Dekoratoren und Exception-Behandlung

Richard Müller, Tom Felber

20. Januar 2022

Python-Kurs

Gliederung

1. Wiederholung

2. Dekoratoren

3. Exception-Behandlung

Gesamtübersicht

Gesamtübersicht

Themen der nächsten Stunden

- Referenzen Erklärung
- Klassen
- Imports
- Nützliche Funktionen zur Iteration
- Lambda
- Unpacking
- File handeling
- Listcomprehension
- Dekoratoren
- Exception-Behandlung

Wiederholung

Wiederholung

Beim letzten Mal:

- Filehandling

```
# reading
f = open("filename.txt", "r")
# writing
f = open("filename.txt", "w")
# append
f = open("filename.txt", "a")
```

- List Comprehensions

```
# Standardweg
liste = []
for i in range(5):
    liste.append(i)

# list comprehension
liste = [i for i in range(5)]
```

Dekoratoren

Dekoratoren

Funktionen und Klassen Definitionen können "dekoriert"werden, indem über der jeweiligen definition `@<funktion>` eingefügt wird.

```
1 @beispiel_dekorator
2 def test(x, y):
3     print(f"test {x} {y}")
```

Hinter dem `@` muss dabei ein Funktionsname stehen.

Funktionen die als Dekorator verwendet werden können, nehmen eine Funktion, Methode oder Klasse als Argument entgegen und geben üblicherweise das gleiche auch wieder aus.

Durch die `@` Notation wird die ursprüngliche Funktion mit dem Ausgabewert der Dekoratorfunktion überschrieben.

```
1 def test(x, y):
2     print(f"test {x} {y}")
3
4 test = beispiel_dekorator(test)
```

Dieser Code bewirkt das Gleiche, was auch mit der `@` Notation bewirkt würde.

Beispiel Funktionsdekorator

Ein Funktionsdekorator nimmt eine Funktion entgegen, und gibt meistens auch wieder eine Funktion aus.

```
def beispiel_dekorator(func):
    def ausgabe_funktion(x, y, z=0):
        func(x, y)
        print(z)
    return ausgabe_funktion
```

```
@beispiel_dekorator
def test(x, y):
    print(f"test {x} {y}")
```

In diesem Fall verändert die Dekoratorfunktion die Signatur der Ausgangsfunktion.

Anwendungsbeispiel Nebeneffekte

Ziel eines Dekorators muss es aber nicht immer sein, das Verhalten der Funktion oder Klasse zu verändern. Er kann auch als Einstiegspunkt dienen.

```
1  triggers = []
2
3  def always_run(func):
4      triggers.append(func)
5      return func
6
7  @always_run
8  def hello():
9      print("hello")
10
11 while(True):
12     inp = input("Wort eingeben:")
13     for t in triggers:
14         t()
```

mehrere Dekoratoren

Es können auch mehrere Dekoratoren angewendet werden:

```
@deko_eins
@deko_zwei
def hi():
    print("hi")

hi()
```

Definitionen für die benutzten Dekoratoren.

```
1  def deko_eins(func):
2      def out():
3          print("deko eins start")
4          func()
5          print("deko eins ende")
6      return out
7
8  def deko_zwei(func):
9      def out():
10         print("deko zwei start")
11         func()
12         print("deko zwei ende")
13     return out
```

mehrere Dekoratoren

Es können auch mehrere Dekoratoren angewendet werden:

```
@deko_eins
@deko_zwei
def hi():
    print("hi")

hi()
```

Ausgabe:

```
deko eins start
deko zwei start
hi
deko zwei ende
deko eins ende
```

Exception-Behandlung

Exceptions

Exceptions sind Fehler, die das Programm wirft, wenn es einen unerwünschten Zustand erreicht hat. Jedem Programmierer werden sie früher oder später begegnen.

Beispiel Exceptions sind:

- `Exception` (Basisklasse, von der alle Exceptions erben)
- `RuntimeError`
- `ZeroDivisionError`
- `ValueError`
- `KeyboardInterrupt`

try und except

Mit den beiden Keywords `try` und `except` können Fehler abgefangen und behandelt werden:

```
1 liste = [0]
2 try:
3     print(liste[1])
4 except:
5     print("Da war ein Fehler")
```

Das Programm wird nicht abstürzen, es wird lediglich 'Da war ein Fehler' auf der Konsole ausgeben.

try und except

Man kann, ähnlich wie bei `if` und `elif`, auch verschiedene Fehler verschieden behandeln:

```
try:
    # irgendwelcher Code
except ValueError:
    print("Ein ValueError")
except ZeroDivisionError:
    print("Man darf nicht durch Null teilen")
except IndexError:
    print("Dieser Index exisitert nicht")
```

finally

Möchte man *auf jeden Fall* etwas ausführen, egal, ob gerade ein Fehler kam, oder nicht, kann man das Keyword `finally` benutzen:

```
try:
    print(1/0)
except ZeroDivisionError:
    print("Man darf nicht durch Null teilen")
finally:
    print("Fertig")
# printet "Man darf nicht durch Null teilen" und danach "
    Fertig" aus
```

Exceptions sind Objekte

Wird in Python eine Exception geworfen, so ist diese als Objekt verfügbar:

```
try:
    print(1/0)
except ZeroDivisionError as e:
    print(e)
# printet "division by zero" aus
```

oder allgemeiner:

```
try:
    print(1/0)
except Exception as e:
    print(e)
# division by zero
```

Selber Exceptions werfen

Man kann auch selber Exceptions werfen, z.B., wenn man gewisse Eingaben unterbinden will. Dies wird mit dem Keyword `raise` gemacht:

```
try:
    raise RuntimeError("Ein Error zum Testen")
except Exception as e:
    print(e)
# printet "Ein Error zum Testen" aus
```

Möglichkeiten für nächste Stunde

- Flask/Django
- Discord Bot
- Selenium
- Erweiterte Themen
  - Multithreading
  - Datetime
  - Regex
  - ...
- ...