



Python für ReMail

Eine Präsentation von uns

Inhaltsverzeichnis

- Konventionen
- Tipps und Tricks
- Decorator
 - Dataclass
 - Property
 - classmethod
- Generator
- Abstrakte Klassen

Python für ReMail

Eine Präsentation von uns

Einrückungen

- 4 Leerzeichen als Einrückung
- Parameter von Funktionen gruppieren
- Typen der Parameter dazu
- Kommentarblock oben drüber

```
# Kommentar
def langer_name(
    varA:str, varB:str,
    var1:int, var2:int) -> bool:
    print("Hello world")
    return True
```

If-Statements

-Operatoren nach Zeilenumbruch
-Lösung: Kommentar einfügen

```
if (argument1  
    and argument2):  
    # Kommentar  
    print("Hello World!")
```

Lesbarkeit mathematische Funktionen

- unnötige Leerstellen vermeiden
- nach der Rechenhierarchie trennen

```
res = (a+b) * (c+d)
```

Variablenbenennung

- packages: klein geschrieben, kurze Namen
- Gruppieren der Variablen durch z.B. Prefixe
- private Objects: _ vor Namen
- functions, variables: klein geschrieben, _ als Trennung
- Constant: alles groß
- class: erster Buchstabe von Wörtern jeweils groß

```
from package1 import MyClass
from dataclasses import dataclass

_private_variable = 10
bread_variable_one = False
bread_variable_two = False
def function():
    print("Hello world!")
CONSTANT = 55

@dataclass
class KlassenName():
    name: str
```

Format Strings

Entweder

```
txt = "For {price} dollars{end}"  
print(txt.format(price = 49.1, end = "!"))
```

Oder als f String

```
price = 49.1  
end = "!"  
print(f"For {price} dollars{end}")
```


Format Strings

Verschiedene Formate möglich:

- `{price:.3f}` gibt 3 Nachkommastellen an
- `{name:>10}` Feld muss 10 Zeichen lang sein, sonst wird von rechts mit Leerzeichen aufgefüllt
- `{number:06}` Feld muss 6 Zeichen lang sein, sonst wird mit führenden 0en aufgefüllt
- `{word: _<10}` Feld muss 10 Zeichen lang sein, sonst wird von links mit _ aufgefüllt

Enumerate()

Anstelle von

```
brote = ["Vollkorn", "Weizen", "Sauerteig", "Hafer"]  
for index in range(len(brote)):  
    print(f"Brotsorte {brote[index]} ist an Index: {index}")
```

besser

```
brote = ["Vollkorn", "Weizen", "Sauerteig", "Hafer"]  
for index, sorte in enumerate(brote):  
    print(f"Brotsorte {sorte} ist an Index: {index}")
```

Enumerate

- Gibt einen Iterator von Tupeln zurück (Index, Element)

List Comprehension

Anstelle von

```
zero_to_ten = [0,1,2,3,4,5,6,7,8,9,10]
iterative = []

for num in zero_to_ten:
    if num <= 5:
        iterative.append(num*num)
```

besser

```
zero_to_ten = [0,1,2,3,4,5,6,7,8,9,10]
l_comprehension = [num*num for num in zero_to_ten if num <= 5]
```

List Comprehension

```
zero_to_ten = [0,1,2,3,4,5,6,7,8,9,10]  
l_comprehension = [num*num for num in zero_to_ten if num <= 5]
```

Allgemein

```
l_comprehension = [expression for item in iterable if condition == True]
```

Sets

```
set1 = set([1,2,3,1,1,5])
print(set1) # {1, 2, 3, 5}

string = "keine doppelten Elemente"
set2 = set(string)
print(set2) # {'d', 'm', 'k', 'p', 'l', 'o', 'E', 'e', ' ', 't', 'n', 'i'}

set3 = set()
print(bool(set3)) # False
print(bool(set1)) # True

print(1 in set1) # True
print(10 in set1) # False

set4 = set([1,6,7])
print(set1 | set4) # {1, 2, 3, 5, 6, 7}
print(set1.union(set4)) # {1, 2, 3, 5, 6, 7}
```

