# Programmieren mit Python

## Theorie

Version vom 21. Oktober 2025

# Inhaltsverzeichnis

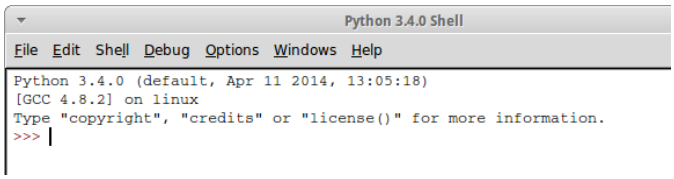
# 1 Einleitung

### Was ist Python?

- Python ist eine universelle, interpretierte Programmiersprache.
- Python wurde von Guido van Rossum zu Beginn der 1990er Jahre entwickelt.
- Der Name Python stammt von der gleichnamigen britischen Komikertruppe *Monty Python*.
- Wir verwenden Python 3.x; das sind Versionen, die nicht mehr zum Entwicklungszweig 2.x kompatibel sind. Die Unterschiede sind jedoch nicht sehr gravierend.

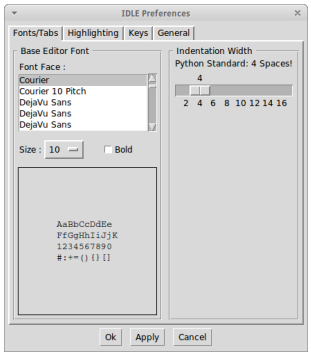
### IDLE

IDLE ist eine integrierte Entwicklungsumgebung (*Integrated Development Environment*, IDE) für Python. In IDLE kann man Python-Code interaktiv ausprobieren oder Programme in einem separaten Editor schreiben und dann zur Ausführung an die Python Shell senden.



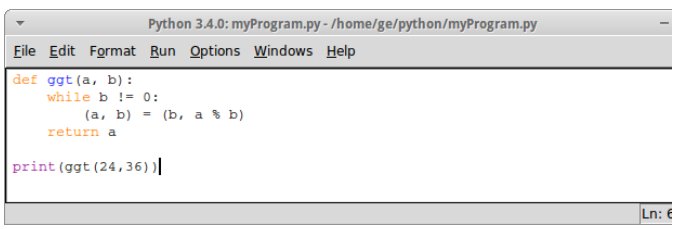

### Konfiguration von IDLE

Unter `Options/Configure IDLE...` lassen sich Schriftgrösse, Einrückungstiefe (standardmässig 4 Leerzeichen) oder spezielle Tastenkombinationen festlegen.




### Die Python-Shell

Mit `File/New File` wird eine leere Programmdatei geöffnet.



Speichert man die Datei unter einem Namen mit der Dateiendung `.py` ab, wird die farbliche Syntax-Hervorhebung eingeschaltet. *Beachte:* Diese Endung wird nicht automatisch angehängt.

Ein so erstelltes Python-Programm kann mit dem Befehl `Run/Run Module` oder mit der F5-Taste gestartet werden.

### Kommentare

In einem Python-Skript werden Kommentare mit einem Doppelkreuz (`#`) gekennzeichnet.

Kommentare haben zwei Funktionen:

- Dokumentation von Programmcode (wichtig!)
- Vorübergehendes Ausschalten „verdächtiger“ Programmzeilen, um Fehler lokalisieren zu können.

## 1.1 Edit-Run-Zyklus

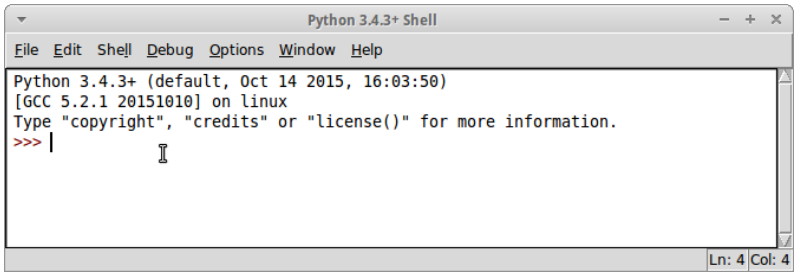
Die Shell ist ein Programm, das die Benutzereingaben entgegennimmt, nach der Betätigung der Enter-Taste den Code ausführt (sofern dies schon möglich ist) und allenfalls das Ergebnis ausgibt.

Dies funktioniert für Programme, die aus einer oder ganz wenigen Zeilen bestehen ganz gut.

Für grössere Projekte sollten Programme in einer eigenen Datei gespeichert werden, damit man sie erneut laufen lassen oder verändern kann.

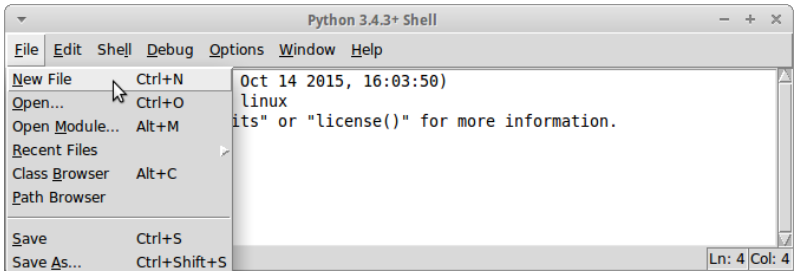
### Schritt 1

Starte die Python Entwicklungsumgebung IDLE.



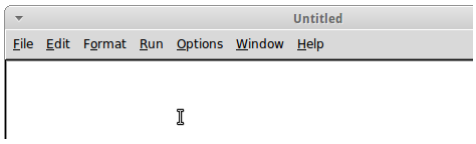
### Schritt 2

New File-Dialog starten.




### Schritt 3

Ein leeres Editorfenster erscheint.

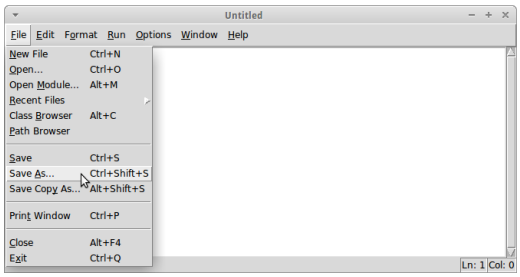



Ein Editor ist ein Programm mit dem man Textdateien schreiben und verändern kann.

Die Textdateien dürfen keine Formatierungen (wie bei Word) enthalten.

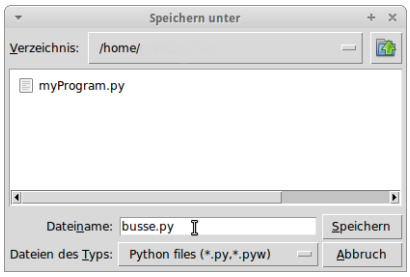
### Schritt 4

Die leere Datei sofort unter einem geeigneten Namen mit der Endung `.py` abspeichern.




Es ist von Vorteil, der Datei gleich zu Beginn einen Namen zu geben und sie in regelmässigen Abständen zu sichern.

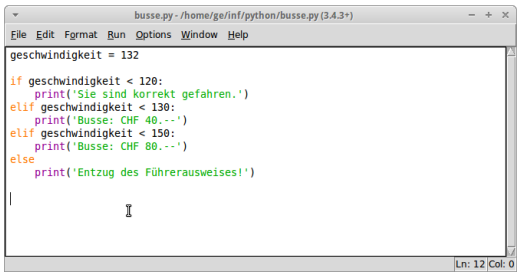
### Schritt 5




Die Endung `.py` wird nicht automatisch an die Datei angehängt. Das Programm würde zwar trotzdem funktionieren (die Endung ist im Grunde egal) aber man muss dann im Editor auf die Syntaxhervorhebung [siehe Schritt 6] verzichten.

### Schritt 6

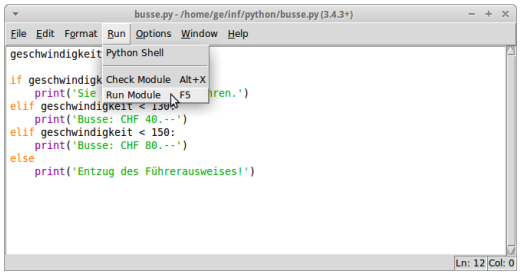
Das Programm kann jetzt geschrieben werden. Für die Einrückung (bei Verzweigungen, Schleifen und Funktionen) drückt man die Tabulator-Taste. Normalerweise wird dann der Text um vier Zeichen nach rechts eingerückt.




Eine Einrückung macht man mit der Backspace-Taste ⟵ wieder rückgängig.

### Schritt 7

Nachdem man das Programm (oder einen lauffähigen Teil) geschrieben hat, kann man es mit dem Run-Befehl (oder der Taste F5) ausführen.




### Schritt 8

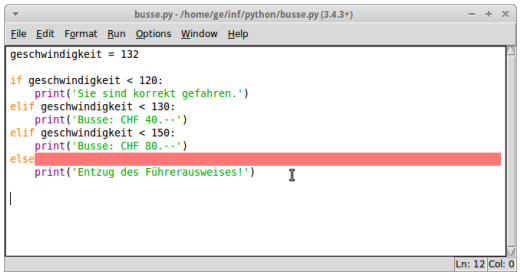
Python erkennt Syntaxfehler und bricht die Ausführung des Programms vorzeitig ab.


Syntaxfehler entstehen beispielsweise durch:

- fehlende Klammern,
- fehlende Doppelpunkte,
- falsch Einrückungen,
- falsch geschriebene Schlüsselwörter oder Bezeichner,
- ...

### Schritt 9

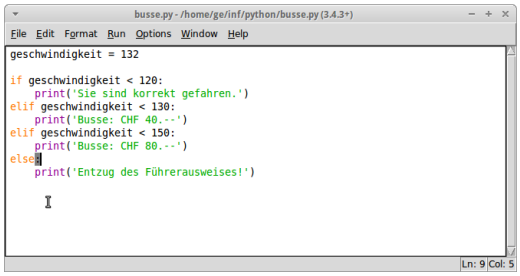
Die Entwicklungsumgebung versucht herauszufinden, wo der Fehler entstanden ist und zeigt die betreffende Zeile an.




Oft entdeckt Python den Fehler erst nach einer oder zwei Zeilen Code. Deshalb kann es sich manchmal lohnen, den Fehler etwas oberhalb der angegebenen Stelle zu suchen.

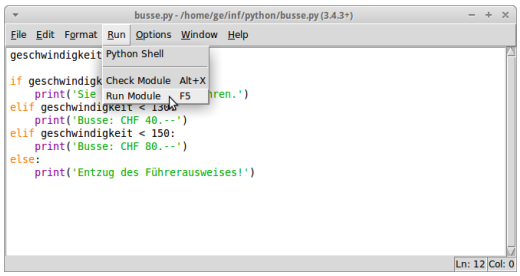
### Schritt 10

Der oder die Fehler können nun im Editor korrigiert werden.



