Einführung in Python - Teil 2

Lernziel

Lernziel

```
class GameResult:
    def __init__(self, home_team: str, away_team: str,
                 points_home: int, points_away: int):
        self.home_team: str = home_team
        self.away_team: str = away_team
        self.points_home: int = points_home
        self.points_away: int = points_away

    def announce(self) -> str:
        result_string: str
        if self.points_home > self.points_away:
            result_string = 'gewonnen'
        elif self.points_home < self.points_away:
            result_string = 'verloren'
        else:
            result_string = 'unentschieden gespielt'
        return f'{self.home_team} hat gegen {self.away_team} ' \
               f'mit {self.points_home} : {self.points_away} {result_string}.'
```

Tuples & Dictionaries

Zusammenfassung Listen

Listen sind ein **Containerdatentyp** für eine beliebige Menge von *beliebigen* Elementen.

Die Definition und der Zugriff (+ Slicing!) erfolgen mittels eckiger Klammern.

Die Indexierung der Listenelemente mit n Elementen beginnt bei 0 und endet bei $n - 1$.

```
from typing import List

my_list: List = [0, 'eins', 2 / 1, 2 * 1.5, 2 ** 2]

my_list[index]
my_list[start_index:end_index]
my_list[start_index:end_index:schrittweise]
```

Tuples

Tuple sind wie Listen Datentypen für Objektmengen. Es gibt aber Unterschiede:

- Tuple werden mit runden Klammern erstellt
- Tupel sind **nicht veränderbar**:
  - Nicht erweiterbar
  - Keine neue Zuweisung
  - Kein Löschen von Elementen
  - ...

  $\Rightarrow$ Führt zu Exception

Ansonsten haben sie die gleiche Funktionsweise:

- Zugriff mit eckigen Klammern, auch Slicing möglich
  ($\rightarrow$ Slice eines Tuples ist wieder ein Tuple)
- Iteration mit Schleifen
- ...

Beispiele: Tuples

```python
from typing import Tuple

my_tuple: Tuple[int, int, int] = (1, 2, 3)

first_elem: int = my_tuple[0]  # 1

new_tuple: Tuple[int, int] = my_tuple[:2]  # Neues Tuple: (1, 2)
other_new_tuple: Tuple[int, int] = my_tuple[::2]  # Neues Tuple: (1, 3)

# (1, 2, 3, 1, 2, 3)
concatenated_tuple: Tuple[int, int, int, int, int, int] = my_tuple * 2

# Nicht möglich, führt jeweils zu einem TypeError:
# 'tuple' object does not support item assignment
del my_tuple[0]
my_tuple[1] = 3
```

Dictionaries

Dictionaries sind Datentypen für Mengen von Schlüssel-Wert-Paaren (wie echte Wörterbücher) und definieren eine Abbildungs-Relation.

Die Definition erfolgt in geschweiften Klammern `{}`, Angabe der Paare als

```
Schlüssel: Wert
```

Die Schlüssel müssen eindeutig und unveränderlich sein (Stichwort: Zugriff, also zum Beispiel Strings, Zahlen, Tuple, keine Liste).

Der Zugriff erfolgt mit eckigen Klammern und Angabe des Schlüssel, es gibt aber kein Slicing.

Beispiel: Dictionary

```
from typing import Dict, Any

persDic: Dict[str, Any] = {'alter': 50, 'groesse': 170, 'nation': 'deutsch'}

print(persDic['alter'], ';', persDic['groesse'])  # 50 ; 170
print(persDic['nation'])  # 'deutsch'

# Update eines Wertes
persDic['alter'] = 51
print(persDic['alter'])  # 51

# Hinzufügen eines Schlüssel-Wert-Paares
persDic['geschlecht'] = 'weiblich'
print(persDic['geschlecht'])  # weiblich
```

Iteration bei Dictionaries

Um über die Schlüssel-Wert-Paare in einem Dictionary zu iterieren, wird die Methode *dict*.items() benutzt.

