Tutorium 11 - 2025-01-09

Vorrechnen 10, Generatoren, OOP, Type-Annotation, Blatt 11

Vorrechnen Blatt 10

Generator

Was ist das?

Generator

- Generatoren sind Funktionen, die sich wie Iteratoren verhalten
- Bei Iteratoren könnt ihr euch entscheiden wann ihr das nächste Element anschauen wollt
- Genauso könnt ihr bei Generatoren entscheiden wann ihr das nächste Element berechnen wollt

Generator - Beispiel

```python
def my_range(start: int, end: int) -> list[int]:
    num = start
    nums = [num]
    while (num := num + 1) < end:
        nums.append(num)
    return nums
```

Generator - Beispiel

```
for i in my_range(0, 101):
    print(i) # 0 .. 100
```

Generator - Beispiel

```python
from typing import Iterator

def my_range(start: int, end: int) -> Iterator[int]:
    num = start
    yield num # start ist inklusive
    while (num := num + 1) < end:
        yield num
    raise StopIteration
```

- `raise StopIteration` wird häufig genutzt um das Ende eines Iterators zu signalisieren
- kann jetzt genauso wie unsere erste `my_range` benutzt werden

Unendlicher Generator goes brrr

```python
from typing import Iterator

def my_range(start: int) -> Iterator[int]:
    num = start
    yield num # start ist inklusive
    while True:
        yield (num := num + 1)
    raise StopIteration
```

OOP

Überschreiben von Methoden und Polymorphie

Override-Dekorator

- ist in `typing`
- Wird über Methoden geschrieben, die überschrieben werden
- Pylance zeigt einen Fehler an, wenn die überschriebene Methode in keiner Oberklasse gefunden wird
- Hilft Fehler vorzubeugen - falsche Signatur, Parameter, ...

Override-Dekorator - Beispiel

```python
from typing import override
from dataclasses import dataclass


@dataclass
class GameObject:

    def on_collision(self, other: 'GameObject'):
        pass

class StaticObject(GameObject):

    @override
    def on_collisoin(self, other: 'GameObject'): # Pylance-Error
        self.alive = False
```

Override-Dekorator - Beispiel

```python
from typing import override
from dataclasses import dataclass


@dataclass
class GameObject:

    def on_collision(self, other: 'GameObject'):
        pass

class StaticObject(GameObject):

    @override
    def on_collision(self): # Pylance-Error
        self.alive = False
```

Warum muss man überhaupt Methoden überschreiben?

Vererbung - Animal

```python
@dataclass
class Animal:
    age: int
    weight: int
    name: str
```

```python
@dataclass
class Cat(Animal):
    pass

@dataclass
class Dog(Animal):
    pass
```

Bisher in der Vorlesung

```
def make_noise(animal: Animal):
    match animal:
        case Dog():
            print("Woof")
        case Cat():
            print("Meow")
        case _:
            pass
```

Bisher in der Vorlesung

```
def make_noise(animal: Animal):
    match animal:
        case Dog():
            print("Woof")
        case Cat():
            print("Meow")
        case _:
            pass
```

das ist kein gutes Design, warum?

was ist wenn wir jetzt eine neue Klasse `Mouse` erstellen wollen.

```
@dataclass
class Mouse(Animal):
    pass
```

jetzt müssen wir `Mouse` zu `make_noise` hinzufügen

```python
def make_noise(animal: Animal):
    match animal:
        case Dog():
            print("Woof")
        case Cat():
            print("Meow")
        case Mouse():
            print("Peep")
        case _:
            pass
```

das kann ziemlich nervig werden. Vor allem wenn wir größere Projekte haben

Polymorphism

> Die Polymorphie der objektorientierten Programmierung ist eine Eigenschaft, die immer im Zusammenhang mit Vererbung und Schnittstellen (Interfaces) auftritt. Eine Methode ist polymorph, wenn sie in verschiedenen Klassen die gleiche Signatur hat, jedoch erneut implementiert ist.

```python
@dataclass
class Animal:
    age: int
    weight: float
    name: str

    def make_noise(self) -> None:
        pass

@dataclass
class Cat(Animal):

    def make_noise(self) -> None:
        print("Meow")

@dataclass
class Dog(Animal):

    def make_noise(self) -> None:
        print("Woof")
```

Geht das schöner in Python 3.12?

