Programmieren mit Python

Skript (Teil II)

5. Februar 2023

4 Listen

Ausblick

Ziele dieser Lektion:

- die Verarbeitung von Daten mit Listen
- das Lesen und Schreiben von Textdateien

Motivation

Oft wollen wir mit einem Computer viele zusammengehörende Werte verarbeiten. Dann ist es nicht sinnvoll, für jeden einzelnen Wert eine separate Variable zu definieren. Hier wäre eine *Datenstruktur* praktisch, in der man alle diese Werte unter einem gemeinsamen Namen speichern und bei Bedarf wieder hervorholen kann. *Beispiele:*

- Die mittleren Tagestemperaturen von Stans im Jahr 2021
- Die Namen der Schülerinnen und Schüler einer Klasse
- Eine Menge von Zahlen, die sortiert werden sollen

Der Python-Datentyp `list`

Python stellt uns mehrere Datenstrukturen zur Verfügung, mit denen wir Sammlungen von Objekten verwalten können. Eine am häufigsten gebrauchte Datenstruktur ist die *Liste*.

In Python ist eine Liste eine Menge von Objekten, die in einer bestimmten Reihenfolge stehen und auf die wir mit einem *Index* zugreifen können.

-7	Hello	3.14159	True
Index 0	Index 1	Index 2	Index 3

Im Gegensatz zu vielen anderen Programmiersprachen verlangt Python nicht, dass alle Elemente der Liste vom gleichen Datentyp sind (z. B. nur ganze Zahlen). Es ist sogar möglich, dass Elemente einer Liste selber wieder Listen sind.

Eingabe von Listen

Die oben symbolisch dargestellte Liste lässt sich in Python wie folgt definieren und beispielsweise der Variablen `L` zuweisen:

```
L = [-7, 'Hello', 3.14159, True]
```

Beachte dass für die eckigen Klammern („`[`“ und „`]`“) auf deutschen Windows-Tastaturen zuvor jeweils die `AltGr`-Taste (*alternate graphic*) gedrückt werden muss.

Wie Listen gespeichert werden

Dieses Modell geht davon aus, dass jedes Byte im Arbeitsspeicher des Computers eine Adresse hat (0, 1, 2, 3, . . .) und dass jedes Objekt genau 4 Byte Speicher benötigt.

Soll eine Liste mit den Durchschnittstemperaturen von 5 aufeinanderfolgenden Tagen unter dem Variablennamen `T` gespeichert werden, wird ein zusammenhängender Speicherbereich aus $5 \cdot 4 = 20$ aufeinanderfolgenden Speicherzellen gesucht und die Daten dort in der entsprechenden Reihenfolge abgelegt.

· · ·	4	3	7	3	2	· · ·
	100	104	108	112	116	120

In der Variablen `T` wird nur die Startadresse (100) und die Anzahl der Objekte (5) gespeichert.

Der Zugriff auf die Elemente

Der Zugriff auf die Elemente einer Liste erfolgt über den Index, der von zwei eckigen Klammern (`[...]`) eingeschlossen wird, die unmittelbar auf den Listennamen folgen.

Beginnt die Liste `T` mit den Temperaturwerten an der Adresse `100` und belegt jeder Wert genau 4 Bytes, dann wird mit `T[2]` die folgende Adresse berechnet:

$$100 + 2 \times 4 = 108$$

Somit bezeichnet `T[2]` den Wert `7`, der die 4 Bytes von Adresse 108 bis und mit Adresse 111 besetzt.

· · ·	4	3	7	3	2	· · ·
	100	104	108	112	116	120

Fragen

Auf welchen Wert in der obigen Liste `T` verweist der Ausdruck?

(a) `T[4]`

(b) `T[0]`

IndexError

Als Programmierer müssen wir uns nicht darum kümmern, wie viele Bytes ein Objekt beansprucht. Es genügt einfach, den *Offset* zum ersten Element anzugeben.

Wenn man jedoch einen Index angibt, der grösser oder gleich der Anzahl Listenelemente ist, dann wird ein `IndexError` ausgelöst, da sich an dieser Stelle keine Daten befinden, die zur Liste gehören.

```
T = [4, 3, 8, 4, 2]
print(T[5]) # IndexError: list index out of range
```

negative Listenindizes

Python bietet noch eine zweite Möglichkeit, um auf die Elemente einer Liste zuzugreifen: mit einem *negativen Listenindex*, der zur Endadresse des letzten Listenelements addiert wird.

Endet die Liste `T` mit den Temperaturwerten an der Adresse `120` und belegt jeder Wert genau 4 Bytes, dann wird mit `T[-2]` die folgende Adresse berechnet:

$$120 + (-2) \times 4 = 120 - 8 = 112$$

...	4	3	7	3	2	...
	100	104	108	112	116	120

Somit bezeichnet `T[-2]` den Wert `3`, der die 4 Bytes von Adresse 112 bis und mit Adresse 115 besetzt.

Listenelemente ändern

Wie bei Variablen, können auch den Listen mit dem Zuweisungsoperator (`=`) neue Werte zugewiesen werden. Dafür kombiniert man die Syntax für das Auslesen von Elementen mit einer Zuweisung. *Beispiel:*

```
1 L = [5, 8, 9, 7, 4]
2 L[3] = 1
3 L[-1] = 0
4 print(L) # => [5, 8, 9, 1, 0]
```

Verschachtelte Listen

In Python können Listen weitere Listen und diese wieder Listen enthalten. Solche Konstrukte nennt man mehrdimensionale Listen oder List of Lists (*LoL*).

Der Zugriff auf die Elemente erfolgt über mehrere Indizes. *Beispiel:*

```
1 M = [[6, 0, 8], [7, 4], [1, 9, 5, 9]]
2 print(M[0][2])  # => 8
3 print(M[1][0])  # => 7
4 print(M[2][-2]) # => 5
5 print(M[1])     # => [7, 4]
```

Slices

Manchmal ist es nützlich, nicht nur ein Element sondern einen zusammenhängenden Bereich (*Slice*) aus einer Liste auszuwählen.

Um aus einer Liste `L` alle Elemente vom Index `i` bis und mit Index `j-1` auszuwählen, schreibt man `L[i:j]`.

Ist $\texttt{i} \geq \texttt{j}$, dann ist die Slice die leere Liste (`[]`). Es dürfen auch Slice-Grenzen verwendet werden, die ausserhalb des gültigen Bereichs liegen (siehe Beispiel).

Wenn der erste Index `i` weggelassen wird, dann beginnt die Slice beim Element mit dem Index 0. Wenn der zweite Index `j` weggelassen wird, dann geht die Teilliste bis zum Listenende.

Beispiele:

```
L = [3, 8, 4, 1, 7, 6, 2]
print(L[1:4])   # => [8, 4, 1]
print(L[5:3])   # => []
print(L[2:2])   # => []
print(L[-5:-1]) # => [4, 1, 7, 6]
print(L[:2])    # => [3, 8]
print(L[3:])    # => [1, 7, 6, 2]
print(L[:])     # => [3, 8, 4, 1, 7, 6, 2]
print(L[4:100]) # => [7, 6, 2]
```

Notiere in den folgenden Übungen die Ausgabe(n) des Programms. Achte darauf, die Ausgabe einer Liste mit eckigen Klammern darzustellen. Notiere `IndexError`, falls ein ungültiger Index einen Fehler verursacht.

Übung 4.1

```
L = [5, 8, 7, 1, 4, 2]
print(L[2])
print(L[-2])
print(L[1+1])
print(L[-6])
print(L[2*3])
print(L[L[3]])
```

Übung 4.2

```
L = [4, 0, 2, 6, 5, 9]
L[0] = 8
L[-2] = 3
L[3] = 7
L[5] = L[5] - L[4]
print(L)
```

Übung 4.3

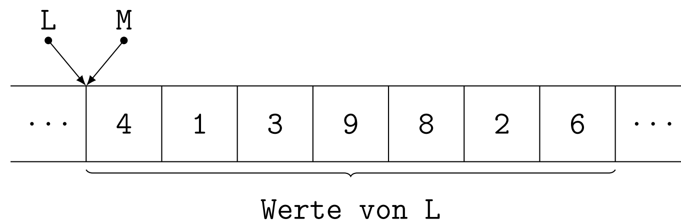
```
L = [[7, 4], [8, 1, 6], [5], [2, [9, 0], 3]]
print(L[1][1])
print(L[0][-1])
print(L[2])
print(L[-1][1][0])
```

Übung 4.4

```
L = [7, 4, 8, 1, 6, 5, 2]
print(L[1:5])
print(L[4:3])
print(L[3:4])
print(L[:2])
print(L[2:])
print(L[:])
print(L[5:100])
```

Listenvariablen sind Referenzen

Wenn man in Python den Wert einer Listenvariable `L` einer anderen Variable `M` zuweist, wird die Liste `L` nicht elementweise an eine andere Stelle im Speicher kopiert, sondern nur die Adresse des ersten Elements. Schematisch sieht das so aus:



Da beide Variablen, auf die gleiche Liste im Speicher verweisen, ist eine Veränderung der Liste mit der Variablen `L` auch in `M` sichtbar und umgekehrt.