Docstrings

```
def funktion():  
    """beschreibung hier"""  
    print("hello world")
```

```
function) def funktion() -> None  
_____  
beschreibung hier
```

Beschreibung wird sichtbar, wenn man mit dem Cursor über die Funktion fährt

Else nach Schleifen

```
example = []  
  
for num in range(1,10):  
    if num <= 5:  
        example.append(num)  
    else:  
        example.append("end")  
  
print(example)
```

[1, 2, 3, 4, 5, 'end']

Else nach Schleifen

```
example = []  
  
for num in range(6,10):  
    if num <= 5:  
        example.append(num)  
else:  
    example.append("end")  
  
print(example)
```

['end']

Else nach Schleifen

```
example = []  
  
for num in range(1,10):  
    break  
else:  
    example.append("end")  
  
print(example)
```

[]

Itertools

```
from itertools import accumulate, chain, compress

res_acc = list(accumulate([1,2,3,4,5]))
print(res_acc)

resChain = list(chain("Halli", "hallo"))
print(res_chain)

res_comp = list(compress("HALLO", [1,0,0,1,1]))
print(res_comp)
```

[1, 3, 6, 10, 15]

['H', 'a', 'l', 'l', 'i', 'h', 'a', 'l', 'l', 'o']

['H', 'L', 'O']

accumulate:

-> [1, 1+2, 1+2+3, 1+2+3+4, 1+2+3+4+5]

compress:

-> Position mit 1 bleibt, 0 fällt weg

<https://docs.python.org/3/library/itertools.html>

Decorator

- Funktionen die eine Funktion übergeben bekommen und modifiziert zurückgeben
- Nicht beschränkt auf Funktionen (z.B. Decorator für Klassen)
- Werden mit @ über das modifizierte Objekt geschrieben

Eigene Decorator

```
@my_decorator  
def funct(i):  
    print("Ich bin die Funktion",i)
```

```
print("-----")  
funct(10)
```

```
-----  
Ich bin die Funktion 10
```

```
def my_decorator(func):  
    return func
```

Eigene Decorator

```
@my_decorator  
def funct(i):  
    print("Ich bin die Funktion",i)
```

```
print("-----")  
funct(10)
```

```
-----  
Ich bin die Funktion 10
```

```
def my_decorator(func):
```

```
    def inner(*args,**kwargs):  
        func(*args,**kwargs)  
    return inner
```

Eigene Decorator

```
@my_decorator  
def funct(i):  
    print("Ich bin die Funktion",i)
```

```
print("-----")  
funct(10)
```

```
-----  
vorher  
Ich bin die Funktion 10  
nachher
```

```
def my_decorator(func):
```

```
    def inner(*args,**kwargs):  
        print("vorher")  
        func(*args,**kwargs)  
        print("nachher")  
    return inner
```

Eigene Decorator

```
@my_decorator  
def funct(i):  
    print("Ich bin die Funktion",i)
```

```
print("-----")  
funct(10)
```

Decorator

vorher

Ich bin die Funktion 10

nachher

```
def my_decorator(func):  
    # Ausführung beim Modulimport  
    print("Decorator")  
    def inner(*args,**kwargs):  
        print("vorher")  
        func(*args,**kwargs)  
        print("nachher")  
    return inner
```


Eigene Decorator

```
@my_decorator_with_arguments(1)
def funct(i):
    print("Ich bin die Funktion",i)
```

```
print("-----")
funct(10)
```

Decorator

vorher 1
Ich bin die Funktion 10
nachher

```
def my_decorator_with_arguments(i):
```

```
    def my_decorator(func):
        # Ausführung beim Modulimport
        print("Decorator")
        def inner(*args,**kwargs):
            print("vorher",i)
            func(*args,**kwargs)
            print("nachher")
        return inner
    return my_decorator
```

