Programmieren I - Python

Sommersemester 2024

Danny Rehl

Institut für Computerlinguistik
Universität Heidelberg

09. Dezember 2024

Übersicht

Dekoratoren

Mehr über Funktionen

- **Alles ist ein Objekt**, also sind auch Funktionen Objekte
- Funktionen sind sog. "Callables" (können aufgerufen werden)
- Funktionsobjekte (ohne Aufruf) können Variablen zugewiesen werden[1]
  - Variable ist Referenz zur Funktion (also auch aufrufbar)
  - Aufruf mit Argumenten muss (immer noch) Parametern entsprechen

Beispiel: Funktion greet wird über Umweg aufgerufen

```
1 def greet(name: str) -> None:
2     print("Hallo " + name)
3
4 # greet_someone ist quasi ein alias für greet
5 greet_someone = greet
6
7 greet_someone("Arthur")
8 Hallo Arthur
```

[1] Zwischen `greet_someone = greet` und `greet_someone = greet()` gibt es einen großen Unterschied! Ersteres ist eine Referenz zur Funktion selbst, Zweiteres weist die Rückgabe der Funktion der Variablen zu (hier: `None`).

Mehr über Funktionen

- Funktionen sind Objekte
- Also können auch Funktionen in Funktionen sein

Beispiel: Funktion + Aufruf innerhalb einer Funktion

```
1  def greet(name: str) -> str:
2
3      # Eine Funktion innerhalb einer Funktion zu schreiben
4      # ist genauso möglich wie in diesem Fall z. B. einfach
5      # x = "Hallo " zu verwenden und später x + name zurückzugeben
6      def get_message() -> str:
7          return "Hallo "
8
9      return get_message() + name
10
11 print(greet("Arthur"))
12 Hallo Arthur
```

Mehr über Funktionen

- Funktionen sind Objekte
- Funktion als Parameter für eine andere Funktion möglich[2]

Beispiel: Aufruf einer übergebenen Funktion in einer Funktion

```
1  from typing import Callable
2
3  def greet(name: str) -> str:
4      return "Hallo " + name
5
6  def call_arthur(func: Callable) -> str:
7      knight = "Arthur"
8      return func(knight)  # Hier wird "greet" (Übergabe s. u.) aufgerufen
9
10 # Das Funktionsobjekt greet wird übergeben, d. h. ohne Klammern!
11 # Der Aufruf vom Callable "greet" passiert dann in "call_arthur".
12 print(call_arthur(greet))
13 Hallo Arthur
```

[2] siehe u. a. `map(func, *iterables)`, Beispiel im Anhang

Mehr über Funktionen

- Funktionen sind Objekte
- Funktion kann Funktionsobjekt zurückgeben[3]

Beispiel: Rückgabe eines Funktionsobjekts (factory function)

```
1  def compose_greet_func() -> Callable:
2      def get_message() -> str:
3          return "Hello there."
4      return get_message  # Rückgabe Funktionsobjekt (ohne Aufruf!)
5
6  # Aufruf der Funktion gibt inneres Funktionsobjekt zurück
7  greet = compose_greet_func()
8  print(greet())
9  Hello there.
10
11 # Insgesamt muss man hier also zweimal aufrufen, so ist es deutlicher:
12 print(compose_greet_func()())
13 Hello there.
```

[3] Die Funktion fabriziert sozusagen ein Funktionsobjekt, welches später aufgerufen werden kann. Daher kann eine solche Funktion auch als *factory function* bezeichnet werden (wie hier die Funktion "`compose_greet_func`").

Kurzer Ausflug zu closure-Funktionen

- Innere Funktionen lesenden Zugang nach außen
- Ein Überschreiben von äußeren Variablen wird verhindert

Beispiel: Innere Funktion lesenden Zugriff nach außen

```
1  def outer_function() -> None:
2      value = 5
3      x = 1
4
5      def inner_function() -> None:  # <- "closure function"
6          value = 10  # value im Skopus/Namespace von inner_function
7          print("Inner function (value):", value)
8          print("Inner function (x):", x)  # lesenden Zugriff
9
10     inner_function()
11     print("Outer function (value):", value)
12
13 outer_function()
14 Inner function (value): 10
15 Inner function (x): 1
16 Outer function (value): 5  # <- Wert ist nicht überschrieben
```

