Programmieren mit Python

Kurztheorie

Inhaltsverzeichnis

Version vom 28. November 2021

1 Allgemeines

Interpreter und Compiler

Python ist eine *interpretierte* Programmiersprache. Das heisst dass ein Python-Programm (Quellcode, Python-Skript) eingelesen und auf syntaktische Korrektheit überprüft wird. Ist dies der Fall, wird es in Maschinencode übersetzt und sofort ausgeführt. Um ein Python-Programm auszuführen benötigt man aber immer den Python-Interpreter für das jeweilige Betriebssystem.

Einen anderen Ansatz verfolgen *compilierte* Programmiersprachen wie Java oder C++. Auch dort wird zuerst überprüft, ob der Quellcode korrekt geschrieben ist. Falls ja, wird das Programm von einem sogenannten Compiler in Maschinenencode übersetzt und für das jeweilige Betriebssystem so aufbereitet, dass es als eigenständiges Programm aufgerufen werden kann.

Python-Code schreiben

Um Python-Programme zu schreiben, braucht es einen Editor, d. h. ein Programm, mit dem man Texte erfassen, verändern und speichern kann. Herkömmliche Textverarbeitungsprogramme sind dafür aber nicht geeignet, da sie neben dem Text auch Formatierungsinformationen abspeichern, die der Python-Interpreter nicht versteht.

Spezielle Editoren unterstützen den Programmierer durch Syntax-Hervorhebung (Syntax-Highlighting). Dabei werden Schlüsselwörter, Variablen, Text, Zahlen und Kommentare durch unterschiedliche Farben gekennzeichnet und erhöhen so die Lesbarkeit der Programme.

Python-Code ausführen

Um Python-Programm auszuführen, benötigt man einen Python-Interpreter. Dies dies ein Programm (`python.exe`, `python3`), das von einem Kommandozeileninterpreter (Shell) aus aufgerufen wird und dem man als Argument den Namen des auszuführenden Programms übergibt.

Ohne Angabe einer Programmdatei wird eine Python-Shell gestartet. Dies ist eine Umgebung, die Python-Code zeilenweise ausführt und sich zum Ausprobieren kleinerer Programmteile eignet.

IDLE

IDLE ist eine Integrierte Entwicklungsumgebung (Integrated Development Environment, IDE) und wird mit der offiziellen Version von Python mitgeliefert. Sie besteht aus einem Editor, der Python-Shell und einem Debugger (Fehlersuchprogramm), die in einer einfachen grafischen Benutzuerschnittstelle zusammengefasst sind.

Das Format von Python-Programmen

Python-Programme werden üblicherweise in einer Textdatei mit der Endung `.py` gespeichert.

Kommentare werden mit einem Doppelkreuz „`#`“ (Raute, Hashtag) eingeleitet. Text zwischen dem Kommentarzeichen und dem Zeilenende wird vom Python-Interpreter ignoriert.

Als Zeichencodierung wird standardmässig UTF-8 verwendet.

Strukturierung

Für Verzweigungen, Schleifen sowie die Definition von Funktionen und Klassen müssen mehrere Anweisungen zu einem Anweisungsblock zusammengefasst werden. Die meisten Programmiersprachen verwenden dafür Klammern.

Python verwendet stattdessen eine Einrückung des zusammengehörenden Codes. Üblicherweise sind das vier Leerzeichen. Diese Einrückung wird normalerweise von einem geeigneten Editor durch Drücken der Tabulator-Taste ausgeführt.

Notation

- Text in `Nichtproportionalschrift` stellt (Teile von) Computercode dar, der genau so eingegeben werden sollte.

```
x = 3
y = 5
print(x + y)
print("hello, world")
```

- Text in *`kursiver Nichtproportionalschrift`* ist durch benutzerdefinierte Werte zu ersetzen.

  `print(`*`Zeichenkette`*`)`

2 Einfache Datentypen

2.1 Ganze Zahlen

Darstellung ganzer Zahlen

Die ganzen Zahlen 741 und −741 können wie folgt dargestellt werden:

- dezimal: `741`, `-741`
- binär: `0b1011100101`, `-0b1011100101`
- oktal: `0o1345`, `-0o1345`
- hexadezimal: `0x2e5`, `-0x2e5`

Operatoren für ganze Zahlen

Operator	Beschreibung	Beispiel	Wert
`-`	monadisches Minus	`-17`	`-17`
`+`	Addition	`23+17`	`40`
`-`	Subtraktion	`23-17`	`6`
`*`	Multiplikation	`4*17`	`68`
`/`	Division	`10/4`	`2.5`
`//`	Ganzzahldivision	`10//4`	`2`
`%`	Divisionsrest	`17%6`	`5`
`**`	Potenzieren	`-2**4`	`-16`

Funktionen für ganze Zahlen (Auswahl)

`abs(`*n*`)` Liefert den Absolutwert der ganzen Zahl *n* zurück.
`bin(`*n*`)` Wandelt die ganze Zahl *n* vom 10er- ins 2er-System um.
`oct(`*n*`)` Wandelt die ganze Zahl *n* vom 10er- ins 8er-System um.
`hex(`*n*`)` Wandelt die ganze Zahl *n* vom 10er- ins 16er-System um.
`str(`*n*`)` Wandelt die ganze Zahl *n* in eine Zeichenkette um.

