/python3
```

```
liste = ['Heute', 'ist', 'ein', 'schöner', 'Tag!']  
trenner = ' '  
str = trenner.join(liste)  
print(str)
```

**Coding 57**

```
Heute ist ein schöner Tag!
```

Zur Bildung eines Satzes ist das Leerzeichen das am besten geeignete Trennzeichen. Diesem Trenner wird dann mit der Methode `join()` die Liste übergeben und diese fügt das Leerzeichen zwischen den einzelnen Elementen ein.

## 13 Tupel

### Dritte Sequenz-Art

Kommen wir zur dritten Werte-Sequenz-Art, die Python kennt: Das Tupel. Sehr ähnlich einer Liste, wird es folgendermaßen aufgebaut: `tupel = ('a', 23, 't', '3b', 4)`

Wie man sehen kann, werden statt der eckigen Klammern (die bei der Liste Verwendung finden) hier in diesem Fall die runden Klammern benutzt. Daran kann man diese beiden sehr ähnlichen Sequenz-Arten voneinander unterscheiden. Aber es gibt noch einen große und wichtigen Unterschied zwischen den beiden: **Die Werte eines Tupels können nicht editiert werden.** Das obige Beispiel zeigt außerdem, das wie bei Listen, die Werte aus Zahlen, als auch Zeichen bzw. Zeichenketten bestehen können.

Manchmal kommt es vor, dass Funktionen in Python Sequenzen in Form von Tupel zurückliefern. Was ist aber, wenn ich einzelne Werte dieses Tupels noch verändern muss? In diesem Fall muss das Tupel in eine Liste konvertiert werden. Aber der Reihe nach ...

```
#!/usr/bin/python3
```

```
tupel = ('Montag', '13:00', 'Mathematik', 'Frau Günzel')
print(tupel)
tupel[2] = 'Biologie'
print(tupel)
```

Coding 58

```
('Montag', '13:00', 'Mathematik', 'Frau Günzel')
Traceback (most recent call last):
  File "./tupel.py", line 5, in <module>
    tupel[2] = 'Biologie'
TypeError: 'tuple' object does not support item assignment
```

In Coding 58 konstruieren wir uns unser eigenes Tupel und dreister Weise versuchen wir dieses nachträglich zu verändern. Es kommt zu der im Output zu sehenden Fehlermeldung. An dieser Stelle scheinen weder Autor noch Python-Dokumentation gelogen zu haben. Also wandeln wir im nächsten Schritt wie angekündigt das Tupel in eine Liste um (Coding 59):

```
#!/usr/bin/python3
```

```
tupel = ('Montag', '13:00', 'Mathematik', 'Frau Günzel')
print(tupel)
liste = list(tupel)
liste[2] = 'Biologie'
print(liste)
```

Coding 59

