Leibniz Universität
IT Services

Leibniz Universität Hannover

Einführung in Python

Handout
erstellt am 26. Oktober 2020
von Anja Aue (aue@luis.uni-hannover.de)

Genutzte Software im Kurs

- Betriebssystem: Windows, Linux
- Standardversion von Python in der Version 3.x:
  https://www.python.org/downloads/

Installation unter Windows

- Download des aktuellsten Pakets
- Ausführen der Datei „python-3.x.exe“.
- Aktivierung der Kontrollkästchen „Install launcher“ bei der Ausführung als Administrator.
- Aktivierung des Kontrollkästchen „Add Python to PATH“ im Dialog
- Mausklick auf den Link „Install now“. Beachte den Installationspfad!

Kursmaterialien und Übungsdateien

- Download beim Anmeldedialog des Tools Webex Training
- Im Anhang der Informationsmail zum Kurs

. . . in den Sitzungsinformationen

- „Kursmaterialien“ im Bereich „Sitzungsinformationen“
- Klick auf den Link der herunter zu ladenden Datei
- Weitere Schritte je nach genutzten Browswer / App

Ansehen des Kurstutorials als pdf-Datei

- Acrobat Reader
- Foxit Reader
- Okular

Entpacken der Übungsdateien

Beschreibung für das Betriebssystem Windows:

- Suchen der gewünschten zip-Datei im Windows Explorer
- Rechter Mausklick auf die Datei. „Alle Extrahieren“ im Kontextmenü

Übungen: Puzzle, Lückentexte

- Öffnen mit einem Webbrowser.
- Inhaltsverzeichnis: index.htm. Jeder Link führt zu einer Übung.
- „.htm“. Eine Übungsdatei.

Navigationshilfen in den Übungsdateien

- Pfeiltasten nach links: Wechsel zur vorherigen Übung
- Pfeiltasten nach rechts: Wechsel zur nächsten Übung
- Schaltfläche „index“: Zurück zum Inhaltsverzeichnis

Hinweise zu den Lückentexten

- Ausfüllen mit Hilfe der Tastatur.
- Keine Einrückungen wie bei einem Programmiertext

Hinweise zu den Puzzeln

- Mit gehaltener Maustaste werden die entsprechenden Elemente an die gewünschte Position gezogen.
- Sobald die Maustaste losgelassen wird, wird das Element an der Position eingefügt.

Python-Dateien

- Dateien mit der Endung „.py“
- Öffnen mit dem Editor IDLE (in der installierten Standardversion enthalten)

Aufbau der Programmierdateien

- Beschreibung der Übung am Anfang der Codedatei als Kommentar. Der Kommentar beginnt und endet mit drei aufeinanderfolgenden Anführungszeichen.
- Fehlerbehafteter Code darunter muss angepasst werden.
- Eingabe von Code durch den Teilnehmenden, um die Programmieraufgabe zu lösen.

Nutzung als Taschenrechner

Lernziel:

- Kennenlernen der Python Shell
- Eigenschaften von Ganzzahlen und Gleitkommazahlen
- Rechnen mit Zahlen

Steckbrief „Python Shell“

- Name: IDLE
- Anwendung: idle.pyw
- Speicherort unter
  Windows:[path]\Programs\Python\Python38-32\Lib\idlelib

Öffnen

- Klick auf die Anwendung
- Klick auf das passende Icon auf dem Desktop
- Klick auf den entsprechenden Eintrag im Startmenü des Betriebssystems

Aufbau

Titelleiste	Python-Version. Minimieren und Maximieren des Fensters.
Menüleiste	Alle Befehle zum Arbeiten mit der Shell und / oder Editor.
Shell	Python-Version. Befehle zum Anzeigen von Informationen. Eingabe von Code.

Wichtige Menübefehle

Shell - Restart Shell	Neustart der Shell.
Options - Configure IDLE	Konfiguration der Shell.

Zahlen

- Bildung aus ein oder mehreren Ziffern
- Zählen von Mengen
- Gleitkommazahlen oder Festkommazahlen

Ganzzahlen / Festkommazahlen

```
1  >>> -123456789
2  -123456789
3  >>> 26
4  26
5  >>> 0b00011010
6  26
7  >>> 0x1A
8  26
9  >>> 0o32
10 26
```

. . . im Dezimalsystem

```
1  >>> -123456789
2  -123456789
3  >>> 26
4  26
```

- Basis von 10
- Standardsystem beim Rechnen

. . . im Binärsystem

```
1  >>> 0b00011010
2  26
```

- Basis von 2
- Nur die Ziffern 0 und 1
- Zahlensystem des Computers
- Aus- und einschalten von Bits
- 8 Bits = 1 Byte

. . . im Hexadezimalsystem

```
1  >>> 0x1A
2  26
3  >>> 0b00011010
4  26
```

- Basis von 16
- Nur die Ziffern von 0 bis 9 plus die Buchstaben A bis F
- 1 Hexadezimalzeichen = 4 Bits
- Kodierung von Farbwerten

. . . im Oktalsystem

```
1  >>> 0o32
2  26
```

- Basis von 8
- Ziffern von 0 bis 7
- Nutzung von Programmiersprachen

Gleitkommazahlen

```
1  >>> -5.3
2  -5.3
3  >>> 5.12345678
4  5.12345678
5  >>> 0.5
6  0.5
7  >>> .5
8  0.5
9  >>> 2.0e+24
10 2e+24
```

Attributes

- $signifikant * basis^{exponent}$
- Nährung an ein Wert
- Keine exakter Wert

Vier Grundrechenarten

```
1  >>> 5 + 2
2  7
3  >>> 5 - 2
4  3
5  >>> 5 * 2
6  10
7  >>> 5 / 2
8  2.5
```

Addition

```
1  >>> 5 + 2
2  7
3  >>> 0xA + 0x8
4  18
5
6  >>> 5.3 + 2.5
7  7.8
```

Subtraktion

```
1  >>> 5 - 2
2  3
3  >>> 0xA - 0x8
4  2
5
6  >>> 5.3 - 2.5
7  2.8
```

Multiplikation

```
1  >>> 5 * 2
2  10
3  >>> 0xA * 0x8
4  80
5
6  >>> 5.3 * 2.5
7  13.25
```

Division

```
1  >>> 5 / 2
2  2.5
3  >>> 5.0 / .5
4  10.0
5  >>> 5 / 2.5
6  2.0
7  >>> 5.0 / 2
8  2.5
```

Division durch Null

```
>>> 5.5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
```

Ganzzahldivision

```
1  >>> 5 // 2
2  2
3  >>> 5.0 // .5
4  10.0
5  >>> 5 // 2.5
6  2.0
7  >>> 5.0 // 2
8  2.0
```

Arbeiten mit Potenzen

```
1 >>> 5.3 ** 2.5
2 64.66803638583747
3 >>> 5 ** 2
4 25
```

Division mit Rest

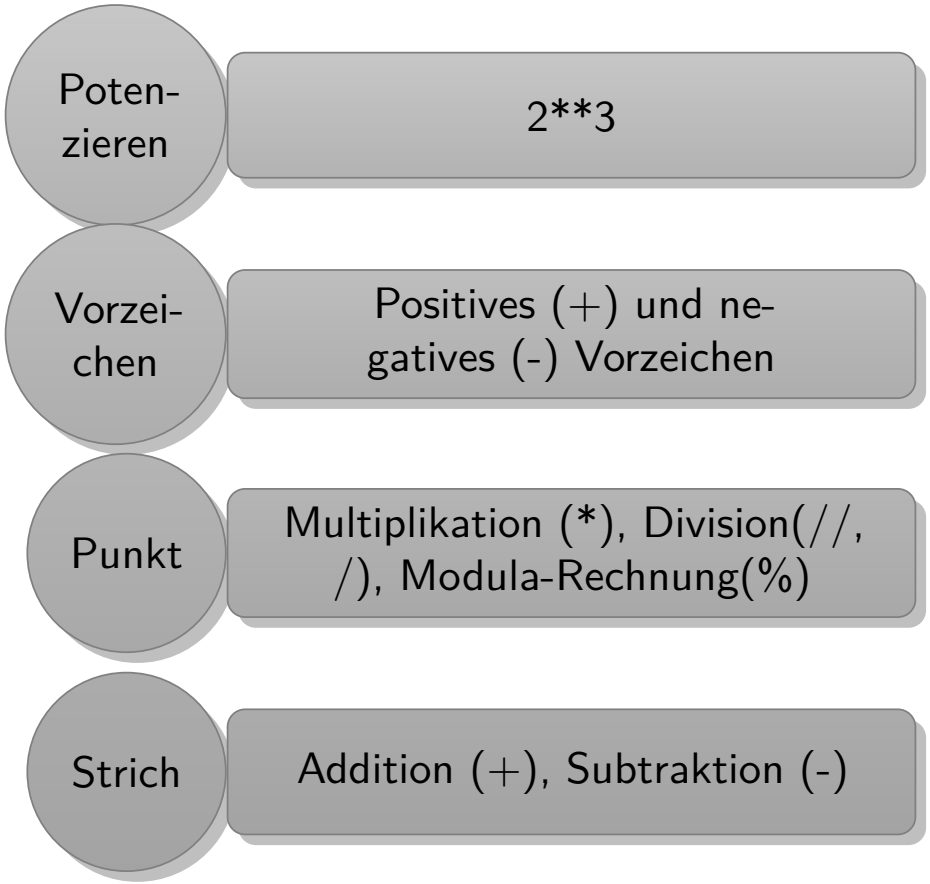
```
1  >>> 5.3 % 2.5
2  0.2999999999999998
3  >>> 5 % 2
4  1
```

- $n = m * a + b, 0 \leq b < |m|$
- 5 % 2 $\leftrightarrow$ 5 = 2 * 2 + 1
- Rückgabe des Rest b

Klammerung von komplexen Ausdrücken

```
1  >>> 3 * 5 + 2
2  17
3  >>> (3 * 5) + 2
4  17
5  >>> 3 * (5 + 2)
6  21
```

Reihenfolge



Beispiele

```
1  >>> (2 ** 3)
2  8
3  >>> (+3.5) ** (-2.7567)
4  0.031634984218318195
5  >>> (300 * 5 / .5 // 4 % 3)
6  0.0
7  >>> (300 + 51245.45 - 1.0)
8  51544.45
```

Übungszettel

Rechenarten

Ordnen Sie der Rechenanweisung die richtige mathematische Operation zu.

0xA + 0x12	=	_____	Division
4 - 8	=	_____	Addition
5 * .5	=	_____	Division mit Rest
7 / 3.4	=	_____	Subtraktion
7 // 3.4	=	_____	Potenzierung
25 % 11	=	_____	Ganzzahl-Division
5.0 ** 2	=	_____	Multiplikation

Übungszettel

Nutzung als Taschenrechner

Schreiben Sie das passende Ergebnis in die Lücken.

8 / (0 * 3)	=	_____
17.0 % 4.3	=	_____
4 - -3 + 2	=	_____
3 + 4 % 5 - 6	=	_____
3 + (4 % 5) - 6	=	_____
5 + 10 * 4 - 2	=	_____
(5 + 10) * (4 - 2)	=	_____

Wiederverwendbarkeit

Lernziel:

- Speicherung von Rechenergebnissen in Variablen
- Speicherung von Code in Dateien
- Ausgabe von Informationen an den Nutzer

Ausdrücke in der Python-Shell

```
1  >>> pi = 3.1415
2  >>> radius = 2
3
4  >>> pi * (radius * radius)
5  12.566
6  >>> ergebnis = pi * (radius * radius)
```

Ausdrücke

- Konstrukt in Abhängigkeit einer bestimmten Semantik
- Rückgabe eines definierten oder undefinierten Wertes

Variablen

- Speicherung des Rückgabewertes eines Ausdruckes
- Speicherung von Literalen, konstanten Werten

Nutzung des Gleichheitszeichens

```
1  >>> pi = 3.1415
2  >>> ergebnis = pi * (2 * 2)
```

- Gleichheitszeichen $\Leftrightarrow$ Zuweisungsoperator.
- Umgangssprachlich: Weise pi den Wert 3.1415 zu. Speichere den Rückgabewert des Ausdrucks $pi * (2 * 2)$ in der Variablen *ergebnis*.
- In Python: Fülle die Speicherstelle an der Position x mit Rückgabewert des Ausdrucks $pi * (2 * 2)$. Klebe auf diesen Behälter das Etikett *ergebnis*.

Literale in Anweisungen

```
1  >>> pi = 3.1415
2  >>> pi * (2 * 2)
3  12.566
```

- Hier in diesem Beispiel: 2, 3.1415
- Konstante Werte, die direkt in den Code geschrieben
- Unveränderliche Werte

Variablen in Anweisungen

```
1  >>> pi = 3.1415
2  >>> ergebnis = pi * (2 * 2)
```

- Hier in diesem Beispiel: *pi*, *ergebnis*
- Label auf einen Behälter
- Platzhalter für einen gespeicherten Wert

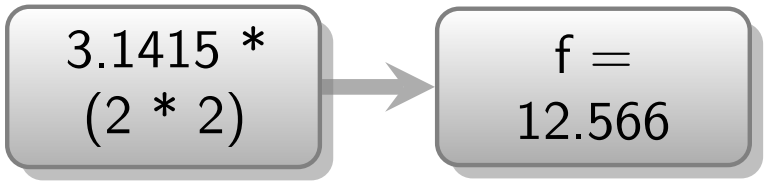
Operatoren in Ausdrücken

```
1  >>> pi = 3.1415
2  >>> pi * (2 * 2)
3  12.566
```

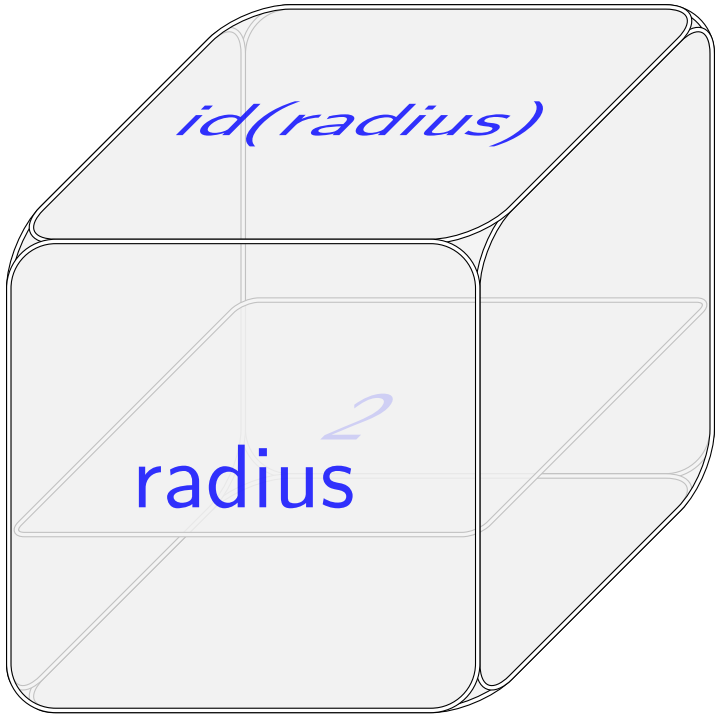
- Hier in diesem Beispiel: *, =
- Zeichen, die zur Verarbeitung von Werten genutzt werden
- Mathematische Operatoren, Zuweisungsoperator et cetera

Auswertung von Ausdrücken

```
1  >>> pi = 3.1415
2  >>> f = pi * (2 * 2)
```



Behälter in Python



Bezeichnung des Behälter

```
1  >>> radius = 2
```

- Frei wählbarer Variablenname
- Platzhalter für einen Wert
- Zugriff auf den Inhalt eines Behälters
- Ein Behälter kann verschiedene Bezeichnungen haben

Regeln

- Beginn mit einem Buchstaben oder Unterstrich.
- Zusammengesetzt aus lateinischen Groß- und Kleinbuchstaben, Ziffern und den Unterstrich.
- Beachtung der Groß- und Kleinschreibung.
- Schlüsselwörter der Programmiersprache und Bezeichnungen für Elemente aus der Standardbibliothek sind nicht erlaubt.

Fehler: Nicht definierte Variable?

```
1  >>> x = 6
2  >>> summe = x * y
3  Traceback (most recent call last):
4    File "<stdin>", line 1, in <module>
5  NameError: name 'y' is not defined
```

Kennnummer des Behälter

```
1  >>> radius = 2
2  >>> id(radius)
3  1274143074640
4  >>> x = 2
5  >>> id(x)
6  1274143074640
7  >>> y = 3
8  >>> id(y)
9  1274143074672
```

Erläuterung

```
1  >>> radius = 2
2  >>> id(radius)
3  1274143074640
4  >>> id(2)
5  1274143074640
```

- Ermittlung des Strichcodes eines Behälters
- Eindeutige Kennung eines Behälters
- Ermittlung der ID eines Literals: (`id(wert)`)
- Ermittlung der ID einer Variablen: (`id(variable)`)

Inhalt des Behälter

```
1  >>> radius = 2
```

- Pro Behälter einen Wert.
- Jeder genutzte Wert, egal welchen Typs, wird nur einmal im Speicher abgelegt.

Kategorisierung des Inhaltes

```
type(variable)
```

- In Abhängigkeit des Datentyps
- Jeder Wert ist ein Objekt mit speziellen Attributen
- Zahlen, Zeichenketten, Boolean et cetera

Zahlen-Kategorien

```
1  >>> radius = 2
2  >>> type(radius)
3  <class 'int'>
4  >>> type(3.4)
5  <class 'float'>
6  >>> type(1.2 + 3j)
7  <class 'complex'>
```

IDs von Gleitkommazahlen

```
1  >>> a = 3.4567
2  >>> id(a)
3  1274182599312
4  >>> id(3.4567)
5  1274181713104
```

- Welche angenährte Darstellung einer reellen Zahl dort gespeichert ist, weiss der Entwickler nicht.
- Siehe https://py-tutorial-de.readthedocs.io/de/python-3.3/floatingpoint.html

Boolsche Werte

```
1  >>> type(True)
2  <class 'bool'>
3  >>> type(False)
4  <class 'bool'>
```

Leerer Behälter

```
1  >>> box = None
2  >>> type(box)
3  <class 'NoneType'>
```

Ausgaben in der Shell

```
1  >>> pi = 3.1415
2  >>> radius = 2
3  >>> flaeche = pi * (radius * radius)
4  >>> print(flaeche)
5  12.566
```

Funktion „print()“

None = print(parameter)

- Ausgabe eines definierten Parameters in der Shell
- Aufruf mit dem Funktionsnamen „print“
- Beachtung der Groß- und Kleinschreibung beim Aufruf

Parameter der Funktion

None = print(parameter)

- Parameterliste direkt nach dem Funktionsnamen
- Beginn und Ende mit den runden Klammern
- Bei dieser Funktion: Was soll ausgegeben werden?

Rückgabewert der Funktion

None = print(parameter)

- Jede Funktion gibt einen Wert in Python zurück.
- Der Rückgabewert kann, muss aber nicht in einer Variablen gespeichert werden.
- Bei dieser Funktion: None. Kein definierter Wert.

Ausdruck von Zeichenketten

```
1  >>> print("Name des Kunden: 'Hans Mustermann'")
2  Name des Kunden: 'Hans Mustermann'
3  >>> print('Name des Kunden: "Elfriede Musterfrau"')
4  Name des Kunden: "Elfriede Musterfrau"
```

- In der Parameterliste: Beliebig viele Parameter
- Trennzeichen zwischen den Parametern in der Liste: Komma
- Trennzeichen zwischen den Parametern beim Ausdruck: Leerzeichen

Ausdruck von x Parametern

```
1  >>> menge = 5
2  >>> print('Bestellung von ', menge, " kg Bananen")
3  Bestellung von  5  kg Bananen
```

- Runde Klammern. Beginn und Ende der Parameterliste
- Komma. Trennung der Parameter in der Liste
- Verbindung mit einem Leerzeichen beim Ausdruck

Erstellung von Codedateien

- Öffnen der Python-Shell IDLE
- Menü File - New File
- Eingabe von Code in das Codefenster.

Hinweise

- Pro Zeile wird eine Anweisung geschrieben.
- Python kennt kein spezielles Zeilenende-Zeichen.

Speicherung des Codes

- File - Save As erzeugt eine Datei mit der „.py“.
- File - Save bei Änderung der Code-Datei.

Durchlaufen des Codes

- Run - Run Module
- Interpretation der Codezeilen in einer Datei von oben nach unten

Öffnen und Schließen Code-Datei

- File - Open
- File - Close

Einzeilige Kommentare

```
# Dies ist ein Kommentar
```

- Hilfe für den Entwickler
- Erläuterung zum folgenden Codeabschnitt

Übungszettel

Anweisungen in Python

Ordnen Sie den Ausdrücken den entsprechenden Python-Code zu.

celsius + 273 = kelvin	⇔	u = a + b + c
Zweierpotenz einer Zahl x	⇔	preisProKilo = 1.99
Addition der drei Seiten eines Dreiecks	⇔	k = temperatur + 273
Preis pro Kilo Bananen: 1,99 Euro	⇔	ergebnis = x ** 2

Übungszettel

Speicherung von Code in einer Datei

Schreiben Sie den passenden Code zu den Aussagen in eine Python-Datei.