Beispiel:

```
L = [4, 1, 3, 9, 8, 2, 6]
M = L
L[2] = 7
M[4] = 0
print(L) # => [4, 1, 7, 9, 0, 2, 6]
print(M) # => [4, 1, 7, 9, 0, 2, 6]
```

Warum ist das so?

Dieses Verhalten beruht auf einer Entscheidung, die von den Erfindern von Python aus Effizienzgründen getroffen wurde. Es geht viel schneller, eine einzelne Adresse zu kopieren als eine Liste mit sehr vielen Elementen zu duplizieren.

Bei Bedarf können wir eine echte Kopie der Liste `L` mit der Slice `L[:]` erzeugen.

Beispiel:

```
L = [4, 1, 3, 9, 8, 2, 6]
M = L[:] # eine 'echte' Kopie von L
L[2] = 7
M[4] = 0
print(L) # => [4, 1, 7, 9, 8, 2, 6]
print(M) # => [4, 1, 3, 9, 0, 2, 6]
```

Übung 4.5

Notiere die Ausgabe(n) des Programms. Achte darauf, die Ausgabe einer Liste mit eckigen Klammern darzustellen.

```
A = [7, 0, 1]
B = A
C = A[:]
A[0] = 3
B[1] = 9
C[2] = 5
print(A)
print(B)
print(C)
```

Operatoren für Listen

- *Addition (Verkettung) von Listen*

```
print([3, 4, 5] + [7, 8]) # => [3, 4, 5, 7, 8]
print([7, 8] + [3, 4, 5]) # => [7, 8, 3, 4, 5]
```

- *Multiplikation von Listen mit ganzen Zahlen*

```
print(3 * [2, 5])     # => [2, 5, 2, 5, 2, 5]
print([0] * 7)        # => [0, 0, 0, 0, 0, 0, 0]
print(0 * [3, 4, 5]) # => []
```

- *Ist ein Wert in einer Liste (nicht) enthalten?*

```
print(3 in [1, 2, 3])       # => True
print(4 in [1, 2, 3])       # => False
print(3 not in [1, 2, 3])  # => False
print(4 not in [1, 2, 3])  # => True
```

Funktionen für Listen

Hier eine Auswahl nützlicher Funktionen für Listen.

- `len(A)`: gibt die Anzahl der Elemente von `A` zurück.

```
A = [7, 1, 5, 3]
print(len(A))  # => 4
```

- `sorted(A)`: gibt die Liste mit den *aufsteigend* sortierten Elementen zurück, sofern sich diese sortieren lassen.

```
A = [7, 1, 5, 3]
print(sorted(A))  # => [1, 3, 5, 7]
```

mit dem Argument `reverse=True` werden die Listenelemente *absteigend* sortiert.

```
A = [7, 1, 5, 3]
print(sorted(A, reverse=True))  # => [7, 5, 3, 1]
```

Methoden für Listen

Auf eine Liste `L` können unter anderem die folgenden Methoden angewendet werden:

- `L.append(x)` fügt das Element `x` am Ende der Liste hinzu.

```
L = ['H', 'E', 'L', 'L']
L.append('O')
print(L)      # => ['H', 'E', 'L', 'L', 'O']
```

- `L.pop()` entfernt das letzte Element aus der Liste und liefert es als Wert zurück.

```
L = [7, 4, 3, 6]
x = L.pop()
print(x)      # => 6
print(L)      # => [7, 4, 3]
```

- `L.pop(i)` entfernt das Element an der Position `i` aus der Liste und liefert es als Wert zurück. Die Elemente an den ursprünglichen Positionen `i` und höher werden nach links verschoben, was ineffizient ist.

```
L = [1, 2, 4, 5]
x = L.pop(0)
print(x)     # => 1
print(L)     # => [2, 4, 5]
```

- `L.insert(i, x)` fügt an der Position `i` das Element `x` in die Liste ein. Die Elemente an den ursprünglichen Positionen `i` und höher werden nach rechts verschoben, was ineffizient ist.

```
L = [1, 2, 4, 5]
L.insert(2, 3)
print(L)        # => [1, 2, 3, 4, 5]
```

- `L.count(x)` zählt, wie oft das Objekt `x` in der Liste vorkommt.

```
L = [5, 4, 3.5, 4.5, 5]
print(L.count(5))    # => 2
print(L.count(5.5))  # => 0
```

- `L.reverse()` kehrt die Reihenfolge der Elemente um.

```
word = ['L', 'E', 'B', 'E', 'N']
word.reverse()
print(word) # => ['N', 'E', 'B', 'E', 'L']
```

Methoden sind Funktionen, die mit der Punkt-Schreibweise auf einen speziellen Datentyp (hier: Listenobjekte) angewendet werden.

Einige von ihnen wie `count()` liefern nur einen Wert zurück. Andere wie `reverse()` verändern allein das Objekt und wiederum andere wie `pop()` verändern das Objekt *und* liefern einen Wert zurück. Diese Semantik muss man entweder auswendig lernen oder in der Dokumentation nachschlagen. Auf der Python-Webseite

`https://docs.python.org/3/tutorial/datastructures.html`

findet man eine Beschreibung aller Listenmethoden. (Stand: 29.1.2022)

Übung 4.6

Notiere die Ausgaben des Programms.

```
print([7, 2, 4] + [5, 3])
print(3 * [1, 0])
print([4] * 5)
print(8 in [1, 2, 3])
print(8 not in [1, 2, 3])
```

Übung 4.7

Notiere die Ausgabe(n) des Programms.

```
A = [3, 8, -1, 5, 7]
B = ['p', 'A', 't']
print(len(A))
print(len(B))
print(len([]))
print(sorted(A))
print(sorted(B, reverse=True))
```

Übung 4.8

Notiere die Ausgabe(n) des Programms.

```
L = []
L.append(2)
L.append(0)
L.append(1)
print(L)
```

Übung 4.9

Notiere die Ausgabe(n) des Programms.

```
L = [3, 9]
L.insert(0, 7)
L.insert(0, 6)
L.insert(1, 5)
print(L)
```

Übung 4.10

Notiere die Ausgabe(n) des Programms.

```
L = [6, 4, 2, 7, 9]
a = L.pop()
b = L.pop(1)
print(a+b)
print(L)
```

Übung 4.11

Notiere die Ausgabe(n) des Programms.

```
L = [3, 1, 1, 3, 1]
a = L.count(1)
b = L.count(2)
print(a)
print(b)
```

Übung 4.12

Notiere die Ausgabe(n) des Programms.

```
L = [4, 1, 2, 5]
L.reverse()
print(L)
```

Listen elementgesteuert durchlaufen

Mit `for <var> in <list>` kann eine Liste elementweise *durchlaufen* werden:

```
words = ['words', 'are', 'very', 'unnecessary']
for w in words:
    print(w, len(w))
# Ausgabe:
# words 5
# are 3
# very 4
# unnecessary 11
```

Bemerkung: Weil Zeichenketten so etwas wie Listen aus Zeichen sind, können viele Listenfunktionen auch auf Zeichenketten angewendet werden. Ist `string` eine Zeichenkette, so gibt `len(string)` die Anzahl der Zeichen einer Zeichenkette oder `str[0]` das erste Zeichen von `string` zurück.

Schleifen mit `break` abbrechen

Mit dem Schlüsselwort `break` kann man eine Schleife frühzeitig beenden.

```
for zahl in [3, 5, 9, 13, 27, 48]:
    if zahl > 9:
        break
    print(zahl)
```

Die `for`-Schleife wird beendet, sobald die Variable `zahl` einen Wert grösser als 9 enthält. Daher gibt das Programm zeilenweise die Zahlen 3, 5 und 9 aus.

Bemerkung: `break` kann auch in `while`-Schleifen stehen.

Schleifendurchläufe mit `continue` überspringen

Steht innerhalb eines Schleifenblocks das Schlüsselwort `continue`, dann wird der Rest des Schleifenblocks übersprungen und der nächste Test im Schleifenkopf durchgeführt.

```
for zahl in [8, -3, 1, -7, 4]:
    if zahl < 0:
        continue
    print(zahl)
```

Das obige Programm lässt den Schleifendurchlauf aus, wenn das Listenelement in der Variablen `zahl` einen negativen Wert hat. Daher gibt es die Zahlen 8, 1 und 4 aus.

Bemerkung: `continue` kann auch in `while`-Schleifen stehen.

