Tutorium 11

Dictionary, List-Comprehensions, OOP nochmal

Dictionary

- Eine Ansammlung aus **Keys** und dessen **Werten**
- Ordnet jedem **Key** einen **Wert** zu
- Ein **Key** muss **immutable** sein, also keine `list`, `Objects`, ...
- **Werte** können mutable sein, also eigentlich alles.

Creating a Dictionary

```
dictionary = {
    <key>: <value>,
    <key>: <value>,
    ...
    <key>: <value>
}
```

Creating a Dictionary - Beispiel

- `Key`: Modul als `str` referenziert
  - **immutable**
- `Value`: Liste aller Stundenten, mutable
  - **mutable**, wir können Stunden entfernen/hinzufügen

```
courses: dict[str, list[str]] = {
    "eidp": ["np19", "az34", "jf334"],
    "mathe": ["aw331", "pl67"],
    "sdp": ["xy111", "xz112"],
}
```

Was passiert wenn wir eine `list` als nehmen?

- `list[Any]` ist mutable, genauer nicht **hashable**
  - `hash([1, 2, 3])` wirft einen Error
  - `dict` nutzt `hash(...)` für lookups
- `dict[list[Any], Any]` wirft also einen `TypeError`, weil `list` nicht **hashable** ist
- `tuple` sind immutable, wenn dessen Elemente immutable sind
  - z.B. `(1, 2)`

Dictionary indizieren

```
print(courses["eidp"]) # ["np19", "az34", "jf334"]

courses["eidp"] += ["jk331"]

print(courses["eidp"]) # ["np19", "az34", "jf334"]

courses["mathe_2"] += ["jk331"] # KeyError

courses["mathe_2"] = ["jk331"] # fügt "mathe_2" hinzu mit dem Wert ["jk331"]

if "logik" not in courses:
    print("logik is not in courses!")
    courses["logik"] = []
    print("now it is!")
```

Was kann man als Value verwenden?

Was kann man als Value verwenden?

Alles!

```
ops: dict[str, Callable] = {
    '+': lambda x, y: x + y,
    '-': lambda x, y: x - y,
    '*': lambda x, y: x * y,
    '/': lambda x, y: x / y,
}

print(ops['+'](3, 1)) # 4
```

Dictionary iterieren

- mit der `items()` Methode bekommt man jeden `Key` mit `Value`

```
for courses, students in courses.items():
    print(f"{courses}: {students}")
```

List-Comprehension

- hattet ihr noch nicht in der Vorlesung
- Viel zu Viele nutzen es schon, und ich will keine 0 Punkte geben
- Syntax-Sugar für das erstellen von Listen basierend auf anderen Listen

```python
even_numbers = [number for number in range(101) if number % 2 == 0] # [0, 2, ..., 100]

# [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
all_permutations = [(a, b)  for a in range(3)
                            for b in range(3)]

# quiet fast actually (for python)
pythagorean_triples = [(a, b, c)    for a in range(101)
                                    for b in range(101)
                                    for c in range(101)
                                    if a ** 2 + b ** 2 == c ** 2]
```

List-Comprehension

```python
# ew no syntax sugar
all_students: set[str] = set()
for _, students in courses.items():
    for student in students:
        all_students.add(student)

# syntax sugar!
print({student  for students in courses.values()
                for student in students})

# flattening stuff
matrix = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
print([num for row in matrix for num in row]) # [1, 0, 0, 0, 1, 0, 0, 0, 1]
```

Übertreibt es aber nicht!