2.2 Gleitkommazahlen

Darstellung von Gleitkommazahlen

- `0.00734`, `.00734`, `7.34e-3`, `7.34E-3`
- `-42598.0`, `-4.2598e4`, `-4.2598E4`,
- `8.0`, `8.`
- `3e4`, `3E4`

Operationen für Gleitkommazahlen

Operator	Beschreibung	Beispiel	Wert
`-`	monadisches Minus	`-17.4`	`-17.4`
`+`	Addition	`23.4+17.6`	`41.0`
`-`	Subtraktion	`23.4-17.6`	`5.79999...`
`*`	Multiplikation	`3.2*4.1`	`13.12`
`/`	Division	`13.12/4.1`	`3.2`
`**`	Potenzieren	`1.3**4`	`2.856100...`

Bemerkungen

- Ohne Klammern werden Operationen gleicher Stufe von links nach rechts gerechnet.
- Alle Operationen bewahren den Datentyp.
- Kommen Gleitkommazahlen und ganze Zahlen in einem arithmetischen Ausdruck vor, so ist das Ergebnis eine Gleitkommazahl.

Funktionen für Gleitkommazahlen

Für die „höheren“ mathematischen Funktion ist vorgängig mit `import math` das `math`-Package zu laden. Hier eine Auswahl:

Funktion	Wert
`abs(`x`)`	Absolutbetrag von x
`math.sqrt(`x`)`	Quadratwurzel von String x
`math.sin(`x`)`	Sinuswert von x (Bogenmass)
`math.cos(`x`)`	Cosinuswert von x (Bogenmass)
`math.tan(`x`)`	Tangenswert von x (Bogenmass)
`math.pow(`x`,`y`)`	Wert der Potenz x^y
`math.exp(`x`)`	Wert der Exponentialfunktion zur Basis e.
`math.log(`x`)`	Wert der Logarithmusfunktion zur Basis e.

2.3 Zeichenketten

Darstellung von Zeichenketten

In Python werden Zeichenketten durch Anführungszeichen `"..."` oder Hochkommas `'...'` begrenzt.

- `"Hello"`, `'Hello'`
- `"Geben Sie eine Zahl ein:"`

Stellt vor bestimmte Zeichen einen Backslash, so erhalten sie eine neue Bedeutung.

Escape-Sequenz	Bedeutung
`\n`	Zeilenschaltung (*newline*)
`\"`	Anführungszeichen
`\'`	Hochkomma (Apostroph)

Funktionen für Zeichenketten

Es seien `s` eine Zeichenkette und `c` ein String aus einem Zeichen.

Funktion	Wert
`len(s)`	Anzahl der Zeichen von `s`
`list(s)`	Liste der einzelnen Zeichen von String `s`
`int(s)`	die von `s` dargestellte ganze Zahl (wenn möglich)
`float(s)`	die von `s` dargestellte Gleitkommazahl (wenn möglich)
`ord(c)`	Unicode-Nummer des Zeichens `c`

Beispiel 2.3.1

```
1 print(len('Hello World!')) # => 12
2 print(list('UHU'))         # => ['U','H','U']
3 print(int('22') + 3)       # => 25
4 print(float('22') + 3)     # => 25.0
5 print(ord('A'))            # => 65
```

Operatoren für Zeichenketten

Es seien *str1*, *str2* und *str* Zeichenketten sowie *n* eine natürliche Zahl.

Operator	Resultat
str1 + *str2*	Verkettung (*concatenation*) von *str1* und *str2*
n * *str*	String aus der *n*-fachen Repetition von *str*

Beispiel 2.3.2

```
1 print('Bon' + 'bon') # => Bonbon
2 print(4 * 'la')      # => lalalala
```

Methoden für Zeichenketten (Auswahl)

Im Folgenden sind `s`, `t`, `u` Python-Zeichenketten und `L` eine Liste mit Zeichenketten.

Methode	Wert
`s.find(t)`	Startindex von `t`, wenn `t` in `s` vorkommt; sonst `-1`
`s.replace(t,u)`	Ersetzt String `t` durch String `u` in `s`
`s.join(L)`	String, der die Elemente von `L` durch `s` verbindet.
`s.lower()`	String `s` mit Grossbuchstaben → Kleinbuchstaben
`s.upper()`	String `s` mit Kleinbuchstaben → Grossbuchstaben
`s.strip()`	String `s` ohne Whitspaces links und rechts
`s.split(t)`	Liste mit Teilstrings von `s`, getrennt durch `t`
`s.format(...)`	Fügt die Argumente bei {0}, {1}, ... ein.

