

# **Tutorium 12 - 19.01.2024**

**Musterlösung 11 - Wiederholung Types - Functions!**

# Musterlösung - Exercise 11

## Aufgabe 11.1 - Generatoren;

`generators.py` [10p]

## Aufgabe 11.1 a - collatz; [2.5p]

Es seien  $i \in \mathbb{N}_0$  und  $n \in \mathbb{N}$ , so ist die Collatz-Folge definiert als

$$c_0 = n$$
$$c_{i+1} = \begin{cases} \frac{c_i}{2}, & c_i \bmod 2 = 0 \\ 3 \cdot c_i + 1, & c_i \bmod 2 = 1 \end{cases}$$

Dabei gilt  $c_i = 1$  als Abbruchbedingung des Generators

## Aufgabe 11.1 a - collatz; [2.5p]

```
def collatz(n: int) -> Generator[int, None, None]:  
    if n < 1:  
        return  
    while n > 1:  
        yield n  
        if n % 2 == 0:  
            n = n // 2  
        else:  
            n = 3 * n + 1  
    yield n
```

## Aufgabe 11.1 b - random; [2.5p]

## Aufgabe 11.1 b - random; [2.5p]

```
def random(seed: int, a: int, b: int, m: int) -> Iterator[int]:  
    yi = seed  
    while True:  
        yield yi  
        yi = (a * yi + b) % m
```

## Aufgabe 11.1 c - chunks; [2.5p]



## Aufgabe 11.1 c - chunks; [2.5p]

```
def chunks[T](iter: Iterator[T], n: int) -> Iterator[list[T]]:
    while True:
        xs = []
        try:
            for _ in range(n):
                xs.append(next(iter))
            yield xs
        except StopIteration:
            if xs:
                yield xs
            break
```

## Aufgabe 11.1 d - flatten; [2.5p]

## Aufgabe 11.1 d - flatten; [2.5p]

```
def flatten[T](iters: Iterator[list[T]]) -> Iterator[T]:  
    for iter in iters:  
        yield from iter
```

# Aufgabe 11.2 - Graphen; `graphs.py` [10p]

## Typaliase als Hilfestellung

```
type GDict[T] = dict[T, set[T]]  
type Graph[T] = GDict[T]
```

## Aufgabe 11.2 a - is\_graph; [2.5p]

## Aufgabe 11.2 a - is\_graph; [2.5p]

```
def is_graph(d: GDict[Any]) -> bool:
    for vals in d.values():
        for val in vals:
            if val not in d.keys():
                return False
    return True
```

## Aufgabe 11.2 b - to\_graph; [2.5p]

## Aufgabe 11.2 b - to\_graph; [2.5p]

```
def to_graph[T](d: GDict[T]) -> Graph[T]:  
    res = dict()  
    for k, vals in d.items():  
        for val in vals:  
            if val not in d:  
                res[val] = set()  
        res[k] = vals  
    return res
```



## Aufgabe 11.2 c - nodes, edges; [2.5p]

## Aufgabe 11.2 c - nodes, edges; [2.5p]

```
def edges[T](graph: Graph[T]) -> Iterator[tuple[T, T]]:
    for key, value in graph.items():
        for v in value:
            yield (key, v)
```

```
def nodes[T](graph: Graph[T]) -> Iterator[T]:
    yield from graph.keys()
```

## Aufgabe 11.2 d - invert\_graph; [2.5p]

## Aufgabe 11.2 d - invert\_graph; [2.5p]

```
def invert_graph[T](graph: Graph[T]) -> Graph[T]:  
    res = dict()  
    for n in nodes(graph):  
        res[n] = set()  
    for a, b in edges(graph):  
        res[b].add(a)  
    return res
```

## Aufgabe 11.2 e - has\_cycle; [0p]

## Aufgabe 11.2 e - has\_cycle; [0p]

```
def find_cycle[T](graph: Graph[T], start: T, visited: set[T]) -> bool:
    assert start in graph
    if start in visited:
        return True
    for value in graph[start]:
        if find_cycle(graph, value, visited | {start}):
            return True
    return False

def has_cycle(graph: Graph[Any]) -> bool:
    return any(find_cycle(graph, node, set()) for node in graph)
```

## Aufgabe 11.3 - Erfahrungen `NOTES.md`; `[0p]`

Tragt eure Stunden ein!

# 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**

# Variablen 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:

```
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 gröÙe
- 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 - Typvariablen (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 variablen 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 Variablen aus vorherigen benutzt werden können

```
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

- Verketteten von Funktionen

```
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**

```
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

```
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: bool(e)  
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

```
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

```
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?