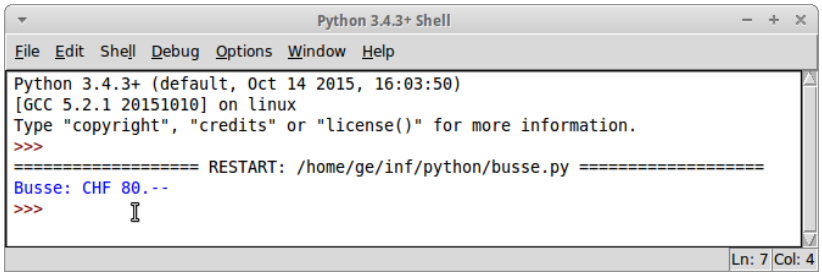

### Schritt 11

Das Programm wird erneut gestartet.




### Schritt 12

Diesmal läuft alles gut und die Ausgabe wird in der Shell angezeigt.



## 1.2 Fehler

### Syntax-Fehler

Können vom Python-Interpreter sofort bei der Prüfung des Codes entdeckt werden. Typische Syntax-Fehler sind falsch geschriebene Variablennamen und Schlüsselwörter, falsche Einrückungen oder eine falsche Anzahl von Parametern beim Aufruf von Funktionen.

### Laufzeit-Fehler

Werden vom Python-Interpreter erst bei der Ausführung des Programms entdeckt. Ein sehr häufiger Fehler in dieser Kategorie ist eine Division durch Null, die vom Programmierer nicht „abgefangen“ wurde.

### Logische Fehler

Dies sind Fehler, die der Programmierer begeht, ohne die Syntax-Regeln von Python zu verletzten. Logische Fehler entstehen z. B. dadurch, dass in einem arithmetischen Ausdruck nötige Klammern vergessen wurden. Das Programm läuft zwar ohne Fehlermeldung, liefert am Ende aber nicht das korrekte Resultat.

### Download und Hilfe

Ausgangspunkt für Alles rund um Python ist die Website

`https://docs.python.org`

Die Hilfeseiten (für die Versionen 3.x) erreicht man hier:

`https://docs.python.org/3/`

# 2 Arithmetik

### Operatoren

Für einfache Berechnungen stehen in Python folgende Operatoren zur Verfügung:

| Operator | Effekt |
|---|---|
| `+` | Addition |
| `-` | Subtraktion |
| `*` | Multiplikation |
| `/` | Division |
| `//` | ganzzahlige Division |
| `%` | Divisionsrest |
| `**` | Potenzieren (und Wurzelziehen) |

### Ganze Zahlen und Gleitkommazahlen

Python kennt bei bei numerischen Werten zwei *Datentypen*:

- *integer:* ganzen Zahlen, wie z. B. $-9$, $-1$, 0, 1, 2
- *float:* Gleitkommazahlen, wie z. B. $-10.75$, 0.0, 3.141592

Bei den Operationen `+`, `-`, `*`, `//` und `**` hat das Ergebnis den gleichen Datentyp wie die Operanden, falls beide den gleichen Datentyp aufweisen.

Haben die Operanden unterschiedliche Datentypen, ist der resultierende Datentyp immer `float`.

Die „normale“ Division (`/`) führt immer zu `float`.

### Beispiel 2.1

```
print(1234 + 5678)
```

### Beispiel 2.2

```
print(1234 - 5678)
```

**Beispiel 2.3**

```
print(1234 * 5678)
```

**Beispiel 2.4**

```
print(20 / 4)
```

**Beispiel 2.5**

```
print(20 // 7)
```

**Beispiel 2.6**

```
print(20 % 7)
```

**Beispiel 2.7**

```
print(2**3)
```

**Beispiel 2.8**

```
print(49**0.5)
```

### Hierarchie

Bezüglich der Rechenreihenfolge gelten die aus der Algebra bekannten Regeln:

1. Klammern
2. Potenzen
3. Multiplikation und Division
4. Addition und Subtraktion

### Beispiel 2.9

```
print(2 * 3 + 4 * 5)
```

### Beispiel 2.10

```
print(2.0 * (3 + 4) * 5)
```

### Casting

Die Funktionen

- `int(...)`
- `float(...)`

wandeln einen Datentyp so gut wie möglich in eine ganze Zahl bzw. Gleitkommazahl um.

### Beispiel 2.11

```
print(int(2.7))
```

**Beispiel 2.12**

```
print(float(42))
```

### Benutzereingaben mit `input(...)`

Benutzereingaben können mit der Funktion `input("...")` an das Programm übergeben werden. Diese Eingabe ist grundsätzlich eine Zeichenkette. Will man mit einer Eingabe rechnen, muss man sie mit `int(...)` oder `float(...)` in den gewünschten Zahlentyp umwandeln.

**Beispiel 2.13**

```
x = input("Geben Sie eine ganze Zahl ein: ")
# der Benutzer gibt 1234.8 ein (!)
x = int(x)
print(x)
```

# 3 Variablen

### Variablennamen

In Python gelten ein paar Regeln für die die Bildung von Variablennamen (*identifier*):

- Der Variablennamen muss aus Buchstaben, Ziffern oder Unterstrichen bestehen.
- Der Variablenname darf nicht mit einer Ziffer beginnen.
- Einige Wörter sind für besondere Zwecke reserviert.

```
False   class     finally  is        return
None    continue  for      lambda    try
True    def       from     nonlocal  while
and     del       global   not       with
as      elif      if       or        yield
assert  else      import   pass
break   except    in       raise
```

### Beispiel 3.1

Handelt es sich um einen korrekten Variablennamen?

(a) `radius`

(b) `4you`

(c) `radius2`

(d) `umsatz-2014`

(e) `anzahlPunkte`

(f) `class`

### Zuweisungen

Um einer Variable einen Wert zuzuweisen, verwendet man folgende Syntax:

`<variablenname> = <ausdruck>`

Das Gleichheitszeichen ist ein *Operator* und hat nichts mit einer Gleichung aus dem Algebra-Unterricht gemeinsam.

Dieser Operator wertet zunächst den Ausdruck rechts des Gleichheitszeichens aus. Wenn der Wert des Ausdrucks feststeht, wird dieser dem Variablennamen zugewiesen. *Beispiele:*

- `a = 5`
- `gruss = 'Hallo'`
- `x = 3 * 7 + 1`
- `x = x + 4`

### Löschen von Variablen

Variablen müssen normalerweise nicht gelöscht werden. Falls nötig, kann man dem Variablennamen einfach einen neuen Wert zuweisen.

```
radius = 2
...
radius = 5
...
```

### Zusammengesetzte Zuweisungen

Will man mit einer Variable rechnen und dann das Resultat wieder der bestehenden Variablen zuweisen, verwendet man eine zusammengesetzte Zuweisung (*augmented assignement*), bei der dem Gleichheitszeichen ein arithmetischer Operator vorangestellt ist.

```
a += 1 # a = a + 1
b -= 8 # b = b - 8
c *= 2 # c = c * 2
d /= 3 # d = d / 3
```

### Beispiel 3.2

Welche Ausgabe macht das folgende Codefragment?

```
a = 5
a -= 3
print(a)
```

### Beispiel 3.3

Welche Ausgabe macht das folgende Codefragment?

```
b = 7
b *= 3
print(b)
```

### Beispiel 3.4

Welche Ausgabe macht das folgende Codefragment?

```
c = 20
c /= 4
print(c)
```

### Beispiel 3.5

Welche Ausgabe macht das folgende Codefragment?

```
b = 7
b *= 3
print(b)
```

### Mehrfachzuweisungen

In Python sind Mehrfachzuweisungen möglich – ein Konzept, das es nicht in jeder Programmiersprache gibt.

```
(a, b, c) = (3, 2, 9)
```

Die Klammern können übrigens auch weggelassen werden.

Auf diese Weise ist auch ein Wertetausch bei zwei oder mehr Variablen möglich.

```
x = 7
y = 4
(x, y) = (y, x)
```

Am Ende dieses Codefragments gilt `x=4` und `y=7`.

### Beispiel 3.6

Welche Ausgabe macht das folgende Codefragment?

```
v, x, y, z = 3, 9, 8, 0
print(x)
```

### Beispiel 3.7

```
x, y, z = 5, 8, 4
y, z, x = x, y, z
print(x)
```

# 4 Wahrheitswerte

### Wahrheitswerte

Python kennt die zwei Wahrheitswerte `True` und `False`. Man beachte die Gross- und Kleinschreibung.

### Vergleichsoperatoren

Die Vergleichsoperatoren liefern jeweils einen Wahrheitswert zurück:

| **Operator** | `True`, **wenn ...** (`False` **sonst**) |
|---|---|
| `x == y` | $x = y$ |
| `x != y` | $x \neq y$ |
| `x < y` | $x < y$ |
| `x <= y` | $x \leq y$ |
| `x > y` | $x > y$ |
| `x >= y` | $x \geq y$ |

### Beispiel 4.1

(a) `5 < 4`

(b) `7 >= 7`

(c) `4 != 4`

(d) `1+8 < 5+6`

### Das logische NEIN

Die Verneinung (Negation) wird mit `not(...)` gebildet:

| **Ausdruck** | **Wert** |
|---|---|
| `not True` | `False` |
| `not False` | `True` |

### Das logische ODER

Das logische ODER (Disjunktion) wird mit `or` gebildet. Ein mit `or` gebildeter logischer Ausdruck ist genau dann wahr, wenn mindestens einer der beiden Operanden wahr ist.

| **Ausdruck** | **Wert** |
|---|---|
| `True or True` | `True` |
| `True or False` | `True` |
| `False or True` | `True` |
| `False or False` | `False` |

### Das logische UND

Das logische UND (Konjunktion) wird mit `and` gebildet. Ein mit `and` gebildeter logischer Ausdruck ist genau dann wahr, wenn beide Operanden wahr sind.

| **Ausdruck** | **Wert** |
|---|---|
| `True and True` | `True` |
| `True and False` | `False` |
| `False and True` | `False` |
| `False and False` | `False` |

### Hierarchie der logischen Operatoren

Wie bei den arithmetischen Operatoren gibt es auch bei den logischen Operatoren eine Hierarchie.

1. Klammern
2. `not`
3. `and`
4. `or`

Bei Operationen gleicher Stufe wird von links nach rechts ausgewertet.

### Beispiel 4.2

(a) `5 < 3 or 7 > 4`

(b) `not False or False`

(c) `not False and False`

(d) `(5 == 7) or (5 != 5)`

(e) `True and False and True and True and True`

Beim letzten Beispiel kann man sich die *Kurzschlusseigenschaft* der logischen Operatoren zunutze machen. Sobald der Wert des Ausdrucks feststeht, wird er nicht mehr weiter ausgewertet.

# 5 Verzweigungen

### Einfache Verzweigung

```
Code vor der Verzweigung

if <bedingung>:
    Code, wenn <bedingung> wahr
    Code, wenn <bedingung> wahr
    ...
    Code, wenn <bedingung> wahr

Code nach der Verzweigung
```

Der eingrückte Code (Block) muss um eine feste Anzahl Leerzeichen eingerückt sein. Üblich sind 4 Leerzeichen. Dies gilt auch für die folgenden Verzweigungstypen.