Beispiel 2.3.3

```
print('Abenteuer'.find('ente'))     # => 2
print('Turm'.replace('T', 'W'))     # => 'Wurm'
print(' & '.join(['Anna', 'Ben'])) # => 'Anna & Ben'
print('CH'.lower())                 # => 'ch'
print('ch'.upper())                 # => 'CH'
print('   test  \n'.strip())        # => 'test'
print('20-06-2020'.split('-'))      # => ['20','06','2020']
print('P({1}|{0})'.format(2,5))     # => 'P(5|2)'
```

2.4 Wahrheitswerte

Darstellung von Wahrheitswerten

`True`, `False`

Operatoren für Wahrheitswerte

x	not x
True	False
False	True

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

Präzedenz (Vorrang) der Operatoren: `not` *vor* `and` *vor* `or`

Beispiel 2.4.1

```
print(not True and False))  # False and False -> False
print(not (True and False)) # not False -> True

print(True or True and False)   # True or False -> True
print((True or True) and False) # True and False -> False
```

Vergleichsoperatoren

Operator	Bedeutung	Operator	Bedeutung
`<`	kleiner als	`<=`	kleiner gleich
`>`	grösser als	`>=`	grösser gleich
`!=`	ungleich	`==`	gleich

Die Vergleichsoperatoren können mit den logischen Operatoren kombiniert werden, haben aber eine höhere Priorität.

Python kennt für `(a < b and b < c)` die Kurzform `a < b < c`.

Beispiel 2.4.2

```
print(4 < 3 or 5 != 8)          # -> True
print(5 >= 4 and not(5 == 8))   # -> True
print(7 > 3 >= 2)               # -> True
```

3 Variablen

Bezeichner

Variablen sind benannte „Behälter“ für Werte. Der Name einer Variablen wird *Bezeichner* (*identifier*) genannt. Die Regeln für die Bildung von Bezeichnern lauten (verkürzt):

- Bezeichner beginnen mit Gross- oder Kleinbuchstaben oder einem Unterstrich. Danach dürfen weitere Gross- oder Kleinbuchstaben sowie Ziffern oder Unterstriche folgen.
- Diese Bezeichner sind reservierte Python-Schlüsselworte:

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

Zuweisungen

Das Symbol „=“ ist der Zuweisungsoperator (*assignment operator*).

Zuerst wird der rechts stehende Ausdrucks ausgewertet und dann der Variablen mit dem links stehenden Bezeichner zugewiesen.

Beispiel 3.1

```
1 x = 2 * 4
2 print(x)   # => 8
3 y = 5 + x
4 print(y)   # => 13
5 x = x - 1
6 print(x)   # => 7
```

Erweiterte Zuweisungen

Operatoren für erweiterte Zuweisungen (*augmented assignments*):

Operator	Beispiel	gleichwertiger Ausdruck
`+=`	`x += 7`	`x = x + 7`
`-=`	`x -= 7`	`x = x - 7`
`*=`	`x *= 7`	`x = x * 7`
`/=`	`x /= 7`	`x = x / 7`
`**=`	`x **= 7`	`x = x ** 7`
`//=`	`x //= 7`	`x = x // 7`
`%=`	`x %= 7`	`x = x % 7`

4 Bedingte Anweisungen und Verzweigungen

Bedingte Anweisung

```
if bedingung:
    codeblock
...
```

Der <codeblock> besteht aus mindestens einer (um 4 Blanks) eingerückten Anweisung. Er wird nur dann ausgeführt, wenn <Bedingung> den Wert True hat. In jedem Fall wird das Programm an der ersten nicht mehr eingerückten Zeile fortgesetzt.

Beispiel 4.1

```
x = 3
if x > 2:
    x = x + 1
x = x + 10
print(x)       # => 14
```

Einfache Verzweigung

```
if bedingung:
    codeblock_1
else:
    codeblock_2
...
```

Wenn <Bedingung> den Wert True hat wird <codeblock_1> ausgeführt, <codeblock_2> übersprungen und das Programm danach fortgesetzt. Andernfalls wird <codeblock_1> übersprungen, <codeblock_2> ausgeführt und das Programm danach fortgesetzt.

Beispiel 4.2

```
x = 3
if x > 4:
    x = x + 1
else:
    x = x - 1
x = x + 10
print(x)      # Ausgabe: 12
```

Mehrfachverzweigung

```
if bedingung_1:
    codeblock_1
elif bedingung_2:
    codeblock_2
elif bedingung_3:
    codeblock_3
...
    ....
else:
    codeblock_n
...
```

Hat eine Verzweigung zusätzlich `elif`-Klauseln, wird nur der Codeblock der ersten wahren Bedingung ausgeführt. Ist keine der Bedingungen wahr, so wird der Codeblock nach `else` ausgeführt. In jedem Fall wird das Programm danach fortgesetzt.