Ja, mit `override` sagt sogar pylance bescheid wenn wir eine polymorphe Methode falsch überschrieben haben!

```
form dataclasses import dataclass
from typing import override

@dataclass
class Cat(Animal):

    @override
    def make_noise(self) -> None:
        print("Meow")
```

Unsere neue polymorphe Methode benutzen

```python
@dataclass
class Zoo:
    animals: list[Animal] = []


zoo = Zoo(animals=[
            Cat(age=6, weight=4.5, name="Milow"),
            Cat(age=7, weight=5.0, name="Blacky"),
            Dog(age=12, weight=40.3, name="Bernd")
        ])

for animal in zoo.animals:
    animal.make_noise() # Meow
                        # Meow
                        # Woof
```

Oberklasse Animal

- Was passiert wenn man `Animal` euzeugt?
  - Wir haben ein Objekt was eigentlich nicht erzeugt werden sollte
  - Es ist eine Art Schablone für alle Klassen von Typ `Animal`

Können wir die Instanziierung verhindern? Ja!

Abstrakte Klassen

- abstrakte Klassen sind kein konkreter Bauplan für ein Objekt
- stattdessen eine Schablone für weiter Baupläne
- `Animal` ist so einen Schablone für alle Tiere

Abstrakte Klassen - in anderen Programmiersprachen

Beispiel Java mit `abstract`

```java
abstract class Animal {
    private int age;
    private float weight;
    private String name;

    abstract void make_noise();
}

class Cat extends Animal {
    @override
    void make_noise() {
        System.out.println("Meow")
    }
}
```

Abstrakte Klassen - Python

- `ABC` aus dem Modul `abc`.
- Ja sie mussten `abstract` wirklich abkürzen mit `ABC` (Abstract Base Class)
- für abstrakte Methoden gibt es die Annotation `@abstractmethod` im selben Modul

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    age: int
    weight: float
    name: str

    @abstractmethod
    def make_noise():
        pass
```

Type annotations

(Wiederholung)

Type annotations - Was ist das?

Type annotations - Was ist das?

- Jedes **Objekt** lässt sich mindestens einem **Typ** zuordnen
  - Objekte im mathematischen Sinne wie z.B. Variablen, Funktionen, ...
- Dieser **schränkt** den Wertebereich ein
  - z.B. ist eine Variable `x` von Typ `int` eine Ganzzahl
  - ähnlich zur mathematischen Schreibweise $x \in \mathbb{Z}$
- In der Informatik nennt man das **Typisierung**
  - Es gibt verschiedene Arten der Typisierung

Type annotations - Typisierung

- **dynamische Typisierung** überprüft die gegebenen Typen zur **Laufzeit**
  - also erst wenn das Programm *läuft*
- **statische Typisierung** überprüft die gegebenen Typen zur **Übersetzungszeit**
  - also während wir den Quellcode übersetzen

Type annotations - Typisierung

- **dynamische Typisierung** überprüft die gegebenen Typen zur **Laufzeit**
  - also erst wenn das Programm *läuft*
- **statische Typisierung** überprüft die gegebenen Typen zur **Übersetzungszeit**
  - also während wir den Quellcode übersetzen

Was ist nun Python?

Was ist nun Python?

- **dynamisch typisiert**
  - wir müssen unsere `.py` Datei ausführen bevor wir wissen ob alles korrekt ist
- **Pylance** ist ein eigenes Programm
  - es soll beim Schreiben bereits **Typverletzungen** erkennen
  - **unvollständige** Typüberprüfung, es soll nur den Entwickler unterstützen

Variabeln Typannotieren

- `variable_name: <Type> = ...`
- Beispiele:

```
x: int = 3
y: int = 5
string: str = "Hello World!"