Listen indexgesteuert durchlaufen

Wenn man die Elemente einer Liste nicht nur verarbeiten sondern auch den Wert in der Liste durch einen neuen ersetzen will, genügt die einfache Schleife über die Liste nicht mehr. In diesem Fall erzeugt `range(a, b)` eine Art Liste von ganzen Zahlen, die in Einerschritten von `a` bis und mit `b-1` hochzählen und mit denen wir die Werte in der Liste mit ihrem Index ansprechen und bei Bedarf ändern können.

```
L = [2, 3, 4, 0, 1]
for i in range(0, len(L)):
    L[i] = 2*L[i]
print(L) # Ausgabe: [4, 6, 8, 0, 2]
```

Listen indexgesteuert durchlaufen (2)

Es gibt noch eine elegantere Variante, die obige Aufgabe zu lösen:

```
L = [2, 3, 4, 0, 1]
for i, x in enumerate(L)
    L[i] = 2*x
print(L) # Ausgabe: [4, 6, 8, 0, 2]
```

Die Funktion `enumerate(L)` erzeugt hier der Reihe nach die geordneten Paare

`(0, 2)`, `(1, 3)`, `(2, 4)`, `(3, 0)`, `(4, 1)`,

die jeweils aus dem Index `i` und dem zugehörigen Element der Liste `L[i]` bestehen. Diese beiden Werte werden für jeden Schleifendurchlauf in dieser Reihenfolge den beiden Bezeichnern `i` und `x` zugewiesen. Es können auch andere Variablen als `i` und `x` verwendet werden.

Übung 4.13

Notiere die Ausgabe(n) des Programms.

```
L = [3, 1, 2, 5]
s = 0
for x in L:
    s = s + x
print(s)
```

Übung 4.14

Notiere die Ausgabe(n) des Programms.

```
L = ['a', 'b', 'c', 'd', 'e', 'f']
s = ''
for x in L:
    if x == 'd':
        break
    s = s + x
print(s)
```

Übung 4.15

Notiere die Ausgabe(n) des Programms.

```
L = [3, 2, 8, 5, 7]
p = 1
for x in L:
    if x > 5:
        continue
    p = p * x
print(p)
```

Übung 4.16

Notiere die Ausgabe(n) des Programms.

```
L = [3, 2, 1, 7]
for i in range(0, len(L)):
    L[i] = L[i] + 10
print(L)
```

Übung 4.17

Notiere die Ausgabe(n) des Programms.

```
L = [5, 7, 4, 6]
for k, e in enumerate(L):
    L[k] = k * e
print(L)
```

5 Funktionen

Ziele dieser Lektion

- Sinn und Nutzen von Funktionen
- Erstellen von Funktionen
- Aufrufen von Funktionen

Motivation

Oft wollen wir ein bestimmtes Programmfragment mehrere Male verwenden. Anstatt bestimmten Code immer neu zu schreiben, können wir dem Code einen klingenden Namen geben und ihn über den Funktionsnamen immer wieder aufrufen.

“Funktions“-tüchtig progammieren

Analysiere folgendes Programm. Wie sieht es aus? Wie könnte man es verbessern?

```
1  hund_name = "Codie";
2  hund_gewicht = 40
3  if hund_gewicht > 20:
4      print(hund_name, 'sagt WAU WAU')
5  else:
6      print(hund_name, 'sagt wuff wuff')
7
8  hund_name = "Sparky";
9  hund_gewicht = 9
10 if hund_gewicht > 20:
11     print(hund_name, 'sagt WAU WAU')
12 else:
13     print(hund_name, 'sagt wuff wuff')

15 hund_name = "Bello";
16 hund_gewicht = 12
17 if hund_gewicht > 20:
18     print(hund_name, 'sagt WAU WAU')
19 else:
20     print(hund_name, 'sagt wuff wuff')
```

Wozu dienen Funktionen?

- Eine Funktion bildet ein Strukturierungselement, da sie eine Folge von Anweisungen unter einem Namen zusammenfasst.
- Ein längeres Programm erhält durch Funktionen eine Struktur, die helfen kann, den Code besser lesen und verstehen zu können.
- Ein guter Funktionsname kann dabei helfen zu verstehen, was das Unterprogramm berechnet oder ausführt.
- Muss eine Codesequenz mehr als einmal ausgeführt werden, so braucht man nur den Funktionsnamen aufzurufen (Vermeidung von Codeduplizität). Auch Verbesserungen am Code müssen nur einmal vorgenommen werden.

Eingebaute Funktionen

Wir müssen nicht alle Funktionen selber definieren. Python stellt uns vordefinierte Funktionen wie `print(...)`, `len(...)`, oder `open(...)` zur Verfügung. Eine vollständige Liste dieser „eingebauten“ Funktionen findet man an folgender Adresse im Web:

`https://docs.python.org/3/library/functions.html`.

Einige dieser vordefinierten Funktionen befinden sich in `Modulen`, die beim Programmstart explizit mit `import` *`modulname`* eingebunden werden müssen. Zum Beispiel `math`, `random` oder `time`. Wenn man ein Modul mit dem `import`-Befehl importiert, stehen danach alle darin definierten Funktionen und Variablen zur Verfügung.

Wie erstellt man eine Funktion?

Eine Funktion muss vor ihrem ersten Gebrauch definiert werden.

```
1 def bellen(name, gewicht):
2     if gewicht > 20:
3         print(name, 'sagt WAU WAU')
4     else:
5         print(name, 'sagt wuff wuff')
```

Das Schlüsselwort `def` leitet die Definition einer Funktion ein. Danach kommt der Name der Funktion, gefolgt von einem Paar runder Klammern `(...)`. Zwischen diesen Klammern stehen allfällige *formale Parameter*. Im Beispiel sind das `name` und `gewicht`. Zum Schluss kommt noch der obligate Doppelpunkt. Alle zur Funktion gehörenden Zeilen (*Funktionskörper* oder *Funktionsrumpf*) müssen wie bei Python üblich eingerückt sein.

Um Funktionen besser zu verstehen, ist es hilfreich, einen genaueren Blick darauf zu werfen, wie Python seine Objekte verwaltet.

Für jedes neue Objekt wird ein Eintrag in einer internen Tabelle angelegt, der den Bezeichner und die Adresse des Objekts enthält.

Bezeichner	Adresse
x	9788960
y	9788992
...	...

Auf diese Weise findet Python die Werte der Variablen wieder, wenn sie später im Programm erneut verwendet werden. Wir nennen den Bereich, in dem diese Variablen abgelegt werden, den *globalen Gültigkeitsbereich*.

Wird nun eine Funktion definiert, dann erhält diese einen separaten Speicherbereich, in dem sie ihre Objekte unabhängig vom globalen Gültigkeitsbereich ablegen kann. Dieser wird daher *lokaler Gültigkeitsbereich* genannt.

Sobald wir eine Funktion aufrufen, verwendet Python für die formalen Parameter und die in der Funktion verwendeten Variablen den lokalen Gültigkeitsbereich.

Gibt es im globalen Gültigkeitsbereich Variablen mit dem gleichen Namen, so werden diese während der Ausführung der Funktion vorübergehend „unsichtbar“. Sobald der Code der Funktion abgearbeitet wurde, wird der lokale Gültigkeitsbereich gelöscht und gleichnamige globale Variablen werden wieder sichtbar.

Enthält eine Funktion eine Variable, die im lokalen Kontext nicht definiert ist, sucht sie im globalen Kontext nach ihr und verwendet deren Wert, falls sie fündig wird.

Beispiel

Hinweis: die Funktion `id(`***`objekt`***`)` gibt die Speicheradresse von ***`objekt`*** aus.

```
def f(x):
    y = 5
    print('lokaler Kontext:')
    print('x =', x, '=>', id(x))
    print('y =', y, '=>', id(y))
    print('z =', z, '=>', id(z))

x = 1
y = 2
z = 3

print('globaler Kontext:')      # globaler Kontext:
print('x =', x, '=>', id(x))    # x = 1 => 140103409107184
print('y =', y, '=>', id(y))    # y = 2 => 140103409107216
print('z =', z, '=>', id(z))    # z = 3 => 140103409107248

f(4)                            # lokaler Kontext:
                                # x = 4 => 140103409107280
                                # y = 5 => 140103409107312
                                # z = 3 => 140103409107248

print('globaler Kontext:')      # globaler Kontext:
print('x =', x, '=>', id(x))    # x = 1 => 140103409107184
print('y =', y, '=>', id(y))    # y = 2 => 140103409107216
print('z =', z, '=>', id(z))    # z = 3 => 140103409107248
```

Aufruf einer Funktion

Sobald wir eine Funktion mit einem Namen (`bellen`) und den nötigen Parametern (`name`, `gewicht`) definiert haben, können wir sie aufrufen:

```
bellen('Fido', 65)
```

Aufgabe 5.1

Hier ein paar weitere Aufrufe unserer Funktion. Schreibe neben jeden Aufruf, wie die Ausgabe aussehen wird.