Anlage der Variablen a und b und Zuweisung eines beliebigen Wertes.
Berechnen von a^2
Berechnen von b^2
Berechnen von $2 * (a + b)$
Addition der Ergebnisse der drei Ausdrücke a^2, $2 * (a + b)$ und b^2
Ausgabe des Ergebnisses der Addition

Code unter bestimmten Bedingungen ausführen

Lernziel:

- Schreiben von Bedingungen in Python
- Abbildung von Fragen, die mit Ja oder Nein beantwortet werden können
- Schreiben und verknüpfen von relationalen Ausdrücken

Vergleichsoperatoren

```
1  >>> 3 == 2          # Ist 3 gleich 2
2  False
3  >>> 3 != 2          # Ist 3 ungleich 2
4  True
5  >>> 3 < 2          # Ist 3 kleiner als 2
6  False
7  >>> 3 <= 2          # Ist 3 kleiner gleich als 2
8  False
9  >>> 3 > 2          # Ist 3 größer als 2
10 True
11 >>> 3 >= 2          # Ist 3 größer gleich als 2
12 True
```

Kombination von Operatoren

```
1  >>> zahl = 3
2  >>> 0 < zahl < 10
3  True
4
5  >>> (zahl * 10) > 50
6  False
```

Hinweise zu „ist gleich“

```
1  >>> # Vergleich von Ganzzahlen
2  >>> 3 == 2
3  False
4
5  >>> # Sind die beiden Gleitkommazahlen gleich
6  >>> 3.123456789123456789123456789 == 3.1234567891234567
7  True
```

Nutzung des Zuweisungsoperators

```
>>> 3.4 = 4
  File "<stdin>", line 1
    3.4 = 4
    ^
SyntaxError: cannot assign to literal
>>> (3.4 = 4)
  File "<stdin>", line 1
    (3.4 = 4)
         ^
SyntaxError: invalid syntax
```

Ist der Behälter leer / nicht leer?

```
1  >>> x = 5
2
3  >>> x == None
4  False
5  >>> x != None
6  True
```

Nutzung in Vergleichsanweisungen

```
1  >>> x = 5
2
3  >>> x > None
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  TypeError: '>' not supported between instances of 'int' and
   ↪  'NoneType'
```

Logische Operatoren

```
1  >>> monat = 5
2
3  >>> not(monat == None)
4  True
5  >>> (monat != None) and (monat != 0)
6  True
7  >>> (monat == 2) or (monat == 3)
8  False
```

Nicht-Operator

```
1  >>> monat = 5
2
3  >>> (not(monat == None))
4  True
```

- Negation eines Ausdruckes
- Ein falscher Ausdruck wird wahr und umgekehrt

Und-Operator

```
1  >>> monat = 5
2
3  >>> (monat != None) and (monat != 0)
4  True
```

- Beide Ausdrücke müssen wahr sein.
- Falls der linke Ausdruck falsch ist, wird der rechte Ausdruck nicht mehr ausgewertet.

. . . zwischen einer oberen und unteren Grenze

```
>>> zahl = 3

>>> # Die Variable zahl ist
>>> # größerer gleich als 0
>>> # und kleiner gleich 10.
>>> (zahl >= 0) and (zahl <= 10)
True

>>> # 0 ist  ist größerer gleich als zahl
>>> # zahl ist kleiner gleich 10.
>>> 0 >= zahl <= 10
False
```

Oder-Operator

```
1  >>> monat = 5
2
3  >>> (monat == 2) or (monat == 3)
4  False
```

- Einer der beiden Ausdrücke muß wahr sein.
- Falls der linke Ausdruck wahr ist, wird der rechte Ausdruck nicht mehr ausgewertet.
- Prüfung einer Liste von Werten.

Wenn-Anweisung

```
1  >>> temperatur = 0.6
2
3  >>> if (temperatur > 0.0):
4  ...         print('Plus-Temperatur')
5  ...
6  Plus-Temperatur
```

Aufbau des Anweisungskopfes

if ([bedingung]):

- Beginn: Schlüsselwort `if`.
- Dem Schlüsselwort folgt die Bedingung `bedingung` zum Starten des Anweisungsrumpfes.
- Ende: Doppelpunkt.

Bedingung im Anweisungskopf

```
1  >>> temperatur = 0.6
2
3  >>> (temperatur > 0.0)
4  True
```

- Vergleichsausdruck, der verknüpft werden kann
- Rückgabe eines boolschen Wertes (True oder False).
- Kann, muss aber nicht geklammert werden.

Anweisungsrumpf

```
1  >>> if (temperatur > 0.0):
2  ...         print('Plus-Temperatur')
3  ...
4  Plus-Temperatur
```

- Eingerückte Zeilen unterhalb des Kopfes.
- Zeilen, die mit vier Leerzeichen im Anschluss an den Kopf beginnen.

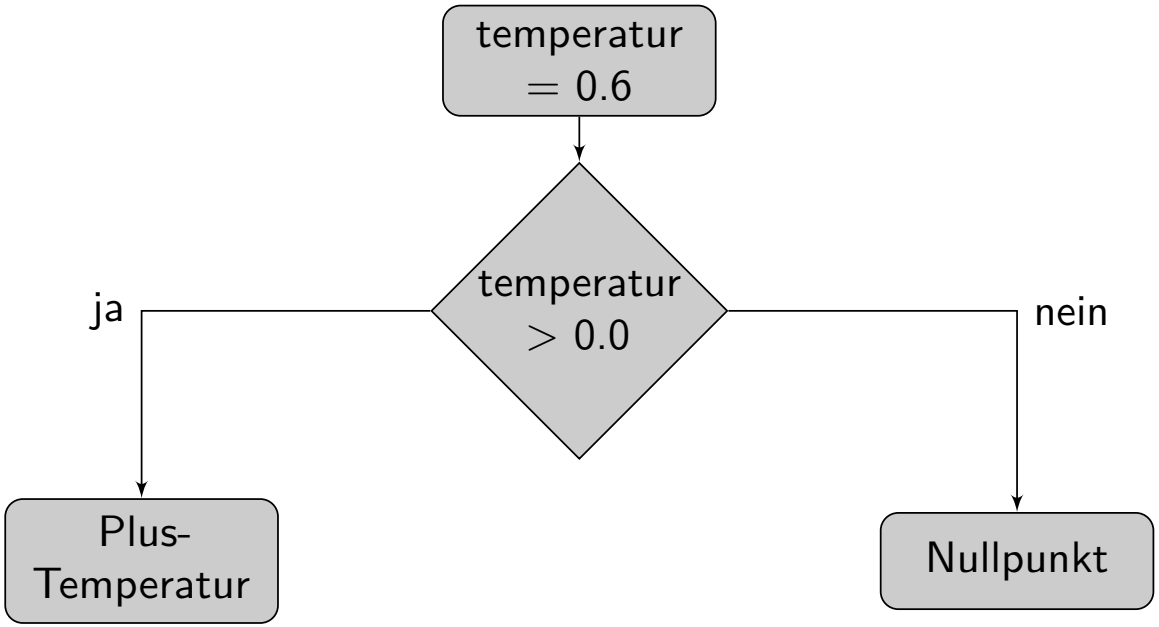
Fehlender Doppelpunkt im Kopf

```
1  >>> if (temperatur > 0.0)
2    File "<stdin>", line 1
3      if (temperatur > 0.0)
4                           ^
5  SyntaxError: invalid syntax
6  >>>         print('Plus-Temperatur')
7    File "<stdin>", line 1
8              print('Plus-Temperatur')
9      ^
10 IndentationError: unexpected indent
```

Wenn-Dann-Anweisung

```
1  >>> if (temperatur > 0.0):
2  ...         print('Plus-Temperatur')
3  ... else:
4  ...         print('Temperatur kleiner gleich Null')
5  ...
6  Plus-Temperatur
```

Wenn-Dann-Anweisung



Hinweise

- Optionale Anweisung: `else`
- Standardfall
- Der, am häufigsten vorkommende Fall

Codeblock in Python

```
1  >>> r = 4.5
2  >>> if (radius > 0.0):
3  ...         print('Fläche: ', (3.1415 * (r * r)))
4  ...         print('Umfang: ', (2 * 3.1415 * r))
5  ... else:
6  ...         print("Bitte geben Sie ein Radius an")
7  ...
8  Fläche:  63.615375
9  Umfang:  28.273500000000002
```

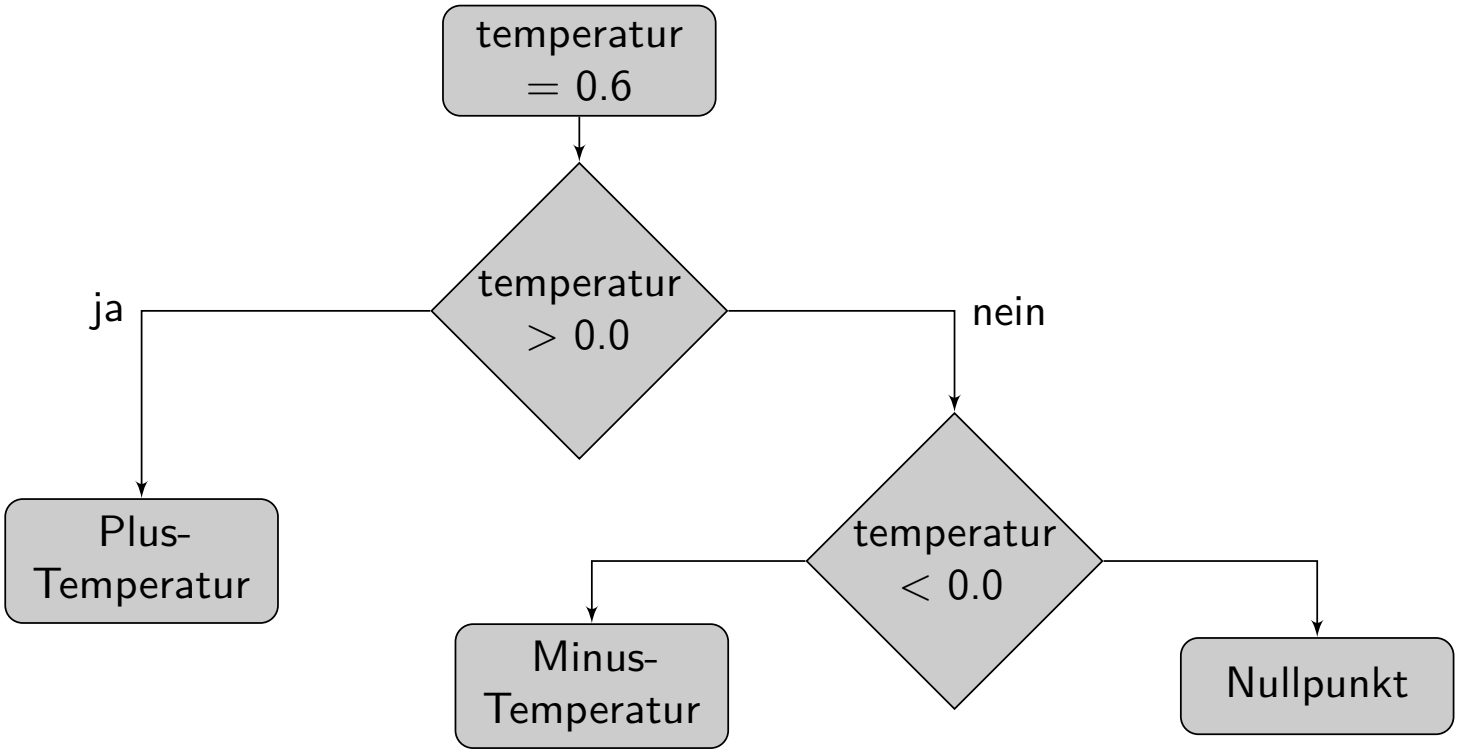
Erläuterung

- Besteht aus mindestens einer Anweisung.
- Zusammenfassung von Zeilen durch Einrückung.
- Jede Einrückung besteht standardmäßig aus vier Leerzeichen.
  Siehe: Options - Configure IDLE - Fonts/Tabs.

Fallunterscheidung

```
1  >>> if (temperatur > 0.0):
2  ...         print('Plus-Temperatur')
3  ... elif (temperatur < 0.0):
4  ...         print('Minus-Temperatur')
5  ... else:
6  ...         print('Nullpunkt')
7  ...
8  Plus-Temperatur
```

Grafische Darstellung



Hinweise

- Optionale Anweisung: `elif`.
- Beschreibung eines möglichen Falls in Bezug zu der Aufgabe
- Sobald eine Bedingung zutrifft, werden die anderen Fälle nicht mehr betrachtet.

Möglicher Fehler

```
>>> t = 37.49
>>> if (t >= 36.9) and (t <= 37.4):
...         print('Erhöhte Temperatur')
... elif (t >= 37.5) and (t <= 39.4):
...         print('Fieber')
... elif (t >= 39.5) and (t <= 40.5):
...         print('Hohes Fieber')
... else:
...         print('Normaltemperatur')
...
Normaltemperatur
```

Schachtelung von Anweisungen

```
>>> if temperatur != None:
...         if (temperatur > 0.0):
...                 print('Plus-Temperatur')
...         elif (temperatur < 0.0):
...                 print('Minus-Temperatur')
...         else:
...                 print('Nullpunkt')
... else:
...         print("Keine Temperaturangaben")
...
Plus-Temperatur
```

Übungszettel

Bedingte Anweisung

Füllen Sie die Lücken mit dem passenden Code aus.

```
breite = 4
[          ] = 6

if (breite [          ] hoehe):
        print("Quadrat")

[          ]
        print("Rechteck")
```

Übungszettel

Bedingte Anweisung

Ordnen Sie die Bestandteile der bedingten Anweisung so, dass sie der Syntax von Python entsprechen.

rabatt = 0.0	
rabatt = 0.05	
elif(bestellmenge >= 50)	
and (bestellmenge <= 99)	
if (bestellmenge > 100):	
rabatt = 0.1	
else:	

Programmierübung - Beschreibung

```
"""
Eine Versuchsreihe besteht aus Werte zwischen 1 und 10.
Mit Hilfe des folgenden Codes sollen Messwerte aus der
↪ Versuchsreihe getestet werden.
Überprüfen Sie den Code, in dem Sie jeweils vor einer der
↪ Zuweisungsanweisungen das Hash-Zeichen entfernen. Die
↪ jeweils drei anderen Zuweisungen werden mit Hilfe des
↪ Hash-Zeichens als Kommentar gekennzeichnet und nicht
↪ durchlaufen.
Falls die Überprüfung der Messwerte nicht das gewünschte Ergebnis
↪ ausgibt, passen Sie den Code an.

"""
```

Programmierübung - Code

```
1  # messwert = -0.5
2  # messwert = 0
3  messwert = 5.3
4  # messwert = 10.003
5
6  if messwert < 0:
7      print("Negative Messung")
8  elif (messwert >= 1) and (messwert <= 10):
9      print("Erlaubte Messwerte")
10 else:
11     print("Zu grosse Messwerte")
```

Programmieraufgabe

Eine quadratische Gleichung in der Form

$$ax^2 + bx + c = 0 \; mit \; a \neq 0$$

kann durch folgenden Term gelöst werden:

$$x_{1,2} = -b \pm \frac{\sqrt{b^2 - 4ac}}{2a}$$

Die Gleichung hat

- zwei Lösungen, wenn der Ausdruck unter der Wurzel positiv ist.
- eine Lösung, wenn der Ausdruck unter der Wurzel gleich Null ist.
- keine Lösung, wenn der Ausdruck unter der Wurzel negativ ist.

Lösung - Deklaration der Variablen

```
1  from math import sqrt
2
3  a = 2
4  b = 6
5  c = 1
6
7  ausdruck = b * b - 4 * a * c
```

Lösung - Bedingung

```
1  if ausdruck > 0:
2     x1 = (-b - sqrt(ausdruck))/(2*a)
3     x2 = (-b + sqrt(ausdruck))/(2*a)
4     print ("Die Gleichung hat die beiden Lösungen:")
5     print ('x1 =',x1,'; x2 =',x2)
6  elif ausdruck == 0:
7     x1 = (-b) / (2 * a)
8     print ("Die Gleichung hat die Lösung:")
9     print ('x =',x1)
10
11 else:
12    print ("Die Gleichung hat keine Lösung!")
```

Wiederholung von Code

Lernziel:

- Schreiben von Bedingungen
- Schreiben von Schleifen / while-Anweisungen

Vergleichsoperatoren

```
1  >>> 3 == 2          # Ist 3 gleich 2
2  False
3  >>> 3 != 2          # Ist 3 ungleich 2
4  True
5  >>> 3 < 2          # Ist 3 kleiner als 2
6  False
7  >>> 3 <= 2          # Ist 3 kleiner gleich als 2
8  False
9  >>> 3 > 2          # Ist 3 größer als 2
10 True
11 >>> 3 >= 2          # Ist 3 größer gleich als 2
12 True
```

Hinweise zu „ist gleich“

```
1  >>> # Vergleich von Ganzzahlen
2  >>> 3 == 2
3  False
4
5  >>> # Sind die beiden Gleitkommazahlen gleich
6  >>> 3.123456789123456789123456789 == 3.1234567891234567
7  True
```

Ist der Behälter leer / nicht leer?

```
1  >>> x = 5
2
3  >>> x == None
4  False
5  >>> x != None
6  True
```

Logische Operatoren

```
1  >>> monat = 5
2
3  >>> not(monat == None)
4  True
5  >>> (monat != None) and (monat != 0)
6  True
7  >>> (monat == 2) or (monat == 3)
8  False
```

Nicht-Operator

```
1  >>> monat = 5
2
3  >>> (not(monat == None))
4  True
```

- Negation eines Ausdruckes
- Ein falscher Ausdruck wird wahr und umgekehrt

Und-Operator

```
1  >>> monat = 5
2
3  >>> (monat != None) and (monat != 0)
4  True
```

- Beide Ausdrücke müssen wahr sein.
- Falls der linke Ausdruck falsch ist, wird der rechte Ausdruck nicht mehr ausgewertet.

. . . zwischen einer oberen und unteren Grenze

```
>>> zahl = 3

>>> # Die Variable zahl ist
>>> # größerer gleich als 0
>>> # und kleiner gleich 10.
>>> (zahl >= 0) and (zahl <= 10)
True

>>> # 0 ist  ist größerer gleich als zahl
>>> # zahl ist kleiner gleich 10.
>>> 0 >= zahl <= 10
False
```

Oder-Operator

```
1  >>> monat = 5
2
3  >>> (monat == 2) or (monat == 3)
4  False
```

- Einer der beiden Ausdrücke muß wahr sein.
- Falls der linke Ausdruck wahr ist, wird der rechte Ausdruck nicht mehr ausgewertet.
- Prüfung einer Liste von Werten.

Wiederholung, while-Anweisung

```
>>> index = 1
>>> summe = 0
>>> while (index >= 1) and (index <= 5):
...         summe = summe + index
...         print(index, ": ", summe)
...         index = index + 1
...
1 :  1
2 :  3
3 :  6
4 :  10
5 :  15
```

Aufbau des Schleifenkopfes

while ([bedingung]):

- Beginn: Schlüsselwort `while`.
- Dem Schlüsselwort folgt die Bedingung `bedingung` zum Starten des Anweisungsrumpfes.
- Ende: Doppelpunkt.

Bedingung im Schleifenkopf

```
1  >>> index = 1
2  >>> (index >= 1) and (index <= 5)
3  True
4  >>> ((index >= 1) and (index <= 5))
5  True
```

- Nutzung von Vergleichsausdrücken.
- Rückgabe eines boolschen Wertes (True oder False).
- Kann, muss aber nicht geklammert werden.

Fehlender Doppelpunkt im Kopf

```
>>> index = 1
>>> while (index >= 1) and (index <= 5)
  File "<stdin>", line 1
    while (index >= 1) and (index <= 5)
                                       ^
SyntaxError: invalid syntax
>>>         index = index + 1
  File "<stdin>", line 1
            index = index + 1
    ^
IndentationError: unexpected indent
```

Schleifenrumpf

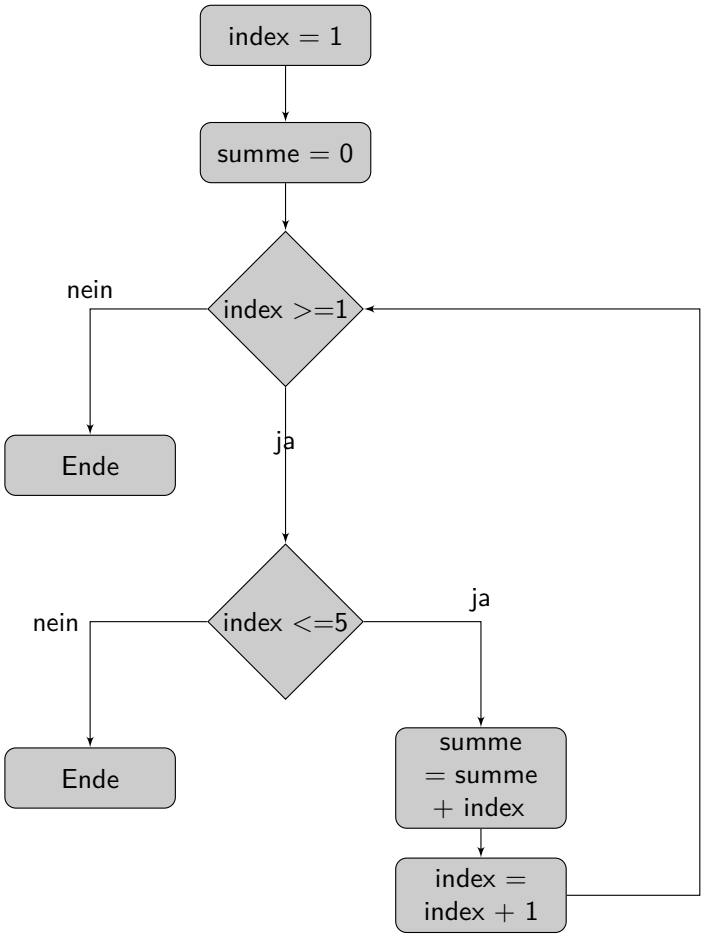
```
1  index = 1
2  while (index >= 1) and (index <= 5):
3          index = index + 1
```

- Eingerückte Zeilen unterhalb des Kopfes.
- Zeilen, die mit vier Leerzeichen im Anschluss an den Kopf beginnen.

Falsch eingerückte Zeilen

```
1  >>> index = 1
2  >>> while (index >= 1) and (index <= 5):
3  ... index = index + 1
4    File "<stdin>", line 2
5      index = index + 1
6      ^
7  IndentationError: expected an indented block
```

Funktionsweise



else-Anweisung einer Schleife

```
1  index = 1
2  summe = 0
3  while (index >= 1) and (index <= 5):
4          summe = summe + index
5          index = index + 1
6
7  else:
8          print("Gesamtsumme: ", summe)
```

Hinweis: Die else-Anweisung wird nur ausgeführt, wenn die Schleife vollständig ohne Fehler durchlaufen wird.