### Beispiel 5.1

```
pwd = input('Passwort: ')

if pwd != 'geheim123':
    exit('Kein Zugang!')

print('Willkommen!')
```

### Einfache Verzweigung mit Alternative

```
Code vor der Verzweigung
if <bedingung>:
    Code, wenn <bedingung> wahr
    ...
    Code, wenn <bedingung> wahr
else:
    Code, wenn <bedingung> falsch
    ...
    Code, wenn <bedingung> falsch
Code nach der Verzweigung
```

### Beispiel 5.2

```
import random

zufallszahl = random.randint(1,10)

if zufallszahl == 7:
    print('Sie haben gewonnen!')
else:
    print('Sie haben nicht gewonnen!')
```

### Mehrfache Verzweigung

```
Code vor der Verzweigung
if <bedingung1>:
    Code, wenn <bedingung1> wahr
    ...
elif <bedingung2>:
    Code, wenn <bedingung2> wahr
    ...
...
else:
    Code, wenn keine der obigen
    Bedingungen wahr ist
    ...
Code nach der Verzweigung
```

### Beispiel 5.3

```
geschwindigkeit = 132

if geschwindigkeit < 120:
    print('Sie sind korrekt gefahren.')
elif geschwindigkeit < 130:
    print('Busse: CHF 40.--')
elif geschwindigkeit < 150:
    print('Busse: CHF 80.--')
else:
    print('Entzug des Führerausweises!')
```

# 6 Schleifen

### Indexgesteuerte for-Schleife

```
for <var> in range(<a>, <b>):
    Codezeile
    Codezeile
    ...
Code nach der Schleife ...
```

Der Code im Schleifenblock (*Schleifenkörper*) wird jeweils für die ganzzahligen Werte `<a>`, `<a>+1`, `<a>+2`, ..., `<b>-2`, `<b>-1` von `<var>` ausgeführt.

Der Schleifenkörper muss um eine feste Anzahl Leerzeichen eingerückt sein. Üblich sind 4 Leerzeichen. Dies gilt auch für die folgenden Schleifentypen.

### Indexgesteuerte for-Schleife (Beispiel)

```
for i in range(0, 5):
    print(2*i)
```

Die Indexvariable (hier wurde `i` gewählt) läuft in Einerschritten von 0 bis 4(!).

Bei jedem Schleifendurchlauf wird das Doppelte der Indexvariable `i` ausgegeben.

Somit gibt das Programm die Zahlen 0, 2, 4, und 8 aus.

### Listengesteuerte for-Schleife

```
for <var> in <liste>
    Codezeile
    Codezeile
    ...
Code nach der Schleife ...
```

Der Code im *Schleifenkörper* wird für jedes Element `<var>` von `<liste>` ausgeführt.

### Listengesteuerte for-Schleife (Beispiel)

```
for e in [2, 0, -2, 9, -7]:
    if e > 0:
        print(e)
```

Die Schleifenvariable (hier wurde `e` gewählt) nimmt der Reihe nach jeden Wert in der Liste an. Also beim 1. Schleifendurchlauf `e=2`, beim 2. Schleifendurchlauf `e=0`, ..., beim 5. Schleifendurchlauf `e=-7`

Bei jedem Schleifendurchlauf wird geprüft, ob der jeweilige Wert der Variable `e` grösser als Null ist. Wenn ja, wird er ausgegeben.

Somit gibt das Programm die Zahlen 2 und 9 aus.

### while-Schleife

```
while <bedingung>:
    Code, falls <bedingung> wahr
    Code, falls <bedingung> wahr
    ...
Code nach der Schleife ...
```

Der Code im *Schleifenkörper* wird so lange ausgeführt, wie `<bedingung>` wahr ist.

### while-Schleife (Beispiel)

```
summe = 0
k = 1

while summe < 1000:
    summe = summe + k
    k = k + 1

print(summe)
```

### Eine (Endlos-)Schleife abbrechen

Die aktuelle Schleife kann in ihrem Innern mit dem Schlüsselwort `break` abgebrochen werden. Danach wird das Programm unmittelbar nach dem Schleifenkörper fortgesetzt.

Oft möchte man zuerst bedingungslos in eine Schleife eintreten und dann am Ende des Schleifenblocks entscheiden, ob man diesen ein weiteres Mal ausführt.

Dies führt zur Konstruktion einer Endlosschleife, die erst beim Eintreten einer Bedingung abgebrochen wird.

### Beispiel

```
daten = []

while True:
      eingabe = input('Eingabe: ')
      if eingabe == 'quit':
          break
      else:
          daten.append(float(eingabe))

print('Sie haben folgende Daten eingegeben: ')
print(daten)
```

### Einzelne Schleifendurchläufe abbrechen

In gewissen Situationen will man nur einzelne unerwünschte Schleifendurchgänge nicht ausführen, ohne jedoch die gesamte Schleife zu beenden.

Dazu verwendet man das Schlüsselwort `continue`. Dieses bricht die Schleife am Ort ihres Auftretens ab und tritt, sofern keine Bedingung verletzt wird, ein weiteres Mal in den Schleifenkörper ein.

### Beispiel

```
import math

for j in [9, -4, 16, 25, -49, 100]:
    if j < 0:
        continue
    print(math.sqrt(j))
```

Das obige Programm durchläuft mit der Hilfsvariablen `j` die Elemente der Liste `[9, -4, 16, ...]`. Sobald die Variable einen negativen Wert hat, wird der aktuelle Schleifendurchgang abgebrochen. Auf diese Weise verhindern wir, dass die Wurzel aus einer negativen Zahl gezoge wird.

Ausgabe:

### Geschwindigkeitsunterschiede

In den folgenden zwei Schleifen wird die Summe der ganzen Zahlen von 1 bis $10^7 - 1$ berechnet:

Mit Hilfe der Funktion `time.clock()` aus dem Modul `time` soll untersucht werden, welcher Schleifentyp das Resultat schneller berechnet.

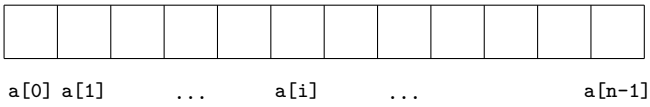
```
import time
n = 10**7

start = time.clock()
summe = 0
for i in range(1, n):
    summe = summe + i
ende = time.clock()
print('for: ', ende-start, ' Sekunden')

start = time.clock()
summe = 0
i = 1
while i < n:
    summe = summe + i
    i = i + 1
ende = time.clock()
print('while: ', ende-start, ' Sekunden')
```

# 7 Listen und Tupel

### Die Liste als Datenstruktur




- Eine Liste ist eine geordnete Sammlung von Elementen.
- Jedes Element in der Liste hat einen *Index*.
- In Python beginnen die Indizes bei Null.
  (Dies ist auch bei vielen anderen Programmiersprachen so.)
- Die Anzahl der Elemente wird *Länge* der Liste genannt.
- Die Elemente können von unterschiedlichem Typ sein. Eine Liste kann selbst Listen als Elemente enthalten.

### Listen erstellen

Eine Liste aus gegebenen Elementen aufbauen:

```
a = [3, -7, 'abc', [1, 0, 5], 8.2]
```

### Die Länge einer Liste bestimmen

```
a = [3, -7, 'abc', [1, 0, 5], 8.2]
len(a) → 5
```

### Ein Listenelement auslesen

```
a = [3, -7, 'abc', [1, 0, 5], 8.2]
a[2] → abc
a[5] → Fehler!
a[3][2] → 5
a[-1] → 8.2
a[-4] → -7
```

### Listenelement ersetzen

```
a = [3, -7, 'abc', [1, 0, 5], 8.2]
a[0] = 99
a[3][1] = 2
print(a) → [99, -7, 'abc', [1, 2, 5], 8.2]
```

### Elemente hinzufügen

Einzelne Elemente werden mit der Methode `append(<element>)` *am Ende* der Liste angehängt:

`a = [7, -3, 0]`

`a.append(29)` → `a = [7, -3, 0, 29]`

Die Methode `insert(<index>, <element>)` fügt `<element>` an der Position `<index>` ein und „schiebt“ die übrigen Elemente nach rechts.

`a = [7, -3, 0]`

`a.insert(1, 99)` → `a = [7, 99, -3, 0]`

### Verketten von Listen

`a = [7, -3, 0]`

`b = [1, 4]`

`a = a + b` → `a = [7, -3, 0, 1, 4]`

### Vervielfachen von Listen

`3 * [9, 2]` → `[9, 2, 9, 2, 9, 2]`

`5 * [0]` → `[0, 0, 0, 0, 0]`

### Elemente entfernen

Einzelne Elemente werden mit der Methode `pop` von Ende der Liste oder von einer legalen Position entfernt. Ohne ein Argument wird das letzte Listenemelent entfernt.

`a = [7, -3, 0, 29, 1, 4]`

`a.pop()` → `a = [7, -3, 0, 29, 1]`

`x = a.pop(1)` → `x = -3, a = [7, 0, 29, 1]`

### Teillisten (*Slices*)

Mit dem Doppelpunkt können Indexbereiche von Teillisten ausgewählt werden:

`a = [3, 5, 8, 2, 0, 7]`

`print(a[1:4])` → `[5, 8, 2]`

*Achtung:* Das Element mit dem Index rechts vom Doppelpunkt gehört nicht zur Teilliste. Daher ist `a[2:6]` erlaubt aber `a[6]` nicht.

Lässt man den Index vor bzw. nach dem Doppelpunkt weg, so werden alle Elemente vom Anfang bzw. bis zum Ende gewählt:

`a[:3]` → `[3, 5, 8]`

`a[2:]` → `[8, 2, 0, 7]`

`a[:]` → `[3, 5, 8, 2, 0, 7]`

Mit einem dritten Argument in den eckigen Klammern kann eine von 1 abweichende Schrittweite angegeben werden.

`a = [3, 5, 8, 2, 0, 7]`

`a[1:5:2]` → `[5, 2]`

`a[4:1:-1]` → `[0, 2, 8]`

`a[::-2]` → `[7, 2, 5]`

### Summen

Die Funktion `sum(...)` bildet die Summe aller Listenelemente.

`s = sum([1, 2, 3, 4])` → `10`

### Umordnen

Die Methode `reverse()` speichert die Elemente *in place* in umgekehrter Reihenfolge:

`a = [3, 0, 5]`

`a.reverse()` → `a = [5, 0, 3]`

### Sortieren

Die Methode `sort()` sortiert die Elemente *in place* in aufsteigender Reihenfolge:

`b = [2, 7, -6]`

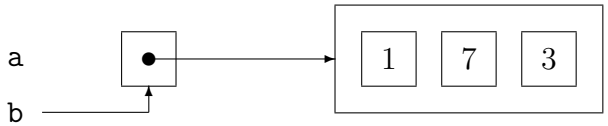
`b.sort()` → `b = [-6, 2, 7]`

### Kopieren von Listen

Die „naive“ Methode funktioniert nicht:

```
a = [1, 2, 3]
b = a
b[1] = 7
print(a) → [1, 7, 3]
```

Aus Effizienzgründen wird nicht die gesamte Liste sondern eine *Referenz* darauf gespeichert. Beim Kopieren wird nur die Referenz kopiert. Daher beziehen sich beide Variablen auf dieselbe Liste.




Für eine „unabhängige“ Kopie wählt man die gesamte Liste als Teilliste aus:

`b = a[:]` funktioniert!

### Mehrfachzuweisungen

Listen erlauben Mehrfachzuweisungen:

`[x, y] = [3, 8]`

weist der Variablen `x` den Wert 3 und der Variablen `y` den Wert 8 zu.