Beispiel 4.3

```
x = 7
if x < 4:
    x = x + 1
elif x < 5:
    x = x + 2
elif x < 6:
    x = x + 3
else:
    x = x + 4
print(x)        # Ausgabe: 11
```

Bemerkung

Bedingte Anweisungen und Verzweigungen können auch verschachtelt werden.

5 Schleifen

Die zählergesteuerte `for`-Schleife

```
for var in range(start, end[, step]):
    codeblock
...
```

var ist ein frei wählbarer Bezeichner, *start*, *end* und *step* sind ganze Zahlen.

Ist *step* positiv [negativ], so werden der Variablen *var* so lange die Werte *start*, *start* + *step*, *start* + 2**step*, ... zugewiesen, wie diese kleiner [grösser] als *end* sind. Damit wird jeweils der *codeblock* ausgeführt.

Für die Schrittweite 1 ist die Angabe von *step* optional.

Beispiel 5.1

```
for i in range(3, 11, 2):  # Ausgabe: 3 9
    print(i, i**2)         #          5 25
                           #          7 25
                           #          9 81
```

Die `while`-Schleife

```
while <bedingung>:
    <codeblock>
...
```

Bei der `while`-Schleife wird der *codeblock* so lange ausgeführt, wie der Wert von *bedingung* wahr ist.

Der Progammierer muss sich selber darum kümmern, dass beim Erreichen des gewünschten Zustands die Bedingung falsch wird, damit die Schleife terminiert.

Beispiel 5.2

```
s = 0
k = 1
while  s < 20:
    s += k
    k += 2
print(s)  # Ausgabe: 25
```

Schleifen frühzeitig abbrechen

Trifft Python bei der Ausführung des Codeblocks einer `for`- oder `while`-Schleife auf das Schlüsselwort `break`, so bricht es dessen Ausführung ab und setzt das Programm nach der Schleife fort.

Beispiel 5.4

```
s = 0
k = 1
while True: # Endlosschleife
    if s > 10:
        break
    else:
        s = s + k
        k = k + 1

print(s)  # Ausgabe: 15
```

Schleifendurchläufe überspringen

Trifft Python bei der Ausführung des Codeblocks einer `for`- oder `while`-Schleife auf das Schlüsselwort `continue`, so bricht es dessen Ausführung ab und springt an den Schleifenkopf, um dort (eventuell) den nächsten Schleifendurchlauf zu starten.

Beispiel 5.5

```
for k in range(1, 10):
    if k % 3 == 0:
        continue
    print(k) # Ausgaben: 1, 2, 4, 5, 7, 8
```

6 Funktionen

Was sind Funktionen?

Eine Funktion ist die Definition eines Codeblocks, der zu einem spätereren Zeitpunkt mit einem Funktionsnamen und variablen Parameterwerten aufgerufen werden kann.

Funktionen verkörpern das wichtige Prinzip der Abstraktion, das Computerprogramme lesbarer macht und das Verbessern von Programmen vereinfacht.

Die Definition von Funktionen

```
1 def fname(p1, p2, ...):
2     codeblock
3 ...
```

Funktionen werden mit dem Schlüsselwort `def` definiert. Es folgt ein gültiger Bezeichner (Funktionsname) auf den unmittelbar ein Paar runder Klammern folgt. Diese Klammern kann eine durch Kommas getrennte Folge von *formalen Parametern* enhalten. Formale Parameter sind Bezeichner, die als Platzhalter im ***codeblock*** eingesetzt werden und die später beim Aufruf der Funktion durch die *aktuellen Parameter* ersetzt werden. Der ***codeblock*** enthält die Anweisungen, die beim Aufruf der Funktion ausgeführt werden sollen.

Rückgabewerte

Steht im Funktionsrumpf eine Anweisung der Form `return` ***wert*** so beendet Python die Abarbeitung allfälliger weiterer Codezeilen und setzt den Rückgabwert ***wert*** an die Stelle des Funktionsaufrufs.

Fehlt eine `return`-Anweisung, so gibt die Funktion den Wert `None` zurück.

Eine Funktion kann nur einen Rückgabewert haben. Diese Einschränkung lässt sich jedoch umgehen, indem man einen zusammengesetzten Datentyp (z. B. eine Liste) als Wert zurückgibt.

Beispiel 6.1

```
1 def f(a, b):
2     return 2*a + b
3
4 print(f(7,1) + 4) # Ausgabe: 19
```

In den Zeilen 1 und 2 wird eine Funktion mit dem Namen `f` definiert, die zum doppelten Wert des ersten formalen Parameters `a` den Wert des zweiten formalen Parameters `b` addiert und das Resultat als Rückgabewert an die Stelle des Funktionsaufrufs setzt.

In der Zeile 4 wird die Funktion mit den aktuellen Parametern `a=7` und `b=1` in der entsprechenden Reihenfolge aufgerufen. Im Funktionsrumpf wird damit $2 \cdot 7 + 1 = 15$ berechnet. Dieser Wert wird an die Stelle `f(7,1)` gesetzt und die `print`-Anweisung gibt 19 ($= 15+4$) aus.