# aber auch eigene Objekte (OOP)
point: Point = Point(3, 1)
```

- diese Annotation ist für uns **optional**

Funktionen Typannotieren

- `def func_name(param1: <Type>, param2: <Type>, ...) -> <Type>`
- Beispiele:

```python
def add(x: int, y: int) -> int:
    return x + y

def div(x: float, y: float) -> Optional[float]:
    if y == 0.0:
        return None
    return x / y
```

- diese Annotation ist **verpflichtend** und muss so vollständig wie möglich sein

Klassen Typannotieren

- ```
  class ClassName:
      attribute_name1: <Type>
      attribute_name2: <Type>
      ...
  ```
- Beispiel:

  ```
  @dataclass
  class Point:
      x: int
      y: int
  ```
- diese Annotation ist **verpflichtend** und muss so vollständig wie möglich sein
  ```

Methoden Typannotieren

- `def method_name(self, param1: <Type>, ...) -> <Type>`
- Beispiel:

```
class Point:
    x: int
    y: int

    def distance_from(self, other: 'Point') -> float:
        return math.sqrt((other.x - self.x) ** 2 + (other.y - self.y) ** 2)
```

- `self` muss **nicht** Typannotiert werden, kann aber
- `other` hingegen schon, wegen Python muss in der Klasse mit `'` annotiert werden
- diese Annotation ist **verpflichtend**

Datentypen von Datentypen

- Manche Datentypen bauen sich aus anderen Datentypen auf
- z.B. `list` ist eine Liste von Elementen mit einem Typ
- hierfür verwenden wir `[]` um den Datentyp in `list` zu annotieren

```
def sum(xs: list[int]) -> int:
    total: int = 0
    for x in xs:
        total += x
    return total
```

- hierbei ist es wichtig so genau wie möglich zu annotieren!
- diese Annotation ist **verpflichtend**

Häufige Fehler mit verschachtelten Typen

Fehlerquelle - `tuple[...]`

- Tuple haben eine feste größe
- Tuple sind endlich
- Tuple können Elemente mit unterschiedlichen Typen haben
- Die Datentypen der Elemente werden mit einem `,` in `[]` getrennt
- Beispiel:

```
tup: tuple[int, int, float, str] = (1, 2, 3.0, "hello world")
```

- Diese Annotation ist **verpflichtend**

Fehlerquelle - `dict[...]`

- Dictionary haben genau zwei zu definierende Typen
  - **Key**
  - **Value**
- Beispiel:

```
number_dictionary: dict[int, str] = {
    0: "zero",
    1: "one",
    2: "two",
}
```

- Diese Annotation ist **verpflichtend**
- Diese kann weiter geschachtelt werden durch z.B. `list` als `Value`:
  - `dict[int, list[str]]`

Fehlerquelle - Typvariabeln (generische Typen)

- manchmal wollen wir nicht genau wissen welchen Datentypen wir haben
- dieser wird dann implizit von Python erkannt
- wir stellen damit sicher dass eine Typvariable **beliebig** aber **fest** ist
- Beispiel:

```
def add[T](x: T, y: T) -> T:
    return x + y
```

- `T` kann nur ein Datentyp sein, also muss `type(x) == type(y)` gelten
- **außer** wir schrenken `T` mit `|` ein: `T: (int | str)` damit müssen x und y nicht den gleichen Datentypen haben
- `T` lässt sich weiter einschränken durch `T: (int, str)`, hierbei ist `T` entweder ein `int` oder (exklusiv) `str`

Fehlerquelle - Was ist TypeVar?

- `TypeVar` ist aus früheren Python-Versionen
- Typvariablen wurden vor der Python 3.12 so definiert:

```
T = TypeVar('T')
```

- sieht dumm aus, ist es auch, benutzt es nicht!

Fragen zu Typannotationen?

Blatt 11