Diese gibt in einer Schleife jeweils ein Tuple aus Schlüssel und Wert zurück:

```
from typing import Dict

values: Dict[int, str] = {1: 'rot', 2: 'blau', 3: 'rot'}

for (key, value) in values.items():
    print("{} -> {}".format(key, value))

# 1 -> rot
# 2 -> blau
# 3 -> rot
```

Aufgabe: Dictionaries

Vervollständigen Sie folgende Funktion `count_frequencies`, die die Häufigkeit der einzelnen Ganzzahlen in der übergebenen Liste berechnet:

```
from typing import Dict, List


def count_frequency(numbers: List[int]) -> Dict[int, int]:
    pass


assert ({} == count_frequency([]))
assert ({1: 3} == count_frequency([1, 1, 1]))
assert ({1: 3, 2: 4} == count_frequency([1, 2, 2, 1, 2, 1, 2]))
```

Lösung: Dictionaries

```python
from typing import Dict, List


def count_frequency(numbers: List[int]) -> Dict[int, int]:
    counts: Dict[int, int] = {}
    for num in numbers:
        if num in counts:
            counts[num] += 1
        else:
            counts[num] = 1
    return counts


assert ({} == count_frequency([]))
assert ({1: 3} == count_frequency([1, 1, 1]))
assert ({1: 3, 2: 4} == count_frequency([1, 2, 2, 1, 2, 1, 2]))
```

Übersicht: Listen, Tuple, Dictionaries

	Liste	Tuple	Dictionary
Containertyp	Beliebig	Beliebig	(Schlüssel-Wert-)Paare
Erstellung	`l = [1, 2, 3]`	`t = (1, 2, 3)`	`d = {key: value}`
Werte veränderbar	Ja	Nein	Ja (Schlüssel konstant)
Zugriff	`l[0]`	`t[0]`	`d[key]`
Slicing	Ja	Ja	Nein

Mehr zu Strings

Strings vs Listen: Ähnlichkeit

Strings haben in Python **große Ähnlichkeit zu Listen (von einzelnen Buchstaben)**. Unter anderem sind folgende Operationen möglich:

```python
my_str: str = 'Hallo!'
# Länge
print(len(my_str))  # 6

# Iteration
for char in my_str:
    print(char)
    # H
    # a
    # l
    # l
    # o
```

```python
# (Mehrfache) Konkatenation
print(my_str + ' Hey!')  # Hallo! Hey!
print(my_str * 2)  # Hallo!Hallo!

# Inhaltstests
print('H' in my_str)  # True
print('i' in my_str)  # False

# Index /Slicing
print(my_str[0] + "::" + my_str[1:])  # H::allo!
print(my_str[::-1])  # !ollaH
```

Strings vs Listen: Umwandlung

Oft muss man einen String in eine Liste von Strings oder umgekehrt eine Liste von Strings in einen einzelnen umwandeln. Je nach *Use Case* gibt es dazu zum Beispiel folgende Funktionen:

- *str*.split(*separator*: str)
- *str*.join(*strs*: List[str]):

```
my_string: str = '1, 2, 3, 4, 5, 6, 7, 8, 9'

my_part_string: List[str] = my_string.split(', ')
# ['1', '2', '3', '4', '5', '6', '7', '8', '9']

my_other_string: str = "-".join(my_part_string)
# '1-2-3-4-5-6-7-8-9'
```

Aufgabe: Palindrom

Implementieren Sie folgende Funktion *is_palindrome*, die überprüft, ob der übergebene String ein Palindrom darstellt! Sie können davon ausgehen, dass der String nur aus Kleinbuchstaben besteht.

Benutzen Sie TDD!

```
def is_palindrome(word: str) -> bool:
    pass
```

Definition Palindrom:

Ein Palindrom ist eine Zeichenkette, die von vorn und von hinten gelesen dasselbe ergibt.

Musterlösung: Palidrom

```python
def is_palindrome(word: str) -> bool:
    for i in range(len(word) // 2):
        if word[i] != word[-i - 1]:
            return False
    return True

    # Andere mögliche Lösung mit Slicing:
    return word == word[::-1]


assert (is_palindrome("otto") is True)
assert (is_palindrome("max") is False)
assert (is_palindrome("lagerregal") is True)
```

String Formatting

Oft müssen in einer Ausgabe verschiedene Datentypen gemischt werden, zum Beispiel „Spieler *name* hat *wert* Punkte erzielt“.