### Tupel

- Tupel sind statische (=unveränderliche) Listen und werden von runden statt von eckigen Klammern eingeschlossen.
- Erlaubt sind alle Listenfunktionen, welche die Elemente oder ihre Anzahl *nicht* verändern.
- Da Tupel unveränderlich sind können sie zum Beispiel als Indizes für Dictionaries verwendet werden. Dazu später mehr.

### Beispiele

- `a = (3, -8, 2, 9)`
- `print(a[-1])` $\rightarrow 9$
- `print(len(a))` $\rightarrow 4$
- `a[2] = 99` $\rightarrow$ Fehler
- `a.pop()` $\rightarrow$ Fehler
- `a[1:3]` $\rightarrow (-8,2)$
- `b = 3 * (0,)` $\rightarrow (0, 0, 0)$

Weshalb braucht es in `b = 3 * (0,)` das Komma?

# 8 Funktionen

### Ausgangslage

Computercode, der gleich aufgebaut ist und an mehreren Stellen in einem (oder verschiedenen) Programmen verwendet wird, *darf unter keinen Umständen kopiert werden.*

**Grund:** Änderungen oder Verbesserungen an diesem diesem Code müssen an allen Stellen durchgeführt werden. Dies kostet sehr viel Zeit und in der Regel verliert man sehr schnell den Überblick, an welchen Stellen der kopierte Code steht.

### Abhilfe

Ein „überschaubares“ Stück Computercode, das eine spezielle Aufgabe an verschiedenen Stellen in einem (oder mehreren Programmen) erfüllen soll, wird als *Funktion* definiert.

Die veränderlichen Objekte des Codes, sofern es solche überhaupt gibt, werden *Parameter* genannt und spielen bei der Definition und der Ausführung von Funktionen eine wichtige Rolle.

### Syntaxbeispiel

```
def <funktionsname>(<p1>, <p2>, ...):
    <anweisung 1>
    <anweisung 2>
    ...
```

Funktionen werden mit dem Schlüsselwort `def` definiert. Dann folgen der Name der Funktion und eine durch runde Klammern eingeschlosse und durch Kommas getrennte Aufzählung der *formalen Parameter*. Diese Aufzählung kann auch leer sein. Danach muss ein Doppelpunkt stehen.

Der eingerückte Funktionskörper enthält den auszuführenden Code. Dieser kann die Anweisung `return <wert>` enthalten, der die Funktionsausführung abbricht und `<wert>` „zurückgibt“.

### Funktion mit Seiteneffekt

```
def hello(name):
    print('Guten Tag', name)

hello('Tarzan')
```

Die Funktion `hello` hat einen formalen Parameter `name`.

Beim Aufruf der Funktion mit `hello('Tarzan')` ersetzt Python den formalen Parameter (`name`) durch eine Kopie des aktuellen Parameters (`'Tarzan'`) und führt damit die Funktion aus. Dies bedeutet im vorliegenden Beispiel, dass der Funktionskörper dafür sorgt, dass ein Begrüssungstext mit dem aktuellen Namen ausgegeben wird.

### Funktion mit Rückgabewert

```
def skalarprodukt(a, b):
    return a[0]*b[0]+a[1]*b[1]+a[2]*b[2]

u = [1,2,3]
v = [4,5,6]
print(skalarprodukt(u, v))
```

Die formalen Parameter sind zwei Listen der Länge 3. Die Funktion berechnet das Skalarprodukt dieser Listen und liefert es als Funktionswert zurück. Dies bewirkt, dass in der `print`-Anweisung der Ausdruck `skalarprodukt(u,v)` durch den Wert 32 ersetzt wird.

### Vorteile

- *Abstraktion*: Komplexe Abläufe können durch Funktionen ersetzt werden, was die Verständlichkeit und Lesbarkeit von Programmen erhöht.
- *Kapselung:* Die Details der Funktionsdefinition sind im Funktionsrumpf „verborgen“. So lange die Schnittstellenparameter (Anzahl und Reihenfolge der formalen Parameter sowie der Typ des Rückgabewerts) unverändert bleiben, kann eine Funktionen verändert werden, ohne dass dies Konsequenzen für die übrigen Teile eines Programms hat.
- *Modularisierung*: Einzelne Funktionen können wie Bausteine zu neuen Funktionen zusammengefügt werden.
- *Einfache Wartung:* Fehlerkorrekturen und Änderungen müssen nur einmal in der Funktionsdefinition vorgenommen werden.

### Mehrere `return`-Anweisungen

In einer Funktion können mehrere `return`-Anweisungen stehen. Sobald das Programm eine dieser Anweisungen erreicht, wird die Ausführung der Funktion abgebrochen und der Funktionswert an den aufrufenden Code zurückgeliefert.

```
def betrag(x):
    if x < 0:
        return -x
    return x
```

### Lokaler Gültigkeitsbereich (*scope*)

Beim Aufruf einer Funktion werden Kopien der aktuellen Parameterwerte vorübergehend an die formalen Parameter gebunden. Das heisst, dass formale Parameter nur innerhalb der Funktion (lokal) gültig sind. Ist ausserhalb der Funktion eine Variable gleichen Namens definiert, so wird sie während der Funktionsausführung gewissermassen „überschattet“.

```
a = 2
def g(x):
    a = 5
    return x + a

print(g(7))
print(a)
```

### Achtung!

Da bei Listen (und anderen zusammengesetzten Objekten) aus Effizienzgründen nicht das gesamte Objekt sondern eine Referenz auf das Objekt kopiert wird, führt dies dies dazu, dass Veränderungen am Original-Objekt durchgeführt werden

```
def f(mylist):
    mylist.pop()

a = [1,2,3]
f(a)
print(a) # => a = [1, 2]
```

### Verschachteln von Funktionen

Funktionen mit passendem Rückgabewert können verschachtelt werden:

```
def f(x):
    return 2*x + 1

print(f(f(f(3))))
```

### Keyword-Argumente

```
def f(a, b=3, c=2):
    return a+b-c

print(f(7, 1, 4))
print(f(b=3, c=1, a=4))
print(f(5, 0))
print(f(7))
```

### Rückgabe mehrerer Werte

```
def f(a, b):
    c = a + b
    d = a - b
    return (c, d) # Werte z.B. als Tupel darstellen

x, y = f(5, 3)
```

### Variable Anzahl Argumente

```
def f(*args):
    # args ist ein Tupel, das alle Argumente enthält
    s = 0
    for element in args:
        s += element
    return s

print(f(7))
print(f(3, 5))
print(f())
print(f(1, 2, 3, 4, 5))
```

### Rekursion

Man spricht von *Rekursion*, wenn eine Funktion sich selber aufruft. Damit dieser Vorgang nicht dazu führt, dass eine Funktion sich unendlich oft aufruft, sind solche Funktionen mit entsprechenden „Bremsen“ auszustatten.

Ferner ist zu berückichtigen, dass durch das rekursive Aufrufen viel Arbeitsspeicher benötigt wird. Das führt dazu, dass der Python-Interpreter bei einer gewissen Rekursionstiefe den Dienst verweigert.

Trotzdem ist die Rekursion bei vielen Programmierproblemen ein geeignetes Lösungsverfahren.

### Fibonacci-Zahlen

Die Fibonacci-Folge ist rekursiv definiert:

$$a_n = \begin{cases} 1 & \text{falls } n < 3 \\ a_{n-2} + a_{n-1} & \text{sonst} \end{cases}$$

```
def fibo(n):
    if n < 3: # base case
        return 1
    else:
        return fibo(n-2)+fibo(n-1)

# Fibonacci-Folge: 1, 1, 2, 3, 5, 8, 13, ...
print(fibo(7)) # => 13
```

*Achtung:* diese Implementation ist äusserst ineffizient!

# 9 Strings

### Was sind Strings?

Ein *String* (engl. *Zeichenkette*) ist ein Datentyp, der aus einer Folge von Zeichen besteht.

Besteht der String aus einem einzelnen Zeichen, wird er *Character* genannt.

Ein String, der gar keine Zeichen enthält, ist der *leere String*.

Strings werden zwischen Hochkommas oder zwischen Anführungszeichen eingeschlossen.

*Literale* (direkte) Verwendung von Strings:

```
print("Hallo")
```

```
Hallo
```

### Variablen und Strings

Speichern von Strings in Variablen

```
x = "Hallo"
print(x)
```

```
Hallo
```

### Mehrzeilige Strings

Benötigt ein String mehr als eine Zeile, so lässt sich das Zeilenende mit einem Backslash unterdrücken und der String auf der folgenden Zeile fortsetzen.

```
text = "Das ist ein \
sehr sehr sehr sehr \
langer String."
print(text)
```

```
Das ist ein sehr sehr sehr sehr langer String.
```

Man beachte, dass bei der Ausgabe des Strings die Zeilenschaltung nicht ausgegeben wird.

Einen mehrzeiligen String einschliesslich Zeilenschaltungen gibt man mit dreifachen Hochkommas (*triple quoted strings*) bzw. dreifachen Anführungszeichen ein

```
text = '''Das ist eine
Zeichenkette, die mehrere
Zeilen lang ist.'''

print(text)
```

```
Das ist eine
Zeichenkette, die mehrere
Zeilen lang ist.
```

## 9.1 Operatoren für Strings

### Konkatenation (Verkettung)

```
a = "Pyt"
b = "hon"
print(a + b)
```

```
Python
```

### Multiplikation

```
print(3 * "7")
```

```
777
```

### Überladen

Offenbar werden hier dieselben Symbole `+` und `*` wie für die Addition und Multiplikation von Zahlen verwendet. Dennoch erkennt der Python-Interpreter anhand des Kontexts, dass er hier Zeichenketten statt Zahlen verarbeiten soll. Operatoren (oder auch Funktionen bzw. Methoden), die abhängig vom Kontext eine unterschidliche Wirkungsweise haben, nennt man *überladen*.

## 9.2 Konversionen (Umwandlungen)

### Umwandlung von Zahlen in Strings

Oft müssen Zahlen und Zeichenketten zu einer neuen Zeichenkette zusammengefügt werden. In diesem Fall weiss Python nicht, ob zwei Zahlen addiert oder zwei Zeichenketten zusammengefügt werden sollen. Mit der Funktion `str()` können wir eine Zahl (oder andere Objekte) als String darstellen, so dass eine Verkettung mit einer Zeichenkette möglich wird.

```
a = 5
b = 7
print(str(a) + ' * ' + str(b) + ' = ' + str(a*b))
```

```
5 * 7 = 35
```

### Konversion von Strings in Zahlen

Wir werden später sehen, dass Benutzereingaben als Zeichenketten an den Python-Interpreter weitergereicht werden. Insbesondere werden auch Zahlen als Zeichenketten interpretiert, so dass man nicht unmittelbar damit rechnen kann. Deshalb müssen solche Zeichenketten vor ihrer weiteren Verarbeitung in den gewünschten Datentyp konvertiert werden.

