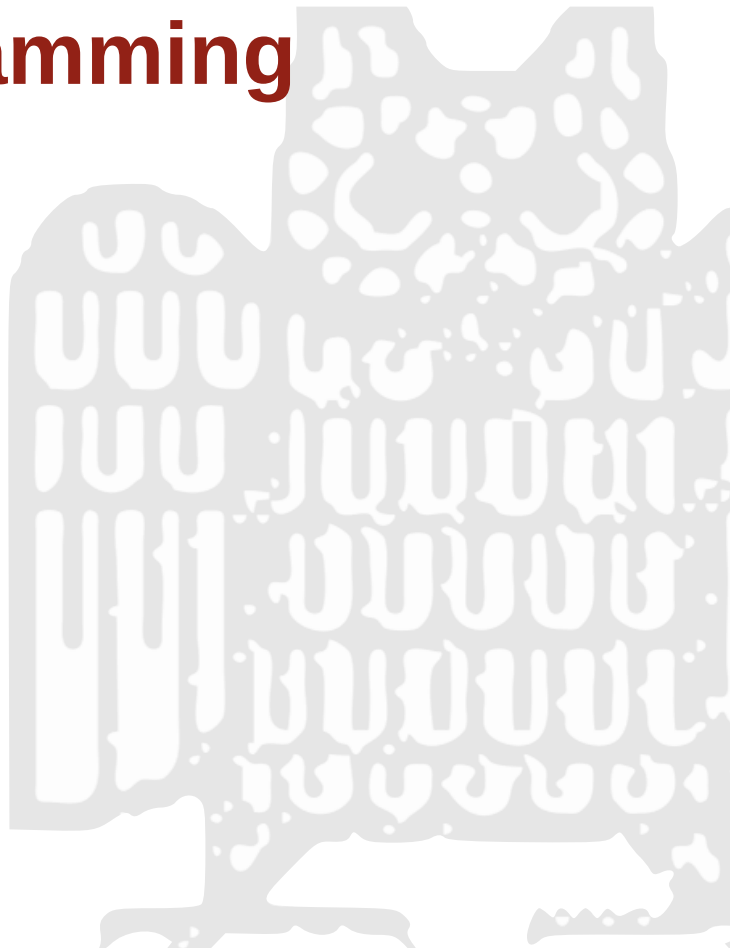
Introduction to Python Programming

20 – Decorators (Advanced Topic)

Josef van Genabith (Stefan Thater)
Dept. of Language Science & Technology
Universität des Saarlandes

WS 2022/23



Remarks

- For a moment (for this slide deck), forget that Python „map“ and „filter“ are built-in functions …
- We are going to „reconstruct/implement“ them using function decorators and a few other bits and pieces
- The purpose of the exercise is to learn how to use function decorators, which can be very handy …

Functions …

- Functions are “first class citizens” in Python
  - functions can be passed as arguments to other functions
  - functions can return other functions
- Functions that take other functions as arguments are called “higher order functions”

Nested Functions

- Functions can be defined within other functions
  - The inner function can only “read” the local variables of the outer functions (no assignment)
  - Outer function can return the inner function, run/evaluated with parameters, here `inner(5)`

```
def outer(values):
    total = sum(values)
    def inner(x):
        return x * total
    return inner(5)
```

```
>>> outer([1, 2, 3])
30
```

Nested Functions

- Functions can be defined within other functions
  - The inner function can only “read” the local variables of the outer functions (no assignment)
  - The outer function can return (a call to!) the inner function, here `add` (without parentheses and argument, i.e it doesn‘t execute it before that call):

```
def make_add(n):
    def add(m):
        return n + m
    return add
```

```
>>> add1 = make_add(1)
>>> add2 = make_add(2)
>>> add1(3)
4
>>> add2(3)
5
```

Function decorators

- In a nutshell, decorators are a simple way of calling a higher order function

Decorators

```
def log(fn):
    def wrap(arg):
        print(fn)
        return fn(arg)
    return wrap

def add17(x):
    return x + 17

add17 = log(add17)
```

```
>>> add17(4)
<function add17 ...>
21
```

Here `log` is a higher order function that takes `add17` as an argument

Decorators

```
def log(fn):
    def wrap(arg):
        print(fn)
        return fn(arg)
    return wrap

@log
def add17(x):
    return x + 17
```

```
def add17(x):
    return x + 17

add17 = log(add17)
```

```
>>> add17(4)
<function add17 ...>
21
```

Decorators:

- a simple syntax for calling higher order functions

An aside

- Functions can be called with any arbitrary number of arguments **`*args`** :

```
def example(*args):
    return args
```

```
>>> example(1,2,3):
(1, 2, 3)
```

```
>>> example(1,2,3,4,5):
(1, 2, 3, 4, 5)
```

Another aside

- Argument unpacking

```
>>> def add(x, y):
...     return x + y
...
>>> pair = (17, 4)
>>> add(pair[0], pair[1])
21
>>> add(*pair)
21
```

Decorators

```
def log(fn):
    def wrap(*args):
        print(fn)
        return fn(*args)
    return wrap

@log
def add17(x):
    return x + 17

@log
def add(x, y):
    return x + y
```

```
>>> add17(4)
<function add17 ...>
21
>>> add(17, 4)
<function add ...>
21
```

