Tutorium 13 - 26.01.2024

Orga - Wiederholung Types - Functions!

Orga

Wegen fehlendem Tutorium am 19.01.

- Jeder kriegt die 6 Punkte für Anwesenheit
  - Auf Blatt 13 als Extrapunkte unter *Anmerkungen*
- Sorry fürs nicht beantworten von manchen Nachrichten
- Falls ihr glaubt ihr bekommt knapp nicht genug Punkte schreibt mich einfach an, man wird schon noch irgendwo Punkte finden

Syntax-Fehler

- Für **Syntax-Fehler** habe ich im allgemeinen **0 Punkte** in der jeweiligen Datei vergeben
  - Das euer Programm ausführbar ist sollte das mindeste sein!
  - Ihr sollt euer Programm sowieso selbständig testen und ich geh mal davon aus das ist nicht passiert wenn sich die Datei nichtmal ausführen lässt
- Zeitdruck kann ich voll nachvollziehen

Nachträgliches ausbessern

- Ihr verbessert euren SyntaxFehler (eure Python-Datei ist ausführbar)
- Ihr schickt mir eine `.zip` oder eine `.tar.gz` mit dem verbesserten Code an nils@narl.io
  - verbessert nichts anderes!
  - Schreibt kurz in die Mail welches Blatt + Aufgabe + Kürzel
- Ich korrigiere eure Abgabe nachträglich und ihr bekommt zumindest mehr als 0 Punkte
- Bitte nur wenn ihr wirklich die Punkte braucht und habt etwas Geduld mit der Korrektur

Allgemeines

- biete euch Übungen passend zur Klausur
  - kein genaues Datum, aber vor dem 09.02
- Klausur ist *wahrscheinlich* am 19.02.
- Short-Link zu der Übung https://s.narl.io/s/eidp-ub
  - aktuell noch nicht online

Link: https://s.narl.io/s/eidp-ub

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

```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?

- Funktionen sind äquivalent zu Datenobjekten
- anonyme Funktionen aka Lambdas
- Closures
- Programmablauf mit Verkettung und Komposition von Funktionen

Funktionen sind Datenobjekte

- Jede Funktion hat den Datentyp `Callable`
- Wir können Funktionen wie alle anderen Objekte variabeln zuweisen

```
def add(a: int, b: int) -> int:
    return a + b

add_but_variable = add

print(add_but_variable(3, 2)) # 5
```

Anonyme Funktionen - `lambda`

- Mit dem `lambda` Keyword lassen sich anonyme Funktionen definieren ohne `def`
- Bietet sich vor allem an für kleine Funktionen und Kompositionen von Funktionen

```
print(reduce(lambda x, y: x + y, [1, 2, 3, 4])) # 10
```

- hat als Datentyp auch `Callable`

```
add: Callable[[int, int], int] = lambda x, y: x + y
```

Closures

- Verkettete Funktionen, bei denen die Variabeln aus vorherigen benutzt werden können

```python
def poly(x: float) -> Callable[[float, float], Callable[[float], float]]:
    return lambda a, b: lambda c: a * x ** 2 + b * x + c

print(poly(3)(2, 3)(5)) # 2 * 3 ** 2 + 3 * 3 + 5 = 32
```

- kein wirklich schönes Beispiel, ein besseres ist `compose` für Kompositionen

Komposition

- Verketten von Funktionen

```python
def compose[T](*funcs: Callable[[T], T]) -> Callable[[T], T]:
    return fold(lambda f, g: lambda n: f(g(n)), funcs)

f: Callable[[int], int] = lambda n: n + 42
g: Callable[[int], int] = lambda n: n ** 2
h: Callable[[int], int] = lambda n: n - 3

print(compose(f, g, h)(0))
```

Higher-Order Functions

- nehmen eine oder mehrere `Callable` als Argument
- geben ein `Callable` zurück

Higher-Order-Function - `map`

- **Wendet ein `Callable` auf jedes Element in einem `Iterable` an**

```python
def map[T, R](func: Callable[[T], R], xs: Iterable[T]) -> Iterable[R]:
    return [func(x) for x in xs]

numeric_list = list(map(lambda e: int(e), ['1', '2', '3']))
print(numeric_list) # [1, 2, 3]
```

Higher-Order-Function - `filter`

- `filter` verarbeitet Datenstrukturen anhand eines Prädikats (`Callable`)
- behält nur Elemente die das Prädikat erfüllen

```python
def filter[T](predicate: Callable[[T], bool], xs: Iterable[T]) -> Iterable[T]:
    return [x for x in xs if predicate(x)]

predicate: Callable[[int | None] bool] = lambda e: e is not None
none_free_list: list[int] = list(filter(predicate, [1, 2, 3, None, 5, 6]))
print(none_free_list) # [1, 2, 3, 5, 6] - kein None
```

Higher-Order-Function - `fold`

- Kombiniert Elemente einer Datenstruktur

```python
def fold[T](func: Callable[[T, T], T], xs: Iterable[T]) -> T:
    it: Iterator[T] = iter(xs)
    value: T | None = None
    for x in it:
        match value:
            case None:
                value = x
            case _:
                value = func(value, x)
    if not value:
        raise TypeError("can't fold empty list")
    return value

sum: Callable[[Iterable[int]], int] = lambda xs: fold(lambda x, y: x + y, xs)
print(sum([1, 2, 3, 4])) # 10
```

keine Higher-Order-Function - `flatten`

- Nimmt mehrdimensionale Listen und macht eine Liste draus

```python
def flatten(xs: Iterable[Any]) -> Iterable[Any]:
    new_list = []
    for s in xs:
        if isinstance(s, Iterable):
            new_list += flatten(s)
        else:
            new_list.append(s)
    return new_list

flattened = list(flatten([[1, 2, 3], 4, [[5, 6], 7, [8, 9]]]))
print(flattened) # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- nimmt weder `Callable` als Argumente
- gibt kein `Callable` zurück
- ist keine Higher-Order-Function

Fragen zur funktionalen Programmierung?

Weitere allgemeine Fragen?