Hat man es mit einer ganzen Zahl zu tun, verwendet man die Funktion `int()`:

```
text = '-123'
print(int(text) + 23)
```

```
-100
```

`int()` kann sogar noch etwas mehr. Gibt man als zweites Argument eine Basis (grösser als 1 und kleiner als 36) an, wandelt Python auch Strings, die ganze Zahlen in anderen Zahlensystemen darstellen, in eine Zahl im Dezimalsystem um:

```
print(int('c', 16))
print(int('1101', 2))
```

```
12
13
```

Bei Fliesskommazahlen verwendet man die Funktion `float()`. Hier werden neben Vorzeichen auch die Symbole 'e' bzw. 'E' als Präfix für einen ganzzahligen Exponenten interpretiert.

```
print(float('.123'))
print(float('1.5e6'))
print(float('2e-1'))
```

```
0.123
1500000.0
0.2
```

## 9.3 Sonderzeichen und Escape-Sequenzen

| **Escape-Zeichen** | **Bedeutung** |
|---|---|
| \<Zeilenschaltung> | Ignoriere die Zeilenschaltung |
| \\ | Backslash |
| \' | Hochkomma (single quote) |
| \" | Anführungszeichen (double quote) |
| \a | ASCII-Signalton (BEL) |
| \b | ASCII-Backspace (BS) |
| \f | ASCII-Formfeed (FF) |
| \n | ASCII-Linefeed (LF) |

| Escape-Zeichen | Bedeutung |
|---|---|
| \N{<name>} | Unicode-Zeichen mit <name> |
| \<ooo> | Zeichen mit (dreistelligem) Oktalcode |
| \r | ASCII-Wagenrücklauf (CR) |
| \t | ASCII-Tabulator (TAB) |
| \u<hhhh> | Unicode-Zeichen mit 16-bit-Hexadezimalwert |
| \U<hhhhhhhh> | Unicode-Zeichen mit 32-bit-Hexadezimalwert |
| \t | ASCII vertikaler Tabulator (VT) |
| \x<hh> | Zeichen mit 8-bit-Hexadezimalwert |

Um innerhalb einer Zeichenkette, die von Anführungszeichen eingeschlossen ist, weitere Anführungszeichen verwenden zu können, muss Python wissen, dass das verwendete Anführungszeichen die Zeichenkette nicht beendet. Daher wird vor das betreffende Anführungszeichen ein Backslash gesetzt, um es zu „maskieren“, um seine eigentliche Bedeutung aufzuheben. Dieses Vorgehen nennt man in der Fachsprache *escapen* von (engl. escape).

Natürlich lässt sich dieses Problem auch eleganter durch Verwendung der jeweils unbenutzten String-Kennzeichners lösen:

```
print('C\'est la vie!') # mit Maskierung
print("C'est la vie!")  # das Problem 'umgehen'

C'est la vie!
C'est la vie!
```

Will man die (Unicode-)Nummer zu einem bestimmten Zeichen haben, gibt es dafür die Funktion `ord()`, wobei als Argument ein einzelnes Zeichen anzugeben ist.

```
print(ord("a"))

97
```

Offenbar gibt `ord()` die Nummer des Zeichens in dezimaler und nicht in hexadezimaler Form an. In diesem Fall hilft die Funktion `hex()`, die eine Zahl vom 10er- ins 16er-System verwandelt:

```
print(hex(ord("a")))

0x61
```

Die Zeichenfolge `0x` zeigt an, dass die folgende Zahl als Hexadezimalzahl zu deuten ist.

Zu `ord()` gibt es die „Umkehrfunktion“ `chr()` (*character*), die zu einer Dezimalzahl das entsprechende (Unicode-)Zeichen liefert:

```
print(chr(97))
print(chr(int(0x3b)))

a
;
```

## 9.4 Strings und Listen

Strings verhalten sich in vielerlei Hinsicht wie Listen. Beispielsweise liefert die Funktion `len()` nicht nur die Länge einer Liste sondern auch die Länge des Strings, der im Argument steht. Ebenso funktioniert auch die Indizierung einzelner Zeichen oder das Slicing von Strings:

```
text = "Das ist ein Satz."
print(len(text))
print(text[2])
print(text[0:3])
print(text[-5:])
```

```
17
s
Das
Satz.
```

Ähnlich wie bei Listen kann mit `in`, geprüft werden, ob eine Zeichenkette in einer anderen enthalten ist:

```
text = "Das ist Wahnsinn!"
print("ist" in text)
print("Sinn" in text)
```

```
True
False
```

In einem wichtigen Punkt unterscheiden sich Listen und Strings:

- Listen sind veränderliche Objekte.
- Strings sind unveränderliche Objekte (*immutable*).

Das folgende Codefragment führt zu einer Fehlermeldung, da ein String nachträglich nicht verändert werden kann.

```
text = 'Kas ist ein Satz.'
text[0] = 'D'
```

```
Traceback (most recent call last):
  File "python-strings-theo-22.py", line 2, in <module>
    text[0] = 'D'
TypeError: 'str' object does not support item assignment
```

Wir werden später sehen, wie man mit einem kleinen Umweg dennoch Änderungen an einem String vornehmen kann.

## 9.5 Die `format`-Methode

Mit der format-Metode können für viele Zwecke aus verschidenen Argumenten massgeschneiderte Zeichenketten erzeugt werden.

```
text = 'Das Doppelte von {0} ist {1}.'.format(7, 2*7)
print(text)
```

```
Das Doppelte von 7 ist 14.
```

Der Ausdruck beginnt mit einem String, der eine Art Schablone für die Ausgabe darstellt und Platzhalter für die später einzusetzenden Werte enthalten kann.

Jeder Platzhalter besteht in der Regel aus einer ganzen Zahl, die von einem Paar geschweifter Klammern umschlossen ist und auf die Position des Arguments der format-Methode verweist. Konkret steht `{0}` für das erste Argument, `{1}` für das zweite, usw.

Wenn die Argumente in der Reihenfolge ihres Auftretens eingesetzt werden, können die Nummern auch weggelassen werden:

```
text = 'Das Doppelte von {} ist {}.'.format(7, 2*7)
```

Neben der Reihenfolge der Einbettung von Werten in einen String, kann auch die Ausrichtung, der Platzbedarf und die Art der Zahlendarstellung kontrolliert werden. Die dafür nötigen Angaben werden innerhalb der geschweiften Klammern nach einem Doppelpunkt eingegeben:

```
text = '{0:.3f} ist ungleich {0:.4f}.'.format(2/3)
print(text)
```

```
0.667 ist ungleich 0.6667.
```

Hier bedeutet `.3f` (`.4f`), dass die Zahl als Fliesskommazahl mit drei (vier) Nachkommastellen angezeigt werden soll.

```
schablone = "{0:>10}{1:>10}{2:>10}"

print(schablone.format(2030, 21325, 760))
print(schablone.format(77, 953, 3234))
```

```
      2030     21325       760
        77       953      3234
```

Hier bedeutet `>10`, dass für den Wert 10 Zeichen reserviert werden und er rechtsbündig dargestellt werden soll. Entsprechend würde `<10` bzw. `^10` bedeuten, dass der Wert linksbündig bzw. zentriert dargestellt wird.

Wer möchte, kann auch noch ein Füllzeichen einsetzen:

```
text = "{:*^10}.".format(3.14)
print(text)
```

```
***3.14***.
```

Ferner kann mit einem Präfix die Art des Zahlensystems eingestellt werden:

```
text = "Dec: {0} Bin: {0:#b} Hex: {0:#x}".format(35)
print(text)
```

```
Dec: 35 Bin: 0b100011 Hex: 0x23
```

Eine Angabe der maximalen Zeichenlänge ist bei den letzten zwei Formatbeschreibern übrigens nicht zulässig, da eine ganze Zahl nach einer Umwandlung länger sein kann und die Gefahr besteht, dass Stellen abgeschnitten werden.

## 9.6 Weitere Methoden für Strings

`s.capitalize()`

Liefert eine Kopie der Zeichenkette `s` zurück, bei der erste Buchstabe ein Grossbuchstabe ist.

```
text = "PYTHON".capitalize()
print(text)
```

```
Python
```

`s.count(`*`t`*`[,`*`start`*`,`*`end`*`])`

Zählt die Vorkommen des Strings *`t`* im String *`s`*. Optional kann ein Bereich (Slice) angegeben werden. Man beachte, dass (wie bei Slices üblich) das Zeichen an der Position *`end`* nicht mehr zum Teilstring gehört.

```
print("Ananas".count("a"))
print("Ananas".count("a", 2, 4))
```

```
2
1
```

`s.encode(`*`encoding=utf8`*`)`

Wandelt einen String in die Binärform um.

```
text = "Sonderzeichen: ä, ö\n".encode()
print(text)
```

```
b'Sonderzeichen: \xc3\xa4, \xc3\xb6\n'
```

`s.join(`*`seq`*`)`

Liefert die Konkatenation (Verkettung) jedes Elements in der Folge *`seq`* zurück, wobei zwischen den Elementen der Folge jeweils die Zeichenkette `s` eingefügt wird.

```
liste = ["Luzern", "Fribourg"]
print("--".join(liste))
```

```
Luzern--Fribourg
```

`s.lower()`

Liefert eine Kopie von `s` in Kleinbuchstaben zurück.

```
text = "Gross- und Kleinschreibung"
print(text.lower())
```

```
gross- und kleinschreibung
```

`s.lstrip([t])`

Liefert eine Kopie des Strings `s` ohne führende Whitespaces (Leerzeichen, Tabulatoren, Zeilenschaltungen) zurück. Optional kann eine Zeichenkette *t* angegeben werden, deren Zeichen (in beliebiger Reihenfolge) links vom String `s` entfernt werden.

```
print("    Aufgabe".lstrip())
print("Aufgabe".lstrip("fuA"))
```

```
Aufgabe
gabe
```

`s.replace(t, u, n)`

Liefert eine Kopie der Zeichenkette `s` zurück, wobei jedes Vorkommen des Strings *t* durch den String *u* ersetzt wird.

```
original = "Drei Chinesen mit dem Kontrabass"
kopie = original.replace("e", "i")
print(kopie)
```

```
Drii Chinisin mit dim Kontrabass
```

`s.rstrip([t])`

Liefert eine Kopie des Strings `s` ohne Whitespaces (Leerzeichen, Tabulatoren, Zeilenschaltungen) am rechten Rand zurück. Optional kann eine Zeichenkette *t* angegeben werden, deren Zeichen (in beliebiger Reihenfolge) rechts vom String `s` entfernt werden.

```
text1 = "Das ist eine Zeile.\n\t  \n\n".rstrip()
text2 = "Tempomat".rstrip("tam")
print(text1, text2)
```

```
Das ist eine Zeile. Tempo
```

`s.split(t[, n])`

Liefert eine Liste von Teilstrings von `s` zurück, die durch das Trennzeichen t enstehen. Optional kann eine maximale Anzahl von Aufspaltungen angegeben werden.

```
L1 = "192.168.1.1".split(".")
L2 = "Aller guten Dinge sind drei".split(" ", 2)
print(L1)
print(L2)
```

```
['192', '168', '1', '1']
['Aller', 'guten', 'Dinge sind drei']
```

`s.strip(zeichen)`

Liefert eine Kopie des Strings `s` zurück, bei der links und rechts die Whitspaces (Leerzeichen, Tabulatoren und Zeilenschaltungen) entfernt sind. Optional kann ein String t angegeben werden dessen Zeichen (in beliebiger Reihenfolge) links und rechts entfernt werden.

```
text = "programmieren".strip("einpro")
print(text)
```

```
gramm
```

`s.upper()`

Liefert eine Kopie von `s` in Grossbuchstaben zurück.

```
text = "usa".upper()
print(text)
```

```
USA
```

`s.zfill(laenge)`

Füllt eine Zeichenkette mit Nullen zur Gesamtlänge *laenge* auf.

```
text = "7".zfill(3)
print(text)
```

```
007
```

Dies ist nützlich, wenn nummerierte Datenamen vom Betriebssystem „richtig“ sortiert werden sollen, da in der lexikographischen Anordnung der Dateiname `datei-12.txt` vor dem Dateinamen `datei-1.txt` aber hinter dem Dateinamen `datei-01.txt` eingeordnet wird.