Decorators with arguments

- Decorators can have arguments
  - or rather, we can can define a function that takes some arguments and returns a decorator

```
def log(comment):
    def decorator(fn):
        def wrap(*args):
            print(comment)
            return fn(*args)
        return wrap
    return decorator

@log("example")
def add1(x): return x + 1
```

Exercise #1

- Write a decorator `expect` that tests whether the arguments of a function are of certain types:

```
@expect(int, int)
def add(x, y):
    return x + y
```

```
>>> add(17, 4)
21
>>> add('py','thon')
Traceback ...
TypeError: expecting
<class 'int'>
```

Functions …

- Functions are “first class citizens” in Python
  - functions can be passed as arguments to other functions
  - functions can return other functions
- Functions that take other functions as arguments are called “higher order functions”

Remarks

- For a moment (for this slide deck), forget that Python has „map“ and „filter“ as built-in functions …
- We are going to „reconstruct/implement“ them using function decorators and a few other bits and pieces
- The purpose of the exercise is to learn how to use function decorators, which can be very handy …

An Example

- Suppose we are given a list of strings and we want to compute a new list with all the strings in the original list in lower case.

```
def map_to_lower_case(strings):
    result = []
    for string in strings:
        result.append(string.lower())
    return result

print(map_to_lower_case(['A', 'B', 'C']))
# ['a', 'b', 'c']
```

An Example

- Now suppose we are given a list of strings and we want to compute a new list with all the strings of the original list in upper case.

```
def map_to_upper_case(strings):
    result = []
    for string in strings:
        result.append(string.upper())
    return result

print(map_to_upper_case(['a', 'b', 'c']))
# ['A', 'B', 'C']
```

An Example

```
def my_map(fn, items):
    result = []
    for item in items:
        result.append(fn(item))
    return result

def lower_case(string): return string.lower()
def upper_case(string): return string.upper()

print(my_map(upper_case, ['a', 'b', 'c']))
# ['A', 'B', 'C']

print(my_map(lower_case, ['A', 'B', 'C']))
# ['a', 'b', 'c']
```

user-defined functions

An Example

```
def my_map(fn, items):
    result = []
    for item in items:
        result.append(fn(item))
    return result

print(my_map(str.upper, ['a', 'b', 'c']))
# ['A', 'B', 'C']

print(my_map(str.lower, ['A', 'B', 'C']))
# ['a', 'b', 'c']
```

Python string methods

Exercise #2

- Implement a function „my_filter“ that takes a function and a list as arguments and returns the list of all elements of the original list for which the function returns True.

```
def even(x):
    return x % 2 == 0

print(my_filter(even, [1,2,3,4,5,6]))
# [2, 4, 6]
```

Remarks

- „map“ and „filter“ are built-in functions in Python …

More Examples

```
def snd(seq): return seq[1]

lop = [(1,9),(2,8),(3,7)]

max(lop) # (3,7)
max(lop, key=snd) # (1,9)

sorted(lop) # [(1, 9), (2, 8), (3, 7)]
sorted(lop, key=snd) # [(3, 7), (2, 8), (1, 9)]
```

More Examples

```
class Person:
    def __init__(self, name, surname):
        self._name = name
        self._surname = surname
    def __repr__(self):
        return 'Person(%r, %r)' % (self._name, self._surname)
    def name(self): return self._name
    def surname(self): return self._surname

p = [Person('Manfred', 'Pinkal'), Person('Hans', 'Uszkoreit')]

sorted(p, key=Person.name)
# [Person('Hans', 'Uszkoreit'), Person('Manfred', 'Pinkal')]

sorted(p, key=Person.surname)
# [Person('Manfred', 'Pinkal'), Person('Hans', 'Uszkoreit')]
```

More Examples

```
d = dict()
d['Company'] = 4000
d['University'] = 2000

sorted(d)
# ['Company', 'University']

sorted(d, key=d.__getitem__)
# ['University', 'Company']
```

lambda expressions

- Functions are usually defined using “def name(...): ...”
  - ⇒ Functions have a name
- We can define anonymous functions as follows:
  - lambda ⟨parameters⟩: ⟨expression⟩
- Lambda expressions:
  - evaluate to a function
  - the return value of the function is the value of ⟨expression⟩
  - are useful when used in combination with higher order functions such as map or filter
- Note: exactly one ⟨expression⟩, statements are not allowed

lambda expressions

```
>>> lambda x, y: x + y
<function <lambda> at 0xb736ddac>
>>> (lambda x, y: x + y)(1, 2)
3
>>> add = lambda x, y: x + y
>>> add(1,2)
3
>>> map(lambda x, y: x + y, [1,2,3],[4,5,6])
<map object at 0xb738edac>
>>> list(map(lambda x, y: x + y, [1,2,3],[4,5,6]))
[5, 7, 9]
```

If-Expressions

- Standard way of expressing conditionals:
  - if ⟨expression⟩: ⟨block⟩ else: ⟨block⟩
  - (this is a statement: if statement)
- Alternatively: conditional expressions
  - ⟨expression$_1$⟩ if ⟨condition⟩ else ⟨expression$_2$⟩
- Conditional expressions …
  - evaluate to the value of ⟨expression$_1$⟩ if ⟨condition⟩ evaluates to True
  - otherwise the conditional expression evaluates to the value of ⟨expression$_2$⟩

Exercise #3

- Solve Exercise #2 using a lambda-expression (instead of using a named function „even“)

```
def even(x):
    return x % 2 == 0

print(my_filter(even, [1,2,3,4,5,6]))
# [2, 4, 6]
```