Informatik I: Einführung in die Programmierung

18. Funktionale Programmierung / Dekoratoren

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Peter Thiemann

23.01.2024

1 Dekoratoren

Was ist ein Dekorator?

Ein **Dekorator** ist eine Funktion, die eine andere Funktion erweitert, ohne diese selbst zu ändern.
Die Syntax von Dekoratoren (Funktion `decorator` angewendet auf `fun`):

```
@decorator
def fun():
    ...
```

Dabei ist `decorator` selbst eine Funktion ...

Dekoratoren

Dekoratoren werden durch Funktionen, die Funktionen als Parameter nehmen und zurückgeben, implementiert.
Dekoratoren, die uns schon früher begegnet sind: `dataclass`, `property`, etc.
Falls der Dekorator `wrapper` definiert wurde, dann hat

```
@wrapper
def confused_cat(*args):
    pass # do some stuff
```

die gleiche Bedeutung wie

```
def confused_cat(*args):
    pass # do some stuff
confused_cat = wrapper(confused_cat)
```

Dekoratoren: `property`, `staticmethod` (1)

decorators.py

```
@dataclass
class C:
    _name : str

    def getname(self):
        return self._name

    # def setname(self, x):
    #     self._name = 2 * x
    name = property(getname)

    def hello():
        print("Hello world")
    hello = staticmethod(hello)
```

lässt sich mittels der @-Syntax schreiben ...

Dekoratoren: `property`, `staticmethod` (2)

```
@dataclass
class C:
    _name : str

    @property
    def name(self):
        return self._name

    # @name.setter
    # def name(self, x):
    #     self._name = 2 * x

    @staticmethod
    def hello():
        print("Hello world")
```

Motivation

Betrachte die Funktion

```
def mult (x:float, y:float) -> float:
    return x * y
```

Zur Fehlersuche möchten wir folgendes Feature:

Aufgabe

Gib bei jedem Aufruf den Namen der Funktion mit ihren Argumenten aus.

Definition eines Dekorators (1)

Naiver Ansatz: Ändere die Funktionsdefinition!

```
verbose = True
def mult(x:float, y:float) -> float:
    if verbose:
        print("--- a nice header ---------")
        print("--> call mult with args: %s, %s" % x, y)
    res = x * y
    if verbose:
        print("--- a nice footer ---------")
    return res
```

Schlecht, weil wir die Funktionsdefinition ändern müssen, wodurch neue Fehler entstehen können! Wir wollen eine modulare Lösung, bei der die Funktionsdefinition unverändert bleiben kann.

Definition eines Dekorators (2)

Wiederverwendbare modulare Lösung

```
def with_trace(f):
    def wrapper(*args, **kwargs):
        print("--- a nice header ---------")
        print("--> call %s with args: %s" %
              (f.__name__, ",".join(map(str, args))))
        res = f(*args, **kwargs)
        print("--- a nice footer ---------")
        return res
    # print("--> wrapper now defined")
    return wrapper

@with_trace
def mult(x:float, y:float) -> float:
    return x * y
```

Definition eines Dekorators (3)

Aufgabe 2

Wie lange dauert die Ausführung eines Funktionsaufrufs?

```
import time

def timeit(f):
    def wrapper(*args, **kwargs):
        print("--> Start timer")
        t0 = time.time()
        res = f(*args, **kwargs)
        delta = time.time() - t0
        print("--> End timer:  %s sec." % delta)
        return res
    return wrapper
```

Definition eines Dekorators (4)

Dekoratoren hintereinander schalten

decorators.py

```
@with_trace
@timeit
def sub(x:float, y:float) -> float:
    return x - y

print(sub(3, 5))
```

liefert z.B.:

decorators.py

```
--- a nice header ---------
--> call wrapper with args: 3,5
--> Start timer
--> End timer:  9.5367431640625e-07 sec.
--- a nice footer ---------
-2
```

Dekoratoren: `docstring` und `__name__` (1)

