Tutorium 12 - 2025-01-16

Datenklassen, Funktionale Programmierung, Blatt 12

Datenklassen - `field( .. )`

- Wird benutzt um zusätziche Funktionalität in Datenklassen für Attribute zu ermöglichen:
  - `init=..`: ob das Attribut als Parameter in der `__init__` (Konstruktor) auftauchen soll
  - `default=..`: ob das Attribut einen Standardwert bekommen soll (nur primitive Datentypen)
  - `default_factory=..`: Falls es ein Objekt ist brauchen wir eine Factory (z.B. Konstruktor) welche das Objekt erstellt
  - `repr=..`: Ob das Attribut in der `__repr__` auftauchen soll

Datenklassen - Beispiel `field( .. )`

```python
@dataclass
class Person:
    __name: str = field(init=False, default="")
    """das Feld taucht nicht im Konstruktor auf und wird mit "" initialisiert
    """
    __my_hobbies: list[str] = field(init=False, default_factory=list)
    """das Feld lässt sich nur mit einer Factory erstellen
    """
    __secrets: list[str] = field(init=False, default_factory=list, repr=False)
    """das Feld taucht nicht in der `__repr__` Dunder-Methode auf
    """

me = Person()
print(me) # Person(_Person__name='', _Person__my_hobbies=[])
```

Datenklassen - `InitVar`

- `InitVar` ermöglicht einen Platzhalter zu erstellen
  - speichert nichts ab
  - wird aber als Parameter im Konstruktor übergeben

```python
@dataclass
class Time:
    hours: InitVar[int]
    minutes: InitVar[int]
    __time: int = field(init=False)

my_time = Time(hours=4, minutes=20)
my_time = Time(4, 20)
```

Datenklassen - `InitVar` und `__post_init__`

- Um `InitVar` zu verwenden brauchen wir die `__post_init__`
  - die `__post_init__` bekommt die `InitVar` übergeben
  - dann können wir die eigentlichen Attribute initialisieren

```python
@dataclass
class Time:
    hours: InitVar[int]
    minutes: InitVar[int]
    __time: int = field(init=False)

    def __post_init__(self, hours: int, minutes: int):
        self__time = hours * 60 + minutes
```

Was ist `property`?

- Syntax-Sugar,

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
```

Higher-Order Functions

- Eine Funktion muss eine der Eigenschaften haben:
  - nimmt eine oder mehrere `Callable` als Argument
  - gibt 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: not e is 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

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

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

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))
```

Fragen zur funktionalen Programmierung?

Blatt 12