(a) `bellen('Speedy', 20)`

(b) `bellen('Bello', -1)`

(c) `bellen('Rex', 20, 0)`

(d) `bellen('Lady', '20')`

(e) `bellen('Rover', 21)`

Vorsicht bei Datentypen mit Referenzen

Übergibt man Listen oder andere zusammengesetzte Objekte an eine Funktion, so wird als Parameter eine *Referenz* auf das Objekt an die Funktion weitergereicht. Auch wenn diese Referenz im lokalen Kontext kopiert und später wieder gelöscht wird, verweist sie immer noch auf das ursprüngliche Objekt und kann es potenziell verändern.

Ist dieser Effekt unerwünscht, muss man der Funktion eine echte Kopie des Objekts übergeben. Bei einer Liste mit dem Namen `L`, die selber keine Listen oder komplexe Objekte als Elemente enthält, kann man eine solche Kopie mit der Slice-Syntax `L[:]` oder mit `L.copy()` erzeugen.

Das folgende Codefragment zeigt, wie dies funktioniert.

```
1  def removeFirst(L):
2      L.pop(0)
3      print(L)
4
5  A = [3, 2, 7, 5]
6  removeFirst(A)     # Ausgabe: [2, 7, 5]
7  print(A)           # Ausgabe: [2, 7, 5]
8
9  B = [4, 9, 3, 1]
10 removeFirst(B[:])  # Ausgabe: [9, 3, 1]
11 print(B)           # Ausgabe: [4, 9, 3, 1]
```

Aufgabe 5.2

Welche Ausgaben macht das folgende Programm?

```
def changeMe1(myList):
    myList.extend([4, 5, 6])
    print(myList)

myList = [1, 2, 3]
changeMe1(myList)
print(myList)
```

Aufgabe 5.3

Welche Ausgaben macht das folgende Programm?

```
def changeMe2(myList):
    myList = [4, 5, 6]
    print(myList)

myList = [1, 2, 3]
changeMe2(myList)
print(myList)
```

Aufgabe 5.4

Erkläre den Unterschied zwischen den Ausgaben der Programme in Aufgabe 5.3 und Aufgabe 5.4.

Funktionen mit Rückgabewerten

Die oben definierte Funktion `bellen(name, gewicht)` hat ihren Output in Form von `print`-Anweisungen ausgegeben.

Oft ist es aber praktischer, wenn das Ergebnis einer Funktion nicht sofort angezeigt, sondern als Wert an die Stelle des Funktionsaufrufs gesetzt wird, damit dieser weiterverarbeitet werden kann.

Beispiel

Eine Firma, die Zäune baut, berechnet die Gesamtkosten für Material und Arbeit mit dem folgenden Python-Programm:

```
def material(meter, kosten_pro_meter):
    return meter * kosten_pro_meter

def arbeit(stunden, kosten_pro_stunde):
    return stunden * kosten_pro_stunde

kosten_total = material(20, 40) + arbeit(5, 60)
print(kosten_total)
```

Erläuterungen zum Programmablauf

1. In der Zeile 7 muss der Python-Interpreter zuerst den Ausdruck rechts vom Zuweisungsoperator (`=`) bestimmen:

   `kosten_total = material(20, 40) + arbeit(5, 60)`

2. Dafür wird zuerst `material(20, 40)` aufgerufen. Die `return`-Anweisung in Zeile 2 bewirkt, dass das Produkt $20 \cdot 40 = 800$ nach der Ausführung der Funktion an die Stelle von `material(20, 40)` gesetzt wird:

   `kosten_total = 800 + arbeit(5, 60)`

3. Dasselbe gilt auch für dein Aufruf von `arbeit(5, 60)`, der dafür sorgt, dass das Ergebnis 300 an die Stelle von `arbeit(5, 60)` gesetzt wird:

   `kosten_total = 800 + 300` $\Rightarrow$ `kosten_total = 1100`

Wichtig: Eine Funktion, in der keine `return`-Anweisung vorkommt, gibt `None` (der Python-Wert für „Nichts“) zurück.

Die Kurzschlusseigenschaft von `return`

Das Auftreten von `return` bewirkt, dass die Abarbeitung der Funktion sofort beendet und ein allfälliger Wert zurückgegeben wird. Alle danach folgenden Anweisungen werden ignoriert, auch wenn sie syntaktisch noch zur Funktion gehören (*dead code*).

```
def summe(a, b, c):
    s = a + b + c
    return s # Funktion stoppt hier
    print(s) # "dead code"

summe(4, 5, 6) # => keine Ausgabe
```

Es ist aber möglich, dass in einer Funktion mehrere `return`-Anweisungen stehen. Der Rückgabewert kann beispielsweise von einer Bedingung abhängig sein oder bereits im Innern einer Schleife feststehen.

```python
def nichtNegativ(L):
    for x in L:
        if x < 0: # beim ersten x < 0 => Abbruch mit False
            return False
    return True # offenbar kein einziges x < 0 => return True

print(nichtNegativ([3, 5, 100, -2, 8])) # => False
print(nichtNegativ([8, 0, 2, 7, 9]))    # => True
```

Aufgabe 5.5

Schreibe neben jeden Funktionsaufruf dessen Rückgabewert.

```python
def gruss(name):
    return 'Hallo ' + name + '!'
```

(a) `gruss('Speedy')`

(b) `gruss('Hallo')`

Aufgabe 5.6

Schreibe neben jeden Funktionsaufruf dessen Rückgabewert.

```python
def berechne(x, y):
    total = x + y
    if (total > 10):
        total = 10
    return total
```

(a) `berechne(2, 3)`

(b) `berechne(11, 5)`

Aufgabe 5.7

Schreibe neben jeden Funktionsaufruf dessen Rückgabewert.

```python
def erlaube_zugang(person):
    if person == 'Dr Evil':
        antwort = 'ja'
    else:
        antwort = 'nein'
    return antwort
```

(a) `erlaube_zugang('Mia')`

(b) `erlaube_zugang('Dr Evil')`

Aufgabe 5.8

Welche Ausgabe macht das folgende Programm?

```
def f(x):
    return 3*x + 4

def f(x):
    return x - 5

y = f(6) + f(7)
print(y)
```

Funktionen ohne Rückgabewert (gibt es nicht)

Grundsätzlich liefert jede Python-Funktion einen *Wert* zurück. Ohne eine `return`-Anweisung gibt Python den „leeren Wert“ `None` zurück.

```
def missingReturn(x):
    y = 2*x

value = missingReturn(3)

print(value) # Ausgabe: None
```

Die oben definierte Funktion ist in einem gewissen Sinne „nutzlos“. Sie berechnet (lokal) das Doppelte des aktuellen Parameters (3) und weist diesen Wert (6) der Variablen `y` zu. Nachdem die Funktion ausgeführt wurde, werden die lokalen Variablen `x` und `y` wieder gelöscht.

Funktion mit mehreren Rückgabewerten

Eine Python-Funktion kann nur *einen* Wert zurückgeben. Es ist aber möglich, mehrere Werte gleichzeitig an den aufrufenden Code weiterzuleiten, wenn man sie in *einer* Liste oder in *einem* Tupel zusammenfasst. Da Python diese Datenstrukturen als *einzelnes* Objekt auffasst, kann es sie (genauer: ihre Adresse) als Wert zurückgeben. *Beispiel:*

```
def punktAufGraph(x):
    y = 2*x + 3
    return [x, y]

print(punktAufGraph(7)) # Ausgabe: [7, 17]
```

Benannte Parameter

Beim Funktionsaufruf müssen die aktuellen Parameter in der gleichen Reihenfolge wie in der Funktionsdefinition angegeben werden. Bei Funktionen mit vielen Argumenten ist es schwierig, sich diese Reihenfolge zu merken. Stimmt die Reihenfolge der aktuellen Parameter nicht mit der Reihenfolge der formalen Parameter in der Funktionsdefinition überein, erhält man im besten Fall eine Fehlermeldung und im schlimmsten Fall ein falsches Resultat, ohne dass man es bemerkt.