- Beim Dekorieren gehen interne Attribute wie Name und `docstring` verloren.
- Ein guter Dekorator muss das wieder richtigstellen:

```
def with_trace(f):
    def wrapper(*args, **kwargs):
        print("--- a nice header ---------")
        print("--> call %s with args: %s" %
              (f.__name__, ",".join(map(str, args))))
        res = f(*args, **kwargs)
        print("--- a nice footer ---------")
        return res
    wrapper.__name__ = f.__name__
    wrapper.__doc__ = f.__doc__
    return wrapper
```

Dekoratoren: `docstring` und `__name__` (2)

- Dieses Problem kann durch den Dekorator `functools.wraps` gelöst werden:

```
import functools
def with_trace(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        print("--- a nice header ---------")
        print("--> call %s with args: %s" %
              (f.__name__, ",".join(map(str, args))))
        res = f(*args, **kwargs)
        print("--- a nice footer ---------")
        return res
    return wrapper
```

Dekoratoren mit Parametern (1)

- Aufgabe: beschränke alle Stringergebnisse auf 5 Zeichen

```
def trunc(f):
    def wrapper(*args, **kwargs):
        res = f(*args, **kwargs)
        return res[:5]
    return wrapper

@trunc
def data():
    return 'foobar'
```

- Ein aktueller Aufruf:

```
print(data())
```

liefert `fooba`

Dekoratoren mit Parametern (2)

- Warum 5 Zeichen? Manchmal sollen es 3 sein, manchmal 6!

```
def limit(length:int):
    def decorator(f):
        def wrapper(*args, **kwargs):
            res = f(*args, **kwargs)
            return res[:length]
        return wrapper
    return decorator

@limit(3)
def data_a():
    return 'limit to 3'

@limit(6)
def data_b():
    return 'limit to 6'
```

Parametrische Dekoratoren (3)

- Was passiert hier?
- Der Aufruf von `limit(3)` erzeugt einen Dekorator, der auf `data_a` angewandt wird; `limit(6)` wenden wir auf `data_b` an:

  `print(data_a())`

  liefert: `lim`

  `print(data_b())`

  liefert: `limit`
- Aber was passiert genau bei der geschachtelten Definition von Funktionen?

2 Funktionsschachtelung, Namensraum und Umgebung

Geschachtelte Funktionsdefinitionen

- Im letzten Abschnitt sind uns geschachtelte Funktionsdefinitionen begegnet.
- Dabei stellt sich die Frage, auf welche Bindung sich die Verwendung einer Variablen bezieht.
- Dafür müssen wir die Begriffe Namensraum (Scope) und Umgebung verstehen.
- Und wir müssen uns mit der Lebensdauer einer Variablen auseinandersetzen.

Namensraum

- Der Namensraum (Scope) ist ein statisches Konzept. Er zeigt an, in welchen Teilen eines Programms ein definierter Name sichtbar und verwendbar ist.
- Ein Name komt “in scope” durch
  - Definition einer Variable, Funktion oder Klasse
  - Import eines Moduls

  und ist verfügbar bis zum Ende des Blocks, in dem er definiert wurde.
- Z.B. der lokale Namensraum einer Funktionsdefinition enthält Parameter und lokale Definitionen (Variable, Funktionen, Klassen, . . .). Er endet am Ende des Funktionsrumpfes.
- Namensräume bilden eine Hierarchie entsprechend der Schachtelung von Funktions- und Klassendefinitionen.

Umgebungen

- Eine Umgebung ist ein dynamisches Konzept (d.h. zur Laufzeit).
- Sie ist eine Abbildung von Namen auf Werte.
  - `Built-in`-Umgebung (`__builtins__`) mit allen vordefinierten Variablen
  - Umgebung von Modulen, die importiert werden
  - globale Umgebung (des Moduls `__main__`)
  - lokale Umgebung innerhalb eines Funktionsaufrufs (vgl. Kellerrahmen) diese können geschachtelt sein.
- Jeder Aufruf einer Funktion erzeugt eine neue lokale Umgebung, die normalerweise beim Ende des Aufrufs wieder gelöscht wird.
- Die Umgebungen bilden eine Hierarchie, wobei die innerste, lokale Umgebung normalerweise alle äußeren überdeckt!
- Jede Umgebung instanziert einen Namensraum.