```
('Montag', '13:00', 'Mathematik', 'Frau Günzel')  
['Montag', '13:00', 'Biologie', 'Frau Günzel']
```

Mit der Funktion `list()` (in Zeile 5) wird das Tupel in eine Liste umgewandelt und, oh Wunder, Wert 2 der Sequenz lässt sich endlich abändern.

## 14 Web-Programmierung mit Python

### 14.1 Grundlagen

#### 14.1.1 Grundlagen HTML

Mit Python ist serverseitige Web-Programmierung möglich. Dies bedeutet, das sämtliche Programm-Intelligenz auf dem Web-Server liegt und die Anwender, die diese Web-App benutzen möchten, brauchen lediglich einen halbwegs modernen Webbrowser. Dazu gelten die aktuellen Versionen von Safari, Firefox, Chrome/Chromium und Edge. Auf der Client-Seite gibt es ansonsten keine weiteren Softwareabhängigkeiten. Web-Applikationen erfreuen sich daher seit etlichen Jahren an extrem hoher Beliebtheit und eigentlich können nur reine Web-Apps am Ende wirklich plattform-unabhängig und plattform-übergreifend sein.

Damit eine Web-Anwendung jedoch laufen kann, benötigt sie einen wichtigen Dienst: Einen Webserver. Sicherlich kann man dies mit dem Internet Information Server IIS von Microsoft tun, jedoch wird dieser wegen großer Sicherheits-Aspekte selten bis nie im Internet verwendet<sup>6</sup>. Einen sehr oft im WWW verwendeter Web-Service ist der Apache 2.X. Eine recht komfortable Installation des Ganzen bietet der Software-Bundle XAMPP, den es für Windows, Mac und Linux gibt. Linux-Anwender können ebenfalls aber auch auf die Softwarepakete ihrer Distribution zurückgreifen. Der Bundle XAMPP hat den Vorteil, dass neben dem Webserver auch noch gleich eine MySQL-Datenbank und PHP mitinstalliert werden (Das zweite P am Ende der Abkürzung steht für Perl – eine weitere, sehr mächtige Interpreter-Sprache, welche auch in der Lage ist, Webprogrammierung anzubieten). Ein Controll-Center ist ebenso mit an Bord. Bezogen werden kann XAMPP unter <https://www.apachefriends.org/index.html>.

Nach der Installation muss Python dem Webserver noch als Script-Sprache bekannt gegeben werden, die ebenfalls für CGI verwendet werden kann. CGI steht für Common Gateway Interface und ist die Schnittstelle zum Webserver für dynamischen Web-Content, den Perl und Python verwenden. Um die Script-Datei-Endung dem Apache ebenfalls als cgi-fähige Sprache bekannt zu geben, muss in der Konfigurationsdatei `C:\xampp\apache\conf\httpd.conf` die Zeile `AddHandler` um die Endung „.py“ erweitert werden. Danach den Webserver über das Controll-Center stoppen und wieder starten, damit die Änderung vom Web-Systemdienst erkannt wird.

Alle folgenden Scripte sind auf einem Windows-System mit vorhandener XAMPP-Installation unter `C:\xampp\cgi-bin` zu hinterlegen.

Eine einfache HTML-Seite würde ich wie folgt aufbauen:

---

6 Bei Web-Servern ist es aus Sicherheitsgründen ebenfalls selten der Fall, dass Microsoft-Server im WWW den Vorzug erhalten.



Rechts ist das HTML zu sehen, wie es am Ende der Browser darstellt. Links steht der dazugehörige HTML-Code. Er beginnt mit einem HTML-Tag. *Tags* sind Abschnitte in einer HTML-Seite, welche sie in bestimmte Bereiche untergliedert. Diese können auch Formatierungsanweisungen sein. In der Regel gibt es das Beginn-Tag (z.B. `<HTML>`) und das Ende-Tag (z.B. `</HTML>`). Die zwei wichtigen Hauptbereiche auf einer HTML-Seite sind der Kopf (`<HEAD>`) und der Dokumenten-Körper (`<BODY>`). Während der Kopf-Bereich eher Metainformationen beinhaltet (der Einfachheit halber wurde er in unserem Beispiel sogar leer gelassen), befinden sich im HTML-Body die Elemente für den eigentlichen Seiteninhalt. Es gibt auch Tags, die kein Ende-Tag benötigen. Als Beispiel seien hier `<BR>` (Break Row – Zeilenumbruch) und `<HR>` (Horizontal Rule – horizontale Linie) genannt.

### 14.1.2 Grundlagen Python-CGI

Nun wandeln wir unsere kleine HTML-Seite in ein Python-Skript um:

```
#!/usr/bin/python3

import cgi

print("Content-type: text/html")
print()
print("<HTML>")
print("<HEAD>")
print("</HEAD>")
print("<BODY>")
print("<H1> Hallo Welt! </H1>")
print("</BODY>")
print("</HTML>")
```

**Coding 60**

Neben dem Import des CGI-Moduls wurden fast ausschließlich nur Print-Anweisungen den HTML-Tags vorangestellt. Statt in einem Terminal gibt das CGI jedoch die Printausgaben an den beim Web-Server anfragenden Browser aus.



Es besteht die Möglichkeit, die Print-Anweisungen in dem Python-Script derart abzuändern, dass mit ganzen HTML-Blöcken innerhalb des Programm-Codes gearbeitet werden kann. Damit wird es möglich, größere Abschnitte von HTML-Code zu übernehmen (z.B. per Copy/Paste aus einem HTML-Editor heraus), ohne dass die HTML-Zeilen jedes Mal in die Print-Funktion eingezwängt werden müssen. Auf Dauer kann dieses recht lästig werden. Siehe hierzu Coding 61:

```
#!/usr/bin/python3

import cgi

print("Content-type: text/html")
print()

print("""
<HTML>
    <HEAD>
    </HEAD>
    <BODY>
        <H1> Hallo Welt! </H1>
    </BODY>
</HTML>
""")
```

**Coding 61**

Der Vorteil dieser Form der Print-Anweisung ist, dass ich hier die Einrückungen der HTML-Struktur mit übernehmen kann. Bei der Output-Variante davor war das nicht möglich, da dann die Einrückungen als Beginn bzw. Zugehörigkeit von Python-Kontrollstrukturen (z.B. If-Anweisungen und For-Schleifen) interpretiert werden.

Aber was wäre die dynamische Webprogrammierung ohne Buttons, Textboxen, Optionsschaltern und weiteren Formular-Elementen? Diese Elemente darzustellen, ist die eine Sache. Ein andere ist es, dafür zu sorgen, dass Python-Code sich dieser Eingaben annimmt und diese entsprechend auswertet.

## 14.2 Formular-Elemente

### 14.2.1 Das Textfeld

Eine Textbox wird wie folgt dargestellt und ausgewertet:

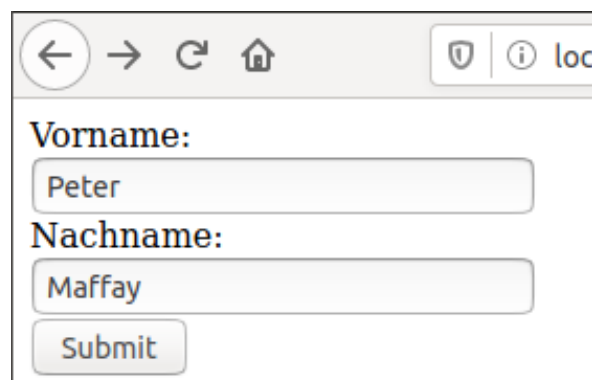
```
#!/usr/bin/python3

import cgi

print("Content-type: text/html")
print()

print("""
<HTML>
    <HEAD>
    </HEAD>
    <BODY>
        <form action="/cgi/auswertung.py">
            <label for="fname">Vorname:</label><br>
            <input type="text" id="fname" name="fname"><br>
            <label for="lname">Nachname:</label><br>
            <input type="text" id="lname" name="lname"><br>
            <input type="submit" value="Submit">
        </form>
    </BODY>
</HTML>
""")
```

Coding 62



The screenshot shows a web browser window. The address bar contains a lock icon, an information icon, and the text 'loc'. The main content area displays a form with the following elements:

- A label 'Vorname:' followed by a text input field containing the text 'Peter'.
- A label 'Nachname:' followed by a text input field containing the text 'Maffay'.
- A 'Submit' button located below the second input field.

Das Aussehen des generierten Formulars ist in Coding 62 zu sehen. Wichtig ist hierbei die Zeile „`<form action = ...`“. In dieser wird nämlich festgelegt, welchen Namen die Python-Datei trägt, die die gesetzten Inhalte des Web-Formulars abfängt und auswertet. Die vier Zeilen darunter

definieren die beiden Textfelder und den Submit-Button am Ende des Formulars, benötigen wir, um die Eingaben an das Auswert-Script zu senden.

Das für die Auswertung zuständige Python-Script nennen wir banaler Weise `auswertung.py`.

```
#!/usr/bin/python3

import cgi

form = cgi.FieldStorage()
vorname = form.getvalue("fname")
nachname = form.getvalue("lname")

print("Content-type: text/html")
print()
print("""
    <html>
    <head><title>Auswertung</title></head>

