Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-09 Dozent: Karsten Flügge

Themen des Tages

Dekoratoren
Server
Datenbanken (ohne SQL)
Tests

Codex ohne Kreditkarte: 5 Anfragen pro Tag
https://chatgpt.com/download/

Dekoratoren

Fortgeschrittene Syntax welche eher als **Ergänzung** des Kern Python verstanden werden kann.

Ein **Dekorator** ist **eine Funktion, die eine andere Funktion umschließt**, um ihr Verhalten zu erweitern oder zu ändern.

```
def dekorator(funktion):
  print("Funktion wird umhüllt", funktion) # wird bei Deklaration einmal aufgerufen
  return funktion # wird bei jedem Aufruf ausgeführt

@dekorator
def hallo():
  print("Funktionsaufruf hallo")


hallo()
```

```
Das @ im @dekorator erinnert an eine Umwicklung/Umschließung/Umhüllung
Der name hinter @ muss eine Funktion sein, welche eine Funktion als Argument erhält
  (hier: def dekorator(funktion))
```

Dekorator Parameter

Fortgeschrittene Syntax welche eher als **Ergänzung** des Kern Python verstanden werden kann.

Ein **Dekorator** kann mit Zusatzparametern k**onfiguriert** werden

```
def dekorator(funktion, args):
  print("Funktion wird umhüllt", funktion) # wird bei Deklaration einmal aufgerufen
  if "hack" in args:
    return hack
  return funktion # wird bei jedem Aufruf ausgeführt

@dekorator(args)
def hallo():
  print("Funktionsaufruf hallo")


# hallo()
```

Dekorator Hacking

Ein Dekorator kann eine beliebige Funktion zurück liefern, welche mit der eigentlichen nichts zu tun hat!

```
def hack():
  print("Funktionsaufruf hack!")


def dekorator(funktion):
  print("Funktion wird umhüllt", funktion) # wird bei Deklaration einmal aufgerufen
  return hack # wird bei jedem Aufruf von hallo ausgeführt!
  # Ignoriere ursprüngliche funktion (hallo)

@dekorator
def hallo():
  print("Funktionsaufruf hallo")


hallo() # Funktionsaufruf hack!
```

Dekoratoren

Da wir Funktionen innerhalb anderer Funktionen erzeugen können, lässt sich das Verhalten der **Umhüllten** beliebig modifizieren:

```
def dekorator(funktion):
  print("Funktion wird umhüllt", funktion)
  def umhuellende(*args, **kwargs): # argumente von hallo
    print("Umhüllende Funktion vor Aufruf von ",funktion)
    ergebnis = funktion(*args, **kwargs)
    print("Umhüllende Funktion nach Aufruf von ",funktion)
    return ergebnis
  return umhuellende


@dekorator
def hallo():
  print("Funktionsaufruf hallo")


hallo(1,2,a=3)
```

Klassen Dekoratoren

Wir können auch Klassen dekorieren, z.B.

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

# __init__ brauchen wir nicht mehr
```

Varianten, welche automatisch Datenbank mapping herstellen

Klassen Dekoratoren

```
def printable(cls):
  def __repr__(self):
    return f"{self.name}, {self.alter}"
  cls.__repr__ = __repr__
  return cls

@printable  # nimmt Klasse und liefert veränderte oder neue zurück
class Person:
  def __init__(self, name, alter):
    self.name = name
    self.alter = alter

john = Person("John", 30)
print(john)  # Ausgabe: John, 30
```

Bekannte Dekoratoren

Bisher haben wir Dekoratoren kennengelernt bei:

```
class Klassenhelfer:

   @staticmethod
   def methode_static(): # z.B. count(), alle_individuen(), KONSTANTEN()
      print("Dies ist eine statische Methode")

   @classmethod
   def methode_cls(cls): # so wie self bei Instanzen
      print("Dies ist eine Klassenmethode von ", cls)


Klassenhelfer.methode_static()    # Dies ist eine statische Methode
Klassenhelfer.methode_cls()       # Dies ist eine Klassenmethode von <class '__main__.Klassenhelfer'>


@app.route('/hallo') # Verbindet server Pfad mit Funktion:
def hallo():
  return "hallo"
```

Server

Python Programm was man per Browser ansprechen kann

```
@app.route('/hallo') # Verbindet server Pfad mit Funktion:
def hallo():
  return "hallo"
```

Nützliche Dekoratoren

Wozu kann man **Dekoratoren** nutzen?

- **Registrierung** von Funktionen
- **Debugging**
- **Performance Benchmarks:** welche Funktionen sind langsam? Speicherverbrauch
- Aspect oriented programming **@before @after**
- **Server @app.route**
- **Daten Klassen, Datenbank mapping**
- **Höhere Magie!**
- **Hacken, *Unfug*, extrem mächtig**
- **@fast** Beschleunigung (z.B. JIT compilation / GPU taichi?)
- **Zugriffsteuerung**
- **Tests!!!**
- Verallgemeinerung von Funktionalität (z.B. @printable)

```
@login_required
def topics(request):
"""Show all topics."""
```

Nützliche Dekoratoren

- Beispiel @property

```
class Person:
   def __init__(self, vorname, nachname):
      self._vorname = vorname
      self._nachname = nachname

   @property
   def voller_name(self):
      return f"{self._vorname} {self._nachname}"

   @voller_name.setter
   def voller_name(self, name):
      vorname, nachname = name.split()
      self._vorname = vorname
      self._nachname = nachname

# Beispiel
p = Person("Max", "Mustermann")
print(p.voller_name)  # Max Mustermann
p.voller_name = "Anna Müller"
print(p.voller_name)  # Anna Müller
```

Vokabular

@dekoratoren (wrapper, reflection) "Magie"

- `@property`