Eigene Decorator

```
@my_decorator_with_arguments(1)
def funct(i):
    print("Ich bin die Funktion",i)
```

```
print("-----")
funct(10)
```

Decorator with arguments
Decorator

vorher 1
Ich bin die Funktion 10
nachher

```
def my_decorator_with_arguments(i):
    # Ausführung beim Modulimport
    print("Decorator with arguments")
    def my_decorator(func):
        # Ausführung beim Modulimport
        print("Decorator")
        def inner(*args,**kwargs):
            print("vorher",i)
            func(*args,**kwargs)
            print("nachher")
        return inner
    return my_decorator
```

Eigene Decorator

```
@my_decorator_with_arguments(1)
def funct(i):
    print("Ich bin die Funktion",i)
```

```
print("-----")
funct(10)
```

```
-----
vorher 1
Ich bin die Funktion 10
nachher
```

```
def my_decorator_with_arguments(i):
    def my_decorator(func):
        def inner(*args,**kwargs):
            print("vorher",i)
            func(*args,**kwargs)
            print("nachher")
        return inner
    return my_decorator
```

Muss ich jetzt ganz viele Decorator schreiben?

Vermutlich nicht

Aber es gibt bereits viele, die man nutzen kann

Dataclass

Property

classmethod

Dataclass

Generiert automatisch Methoden für uns

Init, repr, eq, match_args (Standard)

Order, unsafe_hash, frozen, kw_only, slots, weakref_slot
(Optional)

Anstatt

```
class Person:
    def __init__(self, name, alter, wohnort):
        self.name = name
        self.alter = alter
        self.wohnort = wohnort

    def __eq__(self, other):
        if(self.name != other.name): return False
        if(self.alter != other.alter): return False
        if(self.wohnort != other.wohnort): return False
        return True

    def __repr__(self):
        return f"Person(name='{self.name}', alter={self.alter}, wohnort='{self.wohnort}')"

    __match_args__ = ("name", "alter", "wohnort")
```

Dataclass

Reicht

```
@dataclass
class Person:
    name : str
    alter : int
    wohnort : str
```


Die zu generierende Methoden können angepasst werden

```
@dataclass(order=True)
class Person:
    name : str
    alter : int
    wohnort : str
```

Property

```
kreis = Kreis(3)
kreis.set_radius(4)
print(kreis.get_radius())
```

4

```
class Kreis:

    def __init__(self, radius):
        self._radius = radius

    def get_radius(self):
        return self._radius

    def set_radius(self, value):
        self._radius = value
```

Beispiel für schlechten Code

Property

```
kreis = Kreis(3)
kreis.set_radius(4)
print(kreis.get_radius())
```

4

```
class Kreis:

    def __init__(self, radius):
        self._radius = radius

    def get_radius(self):
        return self._radius

    def set_radius(self, value):
        self._radius = value
```

Getter und Setter sind schlechter
Codestill. Wie geht Python damit um?

Property

```
kreis = Kreis(3)
kreis.radius = 4
print(kreis.radius)
```

4

```
class Kreis:

    def __init__(self, radius):
        self.radius = radius
```

Ja, wirklich!
Aber was machen wir, wenn wir intern
mit dem Durchmesser arbeiten wollen?

Property

```
kreis = Kreis(3)
kreis.radius = 4
print(kreis.radius)
```

4.0

```
class Kreis:

    def __init__(self, radius):
        self._durchmesser = radius * 2

    @property
    def radius(self):
        return self.durchmesser / 2

    @radius.setter
    def radius(self, value):
        self._durchmesser = value * 2
```

Properties!