Wenn man beim Aufrufen einer Funktion in der Parameterliste die aktuellen Parameter

den formalen Parameter zuweist, können die Parameter auch in einer beliebigen Reihenfolge angegeben werden. *Beispiel:*

```
def zins(K, p, t):
    return K * p * t / (100 * 360)

# unklar:
print(zins(1000, 2, 900))        # Ausgabe: 50.0 (Franken)
# besser:
print(zins(K=1000, p=2, t=900))  # Ausgabe: 50.0 (Franken)
# das geht auch:
print(zins(t=900, K=1000, p=2))  # Ausgabe: 50.0 (Franken)
```

Aufgabe 5.9

Welche Ausgaben macht das folgende Programmfragment?

```
def funktion(a, b, c):
    return a * b + c

print(funktion(3, 4, 5))
print(funktion(b=2, c=7, a=1))
print(funktion(c=0, a=1, b=8))
```

Standardargumente

Möchte man, dass Python bei einem Funktionsaufruf bei fehlenden aktuellen Parametern jeweils einen Standardwert einsetzt, so kann man diesen Wert bei der Funktionsdefinition in der Parameterliste angeben. Es muss aber darauf geachtet werden, dass alle Parameter mit Standardargumenten am Ende der Parameterliste stehen.

Das folgende Beispiel ersetzt beim Fehlen des Parameters `ort` jeweils den Wert `'Stans'` ein, wenn dies ein Ort ist, der häufig vorkommt.

```
def adressenAusgabe(name, vorname, strasse, ort='Stans'):
    print(vorname, name)
    print(strasse)
    print(ort)

adressenAusgabe('Ratlos', 'Rudi', 'Feldweg 9')
# Rudi Ratlos
# Feldweg 9
# Stans
```

Aufgabe 5.10

Welche Ausgaben macht das folgende Programmfragment?

```
def rechteckInhalt(a=1, b=1):
    return a * b

print(rechteckInhalt(3, 4))
print(rechteckInhalt(7))
print(rechteckInhalt())
```

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.

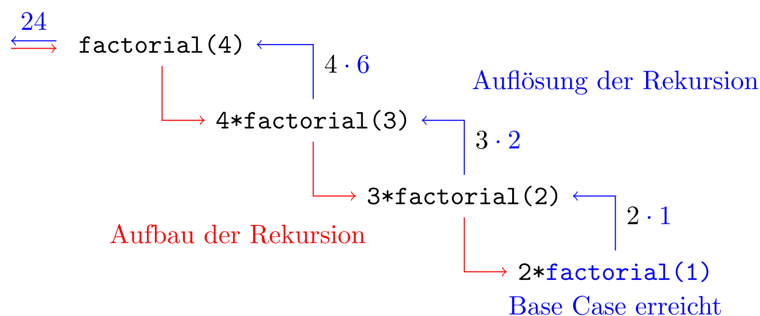
Da bei jedem Funktionsaufruf ein neuer Kontext für die Speicherung von Variablen angelegt werden muss, kostet die rekursive Lösung einer Aufgabe (im Gegensatz zur iterativen Lösung mit Schleifen) zusätzlichen Arbeitsspeicher. Deshalb wird Rekursion dann angewendet, wenn sich die Problemgrösse bei jedem 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 5.11

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

Rekursionsschema für `factorial(4)`:



Beispiel 5.12

Zum Vergleich die iterative Lösung der Fakultätsberechnung:

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

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

Docstrings

Ein *Docstring* ist eine Zeichenkette, die als erster Ausdruck im Funktionsrumpf erscheint und in der Regel Informationen über die Parameter, den Rückgabewert und andere Details der Funktion enthält. Es ist der ideale Ort, die Funktion zu dokumentieren, da der Python-Compiler den Docstring beim ersten Kompilieren der Funktion speichert und später in IDLE und anderen Editoren als Hilfetext anzeigt.

Aufgabe 5.13

Welche Ausgaben macht das folgende Programm?

```
def mittelwert(L):
    '''Gibt den Durchschnitt der Zahlen in der Liste L zurück.'''
    return sum(L)/len(L)

help(mittelwert)

print(mittelwert([13, 4, 7]))
```

Das Importieren von Modulen

Funktionen und Variablen, die sich in Modulen befinden, können auch in anderen Modulen benutzt werden. Auf diese Weise unterstützt Python die Wiederverwendung von bestehendem Code.

Damit Funktionen und Variablen, die in verschiedenen Modulen definiert sind, aber den gleichen Namen haben, beim Importieren keine Namenskonflikte verursachen, werden sie in ihren eigenen Namensraum geladen. Umgekehrt bedeutet dies, dass wir in einem Programm mehrere Funktionen mit dem gleichen Namen verwenden können, sofern sich diese in unterschiedlichen Namensräumen befinden.

Datei `mymodule1.py`:

```
v = 42
def f(x):
    return 2*x + 3
```

Datei `mymodule2.py`:

```
v = 73
def f(x):
    return x**2 + 1
```

Datei `mymodule3.py`:

```
import mymodule1
import mymodule2

print(mymodule1.v)      # => 42
print(mymodule2.v)      # => 73

print(mymodule1.f(3))   # => 9
print(mymodule2.f(3))   # => 10
```

Vorsicht

Bei der Namensgebung eigener Module sollte man vorher prüfen, ob der Name nicht schon von einem Python-Modul verwendet wird. Siehe dazu:

`https://docs.python.org/3/py-modindex.html`

In diesem Fall wird in der Regel das Modul der Python-Installation statt des benutzerdefinierten Moduls geladen, was seltsame Effekte bzw. Fehler(meldungen) zur Folge hat.

Die „Top-Level Code-Umgebung“

Stehen am Ende eines Moduls die Zeilen

```
if __name__ == '__main__':

    # Hier steht Code zum Testen des Moduls
```

prüft der Python-Interpreter, ob es sich beim aktuellen Modul (`__name__`) um das erste vom Benutzer ausgeführte Modul (`'__main__'`) handelt.

Hat diese Zeile den Wert `True` dann wird der danach folgende Code ausgeführt und sonst nicht. Damit können die Funktionen in einem Bibliotheksmodul getestet werden, wenn es zuerst ausgeführt wird. Wird das Bibliotheksmodul hingegen von einem anderen Modul geladen (so wie es gedacht ist), dann wird der Testcode nicht ausgeführt und es werden keine unnötigen Testresultate ausgegeben.

Aufgabe 5.14

Beantworte die Fragen stichwortartig:

- Weshalb sind Funktionen nützlich?

.

- Mit welchem Schlüsselwort definiert man eine Funktion?
- Welche Anweisung braucht es, um mit einer Funktion einen Wert zurückzuliefern?
- Kann eine Funktion mehr als einen Parameter haben?
- Was ist der Unterschied zwischen einer `return`-Anweisung und `print(...)`?
- Wie heissen Variablen, die nur innerhalb einer Funktion verwendet werden?
- Wie nennt sich auf Englisch die Lebensdauer einer Variablen?
- Welchen Rückgabewert hat eine Funktion, die keine `return`-Anweisung enthält?
- Kann mit der `return`-Anweisung mehr als ein Wert zurückgegeben werden?

6 Lesen und Schreiben von Dateien

Dateien lesen

Das Lesen von Textdateien mit Python erfordert drei Schritte:

1. Die Datei mit `myfile = open(`*`dateiname`*`, mode='r')` zum Lesen öffnen und das Dateiobjekt einer Variablen (hier: `myfile`) zuweisen.
2. Die Datei mit `for zeile in myfile` zeilenweise oder mit `myfile.read()` vollständig auslesen und verarbeiten.
3. Die Datei mit `myfile.close()` schliessen.

Beim Öffnen einer Datei können unter anderem folgende Fehler auftreten:

- `FileNotFoundError` (Datei extistiert nicht)
- `PermissionError` (Datei ist geschützt)
- `IsADirectory` (keine Datei sondern Verzeichnis)

Beispiel 6.1

Die Datei `mytest.txt` mit dem Inhalt

```
abc
123
```

wird mit einer `for`-Schleife zeilenweise ausgelesen und ausgegeben:

```python
myfile = open('mytext.txt', mode='r') # r=read
for zeile in myfile:
    print(zeile)
myfile.close()
```

Ausgabe:

```
abc

123

```

Um zu erkennen, warum bei der Ausgabe so viel Zwischenraum entsteht, lesen wir dieselbe Datei binär (byteweise) aus:

```python
myfile = open('mytext.txt', mode='rb') # rb=read binary
for zeile in myfile:
    print(zeile)
myfile.close()
```

Ausgabe:

```
b'abc\n'
b'123\n'
```

Das `b` am Zeilenanfang steht für Binärdaten. Man erkennt, dass am Zeilenende jeweils ein „unsichtbares“ Zeichen (`\n`) steht. Dieses *Newline*-Zeichen stammt vom Zeilenumbruch in der Datei.

Da auch `print(...)` jeweils einen Zeilenumbruch einfügt, erklärt dies die doppelte Zeilenschaltung.

Beispiel 6.1 (verbessert)

```
myfile = open('mytext.txt', mode='r')
for zeile in myfile:
    zeile = zeile.rstrip()
    print(zeile)
myfile.close()
```

Die String-Methode `str.rstrip()` entfernt auf der rechten Seite der Zeichenkette `str` allfällige *Whitespaces* (Leerzeichen, Tabulatoren, Zeilenschaltungen) und gibt die veränderte Zeichenkette als Wert zurück.