Endlosschleife

```
index = 1

while True:
        index = index + 1
        print(index)
```

- Abbruch über die Shell: Shell - Interrupt Execution

Abbruch in der Schleife

```
1  index = 1
2
3  while True:
4          print(index)
5          index = index + 1
6
7          if not((index >= 1) and (index <= 5)):
8                  break
```

Schachtelung von Schleifen

```
1  zeile = 1
2
3  while (zeile >= 1) and (zeile <= 5):
4          spalte = 1
5          while (spalte >= 1) and (spalte <= 5):
6                  ergebnis = spalte * zeile
7                  spalte = spalte + 1
8
9          zeile = zeile + 1
```

Programmierübung Nr. 1

while-Schleife

Die Schleife soll von 0 bis einem Wert x einen Zähler um eins addieren. Führt die Schleife die Aufgabe korrekt aus?

```
zaehler = 0

while (zaehler >= 0):
       print(zaehler )
       zaehler = zaehler + 1
```

Programmierübung Nr. 2

while-Schleife

Die Schleife soll von 5 bis einschließlich 0 den Zähler um eins subtrahieren. Führt die Schleife die Aufgabe korrekt aus?

```
zaehler = 5

while (zaehler >= 0):
       print(zaehler)
       zaehler = zaehler - 1
```

Programmierübung Nr. 3

while-Schleife

Die Schleife soll von 0 bis einem Wert x einen Zähler um eins addieren. Wie oft wird die Schleife durchlaufen?

```
zaehler = 0

while (zaehler <= 0):
       print(zaehler)
       zaehler = zaehler + 1
```

Übungszettel

Ordnen von Code in die richtige Reihenfolge

```
while (xshift <= 10):
xshift = xshift + 1
x = 0
xshift = 1
y = y + yshift
if ((x % 2) == 0):
y = 0
x = x + xshift
yshift = 2
```

Sequenz „Liste“

Lernziel:

- Abbildung von Aufzählungen, Messwerten und so weiter
- Hinzufügen und Löschen von Elementen in einer Sequenz
- Zugriff auf Elemente in einer Sequenz
- Ausschneiden von Teilfolgen aus einer Reihe

. . . in Python

- Aufzählungen, Reihen
- Ablage von Zahlen, Zeichen und boolschen Werten, die einen Bezug zueinander haben

Sequenz-Kategorien

- Listen. Veränderbare Folge von Elementen beliebigen Datentyps.
- Tupel. Nicht veränderbare Folge von Elementen beliebigen Datentyps.
- Range-Objekt. Nicht veränderbare Folge von Zahlen.
- Byte-Sequenz. Nicht oder schreibgeschützte Folge von Ganzzahlen zwischen 0 und 255.
- String. Alphanumerische und numerische Zeichen. Wörter. Sätze.

Eigenschaften

- Zugriff auf die Elemente über ein Index
- In Abhängigkeit der gewählten Kategorie veränderbar oder nicht

Erzeugung von Listen

```
1  >>> xyz = [0, 1.5, -0.5]
2  >>> muster = [True, False, True]
3  >>> messung = [2.3, None, 3.4, None, None, 5.6, 4.5]
```

- Beginn und Ende der Liste: Eckige Klammern.
- Trennzeichen zwischen den Sequenz-Elementen: Komma

Leere neue Listen

```
1  >>> leereListe = []
2  >>> newList = list()
```

- Beide Möglichkeiten können gleichwertig genutzt werden
- Die zweite Variante wird häufig bei der Konvertierung von Sequenzen in Listen genutzt.

Erzeugung eines Objekts „Liste“

```
1  >>> newList = list()
```

- Mit Hilfe einer speziellen Methode (dem Konstruktur) wird ein Objekt vom Typ „Liste“ erzeugt.
- Objekte beschreiben ein bestimmtes Exemplar einer Klasse.

Klasse „Liste“

```
1  >>> newList = list()
2
3  >>> type(newList)
4  <class 'list'>
```

- Allgemeine Definition von Attributen und des Verhalten eines Objekts oder Datentyps
- Zum Beispiel können Autos, Einhörner, Listen und was immer du sonst noch möchtest mit einer Klasse beschrieben werden

Konstruktionsmethode „Liste“

```
1  >>> newList = list()
```

- Spezielle Funktion in jeder Klasse zur Erzeugung eines Objekts.
- Klasse und Konstruktor haben den gleichen Namen. Groß- und Kleinschreibung wird beachtet.

Aufruf der Konstruktionsmethode

```
1  >>> newList = list()
```

- Mit dem Namen der Klasse.
- Dem Namen der Klasse folgt direkt im Anschluss die Parameterliste.
- Die Parameterliste beginnt und endet mit den runden Klammern. Hier ist sie leer. Die Liste hat keine Elemente.

Anzahl der Elemente

```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A']
2  >>> len(zitterrochen)
3  5
```

Leere Liste?

```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A']
2
3  >>> if not (zitterrochen):
4  ...         print("Die Liste ist leer.")
5  ...
```

Nicht leere Liste?

```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A', 'A', 'G']
2
3  >>> if (zitterrochen):
4  ...         print("Die Liste ist nicht leer.")
5  ...
6  Die Liste ist nicht leer.
```

Ist Element . . . enthalten?

```
1  >>> messreihe =
   ↪  [10855.16,10857.42,10743,10743,34183100,10743.009]
2  >>> messung = 10743
3
4  >>> if (messung in messreihe):
5  ...         print(messung, " ist enthalten.")
6  ...
7  10743  ist enthalten.
```

Ist Element . . . nicht enthalten?

```
1  >>> messreihe =
   ↪  [10855.16,10857.42,10743,10743,34183100,10743.009]
2  >>> messung = 10743
3
4  >>> if (messung not in messreihe):
5  ...         print(messung, " ist nicht enthalten.")
6  ...
```

Häufigkeit eines Elements

```
1  >>> messreihe =
   ↪  [10855.16,10857.42,10743,10743,34183100,10743.009]
2  >>> messung = 10743
3
4  >>> messreihe.count(messung)
5  2
```

Methode eines Objekts

```
1  >>> messreihe =
   ↪  [10855.16,10857.42,10743,10743,34183100,10743.009]
2  >>> messung = 10743
3
4  >>> messreihe.count(messung)
5  2
```

- Funktionen, die in der Klasse definiert sind, auf der das Objekt basiert
- Beschreiben das Verhalten von Objekten
- Verändern den Status eines Objekts
- Methoden einer Liste:
  https://www.w3schools.com/python/python_ref_list.asp oder
  https://docs.python.org/3/tutorial/datastructures.html

Aufruf der Methode

```
objekt.function(par01, par02)
```

- Verbindung von Objekt und Methode durch den Punkt
- Anwendung der Methode (rechts vom Punkt) auf das Objekt (links vom Punkt)

Name der Methode

```
objekt.function(par01, par02)
```

- Eindeutig in der Vorlage.
- Beachtung von Groß- und Kleinschreibung.

Parameterliste der Methode

```
objekt.function(par01, par02, . . . )
```

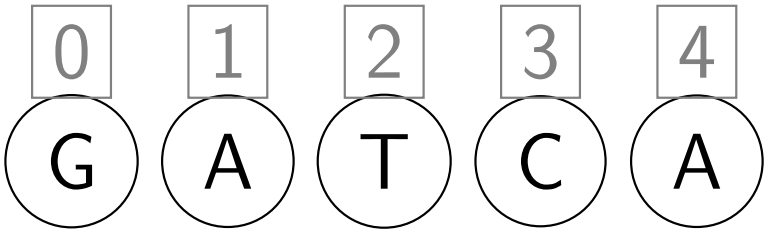
- Beginn und Ende der Parameterliste: Runde Klammern.
- Trennzeichen zwischen den Parametern: Komma.
- Anzahl der Elemente in der Liste in Abhängigkeit der Definition. Kann leer sein.

Rückgabewert der Methode

variable = objekt.function(par01, par02, . . .)

- Rückmeldung an den Aufrufer
- `None` oder ein Wert in Bezug auf das beschriebene Verhalten

Schreiben und lesen eines Elements



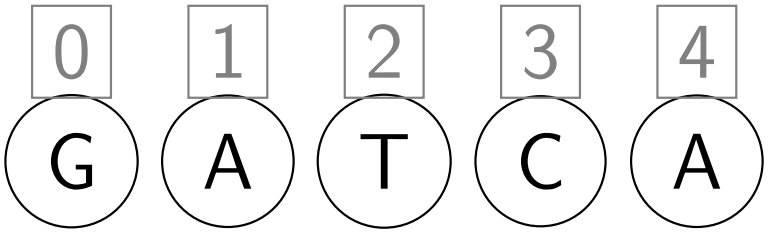

```
>>> zitterrochen = ['G', 'A', 'T', 'C', 'A']

>>> element = zitterrochen[0]
>>> print(element)
G
>>> zitterrochen[3] = element
>>> print(zitterrochen)
['G', 'A', 'T', 'G', 'A']
```

Index des Elements

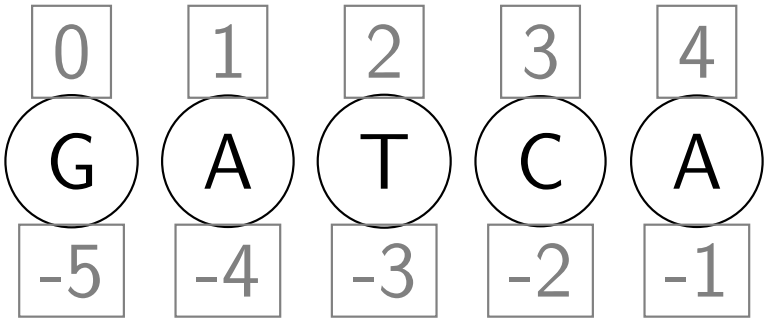
- Eindeutige Position eines Elements in einer Sequenz
- Positive oder negative Ganzzahl
- Folgt direkt dem Sequenznamen in eckigen Klammern.

Nutzung eines positiven Index




```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A']
2
3  >>> zitterrochen[0]
4  'G'
5  >>> zitterrochen[len(zitterrochen) - 1]
6  'A'
```

Nutzung eines negativen Index




```
>>> zitterrochen = ['G', 'A', 'T', 'C', 'A']

>>> zitterrochen[-1]
'A'
>>> zitterrochen[0 - len(zitterrochen)]
'G'
```

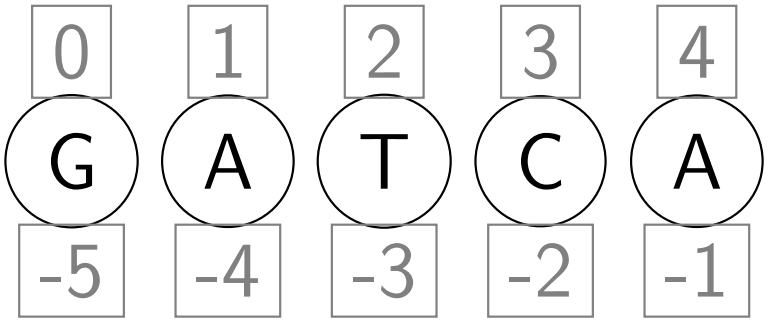
Nutzung in einer Schleife

```
1  zitterrochen = ['G', 'A', 'T', 'C', 'A']
2  index = 0
3  pos = list()
4
5  while index < len(zitterrochen):
6      if zitterrochen[index] == 'A':
7          pos.append(index)
8
9      index = index + 1
10
11 else:
12     print("Häufigkeit von A: ", zitterrochen.count('A'))
13     print("An den Positionen: ", pos)
```

Zu großer Index

```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A']
2  >>> element = 'T'
3  >>> zitterrochen[5] = element
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  IndexError: list assignment index out of range
```

Slicing - Ausschneiden

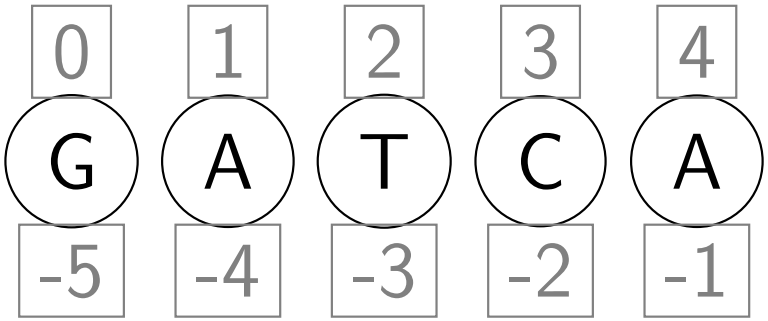



```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A']
2
3  >>> zitterrochen[-4:-2]
4  ['A', 'T']
5  >>> zitterrochen[1:3]
6  ['A', 'T']
```

Hinweise

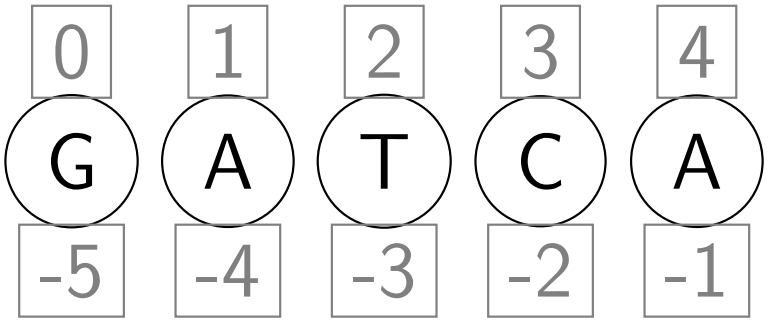
- Startwert. Beliebige positive oder negative Ganzzahl, beginnend beim ersten Element.
- Stopp-Wert. Definition des ersten Elements, welches nicht mehr ausgeschnitten wird. Liegt rechts vom Startwert.
- Start- und Stopp-Werte sind optional.

Keine Angabe zum Startwert



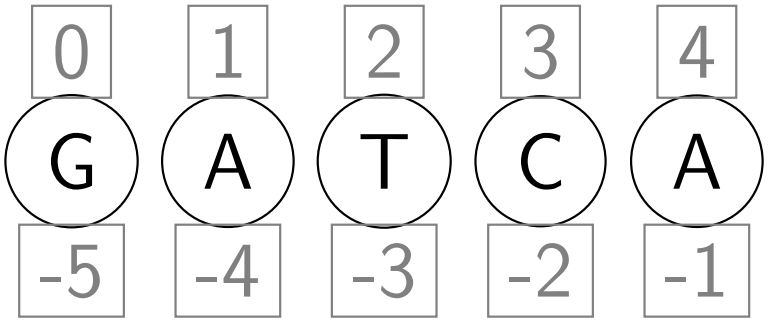

```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A']
2
3  >>> zitterrochen[:-2]
4  ['G', 'A', 'T']
5  >>> zitterrochen[:3]
6  ['G', 'A', 'T']
```

Keine Angabe zum Stopp-Wert



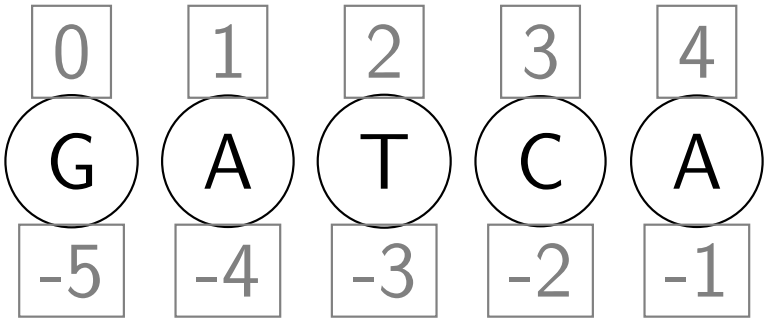

```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A']
2
3  >>> zitterrochen[-4:]
4  ['A', 'T', 'C', 'A']
5  >>> zitterrochen[1:]
6  ['A', 'T', 'C', 'A']
```

Keinerlei Angaben




```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A']
2
3  >>> zitterrochen[:]
4  ['G', 'A', 'T', 'C', 'A']
```

Angabe einer Schrittweite




```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A']
2
3  >>> zitterrochen[::2]
4  ['G', 'T', 'A']
```

Hinweise

- Startwert. Beliebige positive oder negative Ganzzahl, beginnend beim ersten Element.
- Stopp-Wert. Definition des ersten Elements, welches nicht mehr ausgeschnitten wird. Liegt rechts vom Startwert.
- Schrittweite. Standardmäßig wird eine Schrittweite von 1 angenommen. In dem vorherigen Beispiel wird jedes zweite Element in dem Teilbereich angezeigt.
- Alle drei Angaben sind optional.

Ersetzung eines zusammenhängenden Bereichs

```
>>> zitterrochen = ['G', 'A', 'T', 'C', 'A']

>>> zitterrochen[-3:] = ['a', 'T', 'G']
>>> zitterrochen
['G', 'A', 'a', 'T', 'G']
>>> zitterrochen[1:3] = ['t', 'T', 'G']
>>> zitterrochen
['G', 't', 'T', 'G', 'T', 'G']
```

Hinweis: Die Zuweisung eines Literals an ein Teilbereich erzeugt ein Fehler.

Konkatenation von Zeichen

```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A', 'G', 'A',
   ↪  'T', 'C', 'A']
2
3  >>> zitterrochen[1] + zitterrochen[3]
4  'AC'
5  >>> zitterrochen[1:3] + zitterrochen[-3:-1]
6  ['A', 'T', 'T', 'C']
```

- Verknüpfen von Zeichen → String
- Verknüpfen einer Liste von Zeichen → Sequenz

Konkatenation von Zahlen

```
1  >>> messreihe = [0.5, 0.8, 0.5, 0.6, 0.9, 0.5, 0.7, 0.3]
2
3  >>> messreihe[1] + messreihe[3]
4  1.4
5  >>> messreihe[1:3] + messreihe[-3:-1]
6  [0.8, 0.5, 0.5, 0.7]
```

- Verknüpfen von Zahlen → Ergebnis der Addition
- Verknüpfen von Zahlen und Zeichen → Fehler
- Verknüpfen einer Liste von Zahlen → Sequenz

Wiederholung von Zeichen

```
1  >>> zitterrochen = ['G', 'A', 'T', 'C', 'A', 'G', 'A',
   ↪  'T', 'C', 'A']
2
3  >>> zitterrochen[1] * 3
4  'AAA'
5  >>> zitterrochen[1:3] * 2
6  ['A', 'T', 'A', 'T']
```

Hinweis: Die Multiplikation von Teilbereichen erzeugt ein Fehler.

Wiederholung von Zahlen

```
1  >>> messreihe = [0.5, 0.8, 0.5, 0.6, 0.9, 0.5, 0.7, 0.3]
2
3  >>> messreihe[1] * 3
4  2.4000000000000004
5  >>> messreihe[1:3] * 2
6  [0.8, 0.5, 0.8, 0.5]
```

Anhängen von Elementen

```
1  >>> messreihe = [10855.16,10857.42,10743]
2  >>> messung = 10743
3
4  >>> messreihe.append(messung)
5  >>> print(messreihe)
6  [10855.16, 10857.42, 10743, 10743]
7  >>> messreihe.append([34183100,10743.009])
8  >>> print(messreihe)
9  [10855.16, 10857.42, 10743, 10743, [34183100, 10743.009]]
```

Einfügen von Elementen

```
>>> messreihe = [10855.16,10857.42,10743]
>>> messung = 10743

>>> messreihe.insert(0, messung)
>>> print(messreihe)
[10743, 10855.16, 10857.42, 10743]

>>> pos = len(messreihe)
>>> messreihe.insert(pos, [34183100,10743.009])
>>> print(messreihe)
[10743, 10855.16, 10857.42, 10743, [34183100, 10743.009]]
```

Erweiterung der Liste

```
1  >>> messreihe = [10855.16,10857.42,10743]
2
3  >>> pos = len(messreihe)
4  >>> messreihe.extend( [34183100,10743.009])
5  >>> print(messreihe)
6  [10855.16, 10857.42, 10743, 34183100, 10743.009]
```

Hinweis: Eine Liste kann nur am Ende erweitert werden.

. . . mit einer while-Schleife

Defintion der, in der Schleife genutzten Variablen und Listen:

```
1  messreihe = [10855.16,10857.42,10743]
2  neu = [34183100,10743.009, 20456.78]
3  teil = messreihe[1:]
4  index = 0
5  einfuegemarke = 1
```

. . . mit einer while-Schleife

Definition der Schleife:

```
1  while index < len(neu):
2      if einfuegemarke < len(messreihe):
3          messreihe[einfuegemarke] = neu[index]
4          einfuegemarke = einfuegemarke + 1
5      else:
6          messreihe.append(neu[index])
7
8      index = index  + 1
9
10 else:
11     messreihe.extend(teil)
12     print(messreihe)
```

Löschen von Elementen

```
1  >>> messreihe = [10855.16,10857.42,10743]
2
3  >>> del messreihe[0]
4  >>> print(messreihe)
5  [10857.42, 10743]
6
7  >>> pos = len(messreihe) - 1
8  >>> del messreihe[pos]
```

Löschen von allen Elementen

```
1  >>> temperatur = [1.7, 3.8, 4.9, 8.5]
2
3  >>> while (temperatur):
4  ...     print(temperatur.pop(), " wird gelöscht.")
5  ... else:
6  ...     print("Die Liste ist vollständig geleert")
7  ...
8  8.5  wird gelöscht.
9  4.9  wird gelöscht.
10 3.8  wird gelöscht.
11 1.7  wird gelöscht.
12 Die Liste ist vollständig geleert
```

Methode pop()

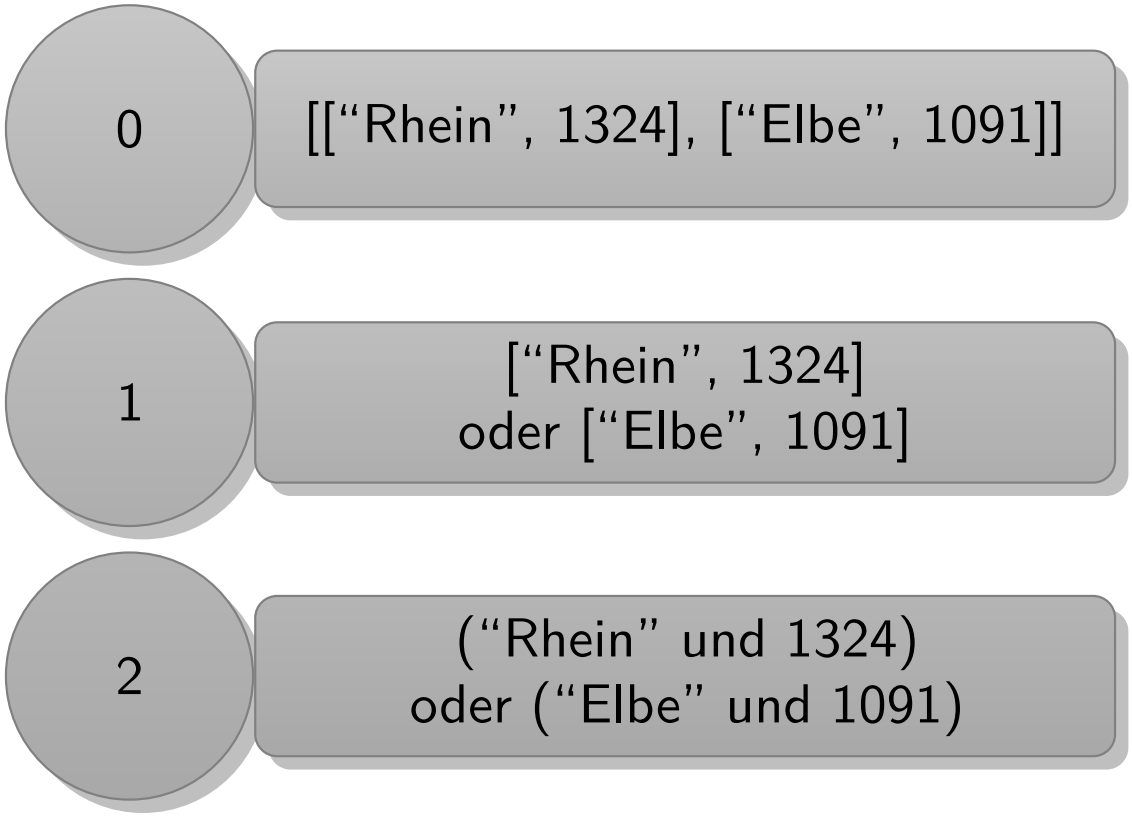
```
1  >>> temperatur = [1.7, 3.8, 4.9, 8.5]
2
3  >>> temperatur.pop()
4  8.5
5  >>> temperatur.pop(0)
6  1.7
```

- `sequenz.pop()`. Löschung des letzten Elements in der Sequenz.
- `sequenz.pop(index)`. Löschung des Elements an der Position / mit dem Index.
- Rückgabe des gelöschten Elements.