Benannte Parameter

Wenn beim Funktionsaufruf die aktuellen Parameter den jeweiligen formalen Parametern zugewiesen werden, spielt die Reihenfolge der Parameter keine Rolle mehr, da dann die Zuordnung eindeutig ist.

```
1 def f(a, b):
2     return 2*a + b
3
4 print(f(b=1, a=7) + 4) # Ausgabe: 19
```

Gültigkeitsbereich von Variablen

Jeder Funktionsaufruf schafft einen neuen Kontext für Variablen, der nur während der Ausführung der Funktion existiert. Zuweisungen innerhalb einer Funktion sind daher nur vorübergehend gültig. Wir unterscheiden zwei Fälle.

- Wird einer bereits existierenden Variablen in einer Funktion ein neuer Wert zugewiesen, so *überschattet* dieser Wert den früheren. Nach der Ausführung der Funktion wird der Kontext gelöscht und der alte Wert kommt wieder zum Vorschein.
- Taucht eine Variable in einer Funktion zum ersten Mal auf, so ist ihr Wert nur während der Ausführung der Funktion gültig. Danach wird diese Variable gelöscht. Diese Variable ist daher ausserhalb der Funktion *nicht sichtbar*.

Beispiel 6.3

```
1 def f(b):
2     a = b
3     print(a)
4
5 a = 7
6 print(a)  # => 7
7 f(5)      # => 5 (a=7 wird überschattet)
8 print(a)  # => 7 (a=7 ist wieder sichtbar)
```

Rekursion

Manchmal ist es sinnvoll, anstelle einer Schleife eine Funktion zu schreiben, die sich selber (rekursiv) aufruft. Dann muss innerhalb der Funktion eine Bedingung definiert sein, die für den Abbruch der Rekursion sorgt (*Base Case*) und zu ihrer Auflösung führt.

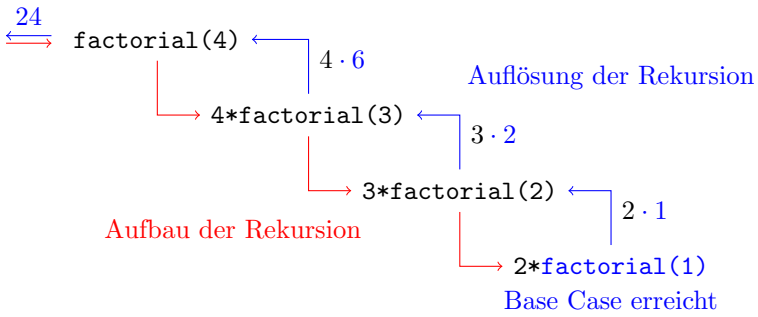
Da bei jedem Funktionsaufruf ein neuer Kontext für die Speicherung von Variablen angelegt werden muss, kostet die rekursive Lösung einer Aufgabe (im Gegensatz zur iterativen Lösung mit Schleifen) zusätzlichen Arbeitsspeicher. Deshalb wird Rekursion dann angewendet, sich die Problemgrösse bei jedem Funktionsaufruf um einen Faktor verkleinert (z. B. halbiert), so dass nur relativ wenige Aufrufe der Funktion nötig sind.

Der Vorteil der Rekursion gegenüber der Iteration besteht darin, dass sich bestimmte algorithmische Probleme damit einfacher lösen lassen – vorausgesetzt man versteht, wie Rekursion funktioniert.

Beispiel 6.4

```
1 def factorial(n): # Rekursive Berechnung von n! (Fakultät)
2     if n == 1:    # Base Case
3         return 1
4     else:         # Löse das nächstkleinere Problem ...
5         return n*factorial(n-1)
```

Rekursionsschema für `factorial(4)`:



Beispiel 6.5

Zum Vergleich die iterative Lösung der Fakultätsberechnung:

```
1 def factorial(n): # Iterative Berechnung von n! (Fakultät)
2     f = 1
3     for k in range(2, n+1):
4         f = k * f
5     return f
```

Man muss hier einräumen, dass die iterative Lösung nicht besonders schwierig zu verstehen bzw. zu programmieren ist

7 Ein- und Ausgabe

Ausgabe in der Shell

`print(`*`wert_1`*`, `*`wert_2`*`, ..., sep=' ', end='\n')`

Gibt die durch Kommas getrennten Werte auf der Standardausgabe (Shell) aus. Ohne Angabe der Parameter `sep=' '` und `end='\n'` werden die Voreinstellungen (Leerzeichen, Zeilenschaltung) verwendet.

Beispiel 7.1

```
1 print(4, 7, 9, 8, sep='+') # => 4+7+9+8
2
3 print(1, 2, 3, sep='\n')   # => 1
4                            #    2
5                            #    3
6
7 print('a', end='***')
8 print('b', end='***')
9 print('c', end='***\n')    # => a***b***c***
```

Eingaben von der Shell

`var` `= input(`*`string`*`)`

Zeigt auf der Shell die Zeichenkette *`string`* an und wartet, bis der Benutzer eine Eingabe gemacht und mit der ENTER-Taste abgeschlossen hat. Die Eingabe wird zusammen mit der „Zeilenschaltung" (von der Enter-Taste) der Variablen *`var`* zugewiesen.

