Einführung in Python - Teil 2

Lernziel

Lernziel

```python
from typing import List


def season_points(game_results: List[List[int]]) -> int:
    points: int = 0

    for game_result in game_results:
        own_goals: int = game_result[0]
        opp_goals: int = game_result[1]
        if own_goals > opp_goals:
            points += 3
        elif own_goals == opp_goals:
            points += 1
        # else:
        #     points += 0

    return points
```

Listen und Schleifen

Listen

Oft müssen in der Programmierung mehrere Elemente (eines Typs) verarbeitet werden, aber es ist nicht klar, wie viele Elemente. Beispiele:

- Minimum / Maximum / Summen
- Sammlung von Texten

Schlechte Lösung: Definition vieler Methoden
Bessere Lösung: Verwendung eines Datentyps, der beliebig viele Elemente (Menge) enthalten kann:

Listen

```
def minimum_0():
    return "Fehler!"


def minimum_1(var1):
    return var1


def minimum_2(var1, var2):
    if var1 < var2:
        return var1
    else:
        return var2
```

Definition Listen

Listen werden in Python in eckigen Klammern definiert. Die einzelnen Werte werden dabei durch ein Komma getrennt.

Achtung: Die Datentypen der einzelnen Elemente müssen **nicht** gleich sein!

```
# Leere Liste
empty = []

# Liste mit einem Element
one = [1]

# Liste mit 'Quadratzahlen'
squares = [1, "vier", 9.0, 16, 16 + 9, 72 / 2]

# Liste mit Primzahlen zwischen 0 und 20
primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

Zugriff auf Listenelemente

Der Zugriff auf Element(e) zum Auslesen und zur Zuweisung erfolgt mit eckigen Klammern und Index:

```
listenname[index]
```

- Index zählt von **0** bis **Anzahl der Elemente - 1** („0-basiert“)
- Alternative: Negative Indizes zählen vom letzten Element bis zum ersten Element
- Bestimmen der Länge einer Liste mit der Funktion `len(liste)`
- Zugriff aus letztes Element der Liste am besten mit Berechnung über Länge der Liste (oder mit -1), da Anzahl der Element dynamisch!

Beispiele: Zugriff auf Listenelemente

```
primes = [2, 3, 5, 7, 11, 13, 17, 19]

# Zugriff auf erstes Element, 2
first = primes[0]

# Zugriff auf drittes Element (5), Setzen auf 7
primes[2] = 7

# Zugriff auf letztes Element, 19
last1 = primes[7]
last2 = primes[-1]
last3 = primes[len(primes) - 1]

# Setzen des letzten Elements auf 0
primes[len(primes) - 1] = 0
```

”Hausaufgabe”: Liste mit Quadratzahlen

- Erstellen Sie eine Liste, die aus 11 Nullen besteht
- Schreiben Sie in die Felder der Liste jeweils die Quadrate aller Zahlen zwischen 0 und 10
- Geben Sie alle Felder der Liste aus und vergewissern Sie sich so, dass ihre Liste korrekt ist

Lösung: Liste mit Quadratzahlen

```
# Liste mit 0en erstellen
squares = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

# Liste füllen
squares[0] = 0 * 0
squares[1] = 1
# ...
squares[10] = 100

# Liste ausgeben
print(squares[0])
print(squares[1])
# ...
```

Listen: Wichtige Methoden

Methode	Funktion
`x in liste`	Test, ob Liste `liste` ein Element `x` enthält
`x not in liste`	Test, ob Liste `liste` ein Element `x` **nicht** enthält
`liste.append(x)`	Element `x` wird an `liste` angehängt
`liste1 + liste2`	Konkatenation von `liste1` und `liste2`
`liste * n`	`n`-fache Konkatenation von `liste`
`len(liste)`	Anzahl der Elemente in `liste`
`min(liste)`	Minimum von `liste`
`max(liste)`	Maximum von `liste`
`liste.sort()`	Sortiert `liste` (falls möglich)
`liste.reverse()`	Kehrt die Reihenfolge von `liste` um

Schleifen

Um den Umgang mit Listen (und anderen Mengen-Datentypen) zu vereinfachen, gibt es in Python zwei Arten von *Schleifen*:

While-Schleifen