### Wichtig

Viele der oben beschriebenen Methoden liefern eine veränderte *Kopie* der Zeichenkette `s` zurück. Falls man (wie in den Beispielen) literale Strings vearbeitet, ist das kein Problem. Sind die Strings jedoch in Variablen gespeichert, so muss man den neuen Wert der ursprünglichen Variablen zuweisen, damit die Änderung wirksam wird. Beispiel:

```
text = "KrEuZ uNd QuEr!"
text.lower()
print(text)
text = text.lower()
print(text)

KrEuZ uNd QuEr!
kreuz und quer!
```

# 10 Input/Output

## 10.1 Ausgabe via Konsole

### Die Funktion `print(...)`

`print(`*`obj1`*`, `*`obj2`*`, ...)` gibt die Textdarstellung der durch Kommas getrennten Objekte auf der Standardausgabe aus.

Bei mehreren Argumenten werden die Textdarstellungen durch jeweils ein Leerzeichen getrennt. Am Ende der Ausgabe erfolgt eine Zeilenschaltung.

Dieses Standardverhalten lässt sich durch die benannten Argumente `sep='...'` und `end='...'` anpassen.

### Beispiel

```
print(1, 2, 3, 4, sep=';', end='')
print(5, 6, 7, 8, sep='*', end='\n')
```

Ausgabe:

```
1;2;3;45*6*7*8
```

## 10.2 Eingaben via Konsole

### Benutzereingaben mit `input(...)`

`input(`*`text`*`)` gibt *`text`* auf dem Bildschirm aus und wartet darauf, bis der Benutzer eine Eingabe gemacht hat und mit RETURN abschliesst. Der Rückgabewert dieser Funktion ist die Zeichenkette, die vom Benutzer eingegeben wurde (ohne RETURN).

```
name = input('Wie heissen Sie? ')
print('Guten Tag, {0}!'.format(name))
```

```
Wie heissen Sie?  Tarzan [ENTER]
Guten Tag, Tarzan!
```

### Benutzereingaben mit Zahlen

`input(...)` liefert immer eine *Zeichenkette* zurück

Wenn also der Benutzer `1234` eingibt, dann wird die Zeichenkette `'1234'` an Python weitergereicht.

Damit Python damit rechnen kann, muss die Zeichenkette in einen Zahl konvertiert (umgewandelt) werden. Für jeden Zahlentyp gibt es eine entsprechende Funktion:

- `int(`*`zeichenkette`*`)`

  konvertiert *`zeichenkette`* in eine *ganze Zahl*.

- `float(zeichenkette)`

  konvertiert `zeichenkette` in eine *Gleitkommazahl*.

### Beispiel

```
x = input('1. Zahl: ')
y = input('2. Zahl: ')
print('Ohne float():', x+y)
x = float(x)
y = float(y)
print('Mit float():', x+y)
```

```
1. Zahl: 5
2. Zahl: 7.2
Ohne float(): 57.2
Mit float(): 12.2
```

## 10.3 Ausgabe via Datei

### Schritt 1

```
fd = open('meinedatei.txt', mode='w')
```

Öffnet eine leere Datei mit dem Namen `meinedatei.txt` und liefert einen Dateideskriptor zurück, der hier in der Variablen `fd` (*file descriptor*) gespeichert wird. Das Argument `mode='w'` steht für *write*. Um Daten an eine bestehende Datei anzuhängen, wählt man `mode='a'`.

*Achtung: Falls es bereits eine gleichnamige Datei gibt, wird diese ohne Warnung überschrieben.*

### Schritt 2

```
fd.write(<zeichenkette>)
```

Schreibt `<zeichenkette>` in das Dateiobjekt.

### Schritt 3

```
fd.close()
```

Schliesst das Dateiobjekt. Diesen Befehl sollte man nicht vergessen, da es sonst zu unerwünschten Effekten kommen kann.

### Beispiel

```
fd = open('output.txt', mode='w')
fd.write('Hallo')
```

```
fd.write('Leute\n')
fd.write(str(5+7))
fd.close()
```

In der Datei `output.txt` steht dann:

```
HalloLeute
12
```

### Bemerkungen

- Im Gegensatz zu `print()` wird bei `write()` nicht automatisch eine Zeilenschaltung angehängt.
- Im Gegensatz zu `print()` wandelt `write()` Zahlen nicht automatisch in Zeichenketten um. Deshalb ist hier die Funktion `str()` nötig. Alternativ kann man eine Schablone mit `format()` verwenden.

## 10.4 Eingabe via Datei

```
1 fd = open('meinedatei.txt', mode='r')
2 for zeile in fd:
3     pass # ... hier stehen die Verabeitungsschritte
4 fd.close()
```

Zeile 1: `open(...)` öffnet sucht und öffnet die Datei mit dem Namen `meinedatei.txt` und liefert einen Dateideskriptor zurück, der hier mit `fd` bezeichnet wird. Das Argument `mode='r'` steht für *read*. Dies ist die Vorgabe und kann auch weggelassen werden.

Zeilen 2 und 3: Die Datei wird zeilenweise bis zum Ende gelesen, wobei sich bei jedem Durchlauf die aktuelle Zeile in der Variablen `zeile` befindet.

Zeile 4: `fd.close()` schliesst das Dateiobjekt `fd`. Ohne `close()` kann es zu unerwünschten Effekten kommen.

### Beispiel

```
fd = open('data.txt')
summe = 0

for zeile in fd:
    summe = summe + float(zeile)

print(summe)
```

### Resultat

Mit der Datei `data.txt`

```
445.7
```

```
39.8
17.0
804.6
266.3
```

erhält man folgende Ausgabe:

```
1573.3999999999999
```

### Zeichencodierung

Python liest und schreibt Text standardmässig im UTF-8-Format. Eine andere Textcodierung muss man als Argument der `open(...)`-Funktion angeben. Hier zwei Beispiele:

- ISO-Latin-1: `mode='latin-1'`
- Windows Code Page 1252: `mode='cp1252'`

### Dateityp

In den obigen Beispielen haben wir Textdateien verwendet. Mit Python können auch Daten im Binärformat geschrieben werden, was beispielsweise für gewisse Bildformate nötig ist.

Genauere Informationen findet man unter dem Stichwort `open` auf der Webseite https://docs.python.org/3/.

## 10.5 Kommandozeilenargumente

Beim Start eines Python-Programms über die Kommandozeile, können zusätzliche Argumente, durch Leerzeichen getrennt, angegeben weden.

Wird das Modul `sys` importiert, kann im aufgerufenen Programm mittels der Liste `sys.argv` auf diese Argumente zugegriffen werden. Es ist zu beachten, dass `sys.argv[0]` den Programmnamen und nicht das erste Argument enthält.

### Beispiel

```
import sys

name = sys.argv[0]
argumente = sys.argv[1:]

print("Programmname: " + pname)
for x in argumente:
    print(x)
```

Beim Aufruf sollte das Programm seinen Namen und allfällige Argumente ausgeben.

# 11 Dictionaries

### Was sind Python-Dictionaries?

In Python ist ein *Dictionary* eine Datenstruktur, in der jeder *Wert* (*value*) mit einem *Schlüssel* (*key*) assoziiert wird. Daher spricht man auch von *Schlüssel-Wert-Paaren*.

In anderen Programmiersprachen werden Dictionaries oft als *assoziative Arrays* oder *Zuordnungstabellen* bezeichnet.

### Dictionaries literal definieren

„literal“ bedeutet hier: die Objekte werden durch ihre jeweilige Zeichefolge dargestellt

```
D = {'rot': 'red', 'blau': 'blue', 'grün': 'green'}
```

Der Schlüssel muss von unveränderlichem Datentyp sein (Zahlen, Strings, Tupel, ...).

### Ein leeres Dictionary erzeugen

```
D = dict() # => {}
```

### Schlüssel-Wert-Paare sukzessive hinzufügen

```
D = dict();

D['rot'] = 'red'
D['blau'] = 'blue'

print(D) # => {'blau': 'blue', 'rot': 'red'}
```

Die Reihenfolge, in der die Schlüssel-Wert-Paare gespeichert werden, wird durch Effizienzüberlegungen bestimmt und erscheint daher etwas willkürlich.

### Dictionaries aus zwei Listen erzeugen

```
namen = ['Joe', 'Lea', 'Kim', 'Max']
noten = [4.5, 5.5, 4.0, 5.0]

D = dict(zip(namen, noten))

print(D) # {'Joe': 4.5, 'Lea': 5.5, 'Kim': 4.0, 'Max': 5.0}
```

### Anzahl Elemente

```
D = {'rot': 'red', 'blau': 'blue', 'grün': 'green'}
print(len(D)) # => 3
```

### Einen Wert ändern

```
D = {'rot':'red', 'blau':'blue', 'grün':'green', 'schwarz':'white'}
D['schwarz'] = 'black'
```

### Schlüssel-Wert-Paare entfernen

```
D = {'rot': 'red', 'blau': 'blue', 'grün': 'green'}

if 'grün' in D:
    del D['grün']
```

Um keinen Fehler zu erzeugen sollte vor dem Entfernen mit `in` getestet werden, ob sich der Schlüssel des zu entfernenden Schlüssel-Wert-Paars im Dictionary befindet

### Dictionaries durchlaufen

```
D = {'rot': 'red', 'blau': 'blue', 'grün': 'green'}

for key in D:
    print(key)       # blau rot grün

for key in D.keys():
    print(key)       # blau rot grün

for val in D.values():
    print(val)       # blue red green

for key, val in D.items():
    print(key, '=>' val)   # blau => blue
                           # grün => green
                           # rot => red
```

**Achtung:** Die Einträge in einem Dictionary werden scheinbar „ungeordnet“ gespeichert. Daher kann man sich nicht darauf verlassen, dass sie in der gleichen Reihenfolge ausgegeben werden, wie man sie dem Dictionary hinzugefügt hat.

### Sortierte Ausgabe

```
D = {'rot': 'red', 'blau': 'blue', 'grün': 'green'}

for wort in sorted(D.keys()):
    print(wort, '=>', D[wort])

# blau => blue
# grün => green
# rot => red
```

# 12 Mengen

Eine Menge ist eine ungeordnete Sammlung von Elementen ohne Duplikate.

Python stellt uns für Mengen die Datenstruktur `set` zur Verfügung.