Property

Ermöglichen uns:

- Getter, Setter und Deleter zu programmieren
- Sind hinter einem Attribut versteckt

Anstelle des Objekts (self) ist der erste Parameter die Klasse (cls)

```
from datetime import date

@dataclass
class Person:
    name : str
    alter : int

    @classmethod
    def fromBirthYear(cls, name, birth_year):
        return cls(name, date.today().year - birth_year)
```

Beispiel einer Factory Method

Generatoren

- Geben einen Iterator zurück
 - Sind Speichereffizienter als Listen
 - Rechnen das nächste Element erst zur Laufzeit aus
 - Dadurch auch unendliche Iteratoren möglich
-
- Ihr kennt bereits einen Iterator
 - `(expression for x in xs if expression)`

Generatoren

```
def is_prime(n):  
    if(n < 2): return False  
    for i in range(2,n):  
        if (n%i) == 0:  
            return False  
    return True
```

```
def my_generator(i):  
    for j in range(i):  
        if (is_prime(j)):  
            yield j
```

- Wert wird mit yield zurückgegeben
- Ausführung wird pausiert
- Wird der nächste Wert abgerufen, wird an der Stelle fortgefahren

Generatoren

```
def is_prime(n):  
    if(n < 2): return False  
    for i in range(2,n):  
        if (n%i) == 0:  
            return False  
    return True
```

```
def my_generator(i):  
    for j in range(i):  
        if (is_prime(j)):  
            yield j
```

```
for i in my_generator(10):  
    print(i)
```

2
3
5
7

Generatoren

```
def is_prime(n):  
    if(n < 2): return False  
    for i in range(2,n):  
        if (n%i) == 0:  
            return False  
    return True
```

```
def my_generator(i):  
    for j in range(i):  
        if (is_prime(j)):  
            yield j
```

```
mg = my_generator(10)  
for i in range(10):  
    print(next(mg))
```

2

3

5

7

StopIteration Fehler

Unendlicher Generator

```
def is_prime(n):  
    if(n < 2): return False  
    for i in range(2,n):  
        if (n%i) == 0:  
            return False  
    return True
```

```
def my_generator():  
    j = 0  
    while True:  
        if (is_prime(j)):  
            yield j  
        j += 1
```

```
mg = my_generator()  
for i in range(10):  
    print(next(mg))
```

```
2  
3  
5  
7  
11  
13  
17  
19  
23  
29
```

Unendlicher Generator

```
def is_prime(n):  
    if(n < 2): return False  
    for i in range(2,n):  
        if (n%i) == 0:  
            return False  
    return True
```

```
def my_generator():  
    j = 0  
    while True:  
        if (is_prime(j)):  
            yield j  
        j += 1
```

```
for i in my_generator():  
    print(i)
```

Ausgabe hört nicht auf

Generatoren

Wofür könnten wir Generatoren gebrauchen?

Zum Beispiel um Datenbankinträge zurückzugeben

Wir laden die Daten erst aus der Datenbank, wenn wir sie wirklich brauchen

Abstract Classes

```
from abc import ABC, abstractmethod

class Abstrakt(ABC):
    @abstractmethod
    def test(self):
        print("this is a test")

class Unter(Abstrakt):
    def test(self):
        pass
    def was():
        print("wasweißichdenn")

u = Unter()
u.test()
```

- Abstrakte Klasse muss von Klasse ABC erben
 - > Aus package „abc“
- Methoden, die in den Unterklassen implementiert werden sollen: decorator „abstractmethod“ dazu
- Unterklasse muss von der erstellten abstrakten Klasse erben
 - > Abstrakte Methoden müssen implementiert werden
 - > **Sonst Fehler**

TypeError: Can't instantiate abstract class Unter with abstract method test

Abstract Classes

```
from abc import ABC, abstractmethod

class Abstrakt(ABC):
    @abstractmethod
    def test(self):
        print("this is a test")

class Unter(Abstrakt):
    def test(self):
        super().test()
    def was():
        print("wasweißichdenn")

u = Unter()
u.test()
```

- Abstrakte Klasse muss von Klasse ABC erben
 - > Aus package „abc“
- Methoden, die in den Unterklassen implementiert werden sollen: decorator „abstractmethod“ dazu
- Unterklasse muss von der erstellten abstrakten Klasse erben
 - > Abstrakte Methoden müssen implementiert werden
 - > **Sonst Fehler**
 - auf super Klasse mit super().methodenname zugreifbar