Dies wird in Python durch String Formatting realisiert:

```
name = "test"
value = 5
print("Spieler {} hat {} Punkte erzielt".format(name, value))
# Spieler test hat 5 Punkte erzielt
```

Es werden alle Vorkommen von geschweiften Klammern durch die Argumente ersetzt, zum Beispiel über

- die Reihenfolge (leere geschweifte Klammern)
- Indizes (Zahlen in den geschweiften Klammern)
- die Namen der Argumente (ähnlich eines Dictionaries)

Beispiele: String Formatting

```
s1 = 'x = {}, y = {}, z = {}'.format(11, 22, 33)
print(s1)  # x = 11, y = 22, z = 33

s2 = 'x = {2}, y = {1}, z = {0}'.format(11, 22, 33)
print(s2)  # x = 33, y = 22, z = 11

s3 = 'x = {fir}, y = {thi}, z = {sec}'.format(fir=11, sec=22, thi=33)
print(s3)  # x = 11, y = 33, z = 22
```

Neu in Python 3.7: `f-Strings`

In Python 3.7 wurde mit den sogenannten f-Strings eine einfacherere und lesbarere Möglichkeit eingeführt, Strings zu formatieren.

Dazu wird dem String ein `f` vorangestellt. Nun wird alles, was im String in geschweiften Klammern steht, als Python-Code ausgewertet:

```
from typing import Dict

x: int = 3
squares_dict: Dict[int, int] = {i: i * i for i in range(x)}

print(f'Die Quadrate der ersten {x} Zahlen sind: {squares_dict}')
# Die Quadrate der ersten 5 Zahlen sind: {0: 0, 1: 1, 2: 4}
```

Strings: Weitere Methoden

```python
my_str: str = 'Dies ist ein Test.'

# Alle Buchstaben groß
my_str_upper: str = my_str.upper()
print(my_str_upper)  # DIES IST EIN TEST.

# Alle Buchstaben klein
my_str_lower: str = my_str.lower()
print(my_str_lower)  # dies ist ein test.

# Teilstrings ersetzen
my_str_replaced = my_str.replace('i', 'o')
print(my_str_replaced)  # Does ost eon Test.
```

Objektorientierte Programmierung: Teil 1

Aufgabe: 2D-Vektoren (1)

Für eine Simulation sollen Sie einige Funktionen zum Rechnen mit zweidimensionalen Vektoren implementieren. Diese werden als Tuple mit 2 Einträgen dargestellt.

Eine Summenfunktion könnte folgendermaßen aussehen:

```
from typing import Tuple

# Benenne 'Tuple[float, float]' um in 'Vec'
Vec = Tuple[float, float]

def vector_2d_sum(vec1: Vec, vec2: Vec) -> Vec:
    return vec1[0] + vec2[0], vec1[1] + vec2[1]

assert (vector_2d_sum((1, 1), (2, 2)) == (3, 3))
```

Aufgabe: 2D-Vektoren (2)

Implementieren Sie folgende Funktionen. Nutzen Sie TDD!

```
def vector_2d_sub(vec1: Vec, vec2: Vec) -> Vec: # Differenz
    pass

def vector_2d_dot(vec1: Vec, vec2: Vec) -> float: # Skalarporodukt
    pass

def vector_2d_abs(vec: Vec) -> float:  # Betrag
    pass

def vector_2d_scale(vec: Vec, scalar: float) -> Vec:
    # Multiplikation mit Skalar
    pass
```

Lösung: 2D-Vektoren (1)

```
def vector_2d_sub(vec1: Vec, vec2: Vec) -> Vec:
    return vec1[0] - vec2[0], vec1[1] - vec2[1]

assert (vector_2d_sub((5, 3), (2, 1)) == (3, 2))
assert (vector_2d_sub((1, 2), (-1, -5)) == (2, 7))

def vector_2d_dot(vec1: Vec, vec2: Vec) -> float:
    return vec1[0] * vec2[0] + vec1[1] * vec2[1]

assert (vector_2d_dot((1, 1), (1, 1)) == 2)
assert (vector_2d_dot((5, 7), (3, 8)) == 71)
```

