Tutorium 12 - 19.01.2024

Musterlösung 11 - Wiederholung Types - Functions!

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

Was ist nun Python?

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?

- **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, soll nur den Programmierer 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

```python
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?

Funktionale Programmierung

Funktionale Programmierung - was ist das?

-