Teil I

Kurze Einführung in Python

3 Kurze Einführung in Python

3.1 Datenstrukturen

3.1.1 Zahlen und Variablen

Wenn man die interaktive Shell von Python oder ipython startet, kann man sofort arithmetische Ausdrücke eingeben:

```
>>> 5 * 8.6 - 4 ** 2
27.0
```

Möchte man Werte speichern, so kann man dies in der Shell ebenso wie in einem Programm mittels Variablenzuweisungen bewerkstelligen. Dies geschieht ebenso wie in den meisten anderen Programmiersprachen durch ein Gleichheitszeichen:

```
>>> x = 42
>>> y = 10
>>> print(x - y)
32
```

Wie wir im vorigen Beispiel gesehen haben, ist eine Typdeklaration dazu nicht nur nicht erforderlich, sondern auch nicht möglich. Python kennt keine Typdeklarationen. Python-Variablen stellen Referenzen auf beliebige Objekte dar.

Auch wenn den Variablennamen keine Typen zugeordnet sind, so entspricht jedes in Python definierte Objekt einem Typ oder genauer gesagt der Instanz einer Klasse.

```
>>> x = 42
>>> type(x)
<class 'int'>
```

Wir sehen also, dass eine Integer-Zahl „42" angelegt worden ist. Diese Integer-Zahl ist ein Objekt der Integer-Klasse. Als Integers oder ganze Zahlen bezeichnet man in der Mathematik die Zahlen

```
..., -3, -2, -1, 0, 1, 2, 3, ...
```

Das heißt, die Zahlen gehen von minus unendlich bis unendlich. Selbstverständlich können wir „unendlich" in „int" nicht darstellen, aber die Zahlen in Python können extrem groß bzw. extrem klein werden:

```
>>> x = 2 ** 64
>>> x
18446744073709551616
>>> len(str(x))
20
>>> x = 2 ** (2 ** 22)
>>> len(str(x))
1262612
```

Benutzt man einen Variablennamen, der nicht definiert ist, erzeugt man eine Ausnahme:

```
>>> counter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'counter' is not defined
```

Im folgenden Beispiel zeigen wir, dass Variablen Referenzen auf Objekte darstellen. Dazu benutzen wir die Funktion „id", deren Rückgabewert ein Integer-Wert ist, der die Objekte eindeutig identifiziert. Wir sehen, dass das Integer-Objekt 10 nur einmal erzeugt wird und dass x und y beide dieses Objekt nach der Zuweisung „`y = x`" referenzieren.

```
>>> x = 10
>>> id(x)
10106112
>>> y = x
>>> id(y)
10106112
>>> y = 12
>>> id(y)
10106176
>>> id(x)
10106112
```

Neben `int` kennt Python auch Float-Zahlen (`float`), die in Ausdrücken auch mit Integer-Zahlen verknüpft werden können:

```
>>> f = 3.5
>>> type(f)
<class 'float'>
>>> f2 = 878.323
>>> x = f * 10
>>> x
35.0
```

3.1.2 Zeichenketten/Strings

Ein weiterer wichtiger Datentyp in Python sind die Zeichenketten, die man meist auch als Strings bezeichnet. Strings können auf verschiedene Arten definiert werden: mit einfachen Anführungszeichen, mit doppelten Anführungszeichen oder mit drei einfachen bzw. drei doppelten Anführungszeichen. Wir demonstrieren diese Varianten im folgenden Beispiel. Wir verwenden auch das Nummernzeichen „#", um Kommentare einzuleiten:

```
>>> 'ein String in einfachen Anführungszeichen'
'ein String in einfachen Anführungszeichen'
>>> 'Miller\s son'  # der Rückwärtsschrägstrich fungiert wie üblich als Escape-
    Zeichen
'Miller\\s son'
```

```
>>> "Miller's son"  # jetzt brauchen wir kein Escape-Zeichen
"Miller's son"
>>> print("""Strings mit drei Anführungszeichen können
... über mehrere
... Zeilen gehen und enden erst, wenn
... drei Anführungszeichen kommen""")
Strings mit drei Anführungszeichen können
über mehrere
Zeilen gehen und enden erst, wenn
drei Anführungszeichen kommen
>>>
```

Bei den Strings in dreifachen Anführungszeichen wartet die Shell noch mit einer Besonderheit auf, wenn wir während der Definition eines solchen Strings die Return-Taste tippen. Die nächste Zeile wird dann nicht mit dem üblichen Prompt „`>>> `" eingeleitet, sondern mit einem aus drei Punkten bestehenden Prompt „`... `". Damit signalisiert uns die Shell, dass der String noch nicht fertig ist. Der String ist erst fertig, wenn wieder entsprechend drei einfache (`'''`) bzw. doppelte (`"""`) Anführungszeichen eingegeben werden.

Strings können indiziert werden, dabei entspricht das erste Zeichen dem Index 0. Das letzte Zeichen eines Strings können wir mit dem Index -1 ansprechen, d.h. mit negativen Zahlen erhält man eine Indizierung von rechts:

```
>>> language = "Python"
>>> language[0]     # 1. Zeichen des Strings an der Stelle 0
'P'
>>> language[2]     # 3. Zeichen des Strings an der Stelle 2
't'
>>> language[-1]     # letztes Zeichen
'n'
>>> language[-2]     # vorletztes Zeichen
'o'
```

Während man mit dem Indizieren nur einzelne Zeichen, also Strings der Länge 1, aus einem String erhalten kann, ermöglicht der Teilbereichsoperator (Slicing) das „Herausschneiden" von beliebigen Teilstrings:

```
>>> s = "Ein grüner Fisch singt nie schräg!"
>>> s[4:10]   # von Position 4 (inklusive) bis Position 10 (exklusive)
'grüner'
>>> s[-7:]   # von Position -7 bis zum Ende des Strings
'schräg!'
>>> s[:16]   # von Anfang bis zur Position 16
'Ein grüner Fisch'
```

Finden von Teilstrings in Strings:

```
>>> s = "A horse, a horse! "
>>> s += "My kingdom for a horse!"
>>> s
'A horse, a horse! My kingdom for a horse!'
>>> s.find("horse")
2
>>> s.find("horse", 3)
11
>>> s.find("horse", 16)
35
```

```
>>> s.find("horse", 36)
-1
>>> s.rfind("horse")   # Suche beginnt von hinten
35
```

Die Methode `index` liefert die gleichen Ergebnisse, außer wenn der Suchstring nicht im String vorkommt:

```
>>> s.index("horse")
2
>>> s.index("horse", 3)
11
>>> s.index("cow")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Ein String kann mit Methode `split` auch in eine Liste von Teilstrings aufgespalten werden. Ohne Parameter dient jede Folge von Leerzeichen (auch `\n`, `\t`, `\r` usw. sind Leerzeichen) als Trenner. Wird ein String als Parameter übergeben, wird dieser als Trenner verwendet. Außerdem kann man noch einen weiteren Parameter übergeben, mit dem man die Anzahl der zu spaltenden Teilstrings angeben kann:

```
>>> s = "The    is \n a\tstring."
>>> s.split()
['The', 'is', 'a', 'string.']
>>> t = "34,12.3,90,0,,1"
>>> t.split(",")
['34', '12.3', '90', '0', '', '1']
>>> t.split(",", 2)
['34', '12.3', '90,0,,1']
>>> t.split(",", maxsplit=2)
['34', '12.3', '90,0,,1']
>>> t.split(sep=",", maxsplit=2)
['34', '12.3', '90,0,,1']
>>> s.split(maxsplit=2)
['The', 'is', 'a\tstring.']
>>> # aufspalten von hinten:
...
>>> t.rsplit(sep=",", maxsplit=2)
['34,12.3,90,0', '', '1']
>>> s.rsplit(maxsplit=2)
['The    is', 'a', 'string.']
```

Mit den Methoden `lstrip`, `rstrip` und `strip` ist es möglich, Leerzeichen oder auch andere Zeichen vom linken bzw. rechten Rand eines Strings oder beidseitig zu entfernen:

```
>>> s = "\n2000 Hamburg\r\n"
>>> s.strip()
'2000 Hamburg'
>>> s.strip("\n\r0123456789 ")
'Hamburg'
>>> s.rstrip()
'\n2000 Hamburg'
>>> s.lstrip()
'2000 Hamburg\r\n'
```

3.1.3 Listen

Listen werden mittels eckiger Klammern in Python erzeugt. Eine Liste kann beliebige Python-Objekte enthalten, die durch Komma getrennt sind:

```
>>> lst = ["rot", "grün", "blau"]
>>> lst2 = ["rot", 12, [3, 6.78]]
```

Auf Listenelemente kann man wie bei Strings über Indices zugreifen oder man greift auf mehrere Listenelemente mit dem Teilbereichsoperator zu. Außerdem können wir mittels Indizierung der Liste auch neue Werte zuweisen:

```
>>> lst = ["rot", 12, "gelb", 123, [3, "Noch ein String"]]
>>> lst[0]
'rot'
>>> lst[4]
[3, 'Noch ein String']
>>> lst[4][1]
'Noch ein String'
>>> lst[1:4]
[12, 'gelb', 123]
>>> lst[0] = "orange"
>>> lst
['orange', 12, 'gelb', 123, [3, 'Noch ein String']]
```

Wir können prüfen, ob ein Element in einer Liste vorhanden ist:

```
>>> farben = ["rot", "grün", "blau", "gelb"]
>>> "rot" in farben
True
>>> "braun" in farben
False
>>> "grau" not in farben
True
```

Anhängen von Objekten an Listen:

```
>>> farben = ["rot", "grün", "blau", "gelb"]
>>> farben.append("silber")
>>> farben
['rot', 'grün', 'blau', 'gelb', 'silber']
```

Kopieren und Konkatenieren von Listen:

```
>>> farben1 = ["rot", "grün"]
>>> farben2 = ["blau", "gelb"]
>>> farben = farben1 + farben2
>>> farben
['rot', 'grün', 'blau', 'gelb']
>>>
>>> farben1 = ["rot", "grün"]
>>> farben2 = ["blau", "gelb"]
>>> farben = farben1.copy()
>>> farben.extend(farben2)
>>> farben
['rot', 'grün', 'blau', 'gelb']
```

Häufig muss man auch Elemente mit einem bestimmten Index aus Listen entfernen, dazu nutzt man meist die Methode `pop`:

```
>>> farben = ["rot", "grün", "blau", "gelb"]
>>> letztes_element = farben.pop()  # das letzte Element wird entfernt
>>> letztes_element
'gelb'
>>> farben
['rot', 'grün', 'blau']
>>>
>>> erstes_element = farben.pop(0)  # erstes Element wird entfernt
>>> erstes_element
'rot'
>>> farben
['grün', 'blau']
```

Möchte man ein bestimmtes Element entfernen, kann man dazu die Methode `remove` nutzen. Falls ein Objekt entfernt werden soll, welches nicht in der Liste enthalten ist, erhalten wir eine Fehlermeldung:

```
>>> farben = ["rot", "grün", "rot", "blau"]
>>> farben.remove("rot") # erstes Vorkommen von "rot" wird entfernt
>>> farben
['grün', 'rot', 'blau']
>>> farben.remove("rot")
>>> farben
['grün', 'blau']
>>> farben.remove("rot")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

3.1.4 Tupel

Tupel werden mit runden Klammern definiert. Allerdings können die Klammern auch weggelassen werden, wie wir in den folgenden Beispielen sehen:

```
>>> tage = ("Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag",
      "Sonntag")
>>> tage
('Montag', 'Dienstag', 'Mittwoch', 'Donnerstag', 'Freitag', 'Samstag', 'Sonntag')
>>> monate = "Frühling", "Sommer", "Herbst", "Winter"   # auch ein Tupel
>>> monate
('Frühling', 'Sommer', 'Herbst', 'Winter')
>>>
>>> wochentage = tage[:5]
>>> wochentage
('Montag', 'Dienstag', 'Mittwoch', 'Donnerstag', 'Freitag')
```

Tupel werden auch für Mehrfachzuweisungen verwendet:

```
>>> x, y = 11, 33
>>> x, y
(11, 33)
>>> x, y = y, x   # Werte werden vertauscht
>>> x, y
(33, 11)
```

Dabei werden zuerst die Ausdrücke auf der rechten Seite ausgewertet und dann den Variablen auf der linken Seite zugewiesen.

3.1.5 Frozensets und Mengen in Python

Ein set-Objekt enthält eine ungeordnete Sammlung von einmaligen und unveränderlichen Elementen, d.h. es gibt keine Mehrfachvorkommen von Elementen, d.h. Elemente können in set-Objekten nicht mehrfach vorkommen. Beim Datentyp „set" handelt es sich um die Python-Implementierung von Mengen, wie wir sie aus der mathematischen Mengenlehre kennen.

Will man eine Menge erzeugen, so ist dies in Python sehr einfach. Wir bedienen uns der geschweiften Klammern, wie dies in der Mathematik üblich ist.

```
>>> staedte = {'Hamburg', 'München', 'Frankfurt', 'Berlin'}
>>> print(staedte)
{'München', 'Berlin', 'Frankfurt', 'Hamburg'}
>>> 'Berlin' in staedte
True
>>> 'Köln' in staedte
False
```

Bei der Ausgabe der Variablen `staedte` erkennen wir, dass die Elemente nicht in der Reihenfolge ausgegeben werden, in der wir Städte eingegeben hatten. Sets haben keine definierte Reihenfolge. Es gibt auch keinerlei Methoden, mit denen man Elemente von Mengen in Abhängigkeit einer Position bearbeiten könnte.

Sets lassen sich ideal einsetzen, wenn wir mehrfache Elemente aus Listen und Tupeln entfernen wollen. Allerdings macht dies nur Sinn, wenn uns die ursprüngliche Reihenfolge nicht mehr interessiert:

```
>>> t = ("a", "b", "b", "a", "c", "d", "a")
>>> s = set(t)
>>> s
{'d', 'a', 'c', 'b'}
>>> t = tuple(s)
>>> t
('d', 'a', 'c', 'b')
>>>
>>> s = "Dies ist ein String mit vielen Buchstaben"
>>> buchstaben = set(s.lower())
>>> buchstaben
{'m', 'e', 'h', 'u', 'c', 'b', 'v', 'r', 'i', 'n', 'd', ' ', 'a', 'l', 'g', 't',
     's'}
```

Für die Elemente einer Menge gelten die gleichen Einschränkungen wie für die Schlüssel eines Dictionaries: Sets können nur unveränderliche Elemente enthalten, also Objekte vom Typ „int", „float", „complex", „str", „byte", „frozen_set" oder „tuple".