```
while bedingung:
    # Schleifenrumpf
```

For-Schleifen

```
for i in iterable:
    # Schleifenrumpf
```

While-Schleifen

While-Schleifen benutzen eine Bedingung, um zu testen, ob der Schleifenrumpf ausgeführt werden soll (also falls Ergebnis der Bedingung „True“)

```
while bedingung:
    # Schleifenrumpf
```

Achtung: Möglichkeit der Endlosschleife (falls Bedingung immer auf „True“ evaluiert)

- Manchmal ist Endlosschleife gewünscht (Eingebettete Systeme, zum Beispiel Snackautomat)
- Meist jedoch ein Programmierfehler, der zu unerwünschten Ergebnissen führt (Programmabbruch mit CTRL + C)

Beispiel: While-Schleifen

```
i = 0
while i < 5:
    print(i)
    i = i + 1  # Ohne Erhöhung: Endlosschleife!

nums = [0, 3, 4, 7, 10, 12]
j = 0
while nums[j] < 10:
    print(nums[j])
    j += 1

while True:
    print("Hilfe, gefangen in Endlosschleife!")
```

For-Schleifen

For-Schleifen benutzen eine Variable (hier: `i`), um über eine Sequenz (z. B. eine Liste, hier: Wert der Variable `iterable`) zu iterieren

```
for i in iterable:
    # Schleifenrumpf
```

For-Schleifen haben (meist) eine feste Anzahl an Ausführungen des Schleifenrumpfs $\Rightarrow$ keine Endlosschleife

Bei einer Iteration über einen Zahlenbereich kann die Funktion `range()` verwendet werden:

- `range(n)`: Alle Zahlen von 0 bis (einschließlich) $n - 1$
- `range(i, j)`: Alle Zahlen von i bis (einschließlich) $j - 1$

Beispiel: For-Schleifen

```
for i in range(11):  # Alle Zahlen von 0 bis 10
    print(i)

for j in range(3, 8):  # Alle Zahlen von 3 bis 7
    print(j)

squares = [0, 1, 4, 16, 25, 36, 49, 64]
for square in squares:
    print(square)

for index in range(len(squares)):
    print(squares[index])
```

Listen und Type hints

Auch bei Listen sind Type hints möglich. Dazu muss jedoch zuerst die spezielle Klasse `List` aus dem Paket `typing` importiert werden (zu beidem später mehr…).

Listen sind *generisch*, d. h. es gibt verschiedene Ausprägungen von Listen mit entsprechenden *Subtypen*:

- Listen von Ganzzahlen
- Listen von Fließkommazahlen
- Listen von String
- …

Der jeweilige Subtyp einer Liste (Ganzzahlen, Fließkommazahlen, etc.) wird in eckigen Klammern angegeben.

Beispiele: Listen und Type hints

```python
from typing import List, Any

squares: List[int] = [i * i for i in range(10)]
names: List[str] = ["Jack", "James", "Jason"]

# Liste von Listen ('Matrix')
matrix: List[List[int]] = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]

# Liste mit verschiedenen Inhalten, daher
# keine 'genaue' Angabe möglich
mixed_list: List[Any] = [1, "Zwei", 3.0, 4]
```

Instanziierung von Listen

Bei einer festen Anzahl an Elementen kann eine Liste direkt erstellt werden:

```
even_squares_1 = [0, 4, 16, 36, 64, 100]
```

Mit einer Schleife kann eine Liste einfach mit Werten gefüllt werden:

```
even_squares_2 = []
for i in range(11):
    if i % 2 == 0:
        even_squares_2.append(i * i)
```

Mit etwas „Syntactic Sugar“ geht dies noch kürzer:

```
even_squares_3 = [i * i for i in range(11) if i % 2 == 0]
```

Beispiele: Instanziierung von Listen

```
# Alle ungeraden/geraden Zahlen von 1 bis 101
unevens = [2 * i + 1 for i in range(51)]
evens = [i for i in range(101) if i % 2 == 0]

# Alle Quadratzahlen der Vielfachen von 3 zwischen 1 und 100
squares = [i * i for i in range(101) if i % 3 == 0]

# Alle Primzahlen von 1 bis 100
# Voraussetzung: Funktion isPrime(k)!
primes = [i for i in range(101) if isPrime(i)]
```

”Hausaufgabe”: Iteration und Instanziierung

Schreiben Sie eine Funktion, die die Summe der Elemente einer Liste (aus Ganzzahlen) berechnet!

Bei einer leeren Liste soll der String „Fehler!“ zurückgegeben werden!