Kurzer Ausflug zu closure-Funktionen

- Innere Funktionen normal lesenden Zugang nach außen
- Schlüsselwort `nonlocal`[4] für Schreibzugriff (vgl. `global`[5])

Beispiel: Innere Funktion schreibenden Zugriff nach außen

```
1  def outer_function() -> None:
2      value = 5
3
4      def inner_function() -> None:
5          nonlocal value  # <- Nun Bezug zu äußerem Skopus/Namespace
6          value = 10
7          print("Inner function:", value)
8
9      inner_function()
10     print("Outer function:", value)
11
12 outer_function()
13 Inner function: 10
14 Outer function: 10  # <- Wert ist überschrieben
```

[4] https://docs.python.org/3/reference/simple_stmts.html#the-nonlocal-statement

[5] https://docs.python.org/3/reference/simple_stmts.html#global

Dekoratoren (Funktions-Wrapper)

Dekoratoren (Funktions-Wrapper)

- Dekoratoren sind Wrapper für beliebige Funktionen
- Eigentliche Funktion wird umhüllt und dadurch erweitert

Beispiel: HTML-Builder (umschließt Text mit HTML tags)

```
1  def text_as_html_site(func: Callable) -> Callable:
2      before = "<html><body>"
3      after = "</body></html>"
4
5      def func_wrapper(text):
6          return f"{before}{func(text)}{after}"
7
8      return func_wrapper
9
10 def message_screamer(text: str) -> str:
11     return text.upper()
12
13 html_site = text_as_html_site(message_screamer)
14
15 print(html_site("Niemand darf passieren."))
16 <html><body>NIEMAND DARF PASSIEREN.</body></html>
```

Dekoratoren, Zugriff über “@”-Syntax

- Python hat eine eingebaute Dekorator-Syntax
- Steht über der Funktion mit einem @-Operator

Beispiel: `message_screamer` nun dauerhaft[6] als HTML-Seite

```
1  def as_html_site(func: Callable) -> Callable:
2      def func_wrapper(text: str) -> str:
3          return f"<html><body>{func(text)}</body></html>"
4      return func_wrapper
5
6  @as_html_site
7  def message_screamer(text: str) -> str:
8      return text.upper()
9
10 print(message_screamer("Wir haben eine Hexe."))
11 <html><body>WIR HABEN EINE HEXE.</body></html>
```

[6] Wenn man `message_screamer` nicht dauerhaft als HTML-Seite anzeigen möchte, dann sollte man diese Funktion auch nicht dekorieren. Dieser Dekorator kann nun für verschiedene Funktionen angewendet werden und diese gleichermaßen erweitern.

Mehrere Dekoratoren

- werden von unten nach oben eingelesen/angewendet

Beispiel: Verwenden von “stacked decorators”

```
1  def body_decorate(func: Callable) -> Callable:
2      """Wrapping text with <body> and </body>"""
3      def func_wrapper(text: str) -> str:
4          return f"<body>{func(text)}</body>"
5      return func_wrapper
6
7  def html_decorate(func: Callable) -> Callable:
8      """Wrapping text with <html> and </html>"""
9      def func_wrapper(text: str) -> str:
10         return f"<html>{func(text)}</html>"
11     return func_wrapper
12
13 @html_decorate  # 2.
14 @body_decorate  # 1.
15 def message_screamer(text: str) -> str:
16     return text.upper()
17
18 print(message_screamer("Nur eine Fleischwunde."))
19 <html><body>NUR EINE FLEISCHWUNDE.</body></html>
```

Parametrisierung von Dekoratoren

- Generalisierung nun möglich
- Durch Hinzufügen einer weiteren (äußeren) Funktion mit Parameter
- Recap: Innere Funktionen können auf äußere Variablen lesend zugreifen