Schachteln von Listen

```
1  >>> fluss = [["Rhein", 1324], ["Elbe", 1091]]
2
3  >>> print("1.Liste in der Liste fluss: ", fluss[0])
4  1.Liste in der Liste fluss:  ['Rhein', 1324]
5  >>> print("2. Element in dieser Liste: ", fluss[0][1])
6  2. Element in dieser Liste:  1324
```

Dimensionen der Liste



Sequenzen in der ersten Dimension

```
1  >>> [1.7, 3.8, 4.9, 8.5]
2  [1.7, 3.8, 4.9, 8.5]
```

- Lineare Aufzählung
- Abbildung eines Zeitstrahls

Sequenzen in der zweiten Dimension

```
1  >>> [["Rhein", 1324], ["Elbe", 1091]]
2  [['Rhein', 1324], ['Elbe', 1091]]
```

- Informationen in Tabellenform
- x-, y-Koordinatensystem
- Matrizen

Sequenzen in der dritten Dimension

```
1  >>> m = [[[1, 2], [1, 3]], [[2, 2], [2, 3]], [[3, 2], [3, 3],
   ↪  [3,4]]]
2  >>> m[2][1][0]
3  3
```

- Räumliche Darstellung von Körpern
- x-, y-, z-Koordinatensystem
- Darstellung von Informationen auf zwei Zeitebenen (Tag-Monat, Jahr–Monat usw.)

Übungszettel

Speicherung als Liste in Python

Grundwasser: 2016, Baden-Württemberg, 352943

Durchschnittstemperatur: 0,6°C, 3,2°C, 6,6°C und 10,9°C

Einkaufszettel: 10 Eier, 1 kg Mehl, 1 kg Zucker, 1 x Backpulver

RGB-Farben: (0,0,255), (255,0,0), (120,120,120)

Programmierübung - Beschreibung

```
1  """
2  Aus einer Buchstabenfolge werden verschiedenen Teilelemente
   ↪ herausgeschnitten. Welche Elemente werden aus der Liste
   ↪ buchstaben herausgeschnitten?
3
4  Hinweis:
5  Bei Eingabe der Codezeilen in der Shell werden die
   ↪ herausgeschnitten Elemente direkt in der nächsten Zeile
   ↪ angezeigt.
6  """
```

Programmierübung - Code

```
1  buchstaben = ['A', 'B', 'C', 'D', 'a', 'b', 'c', 'd']
2
3  buchstaben[2]
4  buchstaben[1:5]
5  buchstaben[-5:-1]
6  buchstaben[5:]
7  buchstaben[:-1]
8  buchstaben[-1:(0 - len(buchstaben))]
9  buchstaben[(0 - len(buchstaben)):-1]
10 buchstaben[(0 - len(buchstaben) + 5):-1]
```

Sequenz „Tupel“

Lernziel:

- Abbildung von Aufzählungen, Messwerten und so weiter
- Erstellung von schreibgeschützten Sequenzen
- Sortierung von Sequenzen

. . . in Python

- Aufzählungen, Reihen
- Ablage von Zahlen, Zeichen und boolschen Werten, einen Bezug zu einander haben

Sequenz-Kategorien

- Listen. Veränderbare Folge von Elementen beliebigen Datentyps.
- Tupel. Nicht veränderbare Folge von Elementen beliebigen Datentyps.
- Range-Objekt. Nicht veränderbare Folge von Zahlen.
- Byte-Sequenz. Nicht oder schreibgeschützte Folge von Ganzzahlen zwischen 0 und 255.
- String. Alphanumerische und numerische Zeichen. Wörter. Sätze.

Eigenschaften

- Zugriff auf die Elemente über ein Index
- In Abhängigkeit der gewählten Kategorie veränderbar oder nicht

Erzeugung von Tupel

```
1  >>> xyz = (0, 1.5, -0.5)
2  >>> muster = (True, False, True)
3  >>> folge = ('A', 5, 'B', 2, 'C', 4)
```

- Runde Klammern. Beginn und Ende des Tupels
- Komma. Trennung der Elemente in der Liste

Anzahl der Elemente

```
1  >>> zitterrochen = ('G', 'A', 'T', 'C', 'A')
2  >>> len(zitterrochen)
3  5
```

Leeres Tupel?

```
1  >>> zitterrochen = ('G', 'A', 'T', 'C', 'A')
2
3  >>> if not (zitterrochen):
4  ...         print("Das Tupel ist leer.")
5  ...
```

Nicht leeres Tupel?

```
1  >>> zitterrochen = ('G', 'A', 'T', 'C', 'A')
2
3  >>> if (zitterrochen):
4  ...         print("Das Tupel ist nicht leer.")
5  ...
6  Das Tupel ist nicht leer.
```

Ist Element . . . enthalten?

```
1  >>> messreihe =
   ↪  (10855.16,10857.42,10743,10743,34183100,10743.009)
2  >>> messung = 10743
3
4  >>> if (messung in messreihe):
5  ...         print(messung, " ist enthalten.")
6  ...
7  10743  ist enthalten.
```

Ersetzung des Operators or

```
1  >>> monat = 1
2
3  >>> if (monat == 1) or (monat == 11) or (monat == 12):
4  ...         print("Winter")
5  ...
6  Winter
7  >>> if (monat in (1, 11, 12)):
8  ...         print("Winter")
9  ...
```

Ist Element . . . nicht enthalten?

```
1  messreihe = (10855.16, 10857.42, 10743, 10743, 34183100,
   ↪  10743.009)
2  messung = 10743
3
4  if (messung not in messreihe):
5      print(messung, " ist nicht enthalten.")
6  else:
7      print(messung, " ist enthalten.")
```

Häufigkeit eines Elements

```
1  ... messreihe =
   ↪  (10855.16,10857.42,10743,10743,34183100,10743.009)
2    File "<stdin>", line 4
3      messreihe =
       ↪  (10855.16,10857.42,10743,10743,34183100,10743.009)
4      ^
5  SyntaxError: invalid syntax
6  >>> messung = 10743
7
8  >>> messreihe.count(messung)
9  2
```

Objektmethode

```
1  >>> messreihe = (10855.16, 10857.42, 10743, 10743, 34183100,
   ↪  10743.009)
2  >>> messung = 10743
3
4  >>> messreihe.count(messung)
5  2
```

- Funktionen, die in der Klasse definiert sind, auf der das Objekt basiert
- Beschreiben das Verhalten von Objekten
- Verändern den Status eines Objekts
- Methoden von Tuples: https://www.w3schools.com/python/python_ref_tuple.asp

Aufruf der Methode

```
objekt.function(par01, par02)
```

- Verbindung von Objekt und Methode durch den Punkt
- Anwendung der Methode (rechts vom Punkt) auf das Objekt (links vom Punkt).

Methodenname

```
objekt.function(par01, par02)
```

- Eindeutig in der Vorlage.
- Beachtung von Groß- und Kleinschreibung.

Parameterliste der Methode

```
objekt.function(par01, par02, . . . )
```

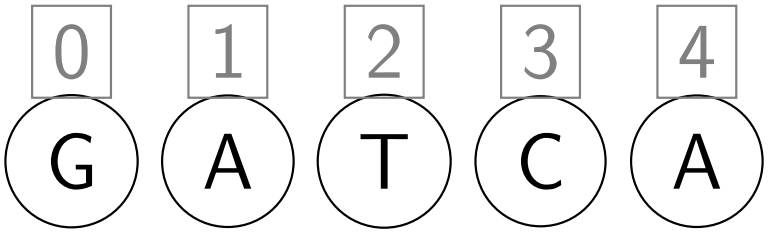
- Runde Klammern. Beginn und Ende der Liste.
- Komma. Trennung der Elemente.
- Anzahl der Elemente in Abhängigkeit der Definition. Kann leer sein.

Rückgabewert der Methode

variable = objekt.function(par01, par02, . . .)

- Rückmeldung an den Aufrufer
- `None` oder ein Wert entsprechend der Aktivität

Lesen von Elementen

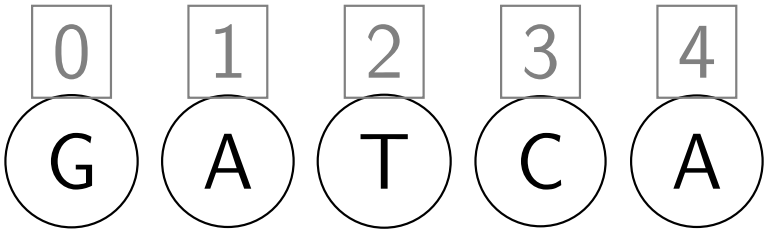



```
1  >>> zitterrochen = ('G', 'A', 'T', 'C', 'A')
2
3  >>> element = zitterrochen[0]
4  >>> print(element)
5  G
```

Index eines Elements

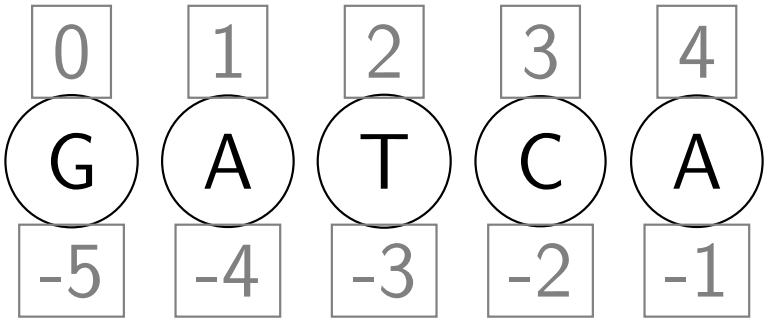
- Eindeutige Position eines Elements in einer Sequenz
- Positive oder negative Ganzzahl
- Folgt direkt dem Sequenz-Namen in eckigen Klammern.

Nutzung eines positiven Index



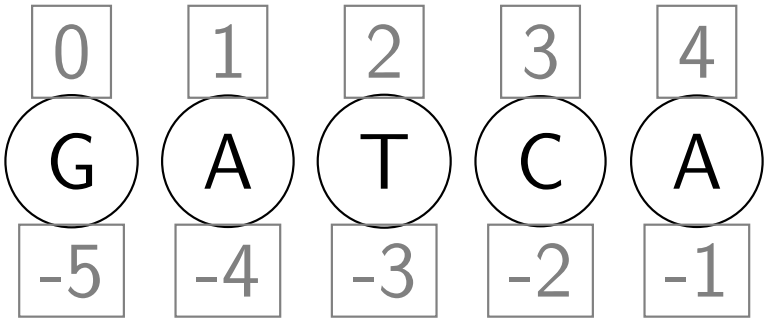

```
1  >>> zitterrochen = ('G', 'A', 'T', 'C', 'A')
2
3  >>> zitterrochen[0]
4  'G'
5  >>> zitterrochen[len(zitterrochen) - 1]
6  'A'
```

Nutzung eines negativen Index




```
1  >>> zitterrochen = ('G', 'A', 'T', 'C', 'A')
2
3  >>> zitterrochen[-1]
4  'A'
5  >>> zitterrochen[0 - len(zitterrochen)]
6  'G'
```

Slicing - Ausschneiden

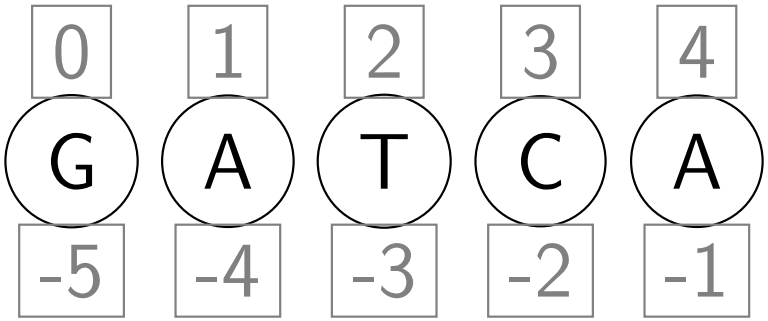



```
1  >>> zitterrochen = ('G', 'A', 'T', 'C', 'A')
2
3  >>> zitterrochen[-4:-2]
4  ('A', 'T')
5  >>> zitterrochen[1:3]
6  ('A', 'T')
```

Hinweise

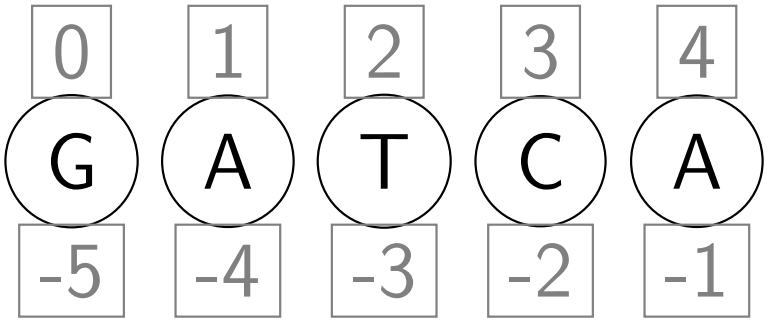
- Startwert. Beliebige positive oder negative Ganzzahl, beginnend beim ersten Element.
- Stopp-Wert. Definition des ersten Elements, welches nicht mehr ausgeschnitten wird. Liegt rechts vom Startwert.
- Start- und Stopp-Werte sind optional.

Keine Angabe zum Startwert



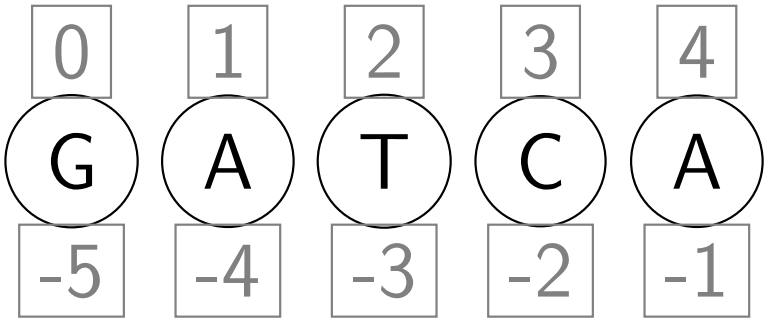

```
1  >>> zitterrochen = ('G', 'A', 'T', 'C', 'A')
2
3  >>> zitterrochen[:-2]
4  ('G', 'A', 'T')
5  >>> zitterrochen[:3]
6  ('G', 'A', 'T')
```

Keine Angabe zum Stopp-Wert




```
1  >>> zitterrochen = ('G', 'A', 'T', 'C', 'A')
2
3  >>> zitterrochen[-4:]
4  ('A', 'T', 'C', 'A')
5  >>> zitterrochen[1:]
6  ('A', 'T', 'C', 'A')
```

Keinerlei Angaben




```
1  >>> zitterrochen = ('G', 'A', 'T', 'C', 'A')
2
3  >>> zitterrochen[:]
4  ('G', 'A', 'T', 'C', 'A')
```

Schreibschutz

```
1  >>> zitterrochen = ('G', 'A', 'T', 'C', 'A')
2
3  >>> zitterrochen[1] = 'T'
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  TypeError: 'tuple' object does not support item assignment
```

Leere neue Tupel

```
1  >>> leereTupel = ()
2  >>> newTupel = tuple()
```

Hinweis:
Eine Hinzufügung von Elementen ist nicht möglich.

Klasse „Tupel“

```
1  >>> newTupel = tuple()
```

- Allgemeine Definition von Attributen und des Verhalten eines Objekts oder Datentyps
- Zum Beispiel können Autos, Einhörner, Listen und was immer du sonst noch möchtest mit einer Klasse beschrieben werden

Erzeugung eines Objekts „Tupel“

```
1  >>> newList = tuple()
```

- Eine spezielle Methode (Konstruktor) wird genutzt, um ein Objekt oder Datentyp zu erstellen.
- Objekte beschreiben eine konkretes Exemplar einer bestimmten Klasse.
- Diese Variante wird ausschließlich genutzt, um Sequenzen in Tupel zu konvertieren.

Konvertierung von Listen in Tupel

```
1  >>> liste = [0, 4.5, 6]
2  >>> tupel = tuple(liste)
3
4  >>> print(tupel)
5  (0, 4.5, 6)
6  >>> type(tupel)
7  <class 'tuple'>
```

Konstruktor „Tupels“

```
objekt = konstruktor(baumaterial)
```

- Spezielle Funktion zum Erzeugen eines Objekts aus einer Klasse heraus
- Name der Vorlage $\leftrightarrow$ Name der Konstruktionsfunktion. Die Groß- und Kleinschreibung wird beachtet.

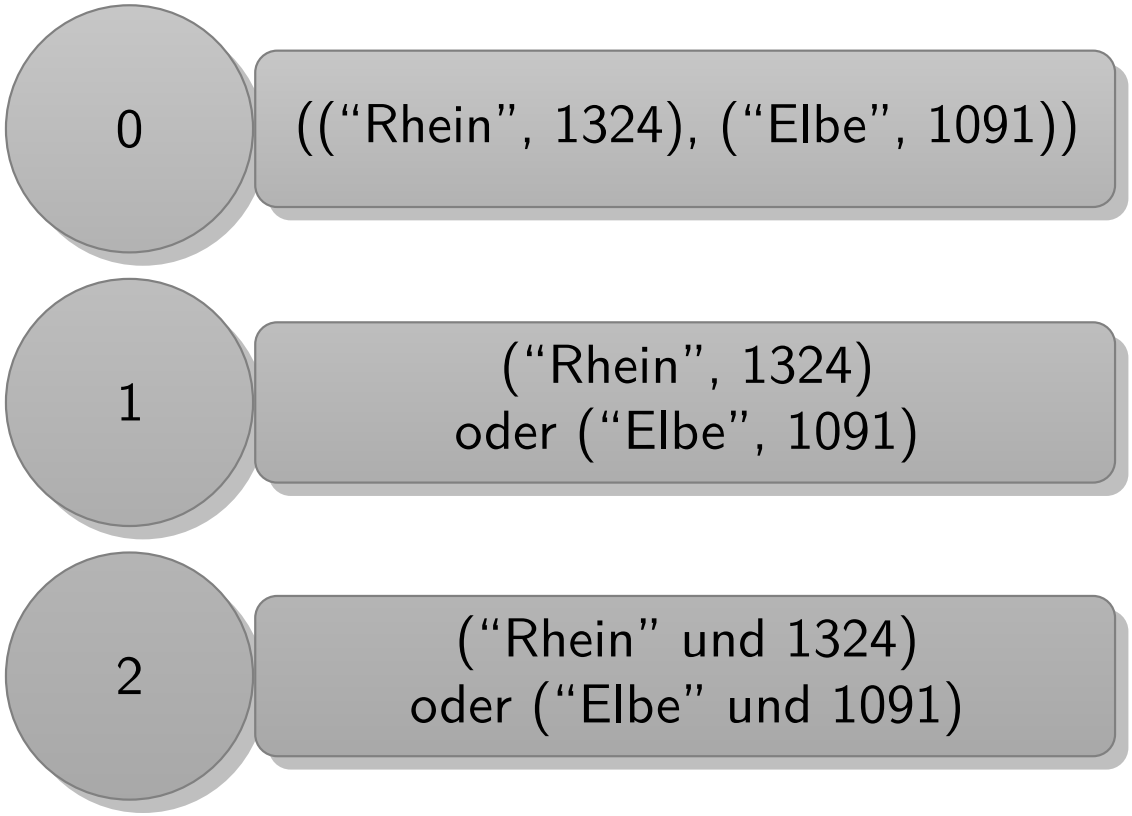
Sortierung von Elementen

```
1  >>> messreihe = (10855.16,10857.42,10743)
2  >>> listeSortiert = sorted(messreihe)
3  >>> print(listeSortiert)
4  [10743, 10855.16, 10857.42]
5
6  >>> listeSortiert = sorted(messreihe, reverse=True)
7  >>> print(listeSortiert)
8  [10857.42, 10855.16, 10743]
```

Schachteln von Tupel

```
1  >>> fluss = (("Rhein", 1324), ("Elbe", 1091))
2
3  >>> print("1.Tupel in dem Tupel fluss: ", fluss[0])
4  1.Tupel in dem Tupel fluss:  ('Rhein', 1324)
5  >>> print("2. Element in diesem Tupel : ", fluss[0][1])
6  2. Element in diesem Tupel :  1324
```

Dimensionen des Tupels



Schachteln von Listen in Tupeln

```
1  >>> fluss = (["Rhein", 1324], ["Elbe", 1091])
2
3  >>> fluss[0] = ["Weser", 452]
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  TypeError: 'tuple' object does not support item assignment
7  >>> fluss[0][1] = 1325
8  >>> print(fluss)
9  (['Rhein', 1325], ['Elbe', 1091])
```

Schachteln von Tupel in Listen

```
1  >>> fluss = [("Rhein", 1324), ("Elbe", 1091)]
2
3  >>> fluss[0] = ["Weser", 452]
4  >>> print(fluss)
5  [['Weser', 452], ('Elbe', 1091)]
6  >>> fluss[0][1] = 1325
7  >>> print(fluss)
8  [['Weser', 1325], ('Elbe', 1091)]
```

Tupel-Wertzuweisungen

```
1  >>> x = 0
2  >>> y = 4.5
3
4  >>> print(x,y)
5  0 4.5
6
7  >>> y, x = x, y
8
9  >>> print(x,y)
10 4.5 0
```

Übungszettel

Speicherung als Tupel in Python

Noten: sehr gut, gut, befriedigend, ausreichend

Chemische Elemente und deren Ordnungszahl: Wasserstoff - 1, Magnesium - 12, Kalium - 19

Winkel: 90°, 180°, 360°

Übungszettel

Sortierung der Tupelelemente

# Werte, aufsteigend sortieren
datum = (11.2, 05.02, 20.02, 3.02)

# Werte, absteigend sortierend
bevoelkerung = (2928671, 2931384, 2939289, 2951272, 2969466, 2988274,3007481, 2948959, 2962728)

Für jedes Element in einer Sequenz

Lernziel:

- Durchlaufen einer Sequenz vom ersten bis zum letzten Element
- Anwendung von Code auf jedes Element in einer Sequenz
- Erstellung von Sequenzen, die Ganzzahlen enthalten

Implementierung mit der while-Schleife

```
1  a = 0
2  index = 0
3  buchstaben = ('G', 'A', 'T', 'C', 'A')
4
5  while buchstaben:
6      if  (buchstaben[index] == "A"):
7          a = a + 1
8
9      index = index + 1
10
11     if (index == len(buchstaben)):
12         break
13
14 if (a > 0):
15     print("Häufigkeit A: ", a)
```

Erläuterung des Schleifenkopfes

while buchstaben:

- Name der Liste als Begingung: Solange die Sequenz ein Element enthält ...
- Andere Möglichkeit:
  `while (index < len(buchstaben))`
- Ende des Schleifenkopfes: Doppelpunkt.

for-Schleife

```
1  a = 0
2  buchstaben = ('G', 'A', 'T', 'C', 'A')
3
4  for zeichen in buchstaben:
5          if  (zeichen == "A"):
6                  a = a + 1
7  else:
8          print("Häufigkeit A: ", a)
```

Aufbau des Schleifenkopfes

for [element] in [sequenz]:

- `for [element]`: Für jedes Element ...
- `in [sequenz]`: In der angegebenen Sequenz.
- Ende des Schleifenkopfes: Doppelpunkt.