Die Klasse „set" bietet eine Vielzahl von Operationen und die aus der Mathematik üblichen Operationen auf Mengen. Wir demonstrieren die wichtigsten in den folgenden Beispielen:

```
>>> s = {"a", "b", "c"}
>>> s2 = {"b", "c", "e"}
>>> s3 = s - s2         # Differenzmenge von s und s2
>>> s3
{'a'}
>>> s | s2              # Vereinigungsmenge
{'e', 'c', 'b', 'a'}
>>> s & s2              # Schnittmenge
{'c', 'b'}
```

```
>>> s2.remove("e")      # Entfernen eines Elementes
>>> s2
{'c', 'b'}
>>> s2.add("x")         # Hinzufügen eines Elementes
>>> s2
{'x', 'c', 'b'}
>>> s2 < s              # Prüfung, ob s2 echte Teilmenge von s ist
False
```

Die Klasse „frozenset" dient zur Erzeugung von unveränderlichen Mengen. Frozensets lassen sich deshalb auch als Schlüssel in Dictionaries und in Mengen verwenden. Sie werden erzeugt, indem man frozenset auf iterierbare Objekte wie z.B. Listen, Tupel oder Sets anwendet:

```
>>> x = frozenset({3, 5, 12})
>>> y = frozenset([33, 15, 112])
>>> m = {x, y}
>>> m
{frozenset({112, 33, 15}), frozenset({3, 12, 5})}
>>> d = {x:300, y:700}
>>> d
{frozenset({112, 33, 15}): 700, frozenset({3, 12, 5}): 300}
```

3.1.6 Dictionaries

Dictionaries gehören wohl zu den Datenstrukturen, die Python besonders attraktiv machen. Während bei Listen der Zugriff auf ein Element über die Position, also den Index erfolgt, geschieht dies bei Dictionaries über Schlüssel (engl. „keys"). Jedem Schlüssel ist ein Wert zugeordnet. Man kann sich ein Dictionary also als eine Menge von Schlüssel-Wert-Paaren vorstellen.

Die Definition und Arbeitsweise mit Dictionaries erläutern wir mit den folgenden Beispielen:

```
>>> empty_dict = {}                # leeres Dictionary
>>> len(empty_dict)                # Anzahl der Elemente
0
>>> haeufigkeit = {"a":4, "b":12, "c":20}
>>> haeufigkeit
{'a': 4, 'c': 20, 'b': 12}
>>> haeufigkeit["a"]               # Zugriff auf den Schlüssel "a"
4
>>> haeufigkeit.get("a")           # Zugriff auf den Schlüssel "a"
4
>>> list(haeufigkeit.items())      # Wandlung in Schlüssel-Wert-Liste
[('a', 4), ('c', 20), ('b', 12)]
>>> list(haeufigkeit.keys())       # Liste mit den Schlüssel
['a', 'c', 'b']
>>> list(haeufigkeit.values())     # Erzeugen einer Liste mit den Werten
[4, 20, 12]
>>> for key in haeufigkeit.keys():  # Iteration über die Schlüssel
...     print(y)
...
a
c
b
```

Die Schlüssel eines Dictionaries können beliebige unveränderliche Objekte sein, also Integers, Floats, Strings und Tupel. Als Werte können beliebige Objekte genutzt werden:

```
>>> staedte_GPS = { (52.520007, 13.404954):"Berlin", (47.767097, 8.872239):"
     Singen am Hohentwiel" }
>>> ports = {21:"File Transfer Protocol (FTP)", 22:"Secure Shell (SSH)", 23:"
     Telnet remote login service"}
>>> adressen = {"Henry":[ ("Henry", "Peterson"), 20016, "Hamburg"]}
```

3.2 Kontrollstrukturen

3.2.1 Bedingte Anweisungen

Bedingte Anweisungen dienen dazu, den Programmfluss unter bestimmten Bedingungen verzweigen zu lassen:

```
x = int(input("Bitte eine ganze Zahl eingeben: "))

if x < -10:
    print("Die eingegebene Zahl ist kleiner als -10")
elif x < 0:
    print("Wert kleiner als 0 aber nicht kleiner als -10")
elif x == 0:
    print("0? Warum nicht :-)")
elif x < 10:
    print("Naja, so richtig groß ist die Zahl nicht!")
else:
    print("Die Zahl könnte ganz schön groß sein!")
    print("Aber ich habe es nicht getestet!")
```

Das Schlüsselwort „elif" ist eine Kurzform für „else if" und verhindert eine „ausufernde" Verschachtelungstiefe, wie wir leicht sehen können, wenn wir das obige Programm ohne „elif" umschreiben:

```
x = int(input("Bitte eine ganze Zahl eingeben: "))

if x < -10:
    print("Die eingegebene Zahl ist kleiner als -10")
else:
    if x < 0:
        print("Wert kleiner als 0 aber nicht kleiner als -10")
    else:
        if x == 0:
            print("0? Warum nicht :-)")
        else:
            if x < 10:
                print("Naja, so richtig groß ist die Zahl nicht!")
            else:
                print("Die Zahl könnte ganz schön groß sein!")
                print("Aber ich habe es nicht getestet!")
```

In Python-Programmen findet man häufig auch das ternäre if:

```
>>> temperatur = 25
>>> wertung = "warm" if temperatur > 20 else "frisch"
>>> wertung
'warm'
```

In diesem Beispiel ist das ternäre if eine abgekürzte Schreibweise für folgenden Code:

```
>>> if temperatur > 20:
...     wertung = "warm"
... else:
...     wertung = "frisch"
```


3.2.2 Schleifen

Schleifen werden benötigt, um einen Codeblock, also eine oder mehrere Python-Anweisungen, wiederholt auszuführen. Einen solchen Codeblock bezeichnet man auch als Schleifenkörper oder Body. Python kennt zwei Schleifentypen: die while- und die for-Schleife.

3.2.2.1 while-Schleife

Die Syntax der while-Schleife sieht wie folgt aus:

```
while <Ausdruck>:
    <Folge1>
else:
    <Folge2>
```

<Folge1> steht für eine beliebige Anweisungsfolge in gleicher Einrückungstiefe. Diese Anweisungsfolge wird solange ausgeführt, wie der Ausdruck „<Ausdruck>" den Wert `True` liefert. Zuerst wird „<Ausdruck>" ausgewertet, und im `True`-Fall wird „<Folge1>" ausgeführt. Falls „<Ausdruck>" den Wert `False` liefert, wird „<Folge2>" ausgeführt, falls vorhanden. Der `else`-Teil ist optional. Eine `while`-Schleife kann auch durch ein `break`-Statement innerhalb von „<Folge1>" abgebrochen werden. In diesem Fall wird jedoch „<Folge2>" übersprungen, falls der `else`-Teil überhaupt vorhanden ist.