    <body>
    <h1> Auswertung: </h1>
""")
print("Dein Vorname lautet: <b>", vorname, "</b>")
print("<br>Dein Nachname lautet: <b>", nachname, "</b>")
print("""
    </body>
    </html>
""")
```

**Coding 63**

Es wird ein Objekt benötigt, welches sich der Inhalte des Formulars der CGI-Seite zuvor annehmen kann. Dies erledigt „`form = cgi.FieldStorage()`“. In den Zeilen darauf werden mit `form.getvalue()` die Inhalte der beiden Textboxen ausgelesen und entsprechenden String-

Variablen zugeordnet. Danach wird mittels Print-Anweisungen HTML-Code erzeugt, über welchen diese beiden Zeichenketten-Variablen ausgegeben werden.

### 14.2.2 Radio-Buttons

Radio-Buttons, zu Deutsch auch Auswahlknöpfe genannt, sind dazu da, sich für genau eine Option bei einer limitierten Auswahl zu entscheiden.

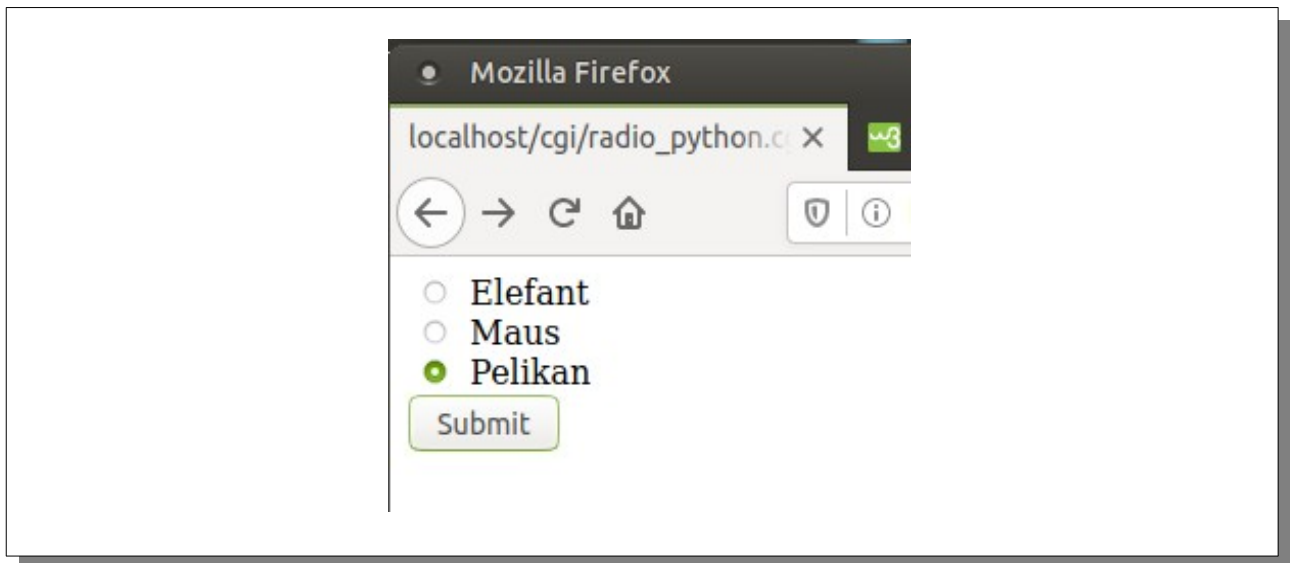
```
#!/usr/bin/python3

import cgi

print("Content-type: text/html")
print()

print("""
        <label for="pelikan">Pelikan</label>
<HTML>
    <HEAD>
    </HEAD>
    <BODY>
        <form action="/cgi/auswertung.py">
            <input type="radio" id="elefant" name="tier"
value="Elefant">
            <label for="elefant">Elefant</label><br>
            <input type="radio" id="maus" name="tier"
value="Maus">
            <label for="maus">Maus</label><br>
            <input type="radio" id="Pelikan" name="tier"
value="Pelikan">
            <br><input type="submit" value="Submit">
        </form>
    </BODY>
</HTML>
""")
```

**Coding 64**



Coding 64 kann der für Radio-Buttons erforderliche Code für das CGI-Script entnommen werden. Darunter ist zu sehen, wie das fertige Formular im Browser dargestellt wird.

Kommen wir nun abermals zum Auswert-Script.