Auf diese Weise gelangt nur noch die Zeilenschaltung von `print()` in die Ausgabe.

Dateien mit Zahlen verarbeiten

Wie bei der Funktion `input()` werden die gelesenen Zeilen einer Textdatei als Zeichenketten interpretiert. Daher müssen Zahlen in Textform nach dem Einlesen umgewandelt werden.

Ist man sich sicher, dass jede Zeichenkette `string` aus einer ganzen Zahl besteht, sollte man zur Umwandlung die Funktion `int(string)` verwenden.

Wenn mindestens eine Zeichenkette `string` eine Zahl mit Dezimalpunkt enthält, muss man zur Umwandlung die Funktion `float(string)` verwenden.

Beispiel 6.2

Die Datei `numbers.txt` enthält folgende Daten:

```
2
3
7
1
```

Das folgende Programm erzeugt eine leere Liste, liest die Datei zeilenweise ein, wandelt die Zeichenkette in eine ganze Zahl um (die Zeilenschaltung wird dabei ignoriert) und fügt die Zahl am Ende der Liste `data` ein.

```
data = []
myfile = open('numbers.txt', mode='r')
for zeile in myfile:
    data.append(int(zeile))
myfile.close()
print(data) # Ausgabe: [2, 3, 7, 1]
```

Text in eine Datei schreiben

Das Schreiben von Textdateien mit Python erfordert drei Schritte:

1. Die Datei mit `myfile = open(`*`dateiname`*`, mode='w')` zum Schreiben öffnen und das Dateiobjekt einer Variablen (hier: `myfile`) zuweisen. **Achtung:** Eine bestehende Datei mit gleichem Namen wird überschrieben.

2. Den gewünschten Text in Form einer Zeichenkette `string` mit `myfile.write(string)` in die Datei schreiben. Dieser Vorgang kann beliebig wiederholt werden.

3. Die Datei mit `myfile.close()` schliessen.

Bemerkung: Solange eine Datei nach dem Beschreiben nicht geschlossen wird, kann sie nicht von einem anderen Programm geöffnet werden. Umgekehrt ist es möglich, dass eine Datei, die von einem anderen Programm geöffnet wurde, nicht von Python überschrieben werden kann.

Beispiel 6.3

```
1 myfile = open('parabel.csv', mode='w')
2 for i in range(-5, 6):
3     myfile.write('{0};{1}\n'.format(i, i**2))
4 myfile.close()
```

- Zeile 1: Die Datei `parabel.csv` wird zum Beschreiben geöffnet.

- Zeile 2: Die Variable `i` durchläuft die ganzen Zahlen von `-5` bis und mit `5`.

- Zeile 3: Der jeweilige Wert von `i` und sein Quadrat `i**2` wird, durch einen Strichpunkt getrennt, in einen Formatstring geschrieben und von einer Zeilenschaltung abgeschlossen. Dieser String wird in die Datei geschrieben.

- Zeile 4: Die Datei wird geschlossen.

Die Datei `parabel.csv` hat dann folgenden Inhalt:

```
1  -5;25
2  -4;16
3  -3;9
4  -2;4
5  -1;1
6  0;0
7  1;1
8  2;4
9  3;9
10 4;16
11 5;25
```

7 Objektorientierte Programmierung

Inhalte

- prozedurale Programmierung
- Objekte sind abstrakte Datentypen
- Definition einer eigenen Klasse
- Der Konstruktor
- Spezialmethoden
- Objekte in Aktion
- Instanzvariablen und -methoden
- Klassenvariablen und -methoden
- Vererbung
- Überladen von Operatoren

7.1 Die Ausgangslage

Wir möchten ein Programm schreiben, das uns Übungsaufgaben für Rechtecke erzeugt. Damit soll überprüft werden, ob die Lernenden

- die Länge des Umfangs,
- den Inhalt der Fläche und
- die Länge der Diagonale

aus den beiden Seitenlängen berechnen können.

7.2 Der bisherige Lösungsansatz

Wir würden ein Rechteck höchstwahrscheinlich durch zwei Zahlen (Länge und Breite) mit den zugehörigen Masseinheiten darstellen. Der Einfachheit halber setzen wir voraus, dass beide Masseinheiten gleich sind. In einem Python-Programm sähe das dann so aus:

```
laenge = 3
breite = 4
einheit = 'cm'
```

Rechtecke als Listen darstellen

Um mehrere Rechtecke zu speichern, können wir eine Liste von Listen (LoL) verwenden. Dazu müssen wir eine bestimmte Reihenfolge der drei Werte festlegen:

```
15 rechtecke = [[3, 4, 'cm'], [5, 3.4, 'm'], [7, 1, 'mm']]
```

Problem: Länge und Breite können verwechselt werden.

Rechtecke als Dictionaries darstellen

Eine zweite Möglichkeit ist die Verwendung einer Liste von Dictionaries. Hier besteht keine Gefahr der Verwechslung. Dafür ist der Aufwand beim Definieren der Rechtecke etwas grösser.

```
rechtecke = [{'L': 3, 'B': 4, 'E': 'cm'},
             {'L': 5, 'B': 3.4, 'E': 'm'},
             {'L': 7, 'B': 1, 'E':'mm'}]
```

Die Berechnung des Umfangs

Auch hier wissen wir, was zu tun ist. Wir schreiben eine Funktion, welche die Parameter eines Rechtecks als Argumente entgegennimmt und seinen Umfang als String zurückgibt.

```
3 def umfang(laenge, breite, einheit):
4     u = 2*(laenge + breite)
5     return '{0}{1}'.format(u, einheit)
```

Die Berechnung des Flächeninhalts

Diese Funktion wird analog definiert. Die hochgestellte „2“ der Masseinheit ist das Unicode-Zeichen mit der Nummer `0x00B2`.

```
7 def inhalt(laenge, breite, einheit):
8     A = laenge * breite
9     return '{0}{1}\u00B2'.format(A, einheit)
```

Die Berechnung der Diagonalen

Das sollte mit dem Satz des Pythagoras kein Problem sein. Beachte, dass die Quadratwurzel als Potenz mit dem Exponenten $\frac{1}{2}$ geschrieben werden kann: $\sqrt{x} = x^{\frac{1}{2}}$.

```
11 def diagonale(laenge, breite, einheit):
12     d = (laenge**2 + breite**2)**0.5
13     return '{0:.4f}{1}'.format(d, einheit)
```

Der Format-Parameter {`0:.4f`} bedeutet, dass der nullte Parameter als `float` mit vier Stellen nach dem Dezimalpunkt dargestellt werden soll.

Erzeugung der Aufgaben und Lösungen

Mit dem bereits geschriebenen Code lassen sich Aufgaben und Lösungen erzeugen:

```
17 for i, r in enumerate(rechtecke):
18     print(f'{i+1}. Berechne Umfang, Inhalt und', end=' ')
19     print('Diagonale des Rechtecks mit', end=' ')
20     print(f'a={r[0]}{r[2]}, b={r[1]}{r[2]}.\n')
21 
22 for i, r in enumerate(rechtecke):
23     u = umfang(r[0], r[1], r[2])
24     A = inhalt(r[0], r[1], r[2])
25     d = diagonale(r[0], r[1], r[2])
26     print(f'{i+1}. u={u}, A={A}, d={d}\n')
```

Erläuterungen zu den Codezeilen 17–26

- `enumerate(<Liste>)` gibt die mit 0 beginnende Nummer zusammen mit dem entsprechenden Listenelement zurück und ordnet sie hier für jeden Schleifendurchlauf den Variablen `i` (Nummer) und `r` (Rechteck) zu.
- `f'{i+1} ...'` ist ein *f-string*. Das `f` vor dem String sorgt dafür, dass Python-Ausdrücke innerhalb geschweifter Klammern ausgewertet und automatisch in einen String umgewandelt werden.
- Das Argument `end='...'` in `print()` ersetzt die Zeilenschaltung bei der Ausgabe am Ende durch den String `'...'` (hier ein Leerzeichen). Es wurde hier verwendet, damit die Zeilen nicht zu lang werden.

Prozedurale Programmierung

Der oben beschriebene Lösungsweg wird als *prozedurale Programmierung* bezeichnet. Bei diesem Vorgehen wird die algorithmische Aufgabe in kleinere (überschaubare) Teilprobleme zerlegt und dann mit Hilfe von Variablen und Funktionen gelöst.

Ein Nachteil bei diesem Vorgehen ist, dass den Funktionen versehentlich falsche Parameter oder Parameter in falscher Reihenfolge übergeben werden können, was zu Fehlern führt.