Falls es sich bei der Eingabe um eine Zahl handelt, muss diese noch noch mit der Funktion `int(..)` in eine ganze Zahl bzw. mit der Funktion `float(..)` in eine Gleitkommazahl umgewandelt werden.

Beispiel 7.2

```
1 a = float(input('1. Zahl: '))
2 b = float(input('2. Zahl: '))
3 print((a + b)/2)
4
5 # Dies ergibt folgenden möglichen 'Dialog':
6 # 1. Zahl: 7 ENTER
7 # 2. Zahl: 2 ENTER
8 # 4.5
```

Ausgabe in eine Datei

1. Einen *File descriptor* zu Beschreiben öffnen:

   fd = open(*dateiname*, mode='w') ['w' steht für *write*]

2. die Ausgabe(n) in den Deskriptor schreiben:

   fd.write(*ausgabe_1*)

   fd.write(*ausgabe_2*)

   ...

3. Den Deskriptor wieder schliessen:

   fd.close()

fd	ein gültiger Bezeichner (meist `fd` oder `fh`)
dateiname	eine Zeichenkette mit dem Dateinamen
ausgabe_i	eine Ausgabe in Form einer Zeichenkette

Achtung: open(*dateiname*, ...) überschreibt eine existierende Datei ohne Rückfrage.

Beispiel 7.3

```
1 fd = open('myfile.txt', mode='w')
2
3 fd.write('Hello World!\n')
4 fd.write('{0}\n'.format(1234))
5 fd.write('{0}\n'.format(5678))
6
7 fd.close()
```

Inhalt der Datei `output.txt`:

```
Hello World!
1234
5678
```

Lesen aus einer Datei

1. Einen *File descriptor* zu Lesen öffnen:

   fd = open(*dateiname*, mode='r') ['r' steht für *read*]

2. die Ausgabe zeilenweise aus dem Deskriptor lesen und in *codeblock* verarbeiten

   ```
   for var in fd:
       codeblock
   ```

3. Den Deskriptor wieder schliessen:

   fd.close()

fd	ein Bezeichner (meist `fd` oder `fh`)
dateiname	eine Zeichenkette mit dem Dateinamen
var	ein Bezeicher für die aktuell gelesene Zeile

Beispiel 7.4

```
1 fd = open('myfile.txt', mode='r')
2
3 for line in fd:
4     # Da die Datei bereits Zeilenschaltungen enthält,
5     # werden die von print(...) erzeugten Zeilenschaltungen
6     # unterdrückt.
7     print(line, end='')
8
9 fd.close()
```

Ausgabe:

```
Hello World!
1234
5678
```

8 Zusammengesetzte Datentypen

8.1 Listen

Darstellung von Listen

Eine Liste wird durch eine kommaseparierte Folge von Werten beliebigen Datentyps definiert, die von einem Paar eckiger Klammern `[...]` eingeschlossen ist. Es ist auch möglich, eine Liste ohne Elemente zu definieren; dann spricht man von *der* leeren Liste.

Die Elemente einer Liste werden durch ihren Index, der bei Null beginnt, referenziert. Negative Indizes bedeuten, dass die Position vom Ende der Liste gezählt wird.

Beispiel 8.1.1

```
1 L = [25, True, -7, 'abc', 1.41421]
2 print(L[2])  # => -7
3 print(L[-2]) # => 'abc'
4 print(L[4])  # => IndexError
```

Slices

Aus einer Liste kann durch Angabe von zwei, durch einen Doppelpunkt getrennte Indizes, eine Teilliste (*slice*) „ausgeschnitten“ werden. Lässt man den ersten [den zweiten] Index weg, so werden alle Elemente vom Beginn [bis zum Ende] der Liste ausgewählt.

Beispiel 8.1.2

```
1 L = [3, 8, 1, 7, 5, 4, 2]
2 print(L[2:5]) # => [1, 7, 5]
3 print(L[3:])  # => [7, 5, 4, 2]
4 print(L[:3])  # => [3, 8, 1]
5 print(L[:])   # => [3, 8, 1, 7, 5, 4, 2]
```

Funktionen für Listen

Im Folgenden sei `L` eine Python-Liste.

Funktion	Wert
`len(L)`	Anzahl der Elemente von `L`
`sorted(L)`	Liste mit aufsteigend sortierten Elementen von `L`
`max(L)`	grösstes Element von `L`
`min(L)`	kleinstes Element von `L`
`sum(L)`	Summe aller Elemente einer Liste `L` aus Zahlen

Beispiel 8.1.3

```
L = [5, 3, 8, 2]
print(len(L))     # => 4
print(sorted(L))  # => [2, 3, 5, 8]
print(max(L))     # => 8
print(sum(L))     # => 18
```

Operatoren für Listen

Im Folgenden seien `A` und `B` Python-Listen sowie `n` eine natürliche Zahl.