```
#!/usr/bin/python3

import cgi

form = cgi.FieldStorage()
radio_button = form.getvalue("tier")

print("Content-type: text/html")
print()
print("""
    <html>
    <head><title>Auswertung</title></head>

    <body>
    <h1> Auswertung: </h1>
    <p> Folgende Wahl wurde getroffen:</p><b>
""")
print(radio_button)
print("""
    </b></body>
    </html>
""")
```

**Coding 65**



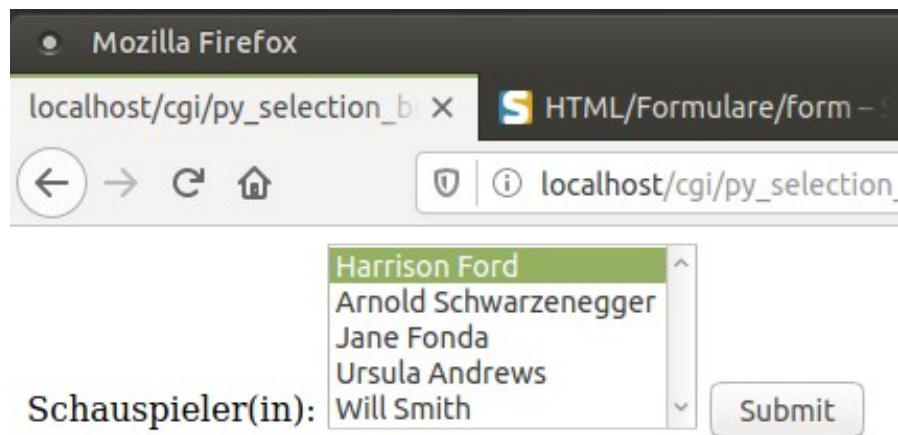
### 14.2.3 Auswahllisten

Eine Alternative zum Radio-Button stellt die Auswahlliste, auch Selection-Box genannt, dar.

```
#!/usr/bin/python3

import cgi

print("Content-type: text/html")
print()
print("""
<HTML>
    <HEAD>
    </HEAD>
    <BODY>
        <form action="/cgi/auswertung.py">
            <label>Schauspieler(in):
                <select name="schauspieler" size="5">
                    <option value="Harrison Ford">Harrison Ford</option>
                    <option value="Joe Black">Joe Black</option>
                    <option value="Jane Fonda">Jane Fonda</option>
                    <option value="Ursula Andrews">Ursula Andrews</option>
                    <option value="Will Smith">Will Smith</option>
                </select>
            </label>
            <input type="submit" value="Submit">
        </form>
    </BODY>
</HTML>
""")
```



Und wir bleiben unserer Tradition treu und betrachten nun das dazugehörige Auswert-Script:

```
#!/usr/bin/python3

import cgi

form = cgi.FieldStorage()
actor = form.getvalue("schauspieler")
print("Content-type: text/html")
print()
print("""
    <html>
    <head><title>Auswertung</title></head>

    <body>
    <h1> Auswertung: </h1>
    Ausgesucht wurde ...
    <b>
""")
print(actor)
print("""
    </b><br>
    </body>
    </html>
""")
```

**Coding 67**



#### 14.2.4 Buttons – Schaltflächen

Manches Mal liefert uns unsere zu programmierende Web-App die Anforderlichkeit, verschiedene Schaltflächen drücken zu können.

```
#!/usr/bin/python3

import cgi

print("Content-type: text/html")
print()
print("""
<HTML>
    <HEAD>
    </HEAD>
    <BODY>
        <H4> Du hast die Wahl</H4>
        <p>Klick Knopf 1 oder Knopf 2 an!</p>
        <form action=\"/cgi/auswertung.py\">
            <button type="submit" name="action" value="K1">Knopf1</button>
            <button type="submit" name="action" value="K2">Knopf2</button>
        </form>
    </BODY>
</HTML>
""")
```

Coding 68





Welche Knopf gedrückt wurde verrät uns einmal wieder das Auswert-Script ...

```
#!/usr/bin/python3

import cgi

form = cgi.FieldStorage()
knopf = form.getvalue("action")

print("Content-type: text/html")
print()
print("""
    <html>
    <head><title>Auswertung</title></head>

    <body>
    <h1> Auswertung: </h1>
    Es wurde ...<br>
    <b>
""")
print(knopf)
print("""
    </b><br>angeklickt!
    </body>
    </html>
""")
```

**Coding 69**