```
from typing import List

def sum_of_elements(numbers: List[int]) -> int:
    pass
```

Testen Sie ihre Funktion mit verschieden Beispielen:

- Alle Zahlen von 0 bis 10 (Ergebnis: 55)
- Der leeren Liste [] (Ergebnis: „Fehler“)
- Der Liste aller Quadrate aller geraden Zahlen zwischen 1 und 7 (56)

Lösung: Iteration und Instanziierung

```python
def sum_of_elements(numbers: List[int]) -> int:
    if len(numbers) == 0:
        return 'FEHLER'
    el_sum = 0
    for element in numbers:
        el_sum = el_sum + element
    return el_sum


print(sum_of_elements([i for i in range(11)]))  # 55
print(sum_of_elements([]))  # 'FEHLER'
print(sum_of_elements([j * j for j in range(8) if j % 2 == 0]))  # 56
```

Listen: Slicing

Anstatt eines einzelnen Elements kann mit dem Operator `[]` auch ein Teil („Slice“) einer Liste von einem Startindex (inklusive) bis zu einem Endindex (**exklusiv**) ausgewählt werden. Der Slice einer Liste ist wieder eine Liste.

- Schreibweise:
  - `[start_index:end_index]` oder
  - `[start_index:end_index:schrittweise]` (Schrittweise ist optional)
- Falls Wert des Parameters ein Standardwert sein soll, kann die Position auch leer gelassen werden (siehe Beispiele). Dann werden folgende Standardwerte angenommen:
  - `start_index = 0`
  - `end_index = len(list)`
  - `schrittweise = 1`
- Bei Slicing können auch negative Indizes verwendet werden

Beispiele: Slicing

```
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

# Zugriff auf alle Elemente, Schrittweite 1
all_primes = primes[:]  # oder: primes[::]

# Zugriff auf erstes bis drittes Element
first_three_primes = primes[0:3]  # oder: primes[:3]

# Zugriff auf letzte zwei Elements
last_two_primes = primes[-2:]

# Zugriff auf El. mit geradem Index (0, 2, 4...)
primes_with_even_indexes = primes[::2]  # oder: primes[0::2]

# Elemente in umgekehrter Reihenfolge
primes_reversed = primes[::-1]
```

Einschub: Importe

Pakete

Um den Type hint für Listen verwenden zu können, mussten wir vorher `List` aus dem Paket `typing` importieren. **Warum?**

Beim Programmieren gilt die Regel:

Do not reinvent the wheel!

Python bringt in der Standardinstallation verschiedene Pakete mit. Pakete sind Sammlungen von Code, der bereits von einem anderen Programmierer geschrieben und ausführlich getestet wurde. Beispiele für solche Pakete sind:

- `math` für erweiterte Mathematik
- `random` für Funktionen mit (Pseudo-)Zufall oder
- `os`, das Funktionen zur Interaktion mit dem Betriebssystem bereitstellt

Importieren von Paketen

Um die Funktionen eines Pakets in einem Skript nutzen zu können, müssen sie zuerst importiert werden.

Dazu gibt es verschiedene Möglichkeiten:

- `import` *`paket_name`*
  Importiert das komplette Paket, Zugriff über *`paket_name.function_name`*
- `from` *`paket_name`* `import` *`to_import1`*`,` *`to_import2`*
  Importiert nur die angegebenen Funktionen, Zugriff nur über Name (ohne Paket), Möglichkeit von Namenskonflikten
- `from` *`paket_name`* `import *`
  Importiert **alle** Funktionen aus dem Paket, Zugriff nur über Name, Möglichkeit von Namenskonflikten

Beispiel: Importe

```
import math
from random import randint

my_list = [math.sqrt(randint(0, 5)) for x in range(5)]

print(my_list)
# Wurzeln 5 'zufälliger' Zahlen zwischen 0 und 5
```

Exceptions

Exceptions

Programmierer machen Fehler. Dann kann es dazu kommen, dass

- der Computer nicht weiß, wie er eine Anweisung interpretieren soll (Kompilier-/Interpretationsfehler)
- die Anweisung nicht erlaubt ist (z. B. Speicherzugriff an falscher Stelle)
- …

In einer solchen Situation wird eine sogenannte *Exception* „geworfen“ und eine Ausnahmesituation eingeleitet.