Das folgende Skript benutzt eine while-Schleife, um die Zahlen von 1 bis 4, gefolgt von ihrem jeweiligen Quadrat, auszugeben:

```
>>> i = 1
>>> while i <= 4:
...     print(i, i**2)
...     i += 1
...
1 1
2 4
3 9
4 16
```

Das nächste Beispiel benutzt ein `break`:

```
>>> numbers = [4, 5, 12, 9, -1, 8, 9]
>>> sum_numbers = 0
>>> while numbers != []:
...     number = numbers.pop()
...     if number >= 0:
...         sum_numbers += number
...     else:
...         break
...
>>> print(sum_numbers, numbers)
17 [4, 5, 12, 9]
```

3.2.2.2 for-Schleife

Die for-Schleife dient in Python dazu, über beliebige iterierbare Objekte zu iterieren. Iterierbare Objekte sind beispielsweise Listen, Tupel, Strings und Dictionaries.

```
einkaufsliste = ["Butter", "Milch", "Brot", "Salat", "Spam"]

for artikel in einkaufsliste:
    if artikel == "Spam":
        print("Spam mag ich nicht!")
    else:
        print("Ich werde " + artikel + " kaufen!")
```

Die Ausgabe lautet dann:

```
Ich werde Butter kaufen!
Ich werde Milch kaufen!
Ich werde Brot kaufen!
Ich werde Salat kaufen!
Spam mag ich nicht!
```

Weitere selbsterklärende Beispiele von for-Schleifen:

```
>>> wort = "Python"
>>> for buchstabe in wort:
...     print(buchstabe)
...
P
y
t
h
o
n
>>>
>>> liste = [(4, 5, 9.1), "Python", [4, ["abc", "xyz"]]]
>>> for element in liste:
...     print(element)
...
(4, 5, 9.1)
Python
[4, ['abc', 'xyz']]
>>>
```

Ein wichtiges iterierbares Objekt für die for-Anweisung stellt die range-Klasse[1] zur Verfügung. Mit den von range erzeugten Objekten lassen sich for-Schleifen im Stil von C und Java simulieren. Das Verhalten von `range` erklären wir durch die folgenden Beispiele:

```
>>> range(4)
range(0, 4)
>>> for i in range(4):
...     print(i, end=", ")
...
0, 1, 2, 3, >>>
>>>
>>> lst = [34, 55, 2, 10]
>>> for i in range(len(lst)):
...     print(i, lst[i])
...
0 34
1 55
2 2
3 10
>>>
>>> von, bis, schrittweite = 3, 21, 4
>>> lst = list(range(von, bis, schrittweite))
>>> print(lst)
[3, 7, 11, 15, 19]
```

Im folgenden Beispiel finden wir noch die break- und die continue-Anweisung im Einsatz. Mit „break" erzeugen wir einen vorzeitigen Abbruch der Schleife, während mit „continue" nur der aktuelle Durchlauf beendet wird, um dann mit dem nächsten Objekt fortzufahren. In diesem Beispiel finden wir auch noch eine Besonderheit der for-Schleife in Python: Die for-Schleife kann auch ein else-Statement enthalten. Nur wenn die Schleife nicht mit einem „break" verlassen worden ist, werden die Anweisungen unter dem else-Teil bearbeitet.

```
string = input("Bitte einen String eingeben: ")

for buchstabe in string:
    zaehler = 0
    if buchstabe.isalpha():
        print(buchstabe)
    else:
        if buchstabe == ".":
            print("Punkt wurde gelesen!")
            print("Die restlichen Zeichen werden nicht mehr bearbeitet!")
            break
        else:
            print("Sonderzeichen wird ignoriert!")
            continue
    zaehler += 1
else:
    print("Der String enthielt keinen Punkt")
```

Starten wir das Programm, erhalten wir folgende Ausgabe:

```
bernd@moon:~$ python3 for_with_break_continue_else.py
Bitte einen String eingeben: Hi, you!
```

[1] range ist keine Funktion, sondern eine Klasse.

```
H
i
Sonderzeichen wird ignoriert!
Sonderzeichen wird ignoriert!
y
o
u
Sonderzeichen wird ignoriert!
Der String enthielt keinen Punkt
bernd@moon:~$ python3 for_with_break_continue_else.py
Bitte einen String eingeben: Hi, you.
H
i
Sonderzeichen wird ignoriert!
Sonderzeichen wird ignoriert!
y
o
u
Punkt wurde gelesen!
Die restlichen Zeichen werden nicht mehr bearbeitet!
```


3.2.3 Funktionen

3.2.3.1 Einfache Funktionen

Funktionen werden mit dem Schlüsselwort „def" eingeleitet. Danach folgt der Name der Funktion, eine Folge von formalen Parametern in runden Klammern und am Ende der Zeile ein Doppelpunkt. Der Funktionskörper besteht aus beliebigen eingerückten Anweisungen. Unmittelbar nach der eben beschriebenen Kopfzeile kann noch optional ein String-Literal stehen. Dies ist ein Dokumentationsstring, docstring, der in der help-Funktionalität verwendet wird.

```
>>> def f():
...     """
...     Diese Funktion hat keine Parameter und liefert None zurück,
...     wenn sie aufgerufen wird!
...     """
...     pass
...
>>> print(f())
None
```

„pass" ist eine leere Anweisung, die als Platzhalter fungiert. Diese Anweisung wird verwendet, wenn in einem Kontext eine Anweisung erforderlich ist, aber keine Aktion ausgeführt werden soll. In Funktionen muss mindestens eine Codezeile nach der Kopfzeile stehen, weswegen wir ein pass verwendet haben.

Ein wesentliches Merkmal von Funktionen ist die Rückgabe von Werten. Unsere Beispielfunktion hat automatisch das None-Objekt zurückgegeben. Man kann auch explizit mit der `return`-Anweisung Rückgaben definieren. Trifft der Programmablauf innerhalb einer Funktion auf eine `return`-Anweisung, wird die Funktion verlassen und das Objekt zurückgeliefert, das in dem Ausdruck der `return`-Anweisung erzeugt wird.

```
>>> def poly(x):
...     return 3 * x**2 + 0.9 * x - 9
...
>>> poly(1)
-5.1
>>> poly(2)
4.800000000000001
```

Eine Funktion kann zwar nur ein Objekt zurückliefern, aber man kann natürlich auch mehrere Objekte, die man zurückgeben möchte, in einem Tupel vereinen. Die folgende Funktion liefert den Umfang und die Fläche eines Rechtecks zurück, dessen Länge und Breite als Parameter an die Funktion übergeben werden:

```
>>> def umfang_flaeche(laenge, breite):
...     return (2*(laenge+breite), laenge*breite)
...
>>> umfang_flaeche(3, 4)
(14, 12)
>>> umfang, flaeche = umfang_flaeche(3, 4)
>>> umfang, flaeche
(14, 12)
```

Funktionsnamen sind – wie andere Variablen auch – Referenzen auf Objekte, in diesem Fall Funktionsobjekte. Wir können also unserer Funktion `umfang_flaeche` einen kürzeren zusätzlichen Namen geben:

```
>>> uf = umfang_flaeche
>>> uf(1, 2)
(6, 2)
```

3.2.3.2 Default-Parameter und Schlüsselwortparameter

Man hat auch die Möglichkeit, die Parameter einer Funktion mit Standardwerten – meist auch als Default-Werte bezeichnet – zu versehen. Diese Parameter sind dann optional beim Aufruf der Funktion, d.h. man kann, aber muss für einen solchen Parameter kein Argument zur Verfügung stellen. Wird kein Wert angegeben, wird der Standardwert für diesen Parameter eingesetzt:

```
>>> def umfang(laenge=2, breite=1):
...     return 2 * (laenge + breite)
...
>>>
>>> umfang(5, 3)
16
>>> umfang(5)       # für breite wird der Standardwert ,,1'' benutzt
12
>>> umfang()        # es werden nun beide Standardwerte benutzt
6
```