Für jedes Element

```
for [element] in [sequenz]:
```

- Synonym für das aktuell zu verarbeitende Element aus der Sequenz
- Der Name des Platzhalters ist frei wählbar

. . . in der Sequenz

```
for [element] in [sequenz]:
```

- Vor der Nutzung muss der Sequenz-Name definiert werden.
- Falls die Sequenz leer, ist wird der Schleifenrumpf nicht durchlaufen.

Schleifenrumpf

```
1  a = 0
2  buchstaben = ('G', 'A', 'T', 'C', 'A')
3
4  for zeichen in buchstaben:
5          if  (zeichen == "A"):
6                  a = a + 1
```

- Eingerückte Zeilen unterhalb des Kopfes.
- Zeilen, die mit vier Leerzeichen im Anschluss an den Kopf beginnen.

else-Zweig

```
1  a = 0
2  buchstaben = ('G', 'A', 'T', 'C', 'A')
3  for zeichen in buchstaben:
4          if  (zeichen == "A"):
5                  a = a + 1
6  else:
7          print("Häufigkeit A: ", a)
```

- Optional
- Ausführung bei einem vollständigen Durchlauf.

Position eines Elements

```
1  a = 0
2  buchstaben = ('G', 'A', 'T', 'C', 'A')
3  pos = -1
4
5  for zeichen in buchstaben:
6          pos = pos + 1
7
8          if  (zeichen == "A"):
9                  print("A gefunden an der Position: ", pos)
10                 a = a + 1
11 else:
12         print("Häufigkeit A: ", a)
```

Schachtelung

```
1  fluss = (("Rhein", 1324), ("Elbe", 1091))
2
3  for zeile in fluss:
4          print(zeile)
5
6          for element in zeile:
7                  print(element)
```

Range-Objekt

```
1  zahlen = range(0,4)
2
3  for element in zahlen:
4          print(element)
```

Hinweis: Nur Ganzzahlen.

Einstellung der Schrittweite

```
1  zahlen = range(0,10,2)
2
3  for element in zahlen:
4          print(element)
```

Erläuterung

```
range(start,stop,step)
```

- Startwert: Optional. Standartwert 0.
- Stoppwert: Nicht in der Sequenz enthalten.
- Step: Optional. Standardwert 1. Ganzzahl.

Nutzung in einer for-Schleife

```
1  for element in range(0, -10, -2):
2          print(element)
```

Sequenz von Gleitkommazahlen

```
1  start = 0
2  stop = 2.5
3  fltCount = start
4  step = 0.2
5  lstNumber = []
6
7  while fltCount < stop:
8          lstNumber.append(fltCount)
9          fltCount = fltCount + step
```

Programmierübung Nr. 1

Arbeiten mit dem Range-Objekt

Ersetzen Sie das Tupel durch ein Range-Objekt.

```
for i in (8, 6, 4, 2, 0):
        ergebnis = i ** 2
        print(i, " ** 2 = ", ergebnis)
```

Programmierübung Nr. 2

Arbeiten mit einer for-Schleife

Die folgende Liste ist gegeben: durchschnitt = [-0.2, 0.1, 3.2, 7.4, 12.1, 15.3, 17.1, 16.8, 13.5, 8.9, 4.3, 1.1]
Berechnen Sie die Differenz zwischen den Listenelementen durchschnitt[index] und durchschnitt[index - 1].

Programmieraufgabe

Überprüfung der Formel:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Lösung

```
1  n = 6
2  i = 1
3  result = (n * (n + 1)) / 2
4  summe = 0
5
6  while i <= n:
7      summe = summe + i
8      i = i + 1
9  else:
10     print("Ist (n * (n + 1)) / 2 = sum(i)? ")
11
12     if result == summe:
13         print("Ja")
14     else:
15         print("Nein")
```

Programmieraufgabe

Die Werte eine Würfels wie zum Beispiel bei dem Spiel „Mensch ärgere dich nicht“ werden in einer Sequenz gespeichert.
Der Computer würfelt eine Zahl.
Der Nutzer hat drei Versuche, die gewürfelte Zahl zu raten. In diesem Beispiel wird der Nutzer durch Auswahl von Zufallszahlen simuliert. Falls der simulierte Nutzer die Zahl richtig rät, wird eine entsprechende Meldung ausgegeben.

Lösung

```
1  from random import randrange
2
3  wuerfel = (1, 2, 3, 4, 5, 6)
4  gewuerfelt = randrange(0, len(wuerfel))
5  print("Gewürfelt: ", wuerfel[gewuerfelt])
6
7  for i in range(1,4):
8      geraten = randrange(0, len(wuerfel))
9      print("Geratene Zahl ", wuerfel[geraten])
10
11     if (wuerfel[geraten] == wuerfel[gewuerfelt]):
12         print("Hurra. Gewonnen.")
13         break
14 else:
15     print("Schade. Der Computer hat gewonnen.")
```

Dictionary

Lernziel:

- Nutzung eines beliebig unveränderbaren Datentyp als Index in einer Sequenz
- Speicherung von Werten unter einem bestimmten Namen
- Arbeiten mit einem ungeordneten Container

. . . in Python

- Identifikation eines Wertes in einer ungeordneten Sammlung mit Hilfe eines Schlüssels
- Beispiele: Beschreibung eines Objekts, Wörterbuch

Erzeugung eines Dictionary

```
1  >>> farbe = {'rot':0xff0000, 'gruen':0x00ff00,
   ↪  'blau':0x0000ff}
2  >>> highScore = {1:'Spieler A', 2:'Spieler C', 3:'Spieler B'}
3  >>> buch ={"autor":"Monty Python", "titel":"Flying Circus"}
```

Leeres neues Dictionary

```
1  >>> dictLeer = {}
2  >>> dictNeu = dict()
```

Objekt „Dictionary“

```
1  >>> leer = dict()
2  >>> highScore = dict([("Spieler 1",80)])
```

- Jede Variable verweist auf ein Objekt.
- Jedes Objekt wird mit Hilfe einer Vorlage erzeugt. Hier: `dict`.
- Vorlagen sind spezielle Funktionen. Der Funktion können Parameter übergeben werden, müssen aber nicht.

Klasse „dict“

```
1  >>> leer = dict()
2  >>> highScore = dict([("Spieler 1",80)])
```

- Allgemeine Definition von Attributen und des Verhalten eines Objekts oder Datentyps
- Zum Beispiel können Autos, Einhörner, Listen und was immer du sonst noch möchtest mit einer Klasse beschrieben werden

Konstruktionsmethode „dict“

```
1  >>> leer = dict()
2  >>> highScore = dict([("Spieler 1",80)])
```

- Mit Hilfe einer speziellen Methode (dem Konstruktur) wird ein Objekt vom Typ „Dictionary“ erzeugt.
- Objekte beschreiben ein bestimmtes Exemplar einer Klasse.

Aufruf der Konstruktionsmethode

```
1  >>> leer = dict()
2  >>> highScore = dict([("Spieler 1",80)])
```

- Mit dem Namen der Klasse.
- Dem Namen der Klasse folgt direkt im Anschluss die Parameterliste.
- Die Parameterliste beginnt und endet mit den runden Klammern. Falls die Liste leer ist, ist das Dictionary leer. Falls der Liste ein Parameter übergeben wird, wird ein Dictionary mit x Elementen erzeugt.

Elemente in einem Dictionary

```
1  >>> buch ={"autor":"Monty Python", "titel":"Flying Circus"}
```

- Beginn und Ende des Dictionary: Geschweiften Klammern
- Trennung der Elemente durch ein Komma
- Zusammensetzung der Elemente: Schlüssel:Wert

Anzahl der Elemente

```
1  >>> buch ={"autor":"Monty Python", "titel":"Flying Circus"}
2  >>> len(buch)
3  2
```

Schüssel-Wert-Paare

```
1  >>> buch ={"autor":"Monty Python"}
```

- Trennung durch den Doppelpunkt
- `key:value`

Wert

```
1  >>> buch ={"autor":"Monty Python"}
```

- Rechts vom Doppelpunkt
- Kann beliebig oft vorkommen
- Jeder Datentyp ist möglich

Schlüssel

```
1  >>> buch ={"autor":"Monty Python"}
```

- Links vom Doppelpunkt
- Kann im Dictionary nur einmal vorkommen
- Unveränderlich
- Datentyp: String oder Ganzzahl.

Zusammengesetzer Schlüssel

```
1  >>> fahrzeug = {("Schiene", "Motor"): "Eisenbahn",
2  ...              ("Strasse", "Motor", "Transport"):"LKW"}
```

- Implementierung als Tupel
- Die Elemente des Tupels ergeben einen eindeutigen Schlüssel

Existiert der Schlüssel?

```
1  >>> {"autor":'Thomas Mann', "autor":'Heinrich Mann'}
2  {'autor': 'Heinrich Mann'}
```

Automatische Ersetzung des dazgehörigen Wertes, aber keine Anzeige eines Fehlers.

Hinzufügung von Elementen

```
1  >>> highscore = {}
2  >>> highscore["L1"] = 50
3  >>> highscore["L2"] = 42
```

Ausgabe von Werten

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> print(highscore["L1"])
4  50
```

Fehler: Nicht vorhandener Schlüssel

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> minute = highscore["L3"]
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  KeyError: 'L3'
```

Holen eines Wertes

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> highscore.get("L3", "Keine Angaben")
4  'Keine Angaben'
```

Methode eines Objekts

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> highscore.get("L3", "Keine Angaben")
4  'Keine Angaben'
```

- Funktionen, die in der Klasse definiert sind, auf der das Objekt basiert
- Beschreiben das Verhalten von Objekten
- Verändern den Status eines Objekts

Aufruf der Methode

```
highscore.get("L3", "Keine Angaben")
```

- Verbindung von Objekt und Methode durch den Punkt
- Anwendung der Methode (rechts vom Punkt, get) auf das Objekt (links vom Punkt, highscore)

Name der Methode

```
highscore.get("L3", "Keine Angaben")
```

- Eindeutig in der Vorlage.
- Beachtung von Groß- und Kleinschreibung.

Parameterliste der Methode

```
objekt.function(par01, par02, . . . )
```

- Beginn und Ende der Parameterliste: Runde Klammern.
- Trennzeichen zwischen den Parametern: Komma.
- Anzahl der Elemente in der Liste in Abhängigkeit der Definition. Kann leer sein.

. . . der Methode get()

```
highscore.get(key, keyNotFound)
```

- Der erste Parameter entspricht einem Schüssel aus dem Dictionary. Der Schlüssel idenfiziert eindeutig einen Wert.
- Falls der angegebene Schlüssel nicht vorhanden ist, wird der Text des zweiten Parameters zurückgegeben werden. Dieser Parameter ist optional.

Rückgabewert der Methode

element = highscore.get(key, keyNotFound)

- Rückmeldung an den Aufrufer
- `None` oder ein Wert in Bezug auf das beschriebene Verhalten
- Methode `get()`: Der, zu dem Schüssel gehörende Wert oder der zweite Parameter der Methode

Ist der Schlüssel vorhanden?

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> if "L3" in highscore:
4  ...         print("L3 ist vorhanden")
5  ...
```

Ist der Schlüssel nicht vorhanden?

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> if "L3" not in highscore:
4  ...         print("L3 ist nicht vorhanden")
5  ...
6  L3 ist nicht vorhanden
```

Ausgabe aller Schlüssel-Wert-Paare

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> for element in highscore.items():
4  ...         print("Schlüssel: ", element[0], ", Wert: ",
   ↪  element[1])
5  ...
6  Schlüssel:  L1 , Wert:  50
7  Schlüssel:  L2 , Wert:  42
```

Weitere Möglichkeit

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> for key, value in highscore.items():
4  ...         print("Schlüssel: ", key, ", Wert: ", value)
5  ...
6  Schlüssel:  L1 , Wert:  50
7  Schlüssel:  L2 , Wert:  42
```

Die Name der Variablen in der Tupel-Anweisung `(key, value)` sind frei wählbar.

Ausgabe aller Schlüssel

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> for key in highscore.keys():
4  ...         print("Schlüssel: ", key)
5  ...
6  Schlüssel:  L1
7  Schlüssel:  L2
```

Weitere Möglichkeit

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> for key in highscore:
4  ...         print("Schlüssel: ", key)
5  ...
6  Schlüssel:  L1
7  Schlüssel:  L2
```

Ausgabe aller Werte

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> for value in highscore.values():
4  ...         print("Wert: ", value)
5  ...
6  Wert:  50
7  Wert:  42
```

Löschen eines Elements

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> del(highscore["L2"])
4  >>> print(highscore)
5  {'L1': 50}
```

Leeren des Dictionary

```
1  highscore = {"L1":50, "L2":42}
2
3  while (len(highscore) > 0):
4      # Welche Schlüssel sind vorhanden?
5      # Alle Schlüssel werden in eine Liste konvertiert
6      schluessel = list(highscore.keys())
7
8      del(highscore[schluessel[0]])
```

Besser: Nutzung der Methode

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> highscore.clear()
4  >>> print(highscore)
5  {}
```

Kopieren des Dictionary

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> laufzeiten = highscore.copy()
4  >>> print(laufzeiten)
5  {'L1': 50, 'L2': 42}
```

Flache Kopie

```
>>> highscore = {"L1":50, "L2":42}

>>> laufzeiten = highscore
>>> print("id laufzeiten: ", id(laufzeiten))
id laufzeiten:  1274183025856
>>> print("id highscore: ",id(laufzeiten))
id highscore:  1274183025856

>>> laufzeiten['L1'] = 120
>>> print("Highscore", highscore)
Highscore {'L1': 120, 'L2': 42}
```

Tiefe Kopie

```
1  >>> highscore = {"L1":50, "L2":42}
2
3  >>> laufzeiten = highscore.copy()
4
5  >>> print("id laufzeiten: ", id(laufzeiten))
6  id laufzeiten:  1274182927872
7  >>> print("id highscore: ",id(laufzeiten))
8  id highscore:  1274182927872
9
10 >>> laufzeiten['L1'] = 120
11 >>> print("Highscore", highscore)
12 Highscore {'L1': 50, 'L2': 42}
```

Übungszettel

Speicherung als Dictionary in Python

Noten: 1 = sehr gut, 2 = gut, 3 = befriedigend, 4 = ausreichend

Flughäfen und ihre Kennung: Berlin-Schönefeld - SXF, Frankfurt - FRA, Hannover - HAJ

Postleitzahlen und ihre Städte: Hannover 30159, Braunschweig 30100 , Hannover 30169, Goslar 38640, Braunschweig 38102, Neustadt am Rübenberge 31535

Programmieraufgabe

Obstkisten

Die folgenden Obstsorten und deren Lagerbestände werden in Python abgebildet.
12 Kisten Bananen, 20 Kisten Äpfel, 2 Kisten Birnen und keine Kiste Weintrauben.

Folgende Kisten werden demnächst ausgeliefert:
10 Kisten Äpfel, 3 Kisten Birnen und 2 Kisten Weintrauben.

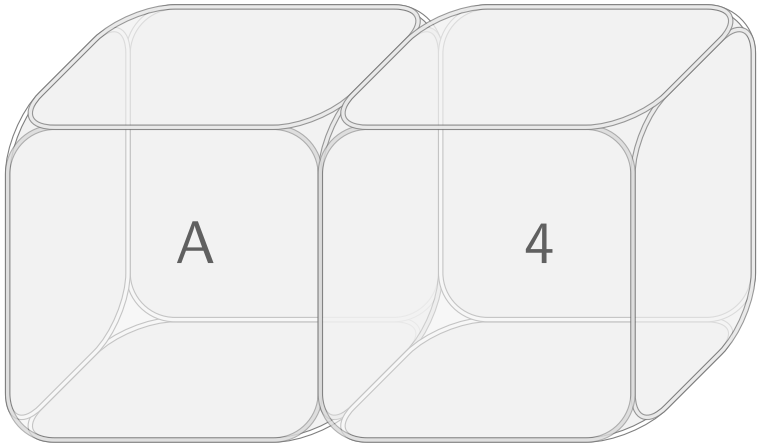
Falls die Auslieferung höher als der Lagerbestand ist, werden neue Waren bestellt. Die Bestellungen werden in einem Dictionary gespeichert.

Strings / Zeichenketten

Lernziel:

- Speicherung von alphanumerischen und numerischen Zeichen
- Verknüpfen von Wörtern
- Finden, Ersetzen und so weiter von Wortbestandteilen in Texten

. . . in Python



- Liste von alphanumerischen und numerischen Zeichen
- Abbildung von Sätzen und Wörtern mit Hilfe von Unicode-Zeichen

Eigenschaften

- Zugriff auf die Elemente über ein Index
- Wie Tupel nicht veränderbar

Erzeugung von Strings

```
1  >>> """
2  ... Dokumentationsstring am Anfang einer Code-Datei
3  ... """
4  '\nDokumentationsstring am Anfang einer Code-Datei\n'
5
6  >>> str01 = "Der String darf keine Anführungszeichen
   ↪  enthalten."
7
8  >>> str02 = 'Der String darf kein Apostroph enthalten.'
```

Dokumentationsstring in Python

- Beginn und Ende: Drei Anführungszeichen oder Apostroph direkt hintereinander
- Mehrzeiliger String
- Ausdruck am Bildschirm wie im Code eingegeben
- Erläuterung zu einer Funktion oder Code in einer Datei

Leerer neuer String

```
1  >>> strLeer01 = ""
2  >>> strLeer01 = ''
3  >>> strLeer03 = str()
```

. . . erzeugen mit

- zwei Anführungszeichen direkt hintereinander
- zwei Apostroph direkt hintereinander
- dem Konstruktor der Klasse string

Erzeugung eines Objekts „String“

```
1  >>> newEmptyString = str()
```

- Mit Hilfe einer speziellen Methode (dem Konstruktur) wird ein Objekt vom Typ „String“ erzeugt.
- Objekte beschreiben ein bestimmtes Exemplar einer Klasse.

Klasse „String“

```
1  >>> newEmptyString = str()
```

- Allgemeine Definition von Attributen und des Verhalten eines Objekts oder Datentyps
- Zum Beispiel können Autos, Einhörner, Listen, Zeichenketten und was immer du sonst noch möchtest mit einer Klasse beschrieben werden

Konstruktionsmethode „String“

```
1  >>> newEmptyString = str()
```

- Spezielle Funktion in jeder Klasse zur Erzeugung eines Objekts.
- Klasse und Konstruktor haben den gleichen Namen. Groß- und Kleinschreibung wird beachtet.

Aufruf der Konstruktionsmethode

```
1  >>> newEmptyString = str()
```

- Mit dem Namen der Klasse.
- Dem Namen der Klasse folgt direkt im Anschluss die Parameterliste.
- Die Parameterliste beginnt und endet mit den runden Klammern. Hier ist sie leer. Die Liste hat keine Elemente.