```
x = 5
y = 'string'
z = x + y
```

```
Traceback (most recent call last):
  File "/home/bjorn/workspace/latex/infohaf_slides/code/02_python/exceptions/exception.py", line 3, in <module>
    z = x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'

Process finished with exit code 1
```

„Werfen“ von Exceptions

Um Fehler anzuzeigen, kann man auch selbst eine Exception werfen. Dies geschieht mit dem Schlüsselwort `raise`:

```
def div(x: float, y: float) -> float:
    if y == 0.0:
        raise Exception("You cannot divide by 0!")
    else:
        return x / y


print(div(5, 2))  # 2.5
print(div(10, 0))  # Exception!
```

Fangen von Exceptions

Mit dem Konstrukt `try - catch` können Exceptions gefangen werden.

Dabei wird der Code im Try-Block normal ausgeführt.

Sollte eine Exception geworfen werden, wird die Ausführung abgebrochen und der Code im Except-Block ausgeführt.

```
try:
    division = 5 / 0
except ZeroDivisionError:
    print("you divided by 0, didn't you?")

print(division)  # Exception - not defined!
```

Fangen von mehreren Exception

```python
# ArgFalseException und ArgZeroException erben
# von Exception und wurden selbst definiert
def some_func(arg1: bool, arg2: int) -> int:
    if not arg1:
        raise ArgFalseException("Argument 1 was false!")
    if arg2 == 0:
        raise ArgZeroException("Argument 2 was zero!")
    return 1

try:
    file = some_func(False, 0)
except ArgFalseException:
    print("that did not work...")
except ArgZeroException:
    print("that did not work either...")
```

Finally-Block

Code, der im Finally-Block steht, wird immer ausgeführt, egal ob eine Exception geworfen wurde oder nicht.

Dies wird zum Beispiel dazu benutzt, Datenbankverbindungen zu schließen.

```
try:
    x = 5 / 0
except ZeroDivisionError:
    print("you divied by 0...")
finally:
    print("im Finally-Block")

# you divied by 0...
# im Finally-Block
```

```
try:
    x = 5 / 1
except ZeroDivisionError:
    print("you divied by 0...")
finally:
    print("im Finally-Block")

# im Finally-Block
```

Test Driven Development (Light)

Bisheriges Vorgehen

> *Beware of bugs in the above code; I have only proved it correct, not tried it.*
>
> *Donald Knuth*

Bisheriges Vorgehen beim Programmieren:

- Programmieren der Funktionalität
- Manuelle Tests mit verschiedenen Ausführungen

Aber ist dies die richtige Vorgehensweise? (Spoiler: **Nein**...)

- Kann man sicher sein, dass ein Programm korrekt ist?
- Wie testet man sein Programm am besten?

Korrektheit eines Programms

Unterscheidung: Verifikation und Validierung

- Verifikation: Programm genügt einer Spezifikation (Programm $\approx$ Algorithmus)
- Validierung: Spezifikation löst die Aufgabe korrekt

Nachfolgend: Vereinfachende Annahme, dass Programm der Spezifikation genügt (Verifikation erfolgreich), also Konzentration auf Validierung

- Gute Nachricht: Nachweis der Korrektheit kann für einzelne Eingaben erfolgen (siehe bisheriges Vorgehen...) $\Rightarrow$ Unittests
- Schlechte Nachricht: „Totale Korrektheit“ ist schwer bis unmöglich nachzuweisen (Überprüfen jeder Eingabemöglichkeit...)

Unittests

> *Durch Testen kann man stets nur die Anwesenheit, nie aber die Abwesenheit von Fehlern beweisen.*
>
> *- Edsger Wybe Dijkstra*

Unittests...

- Testen die funktionalen Einheiten von Programmen (Funktionen, Klassen, ...)
- Garantieren keine Korrektheit des Programms, sondern können nur Fehler nachweisen (siehe Zitat)
- Mit entsprechenden Bibliotheken automatisch durchführbar
  JUnit (Java), JsUnit (Javascript), bei Python Modul „unittest“

Vorgehen bei Unittests

Vorgehen in 3 Schritten:

1. **Arrange** - Aufsetzen Testumgebung
2. **Act** - Ausführung zu testender Code mit definierter Eingabe
3. **Assert** - Vergleich der Ausgabe mit erwarteter Ausgabe

Ein fehlgeschlagener Test wird als „rot“, ein erfolgreicher als „grün“ bezeichnet.

Wichtig: Testen der „Edge Cases“, also zum Beispiel:

- leere Listen
- Division durch 0
- negative Zahlen
- Speziell bei Python: andere Eingabetypen
- ...

Vorgehen: Test Driven Development (Pseudocode, vereinfacht)