Wie sieht es aber aus, wenn wir nur einen Wert für die Breite `breite`, aber nicht für die Länge `laenge` übergeben wollen? Übergibt man nur ein Argument, bedeutet das automatisch, dass dies einen Wert für den ersten Parameter darstellt, also in unserem Fall für `laenge`. Die Schlüsselwortparameter stellen eine Lösung für dieses Problem dar:

```
>>> umfang(breite=1.5)
7.0
```

3.2.3.3 Lokale Funktionen

Man kann innerhalb einer Funktion eine oder mehrere andere Funktionen definieren, die dann lokal in dieser Funktion sind. Sie können also nur innerhalb der Funktion benutzt werden:

```
def f(x):
    def g(x):
        return 2.3 * x

    return g(x) - 2

print(f(1))
```

Das obige Programm gibt `0.2999999999999998` als Ergebnis zurück. Die obige Funktion ist allerdings nicht sehr sinnvoll. Wir hatten festgestellt, dass Funktionen beliebige Objekte zurückliefern können. Dies bedeutet, dass Funktionen auch Referenzen auf Funktionen zurückliefern können. Im Folgenden definieren wir eine Funktion, die eine Referenz auf ein Polynom zweiten Grades zurückliefert:

```
def polynomial_creator(a, b, c):
    def polynomial(x):
        return a * x**2 + b * x + c
    return polynomial


p1 = polynomial_creator(2, 3, -1)
p2 = polynomial_creator(-1, 2, 1)

for x in range(-2, 2, 1):
    print(x, p1(x), p2(x))
```

Im obigen Programm erzeugen wir mithilfe des `polynomial_creator` zwei Polynome p1 und p2. Das Programm liefert die folgende Ausgabe zurück:

```
-2 1 -7
-1 -2 -2
0 -1 1
1 4 2
```

3.2.3.4 Globale und lokale Variablen in Funktionen

Globale Variablen können, auch wenn man es normalerweise nicht tun sollte, innerhalb des Funktionsrumpfs einer Funktion „lesend" benutzt werden. Man spricht dann von freien Variablen:

```
def f():
    print(s)
s = "Ich bin ein String!"
f()
```

Der Funktionsrumpf von f() besteht nur aus der `print(s)`-Anweisung. Weil es keine lokale Variable s gibt, d.h. keine Zuweisung an s innerhalb des Funktionsrumpfs von f, wird der Wert der globalen Variablen s benutzt. Dieser Wert kann allerdings nicht verändert werden, wie wir weiter unten in diesem Kapitel sehen werden. Es wird also der String „Ich bin ein String!" ausgegeben.

Wird innerhalb einer Funktion eine Variable definiert, so ist diese lokal, auch wenn es im aufrufenden Kontext eine Variable mit gleichem Namen gibt:

```
def f():
    s = "Ich bin ein lokaler String!"
    print(s)

s = "Ich bin ein String!"
f()
print(s)
```

Starten wir das Skript, erhalten wir folgende Ausgabe:

```
$ python3 global_lokal2.py
Ich bin ein lokaler String!
Ich bin ein String!
```

Man kann auch den Wert von globalen Variablen innerhalb einer Funktion verändern. Dazu muss man sie jedoch explizit mittels des Schlüsselworts `global` als global deklarieren:

```
def f():
    global s
    print(s)
    s = "Ich bin das globale s!"
    print(s)


s = "f wird mich ändern!"
f()
print(s)
```

Als Ausgabe erhalten wir:

```
$ python3 global_lokal4.py
f wird mich ändern!
Ich bin das globale s!
Ich bin das globale s!
```

3.3 Ausnahmebehandlung

Unter einer Ausnahmebehandlung[2] versteht man ein Verfahren, die Zustände, die während einer Fehlersituation herrschen, an andere Programmebenen weiterzuleiten. Dadurch ist es möglich, per Programm einen Fehlerzustand gegebenenfalls zu „reparieren", um anschließend das Programm weiter auszuführen. Ansonsten würden solche Fehlerzustände in der Regel zu einem Abbruch des Programms führen.

Die Ausnahmebehandlung in Python ist sehr ähnlich zu derjenigen in Java. Der Code, der das Risiko für eine Ausnahme beherbergt, wird in einen try-Block eingebettet. Aber während in Java Ausnahmen durch catch-Konstrukte abgefangen werden, geschieht dies in Python durch das except-Schlüsselwort. Semantisch funktioniert es aber genauso.

[2] engl. „exception handling"

Ausnahmen entstehen beispielsweise, wenn man versucht, durch 0 zu dividieren, eine nicht definierte Variable anzusprechen oder eine nicht unterstützte Operation auszuführen versucht, wie wir im Folgenden demonstrieren:

```
>>> x = 10 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> z = x + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x = 10 + "42"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Diese Fehler kann man in einem Programm wie folgt abfangen:

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Es ist nicht möglich durch 0 zu dividieren!")

try:
    z = x + 1
except NameError:
    print("Die Variable 'x' ist nicht definiert!")

try:
    x = 10 + "42"
except TypeError:
    print("Integer und Strings können nicht addiert werden!")
```

Startet man obiges Programm, erhält man folgende Ausgaben:

```
Es ist nicht möglich durch 0 zu dividieren!
Die Variable 'x' ist nicht definiert!
Integer und Strings können nicht addiert werden!
```

Eine sinnvolle Anwendung stellt das folgende Programmstück dar, in dem wir eine robuste Eingabeaufforderung programmieren. Wir stellen sicher, dass das Programm erst weitermachen kann, wenn eine gültige Integer-Zahl eingegeben wurde:

```
while True:
    try:
        zahl = input("Zahl eingeben: ")
        zahl = int(zahl)
        break
    except ValueError as e:
        print("error message: ", e)
        print("Keine Integer-Zahl!")
```

Statt für jede Ausnahmesituation einen eigenen Exception-Zweig zu programmieren, kann man auch mit einer Exception mehrere Ausnahmen abfangen:

```
while True:
    prompt = """Choose your error:
               1 ZeroDivisionError
```

```
                2 TypeError
                3 NameError
                4 exit
    """
    try:
        error_type = input(prompt)
        if error_type == "1":
            x = 10 / 0
        elif error_type == "2":
            x = 10 + "34"
        elif error_type == "3":
            print(x)
        elif error_type == "4":
            exit()
    except Exception as e:
        print("String-Darstellung der Ausnahme: ", str(e))
        print(type(e))
        print(e.args[0])
```

In folgender beispielhafter Nutzung zeigen wir die Ergebnisse für die verschiedenen Fälle:

```
bernd@marvin ~/tmp $ python3 choose_your_error.py
Choose your error:
                1 ZeroDivisionError
                2 TypeError
                3 NameError
                4 exit
    1
String-Darstellung der Ausnahme:  division by zero
<class 'ZeroDivisionError'>
ZeroDivisionError
division by zero
Choose your error:
                1 ZeroDivisionError
                2 TypeError
                3 NameError
                4 exit
    2
String-Darstellung der Ausnahme:  unsupported operand type(s) for +: 'int' and '
     str'
<class 'TypeError'>
TypeError
unsupported operand type(s) for +: 'int' and 'str'
Choose your error:
                1 ZeroDivisionError
                2 TypeError
                3 NameError
                4 exit
    3
String-Darstellung der Ausnahme:  name 'x' is not defined
<class 'NameError'>
NameError
name 'x' is not defined
Choose your error:
                1 ZeroDivisionError
                2 TypeError
                3 NameError
                4 exit
    4
bernd@marvin ~/tmp $
```

3.3.1 Die optionale else-Klausel

Auch das try ... except-Sprachkonstrukt verfügt über eine optionale else-Klausel, die – falls vorhanden – hinter allen except-Klauseln stehen muss. Dort befindet sich Code, der ausgeführt wird, wenn die try-Klausel keine Ausnahme auslöst.