Konvertierung von Zahlen

```
1  >>> newEmptyString = str(34)
```

- Dem Konstruktor wird das zu konvertierende Element als Parameter übergeben
- Aus dem, in den runden Klammern übergebenen Element wird ein String erzeugt

Nutzung des Unicode-Zeichensatzes

- Kodierung von Zeichen mit Hilfe einer Ziffer.
- Die ersten 256 Zeichen entsprechen dem ISO Latin-1-Zeichensatz.
- Siehe https://docs.python.org/3/howto/unicode.html

Kodierung eines Zeichens

```
1  >>> zeichen = 'a'
2  >>> code = 65
3
4  >>> ord(zeichen)
5  97
6  >>> chr(code)
7  'A'
```

Beginn mit dem Anführungszeichen

```
1  >>> "Monty Python: 'Flying Circus' "
2  "Monty Python: 'Flying Circus' "
3
4  >>> "Monty Python: \"Flying Circus\" "
5  'Monty Python: "Flying Circus" '
```

Beginn mit dem Apostroph

```
1  >>> 'Monty Python: "Flying Circus" '
2  'Monty Python: "Flying Circus" '
3
4  >>> 'Monty Python: \'Flying Circus\' '
5  "Monty Python: 'Flying Circus' "
```

Maskierung von Zeichen

- Beginn mit dem Backslash
- Maskierung von Zeichen, die eine besondere Bedeutung in Python haben
- Steuerzeichen für den Druck
- Siehe http://python-ds.com/python-3-escape-sequences

Beispiele

```
1  >>> print('Erste Zeile \n Zweite Zeile ')
2  Erste Zeile
3   Zweite Zeile
4
5  >>> print('1.5 \t 2.4 \t 0.5 ')
6  1.5          2.4          0.5
7
8  >>> print(" Backslash \\ ")
9   Backslash \
```

Möglicher Fehler

```
1  >>> ordner = "c:\\python\aufgaben\"
2    File "<stdin>", line 1
3      ordner = "c:\\python\aufgaben\"
4                                     ^
5  SyntaxError: EOL while scanning string literal
```

Anzahl der Elemente

```
1  >>> zeichen = 'Monty Python: "Flying Circus" '
2  >>> len(zeichen)
3  30
```

Leere Sequenz?

```
1  >>> zeichen = 'Monty Python: "Flying Circus" '
2
3  >>> if not (zeichen):
4  ...         print("Der String ist leer.")
5  ...
```

Nicht leere Sequenz?

```
1  >>> zeichen = 'Monty Python: "Flying Circus" '
2
3  >>> if (zeichen):
4  ...         print("Der String ist nicht leer.")
5  ...
6  Der String ist nicht leer.
```

Ist das Zeichen . . . enthalten?

```
1  >>> reihe = 'Monty Python: Flying Circus'
2  >>> zeichen = 'n'
3
4  >>> if (zeichen in reihe):
5  ...         print(zeichen, " ist enthalten")
6  ...
7  n  ist enthalten
```

Ist das Zeichen . . . nicht enthalten?

```
1  >>> reihe = 'Monty Python: Flying Circus'
2  >>> zeichen = '@'
3
4  >>> if (zeichen not in reihe):
5  ...         print(zeichen, " ist nicht enthalten")
6  ...
7  @  ist nicht enthalten
```

Häufigkeit eines Zeichens

```
1  >>> reihe = 'Monty Python: Flying Circus'
2  >>> zeichen = 'n'
3
4  >>> reihe.count(zeichen)
5  3
```

Methode eines Objekts

```
1  >>> reihe = 'Monty Python: Flying Circus'
2  >>> zeichen = 'n'
3
4  >>> reihe.count(zeichen)
5  3
```

- Funktionen, die in der Klasse definiert sind, auf der das Objekt basiert
- Beschreiben das Verhalten von Objekten
- Verändern den Status eines Objekts
- Liste von String-Methoden siehe http://python-ds.com/python-3-string-methods

Aufruf der Methode

```
reihe.count(zeichen)
```

- Verbindung von Objekt und Methode durch den Punkt.
- Anwendung der Methode (rechts vom Punkt, count) auf das Objekt (links vom Punkt, reihe).
- Hinweis: Die Methode ist in einer Klasse definiert. Das Objekt basiert auf dieser Klasse.

Name der Methode

```
reihe.count(zeichen)
```

- Eindeutig in der Vorlage.
- Beachtung von Groß- und Kleinschreibung.

Parameterliste der Methode

```
reihe.count(zeichen)
```

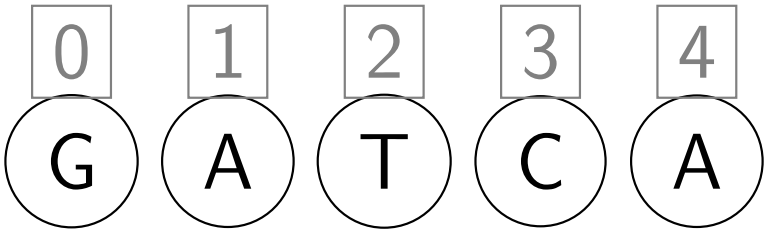
- Beginn und Ende der Parameterliste: Runde Klammern.
- Trennzeichen zwischen den Parametern: Komma.
- Anzahl der Elemente in der Liste in Abhängigkeit der Definition. Kann leer sein.
- In diesem Beispiel: Ein String.

Rückgabewert der Methode

```
anzahlElement = reihe.count(zeichen)
```

- Rückmeldung an den Aufrufer
- `None` oder ein Wert in Bezug auf das beschriebene Verhalten
- `count`: Wie oft kommt der übergebene Parameter in dem String vor?

Zugriff auf die Zeichen

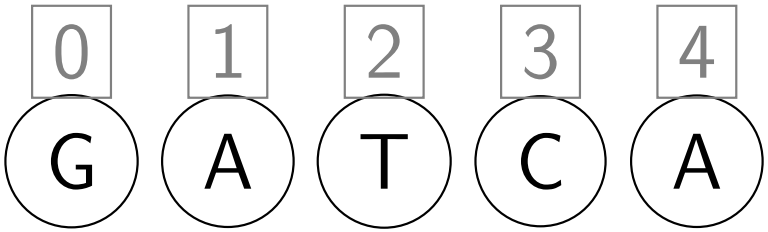



```
1  >>> zitterrochen = 'GATCA'
2
3  >>> element = zitterrochen[0]
4  >>> print(element)
5  G
```

. . . mit Hilfe eines Index

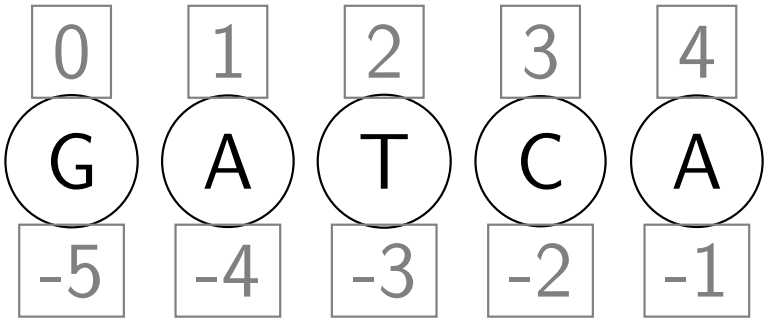
- Definiert eindeutig eine Position in einem String
- Nutzung einer positiven oder negativen Ganzzahl
- Angabe direkt im Anschluss an den Sequenz-Namen in eckigen Klammern

Nutzung eines positiven Index




```
1  >>> zitterrochen = 'GATCA'
2
3  >>> zitterrochen[0]
4  'G'
5  >>> zitterrochen[len(zitterrochen) - 1]
6  'A'
```

Nutzung eines negativen Index

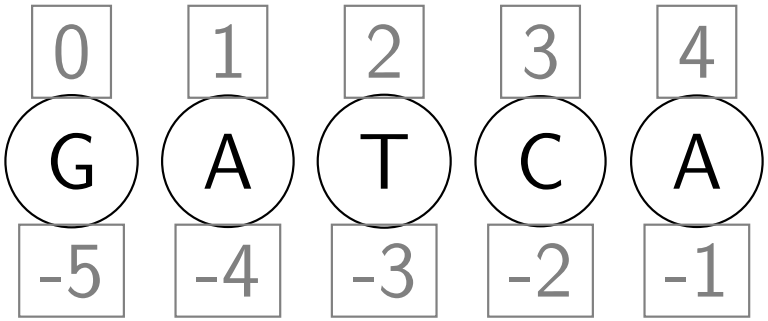



```
1  >>> zitterrochen = 'GATCA'
2
3  >>> zitterrochen[-1]
4  'A'
5  >>> zitterrochen[0 - len(zitterrochen)]
6  'G'
```

Zu großer Index

```
1  >>> zitterrochen = 'GATCA'
2  >>> element = 'T'
3  >>> zitterrochen[5] = element
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  TypeError: 'str' object does not support item assignment
```

Slicing - Ausschneiden

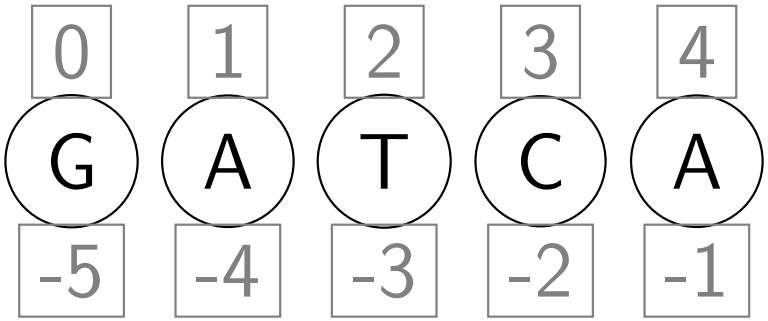



```
1  >>> zitterrochen = 'GATCA'
2
3  >>> zitterrochen[-4:-2]
4  'AT'
5  >>> zitterrochen[1:3]
6  'AT'
```

Hinweise

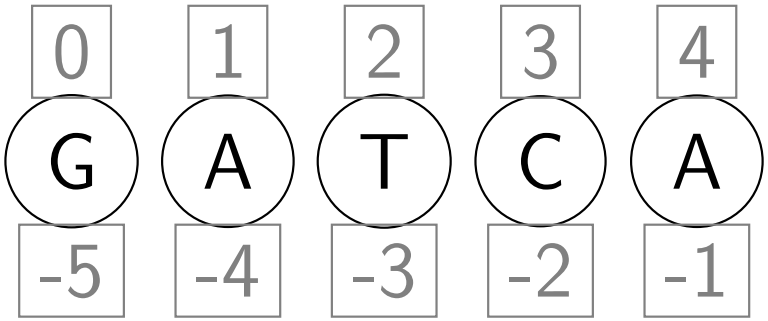
- Startwert. Beliebige positive oder negative Ganzzahl, beginnend beim ersten Element.
- Stopp-Wert. Definition des ersten Elements, welches nicht mehr ausgeschnitten wird. Liegt rechts vom Startwert.
- Start- und Stopp-Werte sind optional.

Keine Angabe zum Startwert



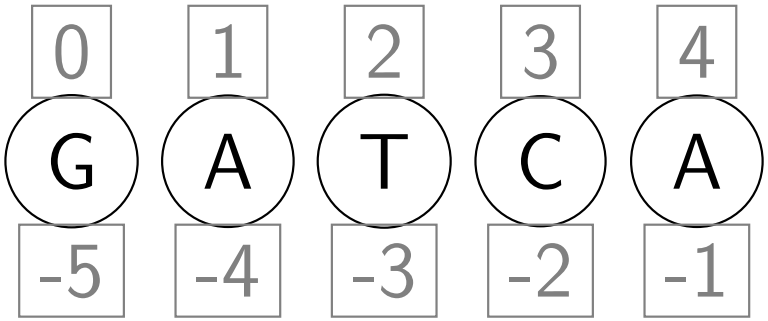

```
1  >>> zitterrochen = 'GATCA'
2
3  >>> zitterrochen[:-2]
4  'GAT'
5  >>> zitterrochen[:3]
6  'GAT'
```

Keine Angabe zum Stopp-Wert



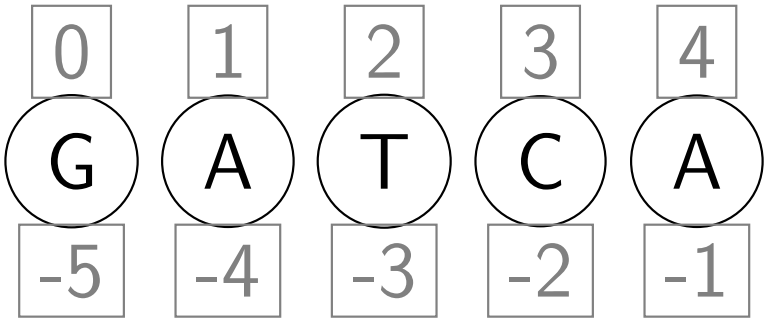

```
1  >>> zitterrochen = 'GATCA'
2
3  >>> zitterrochen[-4:]
4  'ATCA'
5  >>> zitterrochen[1:]
6  'ATCA'
```

Keinerlei Angaben




```
1  >>> zitterrochen = 'GATCA'
2
3  >>> zitterrochen[:]
4  'GATCA'
```

Angabe einer Schrittweite




```
1  >>> zitterrochen = 'GATCA'
2
3  >>> zitterrochen[::2]
4  'GTA'
```

Hinweise

- Startwert. Beliebige positive oder negative Ganzzahl, beginnend beim ersten Element.
- Stopp-Wert. Definition des ersten Elements, welches nicht mehr ausgeschnitten wird. Liegt rechts vom Startwert.
- Schrittweite. Standardmäßig wird eine Schrittweite von 1 angenommen. In dem vorherigen Beispiel wird jedes zweite Zeichen in dem Teilbereich angezeigt.
- Alle drei Angaben sind optional.

Verknüpfung von String

```
1  >>> autor = "Monty Python"
2  >>> buch = "Flying Circus"
3
4  >>> info = autor + ': ' + buch
5  >>> print(info)
6  Monty Python: Flying Circus
```

Zahlen und Strings

```
1  >>> autor = "Monty Python"
2  >>> buch = "Flying Circus"
3
4  >>> info = autor + ' ' + buch
5  >>> info = info + "(TV-Serie " + 1969 + '-' + 1974 + ')'
6  Traceback (most recent call last):
7    File "<stdin>", line 1, in <module>
8  TypeError: can only concatenate str (not "int") to str
```

Behebung des Fehlers

```
1  >>> autor = "Monty Python"
2  >>> buch = "Flying Circus"
3
4  >>> info = autor + ' ' + buch
5
6  >>> info = info + "(TV-Serie " + str(1969) + '-' + str(1974) +
   ↪  ')'
```

x Wiederholungen von Zeichen

```
1  >>> 'ab' * 3
2  'abababab'
```

Ersetzung an der Position

```
1  >>> zitterrochen = "GATCAAGGGCTTTTATATAC"
2
3  >>> rochen = zitterrochen[:3] + 'G' + zitterrochen[4:]
4  >>> print(rochen)
5  GATGAAGGGCTTTTATATAC
```

Ersetzung des Elements

```
1  >>> reihe = 'abcABC'
2
3  >>> reihe.replace('c', 'x')
4  'abxABC'
5
6  >>> reihe.replace('y', 'x')
7  'abcABC'
```

Verknüpfung von Listenelementen

```
1  >>> autor = "Monty Python"
2  >>> buch = "Flying Circus"
3
4  >>> info = "".join([autor, ": ", buch])
5  >>> print(info)
6  Monty Python: Flying Circus
```

Weiteres Beispiel

```
1  >>> obstsorten = ("Bananen", "Äpfel", "Birnen")
2
3  >>> ",".join(obstsorten)
4  'Bananen,Äpfel,Birnen'
```

Erläuterung

```
1  >>> buchstaben = " ab "
2  >>> buchstaben.join("CD")
3  'C ab D'
```

- Jedes Zeichen in einem String ist ein Element in einer Sequenz.
- Das Objekt, links vom Punkt, definiert das Trennzeichen zwischen den Listenelementen.
- Der Methode `.join` werden die zu verknüpfenden Listenelemente übergeben.

Suche nach Teilstrings

```
1  >>> zeichen = "Monty Python:  Flying Circus"
2
3  >>> zeichen.index("Fly")
4  15
5
6  >>> zeichen.index("Brian")
7  Traceback (most recent call last):
8    File "<stdin>", line 1, in <module>
9  ValueError: substring not found
```

Wenn der String nicht gefunden wird, wird eine Exception ausgegeben.

Weitere Möglichkeit

```
1  >>> zeichen = "Monty Python:  Flying Circus"
2
3  >>> zeichen.find("Fly")
4  15
5
6  >>> zeichen.find("Brian")
7  -1
```

Wenn der String nicht gefunden wird, wird der Wert -1 ausgegeben.

Suche das nächste Element

```
1  zitterrochen = "GATCAAGGGCTTTTATATAC"
2  suchbuchstabe = 'A'
3  pos = -1
4
5  while(True):
6      if ((pos + 1) >= len(zitterrochen)):
7              print("Ende des Strings.")
8              break
9      pos = zitterrochen.find(suchbuchstabe, pos + 1)
10     if (pos == -1):
11             print("Nicht vorhanden.")
12             break
```

Groß- und Kleinbuchstaben

```
1  >>> reihe = 'abcABC'
2
3  >>> reihe.lower()
4  'abcabc'
5  >>> reihe.upper()
6  'ABCABC'
```

Eigenschaften der Zeichenkette

```
1  >>> reihe = 'abc123'
2
3  >>> reihe.isalpha()
4  False
5  >>> reihe.isalnum()
6  True
7  >>> reihe.isdigit()
8  False
9  >>> reihe.isnumeric()
10 False
```

Übungszettel

Speicherung der Aussagen in einer Sequenz

Taschenbuch "Titel". Preis: 9,98 €

Wasserstoff, Magnesium, Kalium

Kurs: Einführung in Python

Programmieraufgabe

Suche nach Nachnamen in einer Telefonliste

Folgende Telefonliste ist gegeben:
telefonliste = [["Ruth Baumgartner", 4567], ["Herbert Schulz", 4578], [Ërika Müller", 4677], ["Hans Scholz", 4634], ["Maria Meyer", 4523], ["Richard Beinweil", 4513]]

Aus diese Liste werden alle Personen, deren Nachname mit "M"beginnen, herausgefiltert.

Ein- und Ausgabe

Lernziel:

- Lesen und Schreiben von Werten in die Shell
- Kommunikation mit dem Nutzer
- Formatierung von Strings für die Ausgabe

Ausgabe von Text in eine Konsole

```
1  >>> pi = 3.1415
2  >>> radius = 2
3  >>> print("Fläche", (pi * (radius * radius)), sep=": ",
   ↪  end='\n')
4  Fläche: 12.566
```

. . . in Python

None = print(par01, par02, . . . , sep=separator, end=end)

- Funktion, die im Standardumfang von Python definiert ist
- Kein Schlüsselwort in Python

Name der Funktion

None = print(par01, par02, . . . , sep=separator, end=end)

- `print` = Ausdrucken von etwas
- Eindeutiger Name eines definierten Stückchens Code

Aufruf der Funktion

None = print(par01, par02, . . . , sep=separator, end=end)

- Durch den Programmnamen `print`
- Beachtung der Groß- und Kleinschreibung

Parameterliste der Funktion

None = print(par01, par02, . . . , sep=separator, end=end)

- Beginn und Ende mit den runden Klammern
- Folgt direkt dem Funktionsnamen
- Für die Ausführung des Codes zwingend und optional erforderliche Werte

Parameter in der Liste

None = print(par01, par02, . . . , sep=separator, end=end)

- Trennung durch ein Komma.
- Anzahl der Parameter in Abängigkeit der Definition der Funktion.
- Optionale und nicht optionale Parameter.

Nicht optionale Parameter von print()

None = print(par01, par02, . . . , sep=separator, end=end)

- Mindestens ein Parameter, der ausgedruckt wird
- Die auszudruckende Parameter werden mit dem Parameter `sep` verknüpft

Optionale Parameter von print()

None = print(par01, par02, . . . , sep=separator, end=end)

- Können beim Start übergeben werden, müssen aber nicht.
- `sep`. Trennzeichen zwischen den zu verknüpfenden Parameter. Standardmäßig: Leerzeichen.
- `end`. Ende-Zeichen. Standardmäßig: `n` (neue Zeile).

Ausdruck von einem Wert

```
1  >>> pi = 3.1415
2  >>> radius = 2
3  >>> flaeche = pi * (radius * radius)
4  >>> print(flaeche)
5  12.566
6  >>> print((pi * (radius * radius)))
7  12.566
```

Ausdruck von x Parametern

```
1  >>> pi = 3.1415
2  >>> radius = 2
3  >>> flaeche = pi * (radius * radius)
4  >>> print(pi, "*", radius, "*", radius, "=", flaeche)
5  3.1415 * 2 * 2 = 12.566
```

Änderung des Trennzeichens

```
1  >>> print(0.72, 1.23, 4.5, sep='; ')
2  0.72; 1.23; 4.5
```

Optionale Parameter als Key-Value-Paare

```
1  >>> print(0.72, 1.23, 4.5, sep='; ')
2  0.72; 1.23; 4.5
```

- `name = wert`.
- Dem Schlüsselwort `parametername` wird ein beliebiger Wert zugewiesen.
- Die Reihenfolge der Parameter spielt bei der Nutzung von benannten Argumenten keine Rolle.

Änderung des Ende-Zeichens

```
1  messwerte = (0.72, 1.23, 4.5)
2  anzahl = 1
3
4  for element in messwerte:
5          if (anzahl != len(messwerte)):
6                  print(element, end='\t')
7          else:
8                  print(element)
9          anzahl = anzahl + 1
```

Ausdruck von Zeichenketten

```
1  >>> print("Name des Kunden: 'Hans Mustermann'")
2  Name des Kunden: 'Hans Mustermann'
3  >>> print('Name des Kunden: "Elfriede Musterfrau"')
4  Name des Kunden: "Elfriede Musterfrau"
```

Ausdruck von formatierten Zeichenketten

```
1  >>> formatString = "Temp. zwischen {0} °C und {1} °C"
2  >>> ausgabe = formatString.format(37.5, 39.4)
3  >>> print(ausgabe)
4  Temp. zwischen 37.5 °C und 39.4 °C
```

Methode eines Objekts

```
1  >>> ausgabe = formatString.format(varA, varB)
2  Traceback (most recent call last):
3    File "<stdin>", line 1, in <module>
4  NameError: name 'varA' is not defined
```

- Funktionen, die in der Klasse definiert sind, auf der das Objekt basiert
- Beschreiben das Verhalten von Objekten
- Verändern den Status eines Objekts
- Hinweise zur format-Methode siehe https://www.w3schools.com/python/ref_string_format.asp

Aufruf der Methode

```
ausgabe = formatString.format(varA, varB)
```

- Verbindung von Objekt und Methode durch den Punkt.
- Anwendung der Methode (rechts vom Punkt, format) auf das Objekt (links vom Punkt, formatString).
- Hinweis: Die Methode ist in einer Klasse definiert. Das Objekt basiert auf dieser Klasse.

Objekt „Formatstring“

```
1  >>> formatString = "Temp. zwischen {0} °C und {1} °C"
2  >>> formatString.format(37.5, 39.4)
3  'Temp. zwischen 37.5 °C und 39.4 °C'
```

besteht aus

- einen konstanten Teil: Literale, die nicht verändert werden.
- aus veränderbaren Teilen (Platzhalter): Ganzzahlen in geschweiften Klammern, beginnend mit 1. Identifikation des Parameters in der Parameterliste.

Name der Methode

```
ausgabe = formatString.format(varA, varB)
```

- Eindeutig in der Vorlage.
- Beachtung von Groß- und Kleinschreibung.