```
while not len(todo) == 0:
    current_work = todo[0]
    # 1. Erstelle Tests, die neue Funktionalität testen
    tests = erstelle_unittests(current_work)
    # 2. Implementiere neue Funktionalität
    erweitere_funktionalität(current_work)
    # 3. Sorge dafür, dass alle Test 'grün' sind
    while not alle_erfolgreich(tests):
        korrigiere_funktionalität(current_work)

    todo.remove(0)
```

Durchführung TDD (vereinfacht) mit Python

„Echte“ Unittests verwenden Modul `unittest`, dies beruht jedoch auf der Erstellung von Klassen

Daher: Vereinfachte Unittests, später Vorstellung „richtige“ Tests

Verwendung der Funktion `assert(`*`bedingung1`*`)`, um die Funktion auszuführen und zu testen, ob das Ergebnis mit den Erwartungen übereinstimmt.

Beispiel: TDD in Python

Wir wollen eine Funktion erstellen `test(i)` erstellen, die als Ergebnis `5 / i` zurückgibt. Wenn als Argument eine „0“ übergeben wird, soll die Funktion den String „Fehler“zurückgeben.

```
def test(i):
    if i == 0:
        return "Fehler"
    return 5 / i


assert (test(0) == "Fehler")
assert (test(1) == 5)
assert (test(2) == 2.5)
```

Aufgabe: TDD

Sie sollen eine Funktion `get_index(liste, x)` erstellen, die den Index einer Zahl aus einer Liste von Zahlen heraussucht. Falls die Zahl nicht vorhanden ist, soll eine -1 zurückgegeben werden.

```
from typing import List
def get_index(numbers: List[int], x: int) -> int:
```

Vorgehen:

- Schreiben Sie mindestens 5 Testfälle, die auch die „Edge Cases“ abdecken!
- Schreiben Sie nur den Funktionsrumpf mit einem Standardrückgabewert und führen Sie ihre Testfälle aus!
- Schreiben Sie jetzt die Funktionalität und führen wieder ihre Testfälle aus. Solange diese noch fehlschlagen, müssen Sie ihre Funktion überarbeiten!

Lösung: TDD

```python
def get_index(numbers: List[int], x: int) -> int:
    index = 0
    for v in numbers:
        if v == x:
            return index
        index += 1
    return -1

assert (get_index([], 2) == -1)
assert (get_index([1], 2) == -1)
assert (get_index([1, 2], 2) == 1)
assert (get_index([0, 2, 4, 6, 8], 0) == 0)
assert (get_index([-10, -5, 0, 5, 10, 10], 10) == 4)
```

Wiederholung Lernziel

Lernziel

```python
from typing import List


def season_points(game_results: List[List[int]]) -> int:
    points: int = 0

    for game_result in game_results:
        own_goals: int = game_result[0]
        opp_goals: int = game_result[1]
        if own_goals > opp_goals:
            points += 3
        elif own_goals == opp_goals:
            points += 1
        # else:
        #     points += 0

    return points
```