Lösung: 2D-Vektoren (2)

```
def vector_2d_abs(vec: Vec) -> float:
    return vector_2d_dot(vec, vec) ** 0.5

assert (vector_2d_abs((1, 1)) == 1.4142135623730951)
assert (vector_2d_abs((5, 7)) == 8.602325267042627)

def vector_2d_scale(vec: Vec, scalar: float) -> Vec:
    return vec[0] * scalar, vec[1] * scalar

assert (vector_2d_scale((1, 3), 3) == (3, 9))
assert (vector_2d_scale((5, 7), 0.5) == (2.5, 3.5))
```

Lessons Learned: Vektorrechung

Probleme der aktuellen Umsetzung der Aufgabe:

- Absturz, wenn kein 2D-Tupel übergeben wird, sondern etwas anderes
- 2D als Bezeichnung im Funktionsnamen nötig, so zum Beispiel Funktionen für 3D Vektor analog erzeugbar (alternative ND-Vektor mit Schleifen, allerdings langsamer)
- Wir kennen aber bereits Funktionen die auf einer Variable aufgerufen werden:
  - Listen: `l.append()`, `l.sort()`, `l1 + l2`, ...
  - Strings: `s.split()`, `s.join(l)`, ...
- Schöner wäre bei Vektoren zum Beispiel `vec1.add(vec2)` oder sogar `vec1 + vec2` (allerdings addieren, nicht aneinanderhängen)

Lösung: Verwendung von Objektorientierter Programmierung mit Klassen

Objektorientierte Programmierung

- Bisheriges Programmierkonzept: imperative Programmierung
  - Programm ist eine Folge von Anweisungen
  - Anweisungen geben vor, in welcher Reihenfolge was vom Computer getan werden soll
- Neu: Objektorientierte Programmierung
  - Architektur einer Software wird an die Grundstrukturen der Wirklichkeit angepasst ($\rightarrow$ Objekte)
  - Objekte repräsentieren und abstrahieren die Wirklichkeit
  - Neue Konzepte: Klassen, Vererbung, Polymorphie, ...

$\Rightarrow$ Software ist eine Sammlung von Objekten, die miteinander agieren

Repräsentation komplexer Modelle

Ein Autohaus möchte elektronisch seine verfügbaren Autos speichern, damit für die Kunden schnell ihr passender Wagen gefunden werden kann. Vereinfacht wollen wir die Eigenschaften Idendifikationsnummer (ID), Autotyp, Baujahr, Kilometerstand und Neuwagen (NW) speichern.

```
# ID: 5, Audi A6, Baujahr 2012, 5000 km, kein Neuwagen

# Verschiedene Variablen?
id1, typ1, year1, km1, newCar1 = 5, "Audi A6", 2012, 5000, False
# Liste (bzw. Tuple)?
car1 = [5, "Audi A6", 2012, 5000, False]
# Dictionary?
car2 = {"id": 5, "modell": "Audi A6", "year": 2012,
        "km": 5000, "new": False}
```

Probleme bei jetzigem Ansatz

- Bei Speicherung als Variablen:
  - Speicherung von mehreren Modellen: marke1, marke2, ...?
  - Verwendung in Methode: Übergabe von 10 bis 20 Variablen?
  - Gefahr der Verwechslung bei vielen Variablen?
  - Neue Attribute?
- Bei Speicherung als Listen:
  - Zuordnung der einzelnen Einträge (Reihenfolge?)
  - Neue Attribute
- Bei Speicherung als Dictionary:
  - Zuordnung einzelner Einträge gegeben
  - Neue Attribute (Kommen Objekten sehr nahe)
- Allgemeine Nachteile aller Darstellungen:
  - Zugriffsbeschränkung (Passwörter sollen nicht nach außen sichtbar sein)
  - Was passiert, wenn eine Eigenschaft fehlt?

Ein bisschen Theorie (1)

- Ein Auto wird durch eine Sammlung an Attributen/Eigenschaften/Funktionen beschrieben
- Dies bezeichnen wird in Zukunft als **Objekt**: Ein Auto ist ein Objekt.
- Jedes Auto hat die selben Attribute und Eigenschaften, jedoch mit unterschiedlichen Werten!
- Funktionen, die auf einem Auto arbeiten (z. B. Preisbestimmung, Kostenrechner, ...) sollen abhängig von den gespeicherten Werteneines übergebenen Objektes sein.