Ferner sind nun die Funktionsnamen `umfang(...)` `inhalt(...)` und `diagonale(...)` im Modul bereits besetzt und können beispielsweise nicht mehr für die Berechnung anderer Figuren (Dreiecke, Kreise, ...) verwendet werden.

Die Softwarekrise

In der Mitte der 1990er Jahre stellte man fest, dass es immer schwieriger wurde, für die immer leistungsfähigeren Computer grosse Softwaresysteme termingerecht und fehlerfrei zu entwickeln. Verbesserung versprach man sich unter anderem vom Konzept der „Objektorientierung“, das teilweise bereits in den 1970er Jahren in der Programmiersprache *Smalltalk* realisiert wurde.

7.3 Der objektorientierte Lösungsansatz

Bei der objektorientierten Programmierung werden die Daten (*Eigenschaften*) und die sie verarbeitenden Funktionen (*Methoden*) in einem *Objekt* zusammengefasst. Diese Objekte sind Akteure, die Aufträge erledigen, ihren Zustand ändern oder mit anderen Objekten kommunizieren können.

Um für die für eine Aufgabe massgeschneiderten Objekte zu erhalten, müssen wir diese in einer *Klasse* definieren. Eine Klasse ist der Bauplan, der die Eigenschaften und Handlungsmöglichkeiten der zur gleichen Klasse gehörenden Objekte festlegt.

Die Klassendefinition

```
3 class Rechteck:
```

Eine Klassendefinition wird mit dem Schlüsselwort `class` eingeleitet, auf das ein selbst gewählter Klassenname folgt. Es ist üblich, selbst definierte Klassen mit einem Grossbuchstaben beginnen zu lassen, um sie von den Klassen zu unterscheiden, die uns Python zur Verfügung stellt. Der nach `class <Klassenname>:` folgende Code muss wie bei Funktionen, Schleifen oder Verzweigungen eingerückt werden.

Wie bei Funktionen unterscheiden wir zwischen ihrer Definition und ihrer Anwendung. Was nach dem Klassennamen folgt, ist erst die Definition des Bauplans der Objekte. Danach können wir Objekte gemäss diesem Bauplan erzeugen und sie für uns arbeiten lassen.

Der Konstruktor

```
5     def __init__(self, laenge, breite, einheit):
6         self.a = laenge
7         self.b = breite
8         self.e = einheit
```

Der *Konstruktor* ist eine spezielle Funktion innerhalb einer Klasse mit dem reservierten Namen `__init__(self,...)`. Er definiert, welche *Eigenschaften* (Variablen) ein Objekt haben soll, und weist ihnen später bei der Objekterzeugung konkrete Werte zu. Der erste Parameter (üblicherweise `self`) steht für die Referenz auf das Objekt selbst. Die Schreibweise `self.a` drückt beispielsweise aus, dass es sich um die Länge des jeweiligen Rechtecks handelt.

Der Konstruktor in Aktion

```
class Rechteck:

    def __init__(self, laenge, breite, einheit):
        self.a = laenge
        self.b = breite
        self.e = einheit

r1 = Rechteck(3, 4, 'cm')
r2 = Rechteck(8, 5, 'm')

print(r1.b) # => 4
print(r2.e) # => 'm'
print(r1)   # => <__main__.Rechteck object at 0x7...0730>
print(r2)   # => <__main__.Rechteck object at 0x7...0070>
```

Wir können erkennen, dass in den ersten beiden Ausgaben jedes Objekt weiss, welche Werte zu ihm gehören. Damit das funktioniert, mussten wir in der Methode `__init__(self,...)` den formalen Parameter `self` verwenden.

Gibt man das Objekt selber mit `print(...)` aus, so erscheint die Adresse, an der Python die Daten des jeweiligen Objekts ablegt. Um eine für uns hilfreichere Textdarstellung eines Objekts zu bekommen, schreiben wir eine andere Spezialmethode die unter anderem dann aufgerufen wird, wenn ein Rechteck-Objekt mit `print(...)` ausgeben wird.

Die Spezialmethode `__str__(...)`

```
10    def __str__(self):
11        return 'a={0.a}{0.e}, b={0.b}{0.e}'.format(self)
```

Danach können wir mit `str(r1)` diese Methode auf die Objekte der Klasse anwenden.

Dies ist aber gar nicht nötig da `print(r1)` automatisch die Methode `str(r1)` aufruft, um die Textdarstellung des Objekts zu erhalten. In unserem Beispiel:

```
r1 = Rechteck(3, 4, 'cm')
print(r1)   # => a=3cm, b=4cm
```

(Objekt-)Methoden

In einer Klasse können wir Funktionen definieren, die den Zustand eines Objekts verändern oder die es mit der Aussenwelt oder mit anderen Objekten kommunizieren lässt.

Diese Funktionen werden im Jargon der objektorientierten Programmierung *Methoden* genannt. Eine *Methode* kann man sich als Nachricht an das Objekt vorstellen, etwas bestimmtes *zu tun*.

Da jedes Objekt seine persönlichen Daten „mit sich herumträgt“, müssen diese nicht jedes mal neu als Parameter übergeben werden. Mit der Syntax `self.name` können wir uns bei der Definition der Methoden auf die Variable *name* des jeweiligen Objekts beziehen.

Die Berechnung von Umfang, Inhalt und Diagonale

Nun können wir wie beim prozeduralen Ansatz die Berechnungsfunktionen definieren. Statt jeweils immer alle drei Parameter anzugeben, erhalten wir hier mit dem Parameter `self` Zugriff auf alle Eigenschaften (Variablen) des jeweiligen Objektes.

```
13      def umfang(self):
14          u = 2*(self.a + self.b)
15          return '{0}{1}'.format(u, self.e)
16
17      def inhalt(self):
18          A = self.a*self.b
19          return '{0}{1}\u00B2'.format(A, self.e)
20
21      def diagonale(self):
22          d = (self.a**2 + self.b**2)**0.5
23          return '{0:.4f}{1}'.format(d, self.e)
```

Abstraktion

Der oben beschriebene Bauplan definiert den neuen Datentyp `Rechteck` und stellt uns in der Anwendung eine nützliche *Abstraktion* zur Verfügung. Wir müssen jetzt nur noch wissen, mit welchen Parametern die Rechteck-Objekte erzeugt werden und welche Methoden welche Werte zurückgeben. Die Details der Implementierung können uns als Benutzer der Klasse im Grunde egal sein. Für uns ist nur die *Schnittstelle* zur Klasse von Bedeutung. Eine Schnittstelle ist – vereinfacht ausgedrückt – eine Art Bedienungsanleitung für die Klasse.

Die Klasse als separates Modul

Wir könnten jetzt den Code zur Erzeugung der Aufgaben und Lösungen von Rechteck-Aufgaben direkt nach der Klassendefinition anfügen.

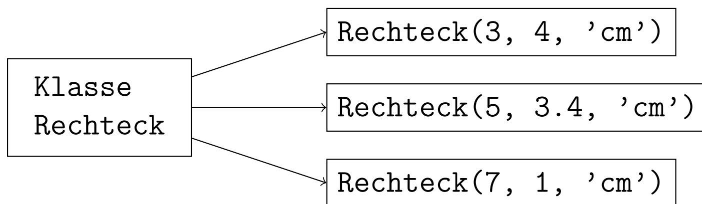
Falls wir jedoch später noch weitere Aufgaben erzeugen möchten, ist es sinnvoll, den Code für die Klasse und deren Anwendung in getrennten Dateien unterzubringen um eine Vermischung zu vermeiden.

Wir erstellen daher eine neue Datei mit dem Namen `rechteck-client-1.py`, welche die Rechteckklasse als Modul importiert und dann verwendet.

```python
# rechteck-client-2.py
from rechteck_oop import Rechteck as Re

rechtecke = [Re(3,4,'cm'), Re(5,3.4,'m'), Re(7,1,'mm')]

for i, r in enumerate(rechtecke):
    print(f'{i+1}. Berechne Umfang, Inhalt und', end=' ')
    print(f'Diagonale des Rechtecks mit {r}.\n')

for i, r in enumerate(rechtecke):
    u = r.umfang()
    A = r.inhalt()
    d = r.diagonale()
    print(f'{i+1}. Lösung: u={u}, A={A}, d={d}\n')
```

Zeile 2 importiert den Konstruktor der Klasse und weist ihm mit dem Schlüsselwort `as` das *Alias* „`Re`" zu, um in Zeile 4 weniger tippen zu müssen.

Instanzvariablen und Instanzmethoden



Eine Klasse ist eine „Fabrik" zur Erzeugung von Objekten des gleichen Typs. Diese Objekte haben zwar alle dieselbe Struktur (Eigenschaften und Methoden), befinden sich aber an unterschiedlichen Stellen im Speicher und können daher auch verschiedene Werte haben. Um diese „Individualität" der Objekte zu unterstreichen, spricht man auch von *Instanzen*. Die Variablen und Methoden dieser Instanzen werden dann *Instanzvariablen* und *Instanzmethoden* genannt.