Operator	Resultat
`A + B`	Liste aus den Elementen von `A` und `B`
`n * A`	Liste aus `n`-facher Repetition der Elemente von `A`

Beispiel 8.1.4

```
A = [1, 4]
B = [3, 2, 5]
print(A + B)  # => [1, 4, 3, 2, 5]
print(3*A)    # => [1, 4, 1, 4, 1, 4]
```

Methoden für Listen

Im Folgenden sind `L` und `M` Python-Listen, `i` ein gültiger Index sowie `e` ein beliebiges Objekt.

Methode	Beschreibung
`L.pop()`	Entfernt `L[-1]` und liefert es als Wert zurück.
`L.pop(i)`	Entfernt `L[i]` und liefert es als Wert zurück.
`L.append(e)`	Fügt `e` am Ende von `L` an.
`L.insert(i,e)`	Fügt `e` an der Position `i` ein.
`L.index(e)`	Index des ersten Auftretens von `e`.
`L.reverse()`	Wendet die Reihenfolge der Elemente *in place*.
`L.remove(e)`	Entfernt erstes Vorkommen von `e`.
`L.sort()`	Sortiert `L` *in place*.
`L.extend(M)`	Erweitert `L` um die Elemente aus `M` .

Beispiel 8.1.5

```
L = [3, -4, 5, 2, 7]
x = L.pop()
print(x, L)          # => 7 [3, -4, 5, 2]
L.append(8)
print(L)             # => [3, -4, 5, 2, 8]
L.insert(4, 5)
print(L)             # => [3, -4, 5, 2, 5, 8]
print(L.index(5))    # => 2
L.reverse()
print(L)             # => [8, 5, 2, 5, -4, 3]
L.remove(5)
print(L)             # => [8, 2, 5, -4, 3]
L.sort()
print(L)             # => [-4, 2, 3, 5, 8]
L.extend([1,4,7])
print(L)             # => [-4, 2, 3, 5, 8, 1, 4, 7]
```

Die listengesteuerte `for`-Schleife

```
for var in liste:
    codeblock
...
```

Die Elemente der Liste ***liste*** werden in der Reihenfolge ihres Auftretens der Variablen ***var*** zugewiesen und damit jeweils jeweils der ***codeblock*** ausgeführt.

Beispiel 8.1.6

```
s = 0
for x in [2, -5, 3, -7, 9]:
    if x > 0:
        s += x
print(s)           # Ausgabe: 14
```

8.2 Dictionaries

Überblick

In Python ist ein Dictionary eine Datenstruktur, die Werte mit Hilfe (unveränderlicher) Schlüssel speichert. In Dictionaries können Werte anhand ihres Schlüssels sehr schnell gefunden werden. Im Gegensatz zu Listen muss nicht jedes einzelne Element geprüft werden. Dafür verliert man die natürliche Ordnung der Elemente, wie man sie bei Listen kennt.

Darstellung von Dictionaries

Eine Dictionary besteht aus einer kommaseparierte Folge von Schlüssel-Wert-Paaren, die von geschweiften Klammern {...} eingeschlossen sind. Schlüssel und Wert werden jeweils durch einen Doppelpunkt getrennt. Die Schlüssel müssen einen konstanten Wert haben, der Wert kann einen beliebigen Datentyp haben. Es ist möglich, mit `dict()` ein leeres Dictionary zu definieren.

Beispiel 8.2.1

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

# Auf Werte zugreifen:
print(D['rot']) # => red

# Werte ändern:
D['blau'] = 'blue'

# Schlüssel-Wert-Paare hinzufügen:
D['schwarz'] = 'black'
D['weiss'] = 'white'

# Schlüssel-Wert-Paare entfernen:
del(D['grün'])
```

Dictionaries durchlaufen

Dictionaries können auf verschiedene Weise mit einer `for`-Schleife durchlaufen werden.

```
for key in mydict:
    print(key)

for key in mydict.keys():
    print(key)

for val in mydict.values():
    print(val)

for (key, val) in mydict.items():
    print(key, val)
```

Achtung: Die Reihenfolge, in der die Elemente ausgegeben werden, ist durch das effiziente Speicherverfahren bestimmt und muss nicht der Ordnung entsprechen, in der die Elemente zum Dictionary hinzugefügt wurden.

Beispiel 8.2.2