Parameterliste der Methode

```
ausgabe = formatString.format(varA, varB)
```

- Beginn und Ende der Parameterliste: Runde Klammern.
- Trennzeichen zwischen den Parametern: Komma.
- Anzahl der Elemente in der Liste in Abhängigkeit der Platzhalter im Formatstring. Kann leer sein.

Zuordnung Parameter und Platzhalter

```
formatString = "Temp. zwischen 0 °C und 1 °C"formatString.format(37.5, 39.4)
```

- Der erste Parameter ersetzt den Platzhalter 0.Der zweite Parameter den Platzhalter 1 und so weiter.
- Jede Nummer eines Platzhalters definiert die Position eines Parameters in der Liste.
- Die Anzahl der Parameter entspricht der Anzahl der Platzhalter.

Rückgabewert der Methode

```
ausgabe = formatString.format(varA, varB)
```

- Rückmeldung an den Aufrufer
- `None` oder ein Wert in Bezug auf das beschriebene Verhalten
- `format`: Der formatierte String

Ausrichtung der Platzhalter

```
>>> str01 = "{0:<}".format("abc") # Linksbündig
>>> str02 = "{0:>}".format("abc") # Rechtsbündig
>>> str03 = "{0:^}".format("abc") # Zentriert

>>> print(str01, '\n', str02, '\n', str03)
abc
 abc
 abc
```

Zahlendarstellung

```
1  >>> "{0:d}".format(0xAF9)
2  '2809'
3  >>> "{0:x}".format(345)
4  '159'
5  >>> "{0:e}".format(1237.545)
6  '1.237545e+03'
7  >>> "{0:f}".format(1237.545)
8  '1237.545000'
```

Feldbreite

```
>>> "{0:10}".format(1237.545)
'  1237.545'
>>> "{0:_<10}".format(1237.545)
'1237.545__'

>>> "{0:6}".format(1237)
'  1237'
>>> "{0:_>6}".format(1237)
'__1237'

>>> "{0:6}".format("abc")
'abc   '
```

Genauigkeit

```
1  >>> "{0:10.2}".format(1237.545)
2  '   1.2e+03'
3
4  >>> "{0:6.2}".format(1237)
5  Traceback (most recent call last):
6    File "<stdin>", line 1, in <module>
7  ValueError: Precision not allowed in integer format specifier
8
9  >>> "{0:6.2}".format("abc")
10 'ab    '
```

Formatspezifikation

```
1  >>> "{0:?>6.2f}".format(1237.3456)
2  '1237.35'
3
4  >>> "{0:?>10.2f}".format(1237.3456)
5  '???1237.35'
6
7  >>> "{0:?>10.2f}".format(1237)
8  '???1237.00'
9
10 >>> "{0:?>10.2s}".format("abc")
11 '????????ab'
```

Weitere Beispiele siehe
https://docs.python.org/3.4/library/string.html#formatspec.

Ausgabe in eine Datei

```
datei = open("daten.txt","w")
print(234567.545, file=datei)
print(1237.545, file=datei)
print(3456.545, file=datei)
datei.close()
```

Öffnen einer Datei

```
datei = open("daten.txt", "w")
```

- Welche Datei wird wie geöffnet?
- In Bezug auf `print()`: w. Schreiben.

Schließen einer Datei

```
datei.close()
```

- Objekt `datei`: Verweis auf die zu schließende Datei.
- Jede geöffnete Dateien muß geschlossen werden.

Eingabe in eine Konsole

```
strTemperatur = input("Temperatur: ")
```

- Lesen von numerischen und alphanumerischen Zeichen
- Lesen von Eingaben mit Hilfe der Tastatur

Eingabe des Nutzers

- Direkt im Anschluß des Eingabe-Prompts tätigt der Nutzer die Eingabe.
- Beendigung der Eingabe durch Return (Eingabetaste). Die Eingabe wird beendet.

Lesen der Eingabe in Python

```
strTemperatur = input("Temperatur: ")

variable = input(prompt)
```

- Die Funktion liest die Eingabe des Nutzers als String.
- `prompt`. Optional. Hinweis zur Eingabe für den Nutzer.

String → Gleitkommazahl

```
strTemperatur = input("Temperatur: ")

strTemperatur.replace(",", ".")
fltTemperatur = float(strTemperatur)
```

String → Ganzzahl

```
strMonat = input("Monat, numerisch: ")

if strMonat.isdecimal() == True:
        intMonat = int(strMonat)
```

Mehrere Zahlen in einer Zeile

```
strMonat = input("Monat, numerisch: ")

lstMonat = strMonat.split(" ")

for element in lstMonat:
        if element.isdecimal() == True:
                intMonat = int(strMonat)
```

Programmieraufgabe

Berechnung des Benzinverbrauchs

Ein Programm berechnet mit Hilfe der Formel $((Liter * 100)/Kilometer)$ den durchschnittlichen Benzinverbrauch eines Autos pro 100 Kilometer.
Die Liter-Angabe und die Kilometerangabe werden von der Konsole eingelesen. Sobald der Benutzer für die Liter- oder Kilometerangabe einen Wert von 0 eingibt, wird die Berechnung gestoppt.
Der durchschnittliche Benzinverbrauch wird auf der Konsole ausgegeben.

Programmieraufgabe

Erfassung von Laufzeiten

Bei einem 10 km Lauf werden folgende Informationen zu den einzelnen Läufern durch Eingabe am Bildschirm erfasst: Startnummer, Laufzeit, Vorname und Nachname. Der Name des Läufers und die Laufzeit werden in Form einer Liste gespeichert. Die Informationen zu allen Läufern werden in einem Dictionary gespeichert. Die Startnummer dient als Schlüssel. Die Läufer-Informationen werden als Wert in dem Dictionary gespeichert. Nach dem Einlesevorgang werden alle Informationen am Bildschirm ausgegeben.

Programmieraufgabe

Der Computer simuliert den Wurf eines Würfels wie zum Beispiel in dem Spiel „Mensch ärgere dich nicht“.
Der Nutzer hat drei Versuche, die gewürfelte Zahl zu raten. Der Nutzer gibt seine Zahl in die Shell ein.

Lösung: Rate Zahl

```
versuch = 3
gewuerfelt = randrange(1, 7)

while versuch > 0:
        strGeraten = input("Raten Sie eine Zahl von 1 bis 6: ")
        geraten = int(strGeraten)

        if (geraten <1) or (geraten > 6):
                print("Falsche Würfelzahl")

        elif (geraten == gewuerfelt):
                print("Hurra. Gewonnen.")
                break

        else:
                print("Leider falsch geraten.")
        versuch = versuch - 1
else:
```

Lösung: Beginn eines neuen Spiels

```
from random import randrange

antwort = "y"
while (antwort == "y") or (antwort == "j"):

        # Code: Rate Zahl

        eingabe = input("Möchten Sie ein weiteres Spiel starten? (y
        → / j) ")
        antwort = eingabe.lower()
```

Schreiben von Funktionen

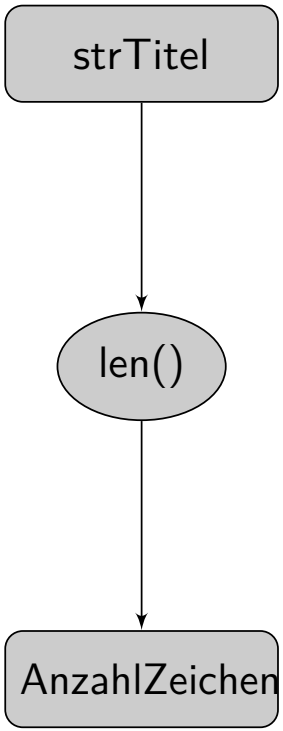
Lernziel:

- Wiederverwendung von Code zu einem bestimmten Thema
- Abbildung einer bestimmten benötigten Handlung mit Hilfe von Code
- Definition von optionalen und nicht optionalen Startwerte für eine bestimmte Funktionalität
- Rückgabe eines Status oder Ergebnisses nach Abschluss der, in dem Code beschriebenen Handlung

Funktionen . . .

- bilden eine bestimmte Aufgabe ab.
- ermöglichen gleiche Handlung in verschiedenen Kontexten einmalig zu definieren.
- kapseln kleine Mengen lauffähigen Code, der zu größeren Blöcken zusammengesetzt werden kann.

Beispiel




```
1  >>> strTitel = "Python"
2  >>> len(strTitel)
3  6
```

Built-in Funktionen

```
1  >>> strTitel = "Python"
2  >>> len(strTitel)
3  6
```

- Integration im Standardumfang von Python
- Siehe https://docs.python.org/3/library/functions.html

Aufruf von Funktion

```
1  >>> strTitel = "Python"
2  >>> len(strTitel)
3  6
```

- Aufruf: `funktionsname(par01, par02, ...)`
- Beachtung der Groß- und Kleinschreibung beim Aufruf

Parameterliste

```
1  >>> strTitel = "Python"
2  >>> len(strTitel)
3  6
```

- Begrenzung durch runde Klammer
- Anzahl und Typ der Parameter in Abhängigkeit der Defintion
- Kann leer sein

Parameter

```
1  >>> strTitel = "Python"
2  >>> len(strTitel)
3  6
```

- Trennung durch das Komma
- Optional oder nicht optional in Abhängigkeit der Definition

Rückgabewert

```
1  >>> strTitel = "Python"
2  >>> anzahl = len(strTitel)
3  >>> print("Rückgabewert: ", anzahl)
4  Rückgabewert:  6
```

- Jede Funktion gibt einen Wert zurück.
- `None` oder einen Wert in Abhängigkeit der Handlung.
- Kann in einer Variablen zur Weiterverarbeitung gespeichert werden.

Bottom-Up-Konstruktion von Funktionen

- Vorhandene Funktionen werden genutzt. Kleinere benötigte Funktionen werden geschrieben.
- Die, in den Funktionen, abgebildeten Handlungen werden in x Schritten zu Modulen zusammengesetzt. Diese Module bilden eine Aktivität vollständig ab.
- Am Ende bilden alle Module gemeinsam die Gesamtaufgabe ab.

Top-Down-Konstruktion von Funktionen

- Zuerst wird die Gesamtaufgabe skizziert.
- Diese Gesamtaufgabe wird in x Schritten in immer kleinere Module zerlegt.
- Am Ende der Planung sind diese Module so klein, dass sie mit Hilfe von Funktionen abgebildet werden können.

Benutzerdefinierte Funktionen in Python

```
1  def taschenrechner():
2          ergebnis = addition(3.5, 4)
3          print("Addition: ", ergebnis)
4
5  def addition(wertL, wertR):
6          summe = wertL + wertR
7          return summe
8
9  def subtraktion(wertL, wertR):
10     pass
```

Funktionskopf

```
def addition(wertL, wertR):
```

- Wie wird die Funktion aufgerufen?
- Beginn mit dem Schlüsselwort `def`.
- Beendigung mit einem Doppelpunkt.

Aufbau des Funktionskopfes

```
def addition(wertL, wertR):
```

- Beginn mit dem Schlüsselwort `def`.
- Dem Schlüsselwort folgt der frei wählbare Funktionsname.
- Dem Funktionsname folgt die Parameterliste.
- Der Funktionskopf endet mit dem Doppelpunkt.

Hinweise zum Funktionsnamen

- Zusammengesetzt aus den Buchstaben a...z, A...Z, den Zahlen 0...9 und dem Unterstrich
- Beginn mit einem Kleinbuchstaben
- Beachtung der Groß- und Kleinschreibung
- Einmalig in der Python-Datei
- Der Name spiegelt die Handlung wieder

Funktionsrumpf

```
1  def addition(wertL, wertR):
2          summe = wertL + wertR
3          return summe
```

- Eingerückte Zeilen unterhalb des Funktionskopfes
- Kapselung der zu beschreibenden Handlung

Leerer Funktionsrumpf

```
1  def subtraktion(wertL, wertR):
2      pass
```

- Jeder Funktionsrumpf besteht aus mindestens einer Zeile.
- `pass`. Die Handlung wird zu einem späteren Zeitpunkt codiert.

Funktionen ohne Rückgabewert

```
1  def addition(wertL, wertR):
2          summe = wertL + wertR
```

- Jede Funktion gibt einen Wert zurück.
- Die obige Funktion gibt `None` zurück.
- Der Rückgabewert ist nicht vom Entwickler definiert.

Funktionen mit Rückgabewert

```
1  def addition(wertL, wertR):
2          summe = wertL + wertR
3          return summe
```

- `return wert.`
- Rückgabe eines Wertes von einem beliebigen Datentyp.
- Der Wert kann durch einen Ausdruck berechnet werden.

. . . in Abhängigkeit einer Bedingung

```
1  def division(wertL, wertR):
2          if (wertR > 0):
3                  return(wertL / wertR)
4          else:
5                  return 0
```

- Beliebige viele return-Anweisungen.
- Rückgabe von Werten in Abhängigkeit der if-Anweisung.

Parameterliste im Funktionskopf

```
def addition(wertL, wertR):
```

- Folgt direkt dem Funktionsnamen
- Begrenzung durch die runden Klammern
- Kann beliebig viele oder keinen Parameter enthalten

Leere Parameterliste

```
def taschenrechner():
```

- Vorbelegung von Variablen, um den Code starten zu können
- Voraussetzungen für den Start der Aktivität

Funktionsaufruf

```
ergebnis = taschenrechner()
taschenrechner()
```

- . . . in der Python-Shell nach dem Starten des Moduls: Aufruf der Start-Funktion in einem Modul.
- . . . in einer Funktion: Verzweigung in die verschiedenen Teilaufgaben der Gesamtaufgabe.

Aufbau der Anweisung

```
rueckgabe = funktionsname()
```

- Aufruf durch den Funktionsnamen so wie im Funktionskopf definiert
- Die Argumentliste folge dem Funktionsnamen
- Wenn die Parameterliste im Funktionskopf leer ist, ist auch die Argumentliste leer.

Hinweise

```
rueckgabe = funktionsname()
```

- Beachtung der Groß- und Kleinschreibung beim Funktionsaufruf.
- Der Rückgabewert kann, muss aber nicht einer Variablen zur Weiterverarbeitung zugewiesen werden.

Nicht optionale Parameter

```
def addition(wertL, wertR):
```

- Trennung durch ein Komma
- Frei wählbarer Parametername
- Zwingend erforderliche Startwerte
- Müssen beim Aufruf der Funktion gesetzt werden

Aufruf der Funktion

```
ergebnis = addition(3.5, 4)

wert01=0.5
wert02=1.4
ergebnis = addition(wert01, wert02)
```

- Trennung durch ein Komma
- Anzahl der Argumente = mindestens Anzahl der nicht optionalen Parameter.
- Bei einer Nicht-Übergabe Anzeige der Meldung „TypeError: funcion() missing x required positional argument: 'name'“

Positionsargumente

```
ergebnis = addition(3.5, 4)
```

- positional argument.
- Zuordnung in Abhängigkeit der Position in der Parameter- und Argumentliste.
- Zuordnung der Elemente von links nach rechts.
- Das erste Argument wird dem ersten Parameter zugeordnet. Das zweite Argument dem zweiten Parameter und so weiter.

Optionale Parameter

```
def addition(wertL, wertR, wert=0):
```

- Trennung durch ein Komma
- Definition als Schlüssel-Wert-Paare. Frei wählbarer Name für den Schlüssel.
- Beim Aufruf kann ein Wert übergeben werden, muss aber nicht.
- Auflistung aller nicht-optionalen Parameter und dann der optionalen Parameter.

. . . als Schlüssel-Wert-Paare definieren

```
def addition(wertL, wertR, wert=0):
```

- `key=value`.
- `key`. Name des Parameters.
- `value`. Der am häufigsten vorkommende Wert bei Zahlen und boolschen Werten. Sequenzen und Dictionaries sollten mit dem Wert `None` vorbelegt werden.

Aufruf der Funktion

```
ergebnis = addition(3.5, 4)
ergebnis = addition(3.5, 4, 6)
```

- Zuerst werden die nicht optionalen Argumente definiert.
- Anschließend werden nur die benötigten optionalen Argumente definiert.

Nutzung von Schlüsselwortargumenten

```
ergebnis = addition(wertR=3.5, wertL=4)
ergebnis = addition(wertL=3.5, wertR=4, wert=6)
```

- keyword argument. Benannte Argumente.
- Zuordnung in der Form `key=value`.
- Die benötigen Argumente werden unabhängig von der Reihenfolge gelistet.

Schlüssel

key=value

- Parametername im Funktionskopf der aufzurufenden Funktion
- Beachtung der Groß- und Kleinschreibung
- Definition links vom Gleichheitszeichen

Wert

key=value

- Ein beliebiger Wert, der von den Anweisungen im Funktionsrumpf interpretiert werden kann.
- Nicht optionalen Argumenten muss ein Wert übergeben werden.
- Optionalen Argumente kann ein Wert übergeben werden. Falls kein Wert beim Aufruf übergeben wird, wird der im Funktionskopf angegebene Standardwert genutzt.

Liste von x Werten

```
1  def addieren(wertL, wertR, *args):
2          summe = wertL + wertR
3
4          if (len(args) > 0):
5                  for element in args:
6                          summe = summe + float(element)
7
8          return summe
9
10 ergebnis = addieren(1, 2)
11 ergebnis = addieren(1, 2, 3, 4, 5)
```

Erläuterung

- Zusammenfassung von beliebig vielen optionalen Werten.
- Der Namen ist frei wählbar, beginnt aber immer mit einem Sternchen.
- Das Sternchen kennzeichnet den Parameter als Tupel.

Liste von x Schlüssel-Wert-Paare

```
1  def ausgabeTemperatur(**kwargs):
2      for key, value in kwargs.items():
3          if (key == "Morgen"):
4              print("Morgentemperatur: ", value)
5          elif (key == "Mittag"):
6              print("Mittagstemperatur: ", value)
7          elif (key == "Abend"):
8              print("Abendtemperatur: ", value)
9          else:
10             print("Temperatur: ", value)
11
12 tag = {"Morgen":2.3, "Mittag":14.5, "Abend":6.5, "A":3.4,
   ↪  "B":5.6}
13 ausgabeTemperatur(**tag)
```

Erläuterung

- Zusammenfassung von beliebig vielen optionalen Schlüssel-Wert-Werten.
- Der Namen ist frei wählbar, beginnt aber immer mit zwei Sternchen.
- Die zwei Sternchen kennzeichnet den Parameter sowohl als auch das Argument als Dictonary.

Lokale Variablen

```
1  def addition(wertL, wertR):
2          summe = wertL + wertR
3          return summe
```

- Parameter und Variablen in einer Funktion
- Nutzung nur in der Funktion, in der sie definiert sind

Modulvariablen

```
1  glExponent = 2
2
3  def potenzieren(basis, exponent= glExponent):
4          ergebnis = basis ** exponent
5          return ergebnis
6
7  ergebnis = potenzieren(2)
8  ergebnis = potenzieren(2, 3)
```

- Globale Variablen
- Nutzung in jeder Funktion in dem Modul
- Nur lesender Zugriff

Nutzung in einer Funktion

```
1  glFaktor = 4
2
3  def multiplikation(wert, faktor = None):
4      global glFaktor
5      if faktor != None:
6          glFaktor = faktor
7      return (wert * glFaktor)
8
9
10 ergebnis  = multiplikation(2)
11 ergebnis = multiplikation(2, 3)
```

Erläuterung

- `global`: Lesen und Verändern einer globalen Variablen in einer Funktion
- Ohne Angabe des Schlüsselwortes: Anzeige des Fehlers „UnboundLocalError: local variable '...' referenced before assignment“
- Nur lesender Zugriff

Programmieraufgabe

Geometrie-Rechner

Das Volumen ($\pi * r^2 * h$), die Mantelfläche ($(2 * \pi * r) * h$) und die Oberfläche ($2 * \pi * r * (h + r)$) eines Zylinders wird berechnet.
Um die Berechnung durchzuführen, wird der Radius (r) und die Höhe (h) als Ganz- oder Gleitkommazahlen vom Bildschirm eingelesen.
Das Ergebnis der Berechnungen „Volumen“, „Mantelfläche“ und „Oberfläche“ wird als Gleitkommazahl in der Shell ausgegeben.

Programmieraufgabe

Telefonliste

Die Namen von neuen Mitarbeitern sowie deren Telefonnummer werden in einem Telefonbuch eingefügt.
In dem Telefonbuch ist es möglich, Mitarbeiternamen zu suchen und die passende Telefonnummer auszugeben.
Mitarbeiter, deren Dienstverhältnis endet, werden aus dem Telefonbuch entfernt.

Fehler im Code

Lernziel:

- Syntaxfehler
- Laufzeitfehler
- Behandlung von Exceptions (Ausnahmen)

Syntaxfehler

- Verletzung der Syntax der Programmiersprache Python.
- Die Interpretation des Codes wird unterbrochen.

Fehlermeldung

```
1  >>> b = 'a
2    File "<stdin>", line 1
3      b = 'a
4            ^
5  SyntaxError: EOL while scanning string literal
```

- `Fehlerkategorie:Beschreibung`
- Zu welcher Kategorie gehört der Fehler?
- Welcher Fehler ist wo aufgetreten?

. . . im Code anzeigen

- Kennzeichnung durch einen roten Balken
- Bei Ausführung in der Python-Shell: Anzeige direkt unterhalb des Codes.
- Bei Ausführung der Codedatei: Anzeige in einem Dialogfenster.

Exception

- Laufzeitfehler. Ausnahmen.
- Fehler, der beim Ausführen des Codes auftritt.
- Situationsbedingter Abbruch während der Laufzeit des Programms.

Fehlermeldung

```
1  >>> 4 / 0
2  Traceback (most recent call last):
3    File "<stdin>", line 1, in <module>
4  ZeroDivisionError: division by zero
```

- Beginn mit dem Wort `Traceback`
- Wo wurde der Fehler ausgelöst?
- Welche Ausnahme wurde ausgelöst?

Sprung zum Auslöser in einer Datei

```
1  >>> 4 / 0
2  Traceback (most recent call last):
3    File "<stdin>", line 1, in <module>
4  ZeroDivisionError: division by zero
```

- Rechter Mausklick auf das Wort `line`.
- Go to file/line

Beschreibung des Fehlers

```
1  >>> 4 / 0
2  Traceback (most recent call last):
3    File "<stdin>", line 1, in <module>
4  ZeroDivisionError: division by zero
```

- `Fehlerkategorie:Beschreibung`
- Zu welcher Kategorie gehört der Fehler?
- Welcher Fehler ist aufgetreten?