Ein bisschen Theorie (2)

- Wir benötigen dazu eine Schablone für ein Auto, die angibt welche Attribute/Eigenschaften/Funktionen ein Objekt haben soll.
- Alle Objekte haben so das selbe „Aussehen“, speichern jedoch unterschiedliche Werte.

Begriffsklärung:

- **Klasse**: Schablone (Blaupause) eines für Objekte (z. B. „Auto“)
- **Instanz**: Objekt, das von einer Klasse erzeugt wurde ($\rightarrow$ „Instantiierung“, zum Beispiel ein konkretes Auto, Audi A6)
- **Attribute**: Eigenschaften eines Objekts

Syntax: Klassendefinition

```
class Car:  # Definition der Klasse Car
    # Konstruktor-'Funktion'
    def __init__(self, id: int, typ: str, year: int,
                 km: int, price: int):
        # self bezieht sich auf aktuelle Instanz
        self.id: int = id
        self.typ: str = typ
        self.year: int = year
        self.km: int = km
        self.price: int = price
        self.newCar: bool = km == 0


# Erzeugen eines neuen Autos gemäß obiger Klasse
car = Car(5, "Audi A6", 2012, 5000, 200000)
```

Aufgabe: Klassen, Konstruktoren und Attribute

Definieren Sie eine Klasse zur Speicherung von Büchern. Es sollen als Attribute jeweils die ISBN, der Autor, der Titel und der Preis über- und ausgegeben werden können:

Erstellen Sie 3 verschiedene Bücher und prüfen Sie jeweils, ob die übergeben Attribute den richtigen Wert haben!

Nutzen Sie Test Driven Development!

```
isbn1, author1, title1, price1 = "1234", "Autor", "Titel", 19.99
book1 = Book(isbn1, author1, title1, price1)
assert (book1.isbn == isbn1)
assert (book1.author == author1)
assert (book1.title == title1)
assert (book1.price == price1)
```

Lösung: Klassen, Konstruktoren und Attribute

```
class Book:
    def __init__(self, isbn: str, author: str, title: str, price: float
        self.isbn: str = isbn
        self.author: str = author
        self.title: str = title
        self.price: float = price


isbn1, author1, title1, price1 = "1234", "Autor", "Titel", 19.99
book1 = Book(isbn1, author1, title1, price1)
assert (book1.isbn == isbn1)
assert (book1.author == author1)
assert (book1.title == title1)
assert (book1.price == price1)
```

Methoden

Wie modellieren wir komplexere Zugriffe auf Objekte? Beispielsweise eine längere Autofahrt: Erhöhen des Kilometerzählers, Tankinhalt verkleinern, Standort ändern, ...

```
# Fahren von Nürnberg nach Würzburg

# Erhöhen Kilometerzähler
car.km = car.km + 100

# Tankinhalt verkleinern
car.fuel = car.fuel - 7.5

# Standort ändern
car.place = "Würzburg"
```

Methoden (2)

Methoden sind die Funktionen eines Objekts. Sie werden auch genau wie Funktionen definiert, müssen jedoch (entsprechend eingerückt) in der Klasse stehen und bekommen als erstes Argument `self` übergeben.

Das Argument `self` beinhaltet beim Aufruf der Methode die Instanz der Klasse, auf der die Methode aufgerufen wird.

```
class Car:
    # Andere Definitionen wie Konstruktur, etc.
    def drive_to(self, dist, consump, dest):
        self.km = self.km + dist
        self.fuel = self.fuel - consump
        self.place = dest


Car().drive_to(100, 7.5, "Würzburg")
```

Aufgabe: Methoden

Erweitern Sie ihre Klasse Buch um zwei neue Attribute:

- Gesamtzahl der Seite und
- die Seite, auf der sich der Leser gerade befindet

Schreiben Sie eine Methode `def lese(self, seiten: int)`, die die Seite, auf der sich der Nutzer befindet, hochzählt. Beachten Sie dabei das Limit der Gesamtzahl an Seiten!

Ist der Nutzer fertig, soll die Methode eine Nachricht ausgeben!