Sichtbarkeit

- Eine Variable heißt sichtbar in dem Teil eines Programms, in dem die Variable ohne die Punkt-Notation referenziert werden kann.
- Wird ein Variablenname zum Lesen referenziert, so durchläuft Python die Hierarchie der Namensräume und versucht der Reihe nach:
  - ihn im lokalen Namensraum aufzulösen;
  - ihn in den nicht-lokalen Namensräumen (die den lokalen Namensraum umschließen) aufzulösen;
  - ihn im globalen Namensraum aufzulösen;
  - ihn im `Builtin`-Namensraum aufzulösen.
- Dabei heißt “auflösen” das Auffinden des Werts der Variable in der zugeordneten Umgebung.

Zuweisungen

- Gibt es eine Zuweisung *var* `=` `...` im aktuellen Scope, so wird von einem lokalen Namen ausgegangen und Referenzen auf *var* dürfen erst nach Ausführung der Zuweisung erfolgen.
- Ausnahmen:
  - „`global` *var*“ bedeutet, dass *var* im globalen Namensraum gesucht werden soll. Zuweisungen an *var* wirken auf die globale Umgebung.
  - „`nonlocal` *var*“ bedeutet, dass *var* in einem nicht-lokalen Namensraum gesucht werden soll, d.h. in den umgebenden Funktionsdefinitionen. Auch Zuweisungen wirken dort.
- Kann ein Name nicht aufgelöst werden, dann gibt es eine Fehlermeldung.

Ein Beispiel für Namensräume (1)

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)
```

Ein Beispiel für Namensräume (2)

Python-Interpreter

```
>>> scope_test()
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
>>> print("In global scope:", spam)
In global scope: global spam
```

3 Closures

Closures in Python

- Eine Closure ist eine von einer anderen Funktion zurückgegebene lokale Funktion, die freie Variable (nicht-lokale Referenzen) enthält:

```
def add_x(x:float) -> Callable[[float], float]:
    def adder(num:float) ->float:
        return x + num
    # adder is a closure
    # x is a free variable of adder
    return adder

add_5 = add_x(5); print(add_5)
```

Ausgabe: `<function add_x.<locals>.adder at 0x10c541080>`

```
print(add_5(10))
```

Ausgabe: 15

Closures in Python (2)

- Dasselbe mit einer `lambda` Abstraktion:

```
def add_x(x:float) -> Callable[[float], float]:
    return lambda num: x + num
    # returns a closure
    # num is a bound variable,
    # x is a free variable of the lambda

add_6 = add_x(6); print(add_6)
```

Ausgabe: `<function add_x.<locals>.<lambda> at 0x10c540e00>`

```
print(add_6(10))
```

Ausgabe: `16`

Closures und Lebensdauer

- Wird eine Funktion mit freien Variablen, wie `x` in `lambda num: x + num` als Ergebnis zurückgegeben, dann verlängert sich die Lebensdauer der Umgebung des Aufrufs von `add_x` und damit von `x`.
- Wenn die zurückgegebene Funktion `add_6` aufgerufen wird, dann wird diese Umgebung und damit der Wert von `x` wieder installiert.

Closures in Python (3)

- Achtung bei der Interaktion von Closures mit Zuweisungen:

```
def clo() -> Callable[[], int]:
    x = 0
    f = lambda : x
    x = x + 1
    return f

fx = clo()
print(fx())
```

Ausgabe: 1

- Nachfolgende Zuweisungen ändern den Wert in der Closure...

Zusammenfassung Closures

- Definition: Eine Variable tritt frei in einem Funktionsrumpf auf, wenn sie zwar vorkommt, aber weder in der Parameterliste noch in einer lokalen Zuweisung gesetzt wird.
- Jede Funktion mit freien Variablen wird durch eine *Closure* repräsentiert.
- Innerhalb einer Closure kann mit Hilfe der Anweisungen `nonlocal` oder `global` auf freie Variable schreibend zugegriffen werden.
- In den beiden letzteren Fällen verlängert sich die Lebensdauer einer Umgebung (nämlich des umschliessenden Funktionsaufrufs)! Sie bleibt so lange erhalten wie die Closure zugreifbar ist!