Im Folgenden verlangen wir solange die Eingabe eines Dateinamens, bis sich dieser zum Lesen öffnen lässt. Der else-Teil der try-Anweisung wird nur ausgeführt, wenn es keinen Ausnahmefehler gegeben hat. Deshalb dürfen wir dann auch auf das Datei-Handle f zugreifen:

```
while True:
    filename = input("Dateiname: ")
    try:
        f = open(filename, 'r')
    except IOError:
        print(filename, " lässt sich nicht öffnen")
    else:
        print(filename, ' hat ', len(f.readlines()), ' Zeilen ')
        f.close()
        break
```

3.3.2 Exceptions generieren

Man kann auch eigene Exceptions generieren:

```
>>> raise SyntaxError("Sorry, mein Fehler!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: Sorry, mein Fehler!
```

Außerdem lassen sich auch eigene Exception-Klassen definieren. Da wir Klassen noch nicht definiert haben, könnte das folgende Beispiel für einige noch nicht verständlich sein:

```
class MyException(Exception):
    pass

raise MyException("Was falsch ist, ist falsch!")
```

Startet man dieses Programm, erhält man folgende Ausgabe:

```
$ python3 exception_eigene_klasse.py
Traceback (most recent call last):
  File "exception_eigene_klasse.py", line 4, in <module>
    raise MyException("Was falsch ist, ist falsch!")
__main__.MyException: Was falsch ist, ist falsch!
```

3.3.3 Finalisierungsaktion

Neben `except` und `else` hat ein try-Konstrukt noch den `finally`-Zweig zu bieten. Man bezeichnet diese Form auch als Finalisierungs- oder Terminierungsaktionen, weil sie immer unter allen Umständen ausgeführt werden müssen, und zwar unabhängig davon, ob eine Ausnahme im try-Block aufgetreten ist oder nicht.

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
finally:
    print("Ich werde immer ausgegeben, ob Fehler oder nicht")

print("Mich sieht man nur, wenn es keinen Fehler gab!")
```

In den folgenden Programmläufen demonstrieren wir einen Fehlerfall und einen Durchlauf ohne Ausnahmen:

```
bernd@saturn:~/bodenseo/python/beispiele$ python3 finally_exception.py
Your number: 42
Ich werde immer ausgegeben, ob Fehler oder nicht
Mich sieht man nur, wenn es keinen Fehler gab!
bernd@saturn:~/bodenseo/python/beispiele$ python3 finally_exception.py
Your number: 0
Ich werde immer ausgegeben, ob Fehler oder nicht
Traceback (most recent call last):
  File "finally_exception.py", line 3, in <module>
    inverse = 1.0 / x
ZeroDivisionError: float division by zero
bernd@saturn:~/bodenseo/python/beispiele$
```


3.4 Dateien lesen und schreiben

3.4.1 Datei lesen

Unser erstes Beispiel zeigt, wie man Daten aus einer Datei ausliest. Um dies tun zu können, muss man zuerst die Datei zum Lesen öffnen. Dazu benötigt man die open()-Funktion. Mit der open-Funktion erzeugt man ein Dateiobjekt, genau genommen ein TextIOWrapper-Objekt, und liefert eine Referenz auf dieses Objekt als Ergebniswert zurück.

Um die Datei `bundeslaender.txt` zum Lesen zu öffnen, genügt die folgende Anweisung:

```
fobj = open("bundeslaender.txt")
```

Alternativ kann kann man auch den Parameter-Mode auf „r" setzen:

```
fobj = open("bundeslaender.txt", "r")
# oder über Schlüsselwort:
fobj = open("bundeslaender.txt", mode="r")
```

Nach dem obigen Befehl kann man über den TextIOWrapper `fobj` verschiedene Methoden aufrufen:

- readline: eine Zeile lesen

```
>>> fobj = open("bundeslaender.txt")
>>> fobj.readline()  # lesen einer Zeile
'Land Flaeche maennlich weiblich\n'
>>> fobj.readline()  # noch eine Zeile
'Baden-Württemberg 35751.65 5271 5465\n'
```

- read: ganzen Text in einen String lesen

```
>>> fobj = open("bundeslaender.txt")
>>> txt = fobj.read()
>>> txt[:60]
'Land Flaeche maennlich weiblich\nBaden-Württemberg 35751.65 5'
```

- readlines: Text in eine Liste mit den Zeilen einlesen

```
>>> fobj = open("bundeslaender.txt")
>>> zeilen = fobj.readlines()
>>> zeilen[:4]    # die ersten vier Zeilen
['Land Flaeche maennlich weiblich\n', 'Baden-Württemberg 35751.65 5271 5465\n
     ', 'Bayern 70551.57 6103 6366\n', 'Berlin 891.85 1660 1736\n']
```

Häufig werden Dateien jedoch mittels einer `for`-Schleife Zeile für Zeile durchlaufen. Dies geschieht dann innerhalb eines `with`-Konstrukts, welches nach Beendigung des Blocks automatisch die Datei schließt:

```
with open("bundeslaender.txt") as fh:
    for zeile in fh:
        print(zeile)
```

3.4.2 Datei schreiben

Will man in eine Datei schreiben, benutzt man „w" statt „r" beim `open`. Mit der Methode `write` kann man einen String in die geöffnete Datei schreiben:

```
>>> fh = open("beispiel.txt", "w")
>>> fh.write("1. Zeile\n")
9
>>> fh.write("2. Zeile\n")
9
>>> fh.close()
>>> txt = open("beispiel.txt").read()
>>> print(txt)
1. Zeile
2. Zeile

>>>
```

Im folgenden Programm lesen wir die Datei `bundeslaender.txt` ein und schreiben die Daten in einem gewandelten Format in eine Datei `bundeslaender2.txt`.

```
with open("bundeslaender.txt") as fh_in, \
     open("bundeslaender2.txt", "w") as fh_out:
    fh_in.readline()   # lesen der headerzeile
    fh_out.write("Land Flaeche Einwohner Dichte\n")
    for zeile in fh_in:
        land, flaeche, maennlich, weiblich = zeile.split()
        flaeche = float(flaeche)
        einwohner = int(maennlich) + int(weiblich)
        dichte = round(einwohner * 1000/ flaeche, 2)
        fh_out.write(land + " " + str(flaeche) + " " + str(einwohner) + " " + str
     (dichte) + "\n")
```

Wir zeigen im Folgenden die ersten sieben Zeilen der durch das obige Programm erzeugten Datei `bundeslaender2.txt`:

```
Land Flaeche Einwohner Dichte
Baden-Württemberg 35751.65 10736 300.29
Bayern 70551.57 12469 176.74
Berlin 891.85 3396 3807.82
Brandenburg 29478.61 2560 86.84
Bremen 404.28 663 1639.95
Hamburg 755.16 1743 2308.12
```

3.5 Modularisierung

Module werden mit der `import`-Anweisung in ein Programm eingebunden:

```
import math
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/2)
1.0
```

Es können auch mehrere Module mit einer `import`-Anweisung importiert werden:

```
import math, random
```

import-Anweisungen können an jeder Stelle des Quellcodes stehen, aber man sollte sie der Übersichtlichkeit willen an den Anfang stellen.

3.5.1 Namensräume von Modulen

Wir haben gesehen, dass die Klassen und Funktionen eines Modules nach einem Import in einem eigenen Namensraum zur Verfügung stehen, d.h. man kann beispielsweise auf die sin()-Funktion nur über den vollen Namen („fully qualified") zugreifen, d.h.