Exception-Handling

```
1  zahlL = 4
2  zahlR = 0
3
4  try:
5      ergebnis = zahlL / zahlR
6
7  except ZeroDivisionError:
8      print("Eine Division durch Null ist nicht erlaubt.")
9
10 except:
11     print("Nicht bekannter Laufzeitfehler.")
```

Versuche . . .

zahlL = 4 zahlR = 0
try: ergebnis = zahlL / zahlR

- Einleitung mit dem Schlüsselwort `try`
- Versuche die eingerückten Anweisungen darunter auszuführen
- Kapselung von Anweisungen in einem Codeblock, die einen Fehler verursachen könnten

Falls ein Fehler auftritt

```
1  zahlL = 4
2  zahlR = 0
3
4  try:
5      ergebnis = zahlL / zahlR
6
7  except:
8      print("Nicht bekannter Laufzeitfehler.")
```

- `except:`
- Abfangen von allen aufgetretenen Laufzeitfehlern.

Abfangen eines bestimmten Fehlers

```
1  zahlL = 4
2  zahlR = 0
3
4  try:
5      ergebnis = zahlL / zahlR
6
7  except ZeroDivisionError:
8      print("Eine Division durch Null ist nicht erlaubt.")
```

- `except fehlerkategorie`
- Abfangen einer bestimmten Fehlerkategorie
- Beliebig viele Fehlerkategorien können abgefangen werden
- Siehe https://docs.python.org/3/library/exceptions.html

Hinweise

- Eine try-Anweisung muss von mindestens einer except-Anweisung gefolgt werden.
- Wenn eventuell auftretende Fehlerkategorien bekannt sind, sollten diese zuerst abgefangen werden.
- Zum Schluss sollten alle Fehler mit der Anweisung `except:` abgefangen werden, die nicht bekannt sind.

Falls kein Fehler auftritt . . .

```
1  zahlL = 4
2  zahlR = 0
3
4  try:
5      ergebnis = zahlL / zahlR
6
7  except ZeroDivisionError:
8      print("Eine Division durch Null ist nicht erlaubt.")
9
10 except:
11     print("Nicht bekannter Laufzeitfehler.")
12
13 else:
14         print("Ergebnis der Division: ", ergebnis)
```

Ausführung des else-Zweigs

- Die, zu `try` gehörenden Anweisungen wurden fehlerfrei ausgeführt.
- Der Codeblock wurde nicht durch `break` oder `return` vorzeitig abgebrochen.

Hinweise zum else-Zweig

- Optional
- Folgt immer nach allen except-Anweisungen

Muß immer ausgeführt werden

```
1  zahlL = 4
2  zahlR = 0
3
4  try:
5      ergebnis = zahlL / zahlR
6  except ZeroDivisionError:
7      print("Eine Division durch Null ist nicht erlaubt.")
8  except:
9      print("Nicht bekannter Laufzeitfehler.")
10 else:
11         print("Ergebnis der Division: ", ergebnis)
12 finally:
13         print("Ende")
14
```

Hinweise zum finally-Zweig

- Optional
- Folgt immer nach der else -Anweisung

Beispiel „ZeroDivisionError“

```
1  zahlL = 4
2  zahlR = 0
3
4  try:
5      ergebnis = zahlL / zahlR
6
7  except ZeroDivisionError:
8      print("Eine Division durch Null ist nicht erlaubt.")
```

Beispiel „ValueError“

```
1  strTemperatur = '3,4'
2
3  try:
4      temperatur = float(strTemperatur)
5
6  except ValueError:
7      print("Kann nicht als Zahl interpretiert werden.")
```

- Der Wert kann nicht entsprechend des Datentyps interpretiert werden.
- Der Wert ist in einer Sequenz nicht enthalten.
- Fehlerhafte Kodierung von Unicode-Zeichen.

Beispiel „TypeError“

```
1  summe = 0
2  ausgabe = ''
3
4  try:
5      summe = 5 + 6
6      ausgabe = 5 + ' + ' + 6 + ' = ' + summe
7
8  except TypeError:
9      print("Keine Verarbeitung möglich")
```

Beispiel „NameError“

```
1  zahlL = 5
2
3  try:
4      summe = zahlL + zahlR
5
6  except NameError:
7      print("Der Bezeichner ist nicht definiert.")
```

- Die genutzte Variable ist nicht definiert
- Die aufgerufene Funktion ist nicht definiert
- Nutzung von unqualifizierten Namen

Beispiel „IndexError“

```
1  buchstaben = ['a', 'b', 'c', 'd']
2  lastIndex = len(buchstaben)
3  index = 0
4
5  try:
6    while(index <= lastIndex):
7        print(buchstaben[index])
8        index += 1
9
10 except IndexError:
11     print("Die Unter- oder Obergrenze wurde überschritten.")
```

Beispiel „KeyError“

```
1  farbe = {'rot':255, 'gruen':0, 'blau':0}
2
3  try:
4      farbton = farbe["gelb"]
5
6  except KeyError:
7      print("Der Schlüssel ist nicht vorhanden")
```

Exceptions in Funktionen

```
def start():
        try:
                zahl = einlesen()
        except ValueError:
                print("Value cannot be converted.")
        else:
                quadrat = zahl * zahl
                print("Square: ", quadrat)
```

Weiterleitung von Exceptions

```
def einlesen():
        strZahl = input("Eingabe einer Zahl: ")

        try:
                zahl = float(strZahl)

        except:
                raise

        else:
                return zahl
```

Erläuterung

- `raise`: Leite die aktuelle Exception eine Stufe höher.
- Optional: Übergabe einer benutzerdefinierten Fehlermeldung.

Benutzerdefinierte Fehlermeldung

```
def einlesen():
        strZahl = input("Eingabe einer Zahl: ")

        try:
                zahl = float(strZahl)

        except:
                raise ValueError("Nur Zahlen sind erlaubt")

        else:
                return zahl
```

Programmieraufgabe

Fläche und Umfang eines Kreises

Die Fläche (π * r * r) und der Umfanges (2 * π * r) eines beliebig großen Kreises wird mit Hilfe von Funktionen abgebildet.
Für die Berechnung wird der Radius des Kreises vom Bildschirm eingelesen. Der Radius muss einen Wert größer Null haben.
Der Wert der Zahl π (3,14159) wird einer globalen Variablen zugewiesen.
Der Nutzer kann in Abhängigkeit eines Buchstabens entscheiden, ob die Fläche oder der Umfang des Kreies berechnet wird.
Fangen Sie eventuelle Fehler in dem Programmcode ab.

Module

Lernziel:

- Zusammenfassung von Code, der thematisch eine Einheit bildet
- Funktionen, die nicht im Standardumfang von Python implementiert sind, nutzen
- Aufrufen von vordefinierten „Module“ aus der Python Library

Was sind Module?

- Programmcode in einer Textdatei.
- Wiederverwendbares Skript, um Handlungen mit Hilfe einer Programmiersprache abzubilden.
- Datei mit der Endung „.py“

Standardmodule

- Alle vordefinierten Funktionen und so weiter, die nicht im Sprachumfang von Python enthalten.
- Sortiert nach Kategorien: https://docs.python.org/3/library/
- Alphabetische Sortierung: https://docs.python.org/3/py-modindex.html

. . . in Bezug auf das Laufzeitsystem

```
1  import sys
2
3  print("Durchsuchbare Pfade / Finden von Modulen in: ")
4  for pfad in sys.path:
5      print(pfad)
6
7  print("\n------------------------------")
8  print("Module im Standardumfang: ")
9
10 for modulName in sys.modules:
11     print(modulName)
12
13 print("\nPlattform: ", sys.platform)
14 print("\nPython-Version: ", sys.version)
```

Import des gesamten Moduls

```
1  >>> import sys
```

- `import modulname`.
- In den ersten Zeilen der Code-Datei werden die import-Anweisungen aufgelistet.
- Alle, in dem Modul definierten Funktionen und so weiter können im Code genutzt werden.

Aufruf der Elemente

```
modulname.funktion(par01, par02, ...)
modulname.variable
```

- Mit Hilfe des qualifizierten Names. Beschreibung des absoluten Pfads zu dem Element.
- Der Aufruf beschreibt vollständig den Pfad zu dem Element. Das Element ist an diesem Ort definiert.
- Der Punktoperator verbindet Module und die darin definierten Element genutzt.

Aufruf von Modulvariablen

```
1  >>> import sys
2
3  >>> sys.version
4  '3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927
   ↪  64 bit (AMD64)]'
```

- Globale Variablen, die in dem Modul definiert sind.
- Variablen haben keine Parameterliste!

Aufruf von Dictionaries und Sequenzen

```
1  >>> import sys
2
3  >>> sys.flags
4  sys.flags(debug=0, inspect=0, interactive=0, optimize=0,
   ↪  dont_write_bytecode=0, no_user_site=0, no_site=0,
   ↪  ignore_environment=0, verbose=0, bytes_warning=0, quiet=0,
   ↪  hash_randomization=1, isolated=0, dev_mode=False,
   ↪  utf8_mode=0)
```

- Name, der häufig mit „s“ endet.
- Der Name der globalen Variablen drückt die Mehrzahl der Elemente aus
- Dictionaries sowie Sequenzen haben keine Parameterliste!

Aufruf von Funktionen

```
1  >>> import sys
2
3  >>> sys.exc_info()
4  (None, None, None)
```

- Aufruf mit dem Funktionsname, definiert im Funktionskopf in dem Modul links vom Punktoperator.
- Die Argumentliste folgt direkt dem Funktionsname. Die Argumentliste beginnt und endet mit den runden Klammern. Wenn kein Argument übergeben wird, ist die Liste wie hier leer.
- Jede Funktion gibt einen Wert zurück, der einer Variablen zugewiesen werden kann.

Informationen zu einer Exception

```
1  import sys
2
3  def start():
4      try:
5          ergebnis =  3 / 0
6
7      except ZeroDivisionError:
8          print("Error-Type: ", sys.exc_info()[0])
9          print("Error-Wert / Beschreibung: ", sys.exc_info()[1])
10         print("Traceback: ", sys.exc_info()[2])
```

Funktion „exc_info“

- Informationen zu einer Exception in Form eines Tupels
- `sys.exc_info()[0]`. Angabe der Fehlerkategorie.
- `sys.exc_info()[1]`. Beschreibung des Fehlers.
- `sys.exc_info()[2]`. Traceback. Wo wurde die Ausnahme ausgelöst?

. . . in Bezug auf das Python-System

```
1  import os
2
3  print("\nBetriebssystem: ", os.name)
4  print("\nPfad der Code file: ", os.getcwd())
5  print("\nPfad zu os.py: ", os.__file__ )
```

Mathematische Funktionen

```
1  from math import sin, pi
2
3  sinus =  sin(pi / 2)
4  print("Sinus: ", sinus)
```

Einige Elemente importieren

```
1  >>> from math import sin, pi
```

- `from modulname import func1, func2, ...`
- Importiere die angegebenen Elemente aus dem Modul ...
- Nur die importieren Elemente können in der Codedatei genutzt werden. In diesem Beispiel: `sin` und `pi`.

Aufruf der importierten Elemente

```
1  from math import sin, pi
2
3  sinus =  sin(pi / 2)
4  print("Sinus: ", sinus)
```

Erläuterung

- Aufruf nur mit dem nicht qualifizierter Name.
- Mit Hilfe des, in dem Modul definierten Namens wird das Element aufgerufen.
- Ein Hinweis auf das Moduls, in dem das Element definiert ist, wird nicht gegeben.

Import aller Elemente

```
1  from math import *
2
3  sinus =  sin(pi / 2)
```

- Entspricht `import modulname`.
- Importiere alle Elemente aus dem angegebenen Modul ...
- Sternchen $\leftrightarrow$ alle Elemente

Zufallszahlen

```
1  import random
2
3  wuerfel = (1, 2, 3, 4, 5, 6)
4  random.randrange(1, len(wuerfel))
5
6  farbe = ["red", "green", "blue", "yellow", "black"]
7  random.choice(farbe)
```

„datetime“ - Datumsangaben

```
1  import datetime
2
3  zeitstempel = datetime.datetime
4  aktuell = zeitstempel.now()
5  zeitstempel.date(aktuell)
6  aktuell.year
7  aktuell.month
8  aktuell.day
```

„datetime“ - Zeitangaben

```
1  import datetime
2
3  zeitstempel = datetime.datetime
4  aktuell = zeitstempel.now()
5
6  zeitstempel.time(aktuell)
7  aktuell.hour
8  aktuell.minute
9  aktuell.utcnow()
```

„time“ - Datumsangaben

```
1  import time
2
3  time.strftime("%d.%m.%Y %H:%M:%S")
4  now = time.localtime()
5  now.tm_mday
6  now.tm_mon
7  now.tm_year
8  # Montag = 0
9  now.tm_wday
```

„time“ - Zeitangaben

```
1  import time
2
3  now = time.localtime()
4  now.tm_hour
5  now.tm_min
6  now.tm_sec
7  # Sommerzeit: 1; Winterzeit: 0
8  now.tm_isdst
```

Berechnung von Zeitangaben

```
1  import datetime
2
3  zeitstempel = datetime.datetime
4  aktuell = zeitstempel.now()
5  stunde = datetime.timedelta(hours=-1)
6  viertelStunde = datetime.timedelta(minutes=15)
7
8  aktuell + stunde
9  aktuell - viertelStunde
```

Berechnung von Datumsangaben

```
1  import datetime
2
3  zeitstempel = datetime.date(2020, 6, 28)
4  monat = datetime.timedelta(days=30)
5
6  zeitstempel - monat
```

. . . ab Python 3.x

```
import datetime
from dateutil.relativedelta import *

zeitstempel = datetime.date(2020, 6, 28)
zeitstempel - relativedelta(months=1)
```

Hierarchie von Modulen

```
from dateutil.relativedelta import *
```

- `modulname.modulname ...` bildet eine Hierarchie ab.
- In diesem Beispiel: In dem Ordner `dateutil` befindet sich ein Ordner `relativedelta`.
- Der Punktoperator trennt die Pfadelemente zu einem Modul.

Zeichnen mit einer Schildkröte

```
1  import turtle
2  import random
3  stern = turtle.Turtle()
4  screen = turtle.Screen()
5  farbenListe = ['black', 'red', 'green', 'yellow', 'blue']
6
7  for i in range(10):
8      farbe = random.choice(farbenListe)
9      stern.color(farbe)
10     stern.forward(200)
11     stern.right(144)
12
13 turtle.done()
```

Nutzung von regulären Expressions

```
1  import re
2
3  strEingabe = "Bananen zum Preis von 1,99"
4  seqEingabe =
   ↪  re.findall(r'[-+]?(\d+([.,]\d*)?|[.,]\d+)([eE][-+]?\d+)?',
   ↪  strEingabe)
5  print(seqEingabe)
```

Hinweise zum Aufbau des Ausdrucks:
https://docs.python.org/3/library/re.html

Dezimalzahlen

```
1  from decimal import Decimal
2
3  Decimal(34)
4  Decimal(0.1)
```

- Nutzung einer dezimalen Darstellung von Zahlen
- Anzeige des intern gespeicherten Nährungswertes
- Höhere Genauigkeit als Gleitkommazahlen

. . . aus Strings

```
1  from decimal import Decimal
2
3  Decimal('0.1')
```

Unendlich oder keine Nummer

```
1  from decimal import Decimal
2
3  Decimal("Infinity")
4  Decimal("-Infinity")
5  Decimal("NaN")
```

Mathematische Operatoren

```
1  from decimal import Decimal
2
3  Decimal('0.1') + Decimal('0.25')
4  Decimal('0.1') - Decimal('0.25')
5  Decimal('0.1') * Decimal('0.25')
6  Decimal('0.1') / Decimal('0.25')
7  Decimal('4.67') % Decimal('2')
8  Decimal('4.67') ** Decimal('2')
```

Mischung von Datentypen

```
1  from decimal import Decimal
2
3  ergebnis = Decimal('0.1') + 34
4  type(ergebnis)
```

Kontext einer Dezimalzahl

```
1  from decimal import Decimal
2  from decimal import getcontext
3
4  getcontext()
```

Genauigkeit

```
1  from decimal import Decimal
2  from decimal import getcontext
3
4  getcontext().prec = 3
5  Decimal("0.1234567")
6  Decimal("0.789645")
```

Rundung von Dezimalzahlen

```
1  from decimal import Decimal
2  from decimal import getcontext, ROUND_HALF_UP
3
4  getcontext().prec = 3
5  getcontext().rounding = ROUND_HALF_UP
6
7  Decimal("0.1234567")
8  Decimal("0.789645")
```

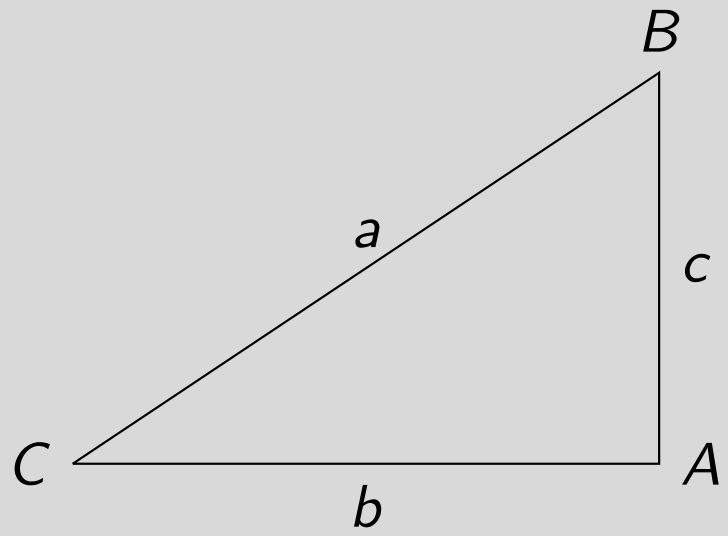
Abfangen von Exception

```
1  from decimal import Decimal
2  from decimal import InvalidOperation
3  from decimal import getcontext
4  import sys
5
6  try:
7          zahlR  = Decimal("Infinity")
8          zahlL = Decimal("0.0000005")
9          getcontext().prec = 3
10         result = Decimal(zahlL * zahlR)
11         print(zahlL, " * ", zahlR, " = ",  result)
12 except InvalidOperation:
13     zahlL = Decimal("NaN")
14 except:
15     print(sys.exc_info())
```

Programmieraufgabe

Geometrie

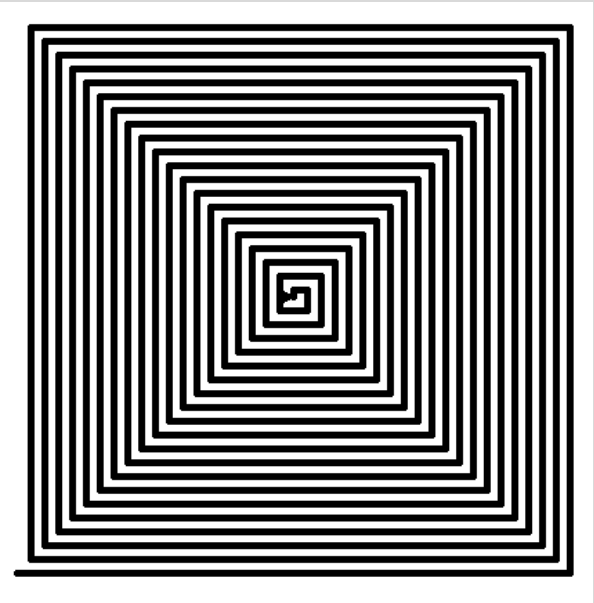
Berechnen Sie die Kante c durch Umstellung des Satz des Pythagoras $a^2 + b^2 = c^2$ mit Hilfe der Funktionen aus dem Modul math (https://docs.python.org/3/library/math.html).



Programmieraufgabe

Zeichnen von Spiralen

Zeichnen Sie folgenden abgebildete Spirale mit Hilfe des Moduls turtle (https://docs.python.org/3.3/library/turtle.html?highlight=turtle).



Programmieraufgabe

Der Computer simuliert den Wurf eines Würfels wie zum Beispiel in dem Spiel „Mensch ärgere dich nicht“.
Der Nutzer hat drei Versuche, die gewürfelte Zahl zu raten. Der Nutzer gibt seine Zahl in die Shell ein. Eventuell falsche Eingaben werden abgefangen.

Funktionen

- `Würfeln`. Der Computer wirft den Würfel.
- `Eingabe`. Der Nutzer gibt eine Zahl zum Vergleich ein.
- `Entspricht dem gewürfelten Wert?`. Entspricht die Eingabe der gewürfelten Zahl?
- `Rate`. Der Nutzer hat x Versuche, um die Zahl zu raten.
- `Start`. Hauptmenü.

Funktion „Würfeln“

```
from random import randrange

def wuerfeln():
        gewuerfelt = randrange(1, 7)

        return gewuerfelt
```

Funktion „Eingabe“

```
def raten():
        geraten = None
        strGeraten = input("Raten Sie eine Zahl von 1 bis 6: ")

        try:
                geraten = int(strGeraten)
        except ValueError:
                print("Falsche Eingabe. Nur Zahlen von 1 bis 6 sind
                → möglich.")
        except TypeError:
                print("Der eingegebene Datentyp kann nicht
                →konvertiert werden")
        except:
                print("Falsche Eingabe.")

        return geraten
```

Funktion „Entspricht dem gewürfelten Wert?“

```
from enum import Enum

class msgGame(Enum):
        Win = 1
        NotWin = 2
        WrongNumber = 3

def isGeratenGleichGewurfelt(intWurf, intRaten):
        if (intRaten <1) or (intRaten > 6):
                print("Falsche Würfelzahl")
                return msgGame.WrongNumber
        elif (intRaten == intWurf):
                print("Hurra Sie haben gewonnen.")
                return msgGame.Win
        else:
                return msgGame.NotWin
```

Funktion „Rate“

```
def rate(versuch, wurf):
        while versuch > 0:
                geraten = eingabe()

                if not(geraten is None):
                        blnStatus = isGeratenGleichGewurfelt(
                                                intWurf=wurf,
                                                intRaten=geraten)

                        if not(blnStatus is msgGame.WrongNumber):
                                if (blnStatus is msgGame.Win):
                                        break
                                else:
                                        versuch = versuch - 1
        else:
                print("Schade. Der Computer hat gewonnen.")
```

Funktion „Start“

```
def start():
        antwort = "y"

    while (antwort == "y") or (antwort == "j"):
        wurf = wuerfeln()
        print("Der Computer hat die Zahl ", wurf, " gewürfelt.")

        versuch = 3
                rate(versuch, wurf)

        eingabe = input("Möchten Sie ein weiteres Spiel starten? (y
        → / j) ")
        antwort = eingabe.lower()
```