7.4 Klasseneigenschaften und Klassenmethoden

Gelegentlich ist es nötig, dass Variablen unabhängig von den Instanzen einer Klasse immer den gleichen Wert haben. Es ist beispielsweise nicht sinnvoll, dass jede Instanz einer Klasse, die Kreisberechnungen durchführt, für sich eine Kopie der Kreiszahl π speichert. In diesem Fall genügt es, wenn die Konstante nur einmal in der Klasse gespeichert wird und alle Instanzen darauf zurückgreifen können. Bei Variablen dieses Typs spricht man von *Klassenvariablen* oder *Klasseneigenschaften*.

Ebenso möchte man Methoden definieren, die unabhängig von den Objekten einer Klasse sind. Diese nennt man entsprechend *Klassenmethoden*.

Das folgende Beispiel mag etwas konstruiert wirken, zeigt aber, dass Klasseneigenschaften und -methoden durchaus ihre Daseinsberechtigung haben.

Zufällige Rechtecke erzeugen

Wir wollen unsere Klasse um eine weitere Methode (=Funktion) erweitern, die ein Rechteck mit zufälliger Länge, zufälliger Breite und zufälliger Einheit erzeugt. Damit dasselbe Rechteck nicht mehrfach entsteht, speichern wir in der Menge mit dem Namen `alt` alle bereits erzeugten Längen und Breiten, um Wiederholungen zu vermeiden.

Der folgende Code ergänzt unsere bisherige Klasse um eine Klassenmethode, die eindeutige zufällige Rechtecke erzeugt:

```
# rechteck_rnd.py
from random import randint, choice
class Rechteck:

    alt = set()

    def zufall(von=1, bis=9):
        while True:
            a = randint(von, bis)
            b = randint(von, bis)
            e = choice(['mm', 'cm', 'dm', 'm'])
            if (a, b) not in Rechteck.alt:
                Rechteck.alt.add((a, b))
                Rechteck.alt.add((b, a))
                return Rechteck(a, b, e)
```

Dann folgt die ursprüngliche Klassendefinition von `Rechteck`.

Kommentare zum obigen Code

- *Zeile 5:* Weist der Variablen `alt` die leere Menge zu.
- *Zeile 7:* Die Klassenmethode `zufall(...)` hat zwei Parameter mit Standardwerten für den Bereich der Zufallszahlen jedoch kein `self`.
- *Zeile 8:* Eine „Endlosschleife“, die nur dann in Zeile 15 mit einer `return`-Anweisung beendet wird, wenn das zufällig erzeugte Rechteck noch nicht früher generiert wurde.
- *Zeilen 9–11:* Erzeugt zufällig zwei ganze Zahlen und eine Einheit.
- *Zeilen 12:* Prüft, ob sich das Paar (Länge, Breite) nicht in der Menge `alt` befindet. Beachte, dass einer Klassenvariable ihr Klassenname vorangestellt werden muss.
- *Zeilen 13–15:* Fügt (Länge, Breite) und (Breite, Länge) der Menge `alt` hinzu, damit sie kein zweites Mal verwendet werden und gibt ein Rechteck-Objekt mit den Zufallswerten zurück.

Der folgende Client importiert die Rechteck-Klasse mit der `zufall(...)`-Klassenmethode und erzeugt damit zehn verschiedene Rechtecke. Beachte, dass auch hier der Klassenname vor dem Methodennamen stehen muss.

```
1 # rechteck-client-2.py
2
3 from rechteck_rnd import Rechteck
4
5 for i in range(0, 10):
6     print(Rechteck.zufall())
```

7.5 Vererbung

Das Konzept der *Vererbung* ermöglicht es, Code von bestehenden Klassen in anderen Klassen wiederzuverwenden.

Dies ist insbesondere dann sinnvoll, wenn ein Objekttyp Spezialfall eines anderen Objekttyps ist (*is-a-Beziehung*).

Wir zeigen nun, wie wir unsere ursprüngliche Rechteck-Klasse um eine Quadrat-Klasse erweitern können, ohne viel neuen Code zu schreiben.

```
25 class Quadrat(Rechteck): # ein Quadrat ist ein Rechteck
26
27     def __init__(self, a, einheit):
28         super().__init__(a, a, einheit)
29
30     def __str__(self):
31         return 'a={0.a}{0.e}'.format(self)
```

Zeile 25: Wenn wir Code von einer *Elternklasse* (=*Superklasse*) an eine *Kindklasse* (=*Subklasse*) *vererben* wollen, müssen wir den Namen der Superklasse in runden Klammern hinter den Namen der Subklasse stellen.

Zeilen 27–28: Da ein Quadrat ein spezielles Rechteck ist, können wir im Konstruktor der Klasse `Quadrat` den Konstruktor der Superklasse (`Rechteck`) mit zwei gleich langen Seiten aufrufen. Damit das funktioniert, greifen wir mit der Funktion `super()` auf den Namen der Elternklasse zu und hängen mit der Punkt-Schreibweise den Namen des Konstruktors an.

Wenn jetzt beispielsweise die Methode `umfang()` auf ein Quadrat-Objekt angewendet wird, findet Python keine Funktion mit diesem Namen. Weil Quadrat-Objekte aber von Rechteck-Objekten erben, sucht Python in der Superklasse nach der Methode und wird dort fündig.

Zeilen 30–31: Da die String-Spezialmethode der Superklasse die Länge *und* die Breite ausgibt, ist das bei einem Quadrat etwas störend. Wenn wir in der Quadrat-Klasse eine eigene Methode für die Textdarstellung definieren, wird nicht mehr nach der gleichnamigen Methode in der Superklasse gesucht. Man spricht dann vom *Überschreiben* einer Methode.

7.6 Zusammenfassung

Klasse und Instanz

Mit dem Begriff *Klasse* wird sowohl der Bauplan für Objekte als auch ein abstrakter Datentyp bezeichnet. Eine Klassendefinition beginnt in Python mit `class <Klassenname>:`

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

Instanzvariable: Eine Variable, deren Wert von Instanz zu Instanz verschieden sein kann. In Python wird in der Klassendefinition einer Instanzvariablen der formale Parameter `self.` vorangestellt.

Instanzmethode: Eine Funktion, die auf eine Instanz angewendet wird. Bei der Definition einer Instanzmethode in Python muss der erste formale Parameter `self` sein. Beim späteren Aufruf der Methode wird dieser Parameter in der Parameterliste weggelassen, da er durch das Objekt selbst dargestellt wird.

Der Konstruktor

Der *Konstruktor* ist eine spezielle Instanzmethode, mit der ein Objekt initialisiert wird. Das bedeutet, dass Speicherplatz für das Objekt reserviert wird und den Instanzvariablen Werte zugewiesen werden. In Python wird der Konstruktor mit der Spezialmethode `__init__(self, ...)` definiert. Um nach der Definition Objekte des Typs zu erzeugen, ruft man den Klassennamen als Funktion mit den aktuellen Parametern auf.

Klassenvariablen und -methoden

Klassenvariable: Eine Variable, deren Wert unabhängig von einer Instanz ist. In Python wird einer Klassenvariable der Klassenname vorangestellt.

Klassenmethode: Eine Funktion, die unabhängig von einer Instanz ist. In Python fehlt bei der Definition der formale Parameter `self`. Beim Aufruf der Methode muss ihr der Klassenname mit einem Punkt vorangestellt werden.

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.

In Python wird bei der Definition einer abgeleiteten Klasse der Name der Superklasse in Klammern hinter den Namen der Subklasse gestellt. Zur Laufzeit sucht Python automatisch in der Elternklasse nach Variablen und Methoden, wenn diese in der Kindklasse nicht definiert sind.

Will man in der Definition einer Kindklasse den Konstruktor der Elternklasse aufrufen, so muss man diesem `super().` voranstellen.

Überladen von Operatoren

Ein Mechanismus, der es erlaubt, den gleichen Operator (den gleichen Funktionsnamen) in unterschiedlichen Klassen zu verwenden. Die unten definierte Klasse `Kreis` verwendet beispielsweise dieselben Methodennamen wie die der Klasse `Rechteck`. Dies ist aber kein Problem, da jedes Objekt „weiss“, welche Methoden zu ihm gehören.

```
class Kreis:

    def PI = 3.141592

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

    def __str__(self):
        return 'r={0.r}{0.e}'.format(self)

    def umfang(self):
        u = 2*Kreis.PI*self.r
        return '{0}{1}'.format(u, self.e)

    def inhalt(self):
        A = Kreis.PI* self.r**2
        return '{0}{1}\u00B2'.format(A, self.e)
```