Beispiel: Parametrisierung durch weitere äußere Funktion

```
1  def tagging(tag: str) -> Callable:  # <- Einschub: Funktion mit Parameter
2      def tags_decorator(func: Callable) -> Callable:
3          def func_wrapper(text: str) -> str:
4              return f"<{tag}>{func(text)}</{tag}>"  # tag anstatt body/html
5          return func_wrapper
6      return tags_decorator
7
8  @tagging("html")
9  @tagging("body")
10 @tagging("p")
11 def message_screamer(text):
12     return text.upper()
13
14 print(message_screamer("Du hast keine Arme mehr."))
15 <html><body><p>DU HAST KEINE ARME MEHR.</p></body></html>
```

Debugging dekorierter Funktionen

- Funktionen werden zurückgegeben
- Der Namespace liegt dann bei diesen Funktionen

Falsche Ausgabe für dekorierte Funktion

```
# Funktionen `tagging` und `message_screamer`: siehe letzte Folie

def say_arthur():
    print("Arthur")

print(say_arthur.__name__)
say_arthur

print(message_screamer.__name__)
func_wrapper  # Eigentlich wollen wir 'message_screamer' !
```

Functools

- Bibliothek `functools.wraps` updated (repariert) alle Attribute[7]

```python
from functools import wraps

def tagging(tag: str) -> Callable:
    def tags_decorator(func: Callable) -> Callable:
        @wraps(func)  # Attribut-Updater als Dekorator
        def func_wrapper(text: str) -> str:
            return f"<{tag}>{func(text)}</{tag}>"
        return func_wrapper
    return tags_decorator

@tagging("p")
def message_screamer(text: str) -> str:
    """A scream something docstring"""
    return text.upper()

print(message_screamer.__name__)
message_screamer                    # Yes! :)
print(message_screamer.__doc__)
A scream something docstring        # Das wollen wir :)
```

[7] siehe u. a. diesen StackOverflow-Beitrag oder die Python-Doku-Seite

Weiteres Beispiel (Profiling)

```python
from functools import wraps
from time import time as timestamp, sleep
from typing import Callable


def profiler(func: Callable) -> Callable:
    @wraps(func)
    def wrapper(*args, **kwargs):
        # before decorated function is called
        started_at = timestamp()

        # actual call of the decorated function
        result = func(*args, **kwargs)

        # after decorated function finished (print duration)
        ended_at = round(timestamp() - started_at, 2)
        print(f"{func.__name__} run {ended_at} second(s).")

        return result  # decorated function behaviour has not changed

    return wrapper


@profiler
def any_function() -> None:
    sleep(2)


if __name__ == "__main__":
    any_function()
```

Weiteres Beispiel (Logging)

```python
import logging
from functools import wraps
from typing import Callable

logging.basicConfig(
    format="%(asctime)s :: %(levelname)-8s :: %(name)s :: %(message)s", datefmt="%Y-%m-%d %H:%M:%S",
    filename="demo.log", filemode="a", level=logging.INFO
)


def log_function_call(func: Callable) -> Callable:
    @wraps(func)
    def wrapper(*args, **kwargs):
        logging.info(f"Calling function {func.__name__} with arguments {args} and {kwargs}")
        result = func(*args, **kwargs)
        logging.info(f"Function {func.__name__} returned {result}")
        return result
    return wrapper


@log_function_call
def tokenize(text: str):
    return text.split()


if __name__ == "__main__":
    tokenize("Dies ist ein einfacher Text.")
    with open("demo.log") as logfile:
        print(logfile.read())
```

Code-Beispiel für custom filter-Funktion

```python
from collections.abc import Generator
from typing import Callable, Iterable


def my_filter(func: Callable, iterable: Iterable) -> Generator:
    """Filter elements of an iterable

    This is a custom variant of the built-in filter function for demo
    purposes. Therefore, less powerful, but hopefully more
    understandable.

    :param func: function to apply to each element
    :param iterable: iterable to filter
    :return: Generator, yielding input items evaluated to True by func
    """
    # Generator, which yields each element of given iterable
    # if the function, which is applied on the current element,
    # will be evaluated to True
    return (element for element in iterable if func(element))


# Now compare "my_filter" with built-in "filter" function

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(list(my_filter(lambda x: x % 2 == 0, numbers)))
# output: [2, 4, 6, 8, 10]

print(list(filter(lambda x: x % 2 == 0, numbers)))
# output: [2, 4, 6, 8, 10]
```