Lösung: Methoden

```
class Book:
    def __init__(self, isbn: str, author: str, title: str,
                 price: int, ges_seiten: int):
        # Andere Attributzuweisungen...
        self.ges_seiten: int = ges_seiten
        self.gelesen: int = 0

    def lese(self, seiten: int):
        self.gelesen = self.gelesen + seiten

        if self.gelesen > self.ges_seiten:
            self.gelesen = self.ges_seiten
            print("Sie haben das Buch komplett gelesen!")
```

Spezialmethoden

Python definiert für Klassen verschiedene spezielle Methoden, die man bei Bedarf überschreiben sollte:

- `__init__(self, *args*)`: Konstruktor, um neue Instanzen einer Klasse (mit Argumenten) zu erstellen
- `__str__(self) -> str`: Wird verwendet, um *Benutzern* des Programms eine lesbare Repräsentation eines Objektes anzuzeigen
- `__repr__(self) -> str`: Wird verwendet, um *Programmiern* eine lesbare Repräsentation eines Objektes anzuzeigen (hilfreich vor allem bei Unittests)
- `__eq__(self, other) -> bool`: Wird zur Bestimmung der Gleichheit mit einem *beliebigen* anderen Objekt `other` beim Vergleich mit `==` benutzt

Beispiel: Spezialmethoden (1)

```
class Rectangle:
    def __init__(self, x: float, y: float, width: float,
            height: float):
        self.x: float = x
        self.y: float = y
        self.width: float = width
        self.height: float = height

    def __repr__(self) -> str:
        return "Rect(({}, {}), w: {}, h: {})".format(
            self.x, self.y, self.width, self.height)

    def __str__(self) -> str:
        return "Rechteck(x: {}, y: {}, breite: {}, hoehe: {})".format(
            self.x, self.y, self.width, self.height)
```

Beispiel: Spezialmethoden (2)

```
    def __eq__(self, other) -> bool:
        return isinstance(other, Rectangle) and \
               self.x == other.x and self.y == other.y and \
               self.width == other.width and self.height == other.height


rect1: Rectangle = Rectangle(1, 2, 3, 4)
print(rect1)  # Rechteck(x: 1, y: 2, breite: 3, hoehe: 4)
print(repr(rect1))  # Rect((1, 2), w: 3, h: 4)

rect2: Rectangle = Rectangle(1, 2, 3, 4)
print(rect1 == rect2) # True
```

Aufgabe: Spezialmethoden

Implementieren Sie die Klasse **`Domino`**, die einen Dominostein repräsentiert. Es sollen folgende Methoden implementiert werden:

- `__init__(self, first: int, second: int)`: Der Konstruktor speichert die Augen auf beiden Seiten.
- `__str__(self)`: Gibt einen String im Format „`[`*`first`*`, `*`second`*`]`“ zurück
- `__repr__(self)`: Gibt einen String im Format „`Domino(`*`first`*`, `*`second`*`)`“ zurück

- `__eq__(self, other)`: Vergleicht einen Dominostein mit einen **beliebigen** anderen Objekt. Zwei Dominosteinen sind gleich, wenn jeweils die Augen (auch „gespiegelt“) übereinstimmen.

Lösung: Spezialmethoden

```
class Domino:
    def __init__(self, first: int, second: int):
        self.first: int = first
        self.second: int = second

    def __str__(self) -> str:
        return "[{}, {}]".format(self.first, self.second)

    def __repr__(self) -> str:
        return "Domino({}, {})".format(self.first, self.second)

    def __eq__(self, other) -> bool:
        return isinstance(other, Domino) and (
                (self.first == other.first and self.second == other.second) or
                (self.first == other.second and self.second == other.first)
        )
```

Wiederholung Lernziel

Lernziel

```
class GameResult:
    def __init__(self, home_team: str, away_team: str,
                 points_home: int, points_away: int):
        self.home_team: str = home_team
        self.away_team: str = away_team
        self.points_home: int = points_home
        self.points_away: int = points_away

    def announce(self) -> str:
        result_string: str
        if self.points_home > self.points_away:
            result_string = 'gewonnen'
        elif self.points_home < self.points_away:
            result_string = 'verloren'
        else:
            result_string = 'unentschieden gespielt'
        return f'{self.home_team} hat gegen {self.away_team} ' \
               f'mit {self.points_home} : {self.points_away} {result_string}.'
```