Mittels geschweifter Klammern können Mengen literal definiert werden:

```
klasse = {'Anna', 'Brad', 'Claudia', 'Daniel'}
```

### Die leere Menge

Die leere Menge kann mit der `set()`-Funktion ohne Argumente erzeugt werden:

```
M = set()
```

**Achtung:** Mit {} wird ein *leeres Dictionary* erzeugt.

### Mächtigkeit einer Menge

Nicht überraschend liefert `len(<menge>)` die Anzahl Elemente des Arguments zurück.

```
M = {'Anna', 'Ben', 'Claudia', 'Daniel'}
len(M) # => 4
```

### Methoden (Auswahl)

$A$ und $B$ bezichnen zwei Mengen und $e$ sei ein Element.

| **Ausdruck** | **Wert** |
|---|---|
| `A.issubset(B)` | `True`, falls $A \subset B$ |
| `A.isdisjoint(B)` | `True`, falls $A \cap B = \emptyset$ |
| `A.union(B)` | $A \cup B$ |
| `A.intersection(B)` | $A \cap B$ |
| `A.difference(B)` | $A \setminus B$ |
| `A.add(e)` | $A \cup \{e\}$ |
| `A.discard(e)` | $A \setminus \{e\}$ (kein Fehler, wenn $e \notin A$) |
| `A.remove(e)` | $A \setminus \{e\}$ (Fehler, wenn $e \notin A$) |

# 13 Module

### Module

Module sind Dateien, welche Definitionen und Anweisungen enthalten, um sie wieder zu verwenden.

Python enthält eine grossen Zahl von Standardmodulen, welche die Funktionalität der Programmiersprache erweitern. Aus Effizienzgründen stehen diese Module jedoch erst nach ihrem Import zur Verfügung. Wenn man beispielsweise Das Modul `math` verwenden möchte, so muss es mit der Anweisung `import math` importiert werden.

Neben diesen Standardmodulen ist es auch möglich, eigene Module zu schreiben oder Module von anderen Programmierern zu verwenden. Dazu später mehr.

### Namensräume

Damit sich die Variablen und Funktionen dieser Module nicht gegenseitig in die Quere kommen, verwendet Python ein Namensschema, das es den verschiedenen Modulen erlaubt, die gleichen Bezeichner zu verwenden.

Wenn also ein Modul mit dem Namen `abc` die Funktion `read()` definiert, so lautet der Name zum Aufruf dieser Funktion `abc.read()`. Dies führt zu keinen Konflikten mit einer Funktion `xyz.read()` aus dem Modul `xyz`.

## 13.1 Das Modul `math`

### Konstanten

- `math.pi`

  Die mathematische Konstante $\pi = 3.141592\ldots$ in der verfügbaren Genauigkeit.

- `math.e`

  Die mathematische Konstante $e = 2.718281\ldots$ in der verfügbaren Genauigkeit.

### Verschiedenes

- `math.ceil(x)`

  kleinste ganze Zahl grösser oder gleich `x`

- `math.floor(x)`

  grösste ganze Zahl kleiner oder gleich `x`

- `math.trunc(x)`

  Wert von `x` ohne Nachkommastellen

- `math.isinf(x)`

  `True`, wenn `x` $\pm\infty$ und `False` sonst.

- `math.isnan(x)`

  `True` wenn `x` NaN ist und `False` sonst.

- `math.factorial(x)`

  Fakultät der nichnegativen ganzen Zahl `x`

### Potenzen und Logarithmen

- `math.exp(x)`

  Potenz `e**x` mit der Basis e = 2.718281 . . .

- `math.log(x)`

  Logarithmus von `x` zur Basis e

- `math.log10(x)`

  Logarithmus von `x` zur Basis 10

- `math.pow(x, y)`

  Potenz $x^y$

- `math.sqrt(x)`

  Quadratwurzel von `x`

### Winkelfunktionen

- `math.acos(x)`, `math.asin(x)`, `math.atan(x)`

  Arcuscosinus, Arcussinus und Arcustangens von `x`

- `math.atan2(y, x)`

  Arcustangens von `y/x` im richtigen Quadranten

- `math.cos(x)`, `math.sin(x)`, `math.tan(x)`

  Cosinus, Sinus und Tangens von `x`

- `math.degrees(x)`

  Wandelt den Winkel `x` vom Bogen- ins Gradmass um.

- `math.radians(x)`

  Wandelt den Winkel `x` vom Grad- ins Bogenmass um

### Bemerkung

Dies war nur eine Auswahl der Methoden aus dem `math`-Modul. Mehr Informationen findet man unter:

https://docs.python.org/3/library/math.html

## 13.2 Das Modul `random`

Da herkömmliche Computer deterministisch arbeiten, können wir mit ihnen keine zufälligen Prozesse nachbilden. Mit einem geeigneten Algorithmus (RNG = random number generator), lassen sich aber Zahlen erzeugen, die sich *wie* Zufallszahlen verhalten. Daher nennt man solche Zufallszahlen *Pseudozufallszahlen* (pseudo random numbers). Der von Python verwendete Zufallzahlengenerator („Mersenne Twister“) ist zwar leistungsfähig aber nicht ausreichend, um sehr sichere Verschlüsselungsverfahren damit zu programmieren. Hier eine Auswahl von Funktionen des Moduls:

`random.seed(x)`

Initialisiere den Zufallszahlengenerator mit der Zahl `x`. Sinnvoll, wenn man die vom Modul `random` erzeugten Pseudozufallszahlen bei jedem Programmlauf reproduzieren möchte.

`random.randint(a, b)`

Liefert eine zufällige ganze Zahl $N$ mit $\texttt{a} \leq N \leq \texttt{b}$ zurück.

`random.choice(L)`

Liefert ein zufälliges Element einer nichtleeren Liste $L$ zurück.

`random.shuffle(L)`

Mischt die Elemente der Liste $L$ in-place. Liefert `None` zurück.

`random.sample(L, k)`

Liefert eine Liste mit `k` zufällig ausgewählten Elementen aus der Liste (oder Menge) `L` zurück. Die Auswahl der Elemente erfolgt ohne Zurücklegen.

`random.random()`

Liefert die nächste zufällige Gleitkommazahl im Intervall $[0, 1)$ zurück.

`random.uniform(a, b)`

Liefert die nächste zufällige Gleitkommazahl im Intervall $[\texttt{a}, \texttt{b}]$ zurück.

## 13.3 Das Modul `re`

Das Modul `re` ermöglicht das Arbeiten mit regulären Ausdrücken (*regular expressions*).

Reguläre Ausdrücke bilden eine Art Sprache, um Klassen von Textmustern zu beschreiben

Die folgenden Beispiele beschreiben nur einen minimalen Teil der Möglichkeiten des `re`-Moduls.

### Beispiel 1

Normale Zeichenketten sind bereits reguläre Ausdrücke. Die Methode

`re.findall(<regulärer Ausdruck>, <Zeichenkette>)`

sucht nach allen Übereinstimmungen von `<regulärer Ausdruck>` in `<Zeichenkette>` und gibt sie als Liste aus.

```
import re
txt = "Das ist das erste Mal, dass ich programmiere."
res = re.findall("das", txt)
print(res)
```

Output:

```
['das', 'das']
```

### Beispiel 2

Sollen an einer Stelle mehrere Zeichen erkannt werden, so können diese innerhalb eckiger Klammern aufgeführt werden.

```
import re
txt = "Im Haus lebt die Maus in Saus und Braus."
res = re.findall("[MH]aus", txt)
print(res)
```

Output:

```
['Haus', 'Maus']
```

### Beispiel 3

Es können auch ganze Bereiche wie `a-f`, `A-Z`, `0-9` usw. angegeben bzw. kombiniert werden.

```
import re
txt = "Im Haus lebt die Maus in Saus und Braus."
res = re.findall("[a-zA-Z]aus", txt)
print(res)
```

Output:

```
['Haus', 'Maus', 'Saus', 'raus']
```

### Beispiel 4

Ein Zirkumflex *zu Beginn* einer Zeichenklasse trifft auf alle Zeichen zu, die *nicht* in der Zeichenklasse aufgeführt sind.

```
import re
txt = "Im Haus lebt die Maus in Saus und Braus."
res = re.findall("[^M-S]aus", txt)
print(res)
```

Output:

```
['Haus', 'raus']
```

### Beispiel 5

Ein Punkt ist ein Platzhalter für ein beliebiges Zeichen.

```
import re
txt = "Das ist ein neues Beispiel."
res = re.findall("ei.", txt)
print(res)
```

Output:

```
['ein', 'eis']
```

### Beispiel 6

Ein Zirkumflex ausserhalb einer Zeichenmenge stellt einem Zeilenanfang dar.

```
import re
txt = '''aufheben
saufen
aufmachen
'''
res = re.findall("^auf", txt, re.MULTILINE)
print(res)
```

Output:

```
['auf', 'auf']
```

`re.MULTILINE`: Zeilenschaltungen in Strings berücksichtigen

### Beispiel 7

Ein Dollarzeichen verankert die Suche am Zeilenende.

```
import re
txt = '''aufheben
senden
```

```
entern
'''
res = re.findall("en$", txt, re.MULTILINE)
print(res)
```

Output:

```
['en', 'en']
```

### Beispiel 8

Ein Stern bedeutet, dass das vorangehende Zeichen nullmal(!), einmal, zweimal, ... wiederholt vorkommen kann.

```
import re
txt = 'abbaaab'
res = re.findall("a*", txt)
print(res)
```

Output:

```
['a', '', '', 'aaa', '', '']
```

Achtung: Reguläre Ausdrücke sind *greedy* (gierig)!

### Beispiel 9

Ein Pluszeichen bedeutet, dass das vorangehende Zeichen mindestens einmal wiederholt vorkommen kann.

```
import re
txt = 'abbaaab'
res = re.findall("a+", txt)
print(res)
```

Output:

```
['a', 'aaa']
```

### Beispiel 10

Ein Fragezeichen bedeutet, dass das vorangehende Zeichen höchstens einmal, d. h. nullmal oder einmal vorkommen darf.

```
import re
txt = 'abbaaab'
res = re.findall("a?", txt)
print(res)
```

Output:

```
['a', '', '', 'a', 'a', 'a', '', '']
```

## 13.4 Das Modul `itertools`

Diese Modul stellt Methoden zur Verfügung, die Iteratoren für effziente Schleifen erzeugen.

### Beispiel 1

`chain(*iterables)` erzeugt einen Iterator, der die Elemente der Input-Iteratoren sukzessive durchläuft.

```
import itertools
iterator = itertools.chain('ABC', 'XY')
for element in iterator:
    print(element, end=" ")
```

```
A B C X Y
```

### Beispiel 2

Dei Methode `combinations(iterable, r)` liefert Teilfolgen der Länge `r` aus der Eingabvariable `iterable` zurück.

Die Kombinationen werden in lexikografischer Ordnung ausgegeben. Elemente sind einzigartig bezüglich ihrer Position. Kommt ein Element in der Eingabe höchstens einmal vor, so wird es auch im Iterator nicht wiederholt vorkommen.

```
import itertools
iterator = itertools.combinations('ABCD', 2)
for element in iterator:
    print(element, end=" ")
```

```
('A', 'B') ('A', 'C') ('A', 'D') ('B', 'C') ('B', 'D') ('C', 'D')
```

### Beispiel 3

```
import itertools
iterator = itertools.combinations(range(0,4), 3)
for element in iterator:
    print(element, end=" ")
```

```
(0, 1, 2) (0, 1, 3) (0, 2, 3) (1, 2, 3)
```

### Beispiel 4

`combinations_with_replacement(iterable, r)`

liefert Teilfolgen der Länge `r` aus der Eingabevariable `iterable` zurück, wobei ein Element wiederholt auftreten darf. Sonst gelten dieselben Eigenschaften wie bei `combinations()`.

```
import itertools
iterator = itertools.combinations_with_replacement('ABC', 2)
for element in iterator:
    print(element, end=" ")
```

```
('A', 'A') ('A', 'B') ('A', 'C') ('B', 'B') ('B', 'C') ('C', 'C')
```

### Beispiel 5

`permutations(iterable, r)` liefert die Permutationen der Länge `r` der Elemente in `iterable` zurück. Falls `r` fehlt, werden alle möglichen Permutationen erzeugt.