```
D = {1: 'A', 2: 'B', 3: 'C', 4: 'D', 5: 'E'}

for k in D:
    print(k, D[k]) # 1 A, 2 B, 3 C, 4 D

for k in D.keys():
    print(k, D[k]) # wie oben

for v in D.values():
    print(v)       # A, B, C, D

for (k, v) in D.items():
    print(k, v)    # 1 A, 2 B, 3 C, 4 D
```

9 Objektorientierung

Das Konzept

Eine Klasse ist ein Bauplan für *Objekte*. Jedes Objekt (*Instanz*) einer Klasse ...

- hat dieselben Variablen (*Eigenschaften*) mit individuellen Werten.
- kann Funktionen (*Methoden*) aufrufen, die Zugriff auf die Eigenschaften des Objekts haben.

Daneben können in einer Klasse auch Variablen und Funktionen definiert werden, die unabhängig von den Objekten sind.

- *Klassenvariablen* haben für alle Instanzen denselben Wert.
- *Klassenmethoden* können unabhängig von einem Objekt aufgerufen werden.

Grundstruktur einer Klassendefinition (ohne Vererbung)

```
class Myclass:

    def __init__(self, a1, a2, ...): # Konstruktor
        self.a1 = a1  # Objektvariable = Objekteigenschaft
        self.a2 = a2  # Objektvariable = Objekteigenschaft
        ...

    def objMethod(self, b1, b2, ...): # Objektmethode
        codeblock

    ...

    c1 = ... # Klassenvariable (ohne 'self')
    c2 = ... # Klassenvariable
    ...

    def clsMethod(d1, d2, ...): # Klassenmethode (ohne 'self')
        codeblock

    ...
```

Der Konstruktor

Die Spezialfunktion

```

    def __init__(self, a1, a2, ...): # Konstruktor
        self.a1 = a1  # Objektvariable = Objekteigenschaft
        self.a2 = a2  # Objektvariable = Objekteigenschaft
        ...
```

definiert innerhalb einer Klasse eine Funktion, die immer dann aufgerufen wird, wenn eine neues Objekt mit dem Klassennamen als Funktionsname und den Parametern `a1`, `a2`, ... aufgerufen wird. Die Variable `self` ist willkürlich gewählt und steht als Referenz auf das Objekt selbst.

Im Beispiel oben werden beim Aufruf des Konstruktors die aktuellen Parameter `a1`, `a2`, ... den entsprechenden Variablen des Objekts zugewiesen, womit es initialisiert bzw. erzeugt wird.

Objektmethoden

Jede Funktionsdefinition in einer Klasse, welche als ersten Parameter die Variable `self` enthält, kann im Funktionsrumpf auf die Eigenschaften des Objekts, d. h. `self.a1`, `self.a1`, ... zugreifen. Der Funktionsrumpf kann zusätzlich lokale Variablen enthalten.

```
1     def objMethod(self, b1, b2, ...): # Objektmethode
2         codeblock
```

Ist `x` eine Objekt der Klasse `Myclass`, dann wird die Objektmethode `objMethod(...)` mit der Punkt-Schreibweise auf das Objekt angewendet:

```
x.objMethod(b1, b2, ...)
```

Hier darf die Referenz `self` nicht mehr als Parameter angegeben werden, da sie der Objektname `x` bereits explizit enthält.

Klassenvariablen

Im Gegensatz zu Objektvariablen existiert eine Klassenvariable nur einmal. Sie wird innerhalb der Klasse ohne Voranstellen der Objektreferenz `self` definiert.

Jedes Objekt kann die Klassenvariable sehen und sie ändern. Die Änderung ist aber für alle anderen Objekte auch sichtbar.

```
1
2     c1 = ... # Klassenvariable (ohne 'self')
3     c2 = ... # Klassenvariable
4     ...
```

Ist die Klasse in Gebrauch, so können wir auf die obigen Klassenvariablen mit der Syntax `Myclass.c1` und `Myclass.c2` zugreifen.

Klassenmethoden

Soll eine Methode (Funktion in einer Klasse) unabhängig von einem Objekt definiert werden, so definiert man sie ohne den ersten formalen Parameter `self`.

```
1     def clsMethod(d1, d2, ...): # Klassenmethode (ohne 'self')
2         codeblock
3
4     ...
```

Klassenmethoden werden aufgerufen, inde man ihnen den Klassennamen und eine Punkt voranstellt. Hier würde das so aussehen:

```
Myclass.clsMethod(d1, d2, ...)
```

Beispiel 9.1

```
class Vector:

    count = 0 # Klassenvariable

    def __init__(self, x, y): # (Spezial)Objektmethode
        self.x = x  # Objektvariable
        self.y = y  # Objektvariable
        Vector.count += 1

    def __str__(self): # (Spezial)Objektmethode
        return '({0}|{1})'.format(self.x, self.y)

    def length(self): # Objektmethode
        return (self.x**2 + self.y**2)**0.5

    def add(u, v): # Klassenmethode
        return Vector(u.x + v.x, u.y + v.y)


a = Vector(3, 4)      # Aufruf des Konstruktors
b = Vector(-1, 2)     # Aufruf des Konstruktors
c = Vector.add(a, b)  # Vektoraddition

print(a.length())   # => 5.0     (Ausgabe der Länge von 'a')
print(c)            # => '(2|6)' (Ausgabe von Vektor 'c')
print(Vector.count) # => 3       (Ausgabe der Anzahl Vektoren)
```