```
>>> math.sin(3.1415)
9.265358966049024e-05
```

Möchte man „bequem" auf Funktionen wie z.B. die Sinus-Funktion zugreifen können, so kann man die entsprechenden Funktionen direkt importieren:

```
>>> from math import sin, pi
>>> print(pi)
3.141592653589793
>>> sin(pi/2)
1.0
```

Die anderen Methoden der Bibliothek stehen dann nicht zur Verfügung. Auch nicht mit ihrem vollen Namen.

Man kann auch eine Bibliothek komplett in den globalen Namensraum einbinden. Dabei werden dann gegebenenfalls bereits vorhandene gleichlautende Namen überschrieben, wie dies im folgenden Beispiel dargestellt wird:

```
>>> pi = 57.898 # eigene Variable, die überschrieben wird
>>> print(pi)
57.898
>>> from math import *
>>> print(pi)
3.141592653589793
```

Außerdem ist es möglich, beim Import einer Bibliothek einen neuen Namen für den Namensraum zu wählen. Im Folgenden importieren wir math als m:

```
>>> import math as m
>>> m.pi
3.141592653589793
>>> m.sin(m.pi)
```

3.5.2 Suchpfad für Module

Wenn man ein Modul importiert, z.B. xyz, sucht der Interpreter nach xyz.py in der folgenden Reihenfolge:

- im aktuellen Verzeichnis
- in der Umgebungsvariablen `PYTHONPATH`, die auf Betriebssystemebene gesetzt werden muss
- Falls `PYTHONPATH` nicht gesetzt ist, wird installationsabhängig im Default-Pfad gesucht, also unter Linux/Unix z.B. in /usr/lib/python3.7.

3.5.3 Inhalt eines Moduls

Mit der built-in-Funktion dir() kann man sich die in einem Modul definierten Namen ausgeben lassen:

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
      'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh
    ', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', '
    floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
    'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', '
    log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
    'tanh', 'trunc']
```

3.5.4 Eigene Module

Jede Datei, die gültigen Python-Code enthält und die Dateiendung `.py` hat, ist ein Modul.

Die beiden folgenden Funktionen fib(), die den n-ten Fibonacci-Wert zurückliefert, und die Funktion fiblist() werden in einer Datei fibonacci.py gespeichert:

```
def fib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

```
def fiblist(n):
    fib = [0,1]
    for i in range(1,n):
        fib += [fib[-1]+fib[-2]]
    return fib
```

Von einem anderen Programm oder von der interaktiven Shell kann man nun, falls fibonacci.py innerhalb des Suchpfads zu finden ist, die Datei mit den beiden Fibonacci-Funktionen als Modul aufrufen:

```
>>> import fibonacci
>>> fibonacci.fib(10)
55
>>> fibonacci.fiblist(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fibonacci.__name__
'fibonacci'
>>>
```

3.5.5 Dokumentation für eigene Module

Rufen wir help für unser fibonacci-Modul auf, erhalten wir folgende Ausgabe:

```
Help on module fibonacci:

NAME
    fibonacci

FUNCTIONS
    fib(n)

    fiblist(n)

FILE
    /home/data/bodenseo/python/fibonacci.py
```

Die `help`-Informationen können wir sehr einfach erweitern. Eine allgemeine Beschreibung des Moduls kann man in einem Docstring zu Beginn einer Moduldatei verfassen. Die Funktionen dokumentiert man wie üblich mit einem Docstring unterhalb der ersten Funktionszeile:

```
""" Modul mit wichtigen Funktionen zur Fibonacci-Folge """

def fib(n):
    """ Iterative Fibonacci-Funktion """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

def fiblist(n):
    """ produziert Liste der Fibo-Zahlen """
    fib = [0,1]
    for i in range(1,n):
        fib += [fib[-1]+fib[-2]]
    return fib
```

Die help-Ausgabe sieht nun zufriedenstellend aus:

```
Help on module fibonacci:

NAME
    fibonacci - Modul mit wichtigen Funktionen zur Fibonacci-Folge

FUNCTIONS
    fib(n)
        Iterative Fibonacci-Funktion

    fiblist(n)
        produziert Liste der Fibo-Zahlen

FILE
    /home/data/bodenseo/python/fibonacci.py
```


3.6 Klassen-Definition

3.6.1 Eine einfache Klasse

Die einfachst mögliche Definition einer Klasse sieht wie folgt aus:

```
class A:
    pass
```

Auch wenn in dieser Klasse keine Methoden definiert worden sind, kann man Instanzen erzeugen, was man auch als Instanziierung bezeichnet. Im folgenden Beispiel erzeugen wir die Instanzen `a1` und `a2`:

```
>>> class A:
...     pass
...
>>> a1 = A()
>>> a2 = A()
>>> print(a1)
<__main__.A object at 0x7fbbb508def0>
```

3.6.2 Attribute

Die meisten Python-Objekte können attribuiert werden:[3]

```
>>> import math
>>> math.f = 2.3
>>>
>>> def f():
...     pass
...
>>> f.counter = 0
```

[3] Ausnahmen sind beispielsweise Instanzen der int-, float- und str-Klassen.

Auch eigene Klassen und deren Instanzen lassen sich attribuieren:

```
>>> class Robot:
...     pass
...
>>>
>>> r1 = Robot()
>>> r2 = Robot()
>>>
>>> Robot.name = "Marvin"
>>> r1.name = "Henry"
>>> print(r1.name, r2.name, Robot.name)
Henry Marvin Marvin
```

Bei `Robot.name` im vorigen Beispiel handelt es sich um ein sogenanntes Klassenattribut, während es sich bei `r1.name` um ein Instanzattribut handelt. Klassenattribute werden auch von allen Instanzen geteilt, falls sie kein Instanzattribut mit dem gleichen Namen haben wie im Falle von `r2.name`. Instanzattribute sind individuell für jede Instanz.

Definiert man Variablen oder Funktionen innerhalb einer `class`-Definition, also innerhalb des eingerückten Blocks, so werden diese zu Klassenattributen:

```
class Robot:

    name = "Marvin"

    def say_hi(self):
        return "Hi, I am " + self.name
```

Attribute sind Referenzen auf beliebige Objekte, im obigen Fall also auf ein String-Objekt und ein Funktionsobjekt.

Im folgenden Beispiel nehmen wir an, dass obige Klassendefinition als `simple_class.py` abgespeichert worden ist:

```
>>> from simple_class import Robot
>>> r = Robot()
>>> r.say_hi()
'Hi, I am Marvin'
>>> r.name
'Marvin'
>>> Robot.name
'Marvin'
>>> Robot.say_hi
<function Robot.say_hi at 0x7fdc61aeeea0>
```

`say_hi` im vorigen Beispiel ist eine gewöhnliche Funktion, die allerdings an die Klasse `Robot` gebunden ist. Der „volle" Name der Funktion lautet damit `Robot.say_hi`. Der Aufruf `r.say_hi()` ist gewissermaßen eine abgekürzte Schreibweise für den Aufruf `Robot.say_hi(r)`. Man sieht dann auch, dass `r` zum Argument für den Parameter `self` wird:

```
>>> from simple_class import Robot
>>> r = Robot()
>>> Robot.say_hi(r)
'Hi, I am Marvin'
```

Außerdem wird damit auch klar, dass man statt des Namens `self` auch einen beliebigen anderen Namen hätte nehmen können. Im OOP-Jargon bezeichnet man solche Funktionen üblicherweise als Methoden.

3.6.3 Initialisierung von Instanzen

Bei der Instanziierung, d.h. beispielsweise beim Aufruf `r = Robot()`, wurde bisher ein leeres Objekt erzeugt. Bei den meisten Klassen möchte man jedoch nach der Erzeugung einer Instanz diese initialisieren, d.h. bestimmte Attribute auf einen bestimmten Initialwert setzen. Zu diesem Zweck definiert man in einer Klasse die Methode `__init__`. Diese Methode wird automatisch nach der Instanziierung einer Instanz aufgerufen. In unserem Fall könnten wir beispielsweise einen Roboter auf einen individuellen Namen setzen:

```
class Robot:

    def __init__(self, name="Marvin"):
        self.name = name

    def say_hi(self):
        return "Hi, I am " + self.name


r = Robot("Henry")
print(r.say_hi())
```

Die Ausgabe lautet in obigem Beispiel nun `Hi, I am Henry`.

3.6.4 Vererbung

Klassen können auch von anderen Klassen erben. So erbt im Folgenden die Klasse `MedicalRobot` von `Robot`:

```
class Robot:

    def __init__(self, name="Marvin"):
        self.name = name

    def say_hi(self):
        return "Hi, I am " + self.name


class MedicalRobot(Robot):

    def heal(self, x):
        return x.name + " is healed now, or maybe not!"


mr = MedicalRobot()
print(mr.say_hi())

y = "A String"
print(mr.heal(y))
```

Die Methoden der Oberklasse, auch Basisklasse genannt, also `Robot` in unserem Fall, werden von der abgeleiteten Klasse geerbt. In unserem Fall übernimmt die abgeleitete Klasse, auch Unterklasse oder Kindklasse genannt, `MedicalRobot` die Methoden `__init__` und `say_hi` von `Robot`. Außerdem wird die Kindklasse um eine Methode `heal` erweitert.

Es ist auch möglich, eine bereits definierte Methode zu überlagern. Im folgenden Beispiel überlagern wir die Methode `say_hi`:

```
class Robot:

    def __init__(self, name="Marvin"):
        self.name = name

    def say_hi(self):
        return "Hi, I am " + self.name


class MedicalRobot(Robot):

    def heal(self, x):
        return "x is healed now, or maybe not!"

    def say_hi(self):
        return "Hi, I am " + self.name + ", your personal nurse!"



mr = MedicalRobot()
print(mr.say_hi())

r = Robot("Henry")
print(r.say_hi())
```

Wir erhalten die folgende Ausgabe:

```
Hi, I am Marvin, your personal nurse!
Hi, I am Henry
```

3.6.5 Private, geschützte und öffentliche Attribute

In der Objektorientierung werden prinzipiell drei Attributarten unterschieden:

- Public oder öffentliche Attribute sind Attribute, auf die man von überall, d.h. innerhalb der eigenen Klasse, von anderen Klassen oder von Anwendungen, welche die Klasse nutzen, lesend und schreibend zugreifen kann und darf.
- Protected oder geschützte Attribute sind Attribute, auf die man nur aus der Klasse selbst oder von abgeleiteten Klassen zugreifen darf. In Python werden solche Attribute mit einem führenden Unterstrich versehen. Allerdings sind diese Attribute nicht „geschützt", wie es der Name vermuten lässt. Der Unterstrich ist eine Konvention, um kenntlich zu machen, dass diese Klassen als `protected` anzusehen sind und nicht außerhalb der Vererbungshierarchie benutzt werden dürfen.
- Private Attribute dürfen und können nur innerhalb der Klasse, in der sie definiert worden sind, benutzt werden. Private Attribute führen zwei führende Unterstriche im Namen.

Wir demonstrieren dies im folgenden Beispiel:

```
class A:

    def __init__(self):
        self.pub = "Ich bin ein public-Attribut"
        self._prot = "Ich bin ein geschütztes Attribut"
        self.__priv = "Ich bin privat!"


a = A()
print(a.pub)
print(a._prot)
print(a.__priv)
```

In der Ausgabe des obigen Programms können wir sehen, dass wir auf das öffentliche und das geschützte Attribut von außen zugreifen können, aber nicht auf das private Attribut:

```
Ich bin ein public-Attribut
Ich bin ein geschütztes Attribut
Traceback (most recent call last):
  File "attribut_arten.py", line 12, in <module>
    print(a.__priv)
AttributeError: 'A' object has no attribute '__priv'
```

3.6.6 Properties

Sollte es notwendig sein, `public`-Attribute zu kapseln, so geschieht dies in Python mithilfe von Properties. Üblicherweise, d.h. in vielen anderen OO-Programmiersprachen, werden private Attribute (Daten) einer Klasse mithilfe von speziellen Zugriffsfunktionen, häufig als Getter und Setter bezeichnet, gekapselt. Im folgenden Beispiel zeigen wir dies mit der Klasse `Robot`. Wir definieren ein privates Klassenattribut für nicht-erlaubte Namen `__forbidden_names`. Der Name eines Roboters wird in einem privaten Attribut `__name` versteckt, d.h. man kann von außen nicht mehr direkt darauf zugreifen, sondern nur noch über die Methoden `get_name` und `set_name`. Die Methode `set_name` verhindert nun, dass ein Roboter einen unzulässigen Namen erhält:

```
class Robot:

    __forbidden_names = {"Henry", "Oscar"}

    def __init__(self, name="Marvin"):
        self.set_name(name)

    def get_name(self):
        return self.__name

    def set_name(self, name):
        if name in Robot.__forbidden_names:
            self.__name = "Marvin"
        else:
            self.__name = name
```

Dies ist aber nicht „pythonisch". Nach den offiziellen Stilrichtlinien von Python[4] implementiert man Dateninhalte, die man Benutzern der Klasse zugänglich machen will, in `public`-Attributen. Benötigt ein solches Attribut später eine spezielle Behandlung, wie wir es oben in der `set_name`-Methode hatten, dann können wir eine `property` einführen. Das Interface-Verhalten wird dadurch für die Benutzer der Klassen nicht verändert.

```
class Robot:

    __forbidden_names = {"Henry", "Oscar"}

    def __init__(self, name="Marvin"):
        self.name = name

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, name):
        if name in Robot.__forbidden_names:
            self.__name = "Marvin"
        else:
            self.__name = name


x = Robot("Isidor")
print(x.name)
x.name = "Henry"
print(x.name)
```

[4] PEP 8 – Style Guide for Python Code, https://www.python.org/dev/peps/pep-0008/