Permutationen werden lexikografisch sortiert ausgegeben.

Solange keine Elemente in `iterable` mehfach vorkommen, wird es auch in den Permutationen keine Wiederholungen geben.

```
import itertools
iterator = itertools.permutations('ABC', 2)
for element in iterator:
    print(element, end=" ")
```

```
('A', 'B') ('A', 'C') ('B', 'A') ('B', 'C') ('C', 'A') ('C', 'B')
```

### Beispiel 6

```
import itertools
iterator = itertools.permutations(range(0,3))
for element in iterator:
    print(element, end=" ")
```

```
(0, 1, 2) (0, 2, 1) (1, 0, 2) (1, 2, 0) (2, 0, 1) (2, 1, 0)
```

### Beispiel 7

`product(*iterables, repeat=1)` erzeugt das kartesische Produkt der Eingabe-Iterablen. „Potenzen“ eines Iterators lassen sich mit dem `repeat`-Argument darstellen.

```
import itertools
iterator = itertools.product('abc', '12')
for element in iterator:
    print(element, end=" ")
```

```
('a', '1') ('a', '2') ('b', '1') ('b', '2') ('c', '1') ('c', '2')
```

### Beispiel 8

```
import itertools
iterator = itertools.product([0,1], repeat=3)
for element in iterator:
    print(element, end=" ")
```

```
(0, 0, 0) (0, 0, 1) (0, 1, 0) (0, 1, 1) (1, 0, 0) (1, 0, 1) (1, 1, 0) (1, 1, 1)
```

# 14 Exceptions

Während der Laufzeit eines Programms können unvorhergesehene Ereignisse eintreten, die ein Programm in einen undefinierten Zustand versetzen:

- falsche Benutzereingabe
- das Fehlen von Ressourcen (Datei, Netzwerkverbindung, ...)
- bisher unentdeckte Programmfehler
- ...

In der Regel möchte man nicht, dass das Programm dann abstürzt, sondern dass es zu einer erneuten Eingabe aufruft oder mindestens das Problem meldet und dann ordnungsgemäss endet.

Der folgende Beispielcode zeigt, wie ein Stück Code, das möglicherweise Laufzeitfehler verursacht, diesen Fehler erkennt und ohne einen Programmabbruch darauf reagiert.

### Ausnahmen

Welche Benutzereingaben führen beim folgenden Code zu einem Programmabbruch?

```
def umrechnung():
    x = float(input("Masszahl in Zentimetern: "))
    return 2.54*x
```

Mit der `try -- except`-Anweisung können wir einen der Fehler erkennen und angemessen darauf reagieren:

```
def umrechnung():
    try:
        x = float(input("Masszahl in Zentimetern: "))
        return 2.54*x
    except:
        return 'Bitte Geben Sie eine *Zahl* ein.'
```

# 15 Objektorientierte Programmierung

## 15.1 Klassen und Instanzen

Eine *Klasse* ist ein massgeschneiderter Datentyp. In Python definiert man eine Klasse mit dem Schlüsselwort `class`:

```
class MyClass:
    <klassenrumpf>
```

Der Klassenname wird üblicherweise mit einem Grossbuchstaben begonnen. Der Klassenrumpf enthält Variablenzuweisungen und Funktionsdefinitionen; er darf aber auch nur aus einer `pass`-Anweisung bestehen.

Nachdem eine Klasse so definiert wurde, können Instanzen (Objekte) dieser Klasse erzeugt werden, indem man den Klassennamen wie eine Funktion aufruft.

```
x = MyClass():
```

## 15.2 Attribute

Einer Instanz kann man mit der Punkt-Schreibweise individuelle Variablen (*Attribute*, *Eigenschaften*, *Instanzvariablen*) zuordnen. Mit der gleichen Notation kann auch auf die Werte dieser Variablen zugegriffen werden.

```
class Kreis:
    pass

a = Kreis()
a.radius = 5

b = Kreis()
b.radius = 9

print(a) # <__main__.Kreis object at 0xb71c5c4c>
print(b) # <__main__.Kreis object at 0xb71c5ccc>

print(a.radius) # => 5
print(b.radius) # => 9
```

Jede Instanz verfügt also über einen separaten Speicherort, den die `print`-Funktion anzeigt.

Statt aber jeder Instanz manuell Variablen und Werte zuzuweisen, kann man der Klassendefinition eine Initialisierungsfunktion hinzufügen, um diese Arbeit automatisch auszuführen.

Im Jargon der objektorientierten Programmierung spricht man aber nicht mehr von Funktionen sonder von *Methoden*, wenn eine Funktion zu einem Objekt gehört.

Daher spricht man von einer Initialisierungsmethode bzw. einem *Konstruktor*.

```
class Kreis:

    def __init__(self, r):
        self.radius = r

a = Kreis(5)
b = Kreis(9)

print(a.radius) # => 5
print(b.radius) # => 9
```

Nach Konvention ist `self` immer das erste Argument eines Konstruktors. Weitere Parameter (im Beispiel: `r`) können folgen, um Instanzvariablen mit Anfangswerten zu versehen.

## 15.3 Methoden

Eine Methode ist eine Funktion, die zu einer bestimmten Klasse gehört. Neben der speziellen Methode `__init__(...)`, die bei jeder neu erzeugten Instanz als erstes ausgeführt wird, lassen sich auch eigene Methoden definieren.

```
class Kreis:

    def __init__(self, r):
        self.radius = r

    def flaeche(self):
        return 3.141 * self.radius**2

c = Kreis(10)

print(c.flaeche()) # => 314.1
```

Soll eine Methode auf die jeweilige Instanz angewendet werden, so steht der erste Parameter für das Objekt selbst. Üblicherweise wird dieser Parameter `self` genannt. Damit kann man innerhalb der Klassendefinition auf alle Eigenschaften und Methoden des Objekts zugreifen.

Wie bei den Instanzvariablen, werden Methoden auf Instanzen angewendet, indem man sie mit der Punkt-Notation an den Instanznamen bindet.

## 15.4 Klassenvariablen

Soll eine Variable für die gesamte Klasse (unabhängig von einer Instanz) denselben Wert haben, so muss sie ausserhalb der `__init__`-Methode definiert werden.

```
class Kreis:

    pi = 3.141

    def __init__(self, r):
        self.radius = r

    def flaeche(self):
        return Kreis.pi * self.radius**2

print(Kreis.pi)
```

Um auf den Wert einer Klassenvariable zuzugreifen, muss ihr jeweils der Klassenname vorangestellt werden.

Etwas unschön ist der Umstand, dass Python bei einer Instanzvariablen auf eine gleichnamigen Klassenvariable zugreift, wenn die Instanzvariablen nicht definiert ist:

```
class Kreis:

    pi = 3.141

    def __init__(self, r):
        self.radius = r

    def flaeche(self):
        return Kreis.pi * self.radius**2

c = Kreis(10)
print(c.pi) # => 3.141
```

## 15.5 Klassenmethoden

Soll eine Methode unabhängig von den Instanzen, d. h. für die ganze Klasse gelten, so muss der erste Parameter `self` weggelassen werden.

```
import random

class Kreis:

    # Kreis mit Zufallsradius
    def zufall():
        r = random.randint(1,20)
        return Kreis(r)

    def __init__(self, r):
        self.radius = r

a = Kreis.zufall()
print(a.radius) # => (pseudo-)zufaelliger Radius
```

Stellt man den `@classmethod`-Dekorator vor eine Klassenmethode, so steht ihr erstes Argument (üblich ist `cls`) für den Klassennamen.

```
import random

class Kreis:

    # Kreis mit Zufallsradius
    @classmethod
    def zufall(cls):
        r = random.randint(1,20)
        return cls(r)

    def __init__(self, r):
        self.radius = r

a = Kreis.zufall()
print(a.radius) # => (pseudo-)zufälliger Radius
```

## 15.6 Vererbung

Das Konzept der *Vererbung* ermöglicht es, Variablen und Methoden von Klassen in anderen Klassen wiederzuverwenden. Das folgende Beispiel zeigt den Fall, bei dem ein Objekt ein Spezialfall eines allgemeineren Objekts ist

```
class Rechteck:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def flaeche(self):
        return self.a * self.b

class Quadrat(Rechteck):

    def __init__(self, a):
        super().__init__(a, a)

q = Quadrat(3)
print(q.flaeche()) # => 9
```

Da ein Quadrat eine spezielles Rechteck mit zwei gleich langen Seiten ist, muss dafür kein neuer Code geschrieben werden. Stattdessen wird mit der Methode `super()` auf den Konstruktor der Elternklasse zugegriffen. Da es sich in dieser Zeile um die Anwendung und nicht um die Definition des Konstruktors handelt, darf hier kein `self` stehen.

## 15.7 Spezielle Methoden

Neben dem Konstruktor gibt es noch andere spezielle Methoden, um den Code besser lesbar zu machen. Beispielsweise um die Objekte mit den bestehenden Funktionen und Operatoren zu verarbeiten.

```
class Vektor:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return '({0.x}, {0.y})'.format(self)

    def __add__(self, other):
        return Vektor(self.x+other.x, self.y+other.y)

    def __eq__(self, other):
        return (self.x==other.x and self.y==other.y)

a = Vektor(2,1)
b = Vektor(3,-4)

print(a)      # => (2, 1)
print(a+b)    # => (5, -3)
print(a==b)   # => False
```

## 15.8 Zusammenfassung

*Klasse:* Ein Bauplan für Objekte (abstrakter Datentyp)

*Instanz:* Ein Objekt, das zur Laufzeit des Programms erzeugt wird.

*Instanzvariable/Attribut:* Eine Variable, deren Wert von Instanz zu Instanz verschieden sein kann.

*Klassenvariable:* Eine Variable, deren Wert unabhängig von einer Instanz ist.

*Instanzmethode:* Eine Funktion, die auf eine Instanz angewendet wird.

*Klassenmethode:* Eine Funktion, die unabhängig von einer Instanz ist.

*Konstruktor:* Eine Methode, mit der (in Python) ein Objekt initialisiert wird.

*Vererbung:* Ein Konzept, das darin besteht, neue Klassen (Sub- oder Kindklasse) aus vorhandenen Klassen (Super- oder Elternklasse) abzuleiten. Die von der Superklasse geerbten Eigenschaften und Methoden können beibehalten oder überschrieben (abgeändert) werden.

*Überladen von Operatoren* Ein Mechanismus, der es erlaubt, den gleichen Operator (den gleichen Funktionsnamen) in unterschiedlichen Kontexten zu verwenden. Beispielsweise verwendent Python den +-Operator für das Addieren von ganzen Zahlen und das Verketten von Strings.