- List-Comprehensions sind zum *erstellen* von Listen.
- List-Comprehensions sollten **nichts** *machen*

Also **kein** side effects oder Funktionsaufrufe!

```
x = [1, 2, 3, 4]
[x.append(num) for num in range(5, 11)] # really bad
```

```
def f(x: float) -> float:
    return x ** 2 + 3 * x + 1

[f(x) for x in range(101)] # bad
```

OOP - Funktion oder Methode?

```python
import math
from dataclasses import dataclass


@dataclass
class Position:
    x: float
    y: float

    def distance(self, other: Position) -> float:
        return math.sqrt((other.x - self.x) ** 2 + (other.y - self.y) ** 2)

def distance_of(a: Position, b: Position) -> float:
    return math.sqrt((b.x - a.x) ** 2 + (b.y - a.y) ** 2)
```

OOP - Funktion oder Methode!

- `distance(self, other: Position)` ist eine Methode.
  - Gehört zu einer Klasse und hat `self` als Parameter
- `distance_of(a: Position, b: Position)` ist eine Funktion.
  - unabhängig (normalerweise kein **state**)

Was ist ein State (Zustand)?

```
class GameState(Enum):
    RUNNING = auto()
    PAUSED = auto()
    ENDED = auto()

@dataclass
class Game:
    state: GameState
```

- Unser `Game`-Objekt hat einen Zustand der sich ändern kann
- Unser `Game` kann pausiert, beendet oder am Laufen sein
- Dieser Zustand kann sich ändern

Was ist ein State (Zustand)?

```python
@dataclass
class Position:
    x: float
    y: float
```

Ebenso sind `x` und `y` Zustände von `Position`, wenn auch nicht ganz so offensichtlich.

- Beschreiben das Objekt
- Können sich ändern

Was ist ein State (Zustand)?

Wie sieht es mit `distance_of(...)` aus?

```
def distance_of(a: Position, b: Position) -> float:
    return math.sqrt((b.x - a.x) ** 2 + (b.y - a.y) ** 2)
```

- Verhält sich immer gleich
  - also hat keinen **State**
- ändert keine **States**
  - manchmal passiert das leider, ist aber ein schlechter Stil!

```
def move_to(pos: Position, x: float, y: float):
    pos.x = x
    pos.y = y
```

  - Guter Stil ist es eigentlich immer die Parameter in Ruhe zu lassen!

Funktionen mit State

- Ihr kriegt 0 Punkte für die gesamte Abgabe.

```
can_execute = True

def function(x: int) -> int:
    global can_execute
    if can_execute:
        can_execute = False
        return x + 1
    return x

def can_execute_again():
    global can_execute
    can_execute = True
```

- Ich meine das ernst mit den 0 Punkten

Dazu zählt auch sowas!

```
def main():
    session = Session()

    def do_something():
        session.do()
    # ...
```

Private, Getter, Setter

Bei `@dataclass`:

- `InitVar` verwenden wenn im Klassenrumpf deklariert
- Sollen mit `_<variable_name>` benannt werden
- `__post_init__(self, <variable>)` muss definiert werden und `__<variable_name>` erstellen!

```python
@dataclass
class Point:
    _x: InitVar[float]
    _y: InitVar[float]

    def __post_init__(self, x, y):
        self.__x = x
        self.__y = y
```

Private, Getter, Setter

- Geht auch ohne `InitVar`
  - Keine Parameter für `__post_init__`
  - Also auch keine Parameter beim erstellen

```
@dataclass
class Point:
    def __post_init__(self):
        self.__x = 0
        self.__y = 0
```

Private, Getter, Setter

- `x` und `y` sind nicht mehr von außen sichtbar

```
p = Point()
print(p.__x) # Error
```

- außer man erstellt einen *Getter* (`@property`)

```
@dataclass
class Point:
    # ...

    @property
    def x(self) -> float:
        return self.__x

print(Point(3, 1).x) # Prints 3
```

Private, Getter, Setter

Genauso kann man auch private Attribute setzbar machen:

```python
@dataclass
class Point:
    # ...

@x.setter
def x(self, new_value: float):
    self.__x = new_value

p = Point(3, 1)
print(p.x) # 3
p.x = 4
print(p.x) # 4
```

Fragen?