Decorators

Adding Functionality to Existing Code

@decorators

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#python-advanced

Table of Contents

Functions Returning Functions

Function Returning Function

- A function can also generate another function

```
def hello_function():
    def say_hi():
        return "Hi"
    return say_hi
hello = hello_function()
print(hello())
```

Prints "Hi" on the console

Closure

- Python allows a nested function to access the **outer scope** of the enclosing function
- This is called **closure** and is a critical concept in decorators

```
def print_message(message):
    def message_sender():
        "Nested Function"
        print(message)
    message_sender()

print_message("Some random message")
```

Problem: Number Increment

- You are given the following code

```
def number_increment(numbers):
    def increase():
        # TODO: Implement
    return increase()
```

- Complete the code so it works as expected

```
print(number_increment([1, 2, 3]))
```

```
[2, 3, 4]
```

Solution: Number Increment

```
def number_increment(numbers):
    def increase():
        increased = [x + 1 for x in numbers]
        return increased
    return increase()
```

The `increase` function increases each element and returns the new numbers

Decorators

Definition and Usage

Decorators Definition

- **Decorators** are a very powerful and useful tool
- It allows programmers to **modify** the behavior of a function or a class
- Decorators allow us to **wrap** another function in order to extend the behavior of the wrapped function

Creating Decorators

- In the example below we create a decorator function that **converts** a sentence to **upper case**

```
def uppercase(function):
    def wrapper():
        result = function()
        uppercase_result = result.upper()
        return uppercase_result

    return wrapper
```

Using Decorators

- Our decorator function takes **a function as an argument**, so let us **define a function** and **pass it to our decorator**
- We learned earlier that we could assign a function to a variable
- We'll use that trick to call our decorator function

```
def say_hi():
    return 'hello there'

decorate = uppercase_decorator(say_hi)
decorate()
```

Decorators and "@"

- However, Python provides a much easier way for us to apply decorators
- We simply use the @ symbol before the function we would like to decorate23456

```
@uppercase
def say_hi():
    return 'hello there'

print(say_hi()) # HELLO THERE
```

functools.wraps()

- In the given example, if we try to call the **name** of the wrapped function the result is "**wrapper**", and its **docstring** is **lost**

```
@uppercase
def say_hi():
    """Saying Hi"""
    return "hello there"

print(say_hi.__name__) # wrapper
print(say_hi.__doc__)  # None
```

functools.wraps()

- To solve this problem, we use a decorator factory as a function **decorator** when defining a **wrapper** function

```
from functools import wraps

def uppercase(function):
    @wraps(function)
    def wrapper():
        result = function()
        uppercase_result = result.upper()
        return uppercase_result

    return wrapper
```

Problem: Vowel Filter

- You are given the following code

```
def vowel_filter(function):
    def wrapper():
        # TODO: Implement
    return wrapper
```

- Complete the code so it works as expected

```
@vowel_filter
def get_letters():
    return ["a", "b", "c", "d", "e"]
print(get_letters())
```

```
["a", "e"]
```

Solution: Vowel Filter

```
def vowel_filter(function):
    def wrapper():
        res = function()
        filtered = [x for x in res if x.lower() in "aeiou"]
        return filtered
    return wrapper
```

Accepting Arguments

Accepting Arguments

- Sometimes, we might need to define a decorator that accepts **arguments**
- We achieve this by passing the arguments to the **wrapper** function
- The arguments will then be passed to the function that is being decorated at call time

Accepting Arguments: Example

```
from time import time

def measure_time(func):
    def wrapper(*args, **kwargs):
        start = time()
        result = func(*args, **kwargs)
        end = time()
        print(end - start)
        return result
    return wrapper
```

Problem: Even Numbers

- You are given the following code

```
def even_numbers(function):
    def wrapper(numbers):
        # TODO: Implement
    return wrapper
```

- Complete the code so it works as expected

```
@even_numbers
def get_numbers(numbers):
    return numbers
print(get_numbers([1, 2, 3, 4, 5]))
```

```
[2, 4]
```

Solution: Even Numbers

```
def even_numbers(function):
    def wrapper(numbers):
        res = [x for x in numbers if x % 2 == 0]
        return function(res)
    return wrapper
```

Passing Arguments

Passing Arguments

- In order to achieve this, we define a **decorator maker** that accepts arguments
- Then we define a **decorator** inside it
- We then define a **wrapper function** inside the decorator as we did earlier

Passing Arguments: Example

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator


@repeat(4)
def say_hi():
    print("Hello")
```

Problem: Multiply

- You are given the following code

```
def multiply(times):
    def decorator(function):
        # TODO: Implement
    return decorator
```

- Complete the code so it works as expected

```
@multiply(3)
def add_ten(number):
    return number + 10
print(add_ten(3))
```

```
39
```

Solution: Multiply

```
def multiply(times):
    def decorator(function):
        def wrapper(params):
            return times * function(params)
        return wrapper
    return decorator
```

Decorating Methods in Classes

Examples

- **@classmethod** - decorator function that converts a method to a class method
- **@abstractmethod** - decorator function that converts an instance method to an abstract instance method
- **@abstractclassmethod** - decorator function that converts a class method to an abstract class method
- **@property** - change your class methods/attributes so that the user of a class doesn't need to make any change in their code

Example: property decorator

```
class Person:
    def __init__(self):
        self.__name = ''

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

    @name.setter
    def name(self, value):
        self.__name = value
```

Classes as Decorators

Classes as Decorators

- We can also use **classes** as **decorators**
- We usually do that when we need to **maintain** a **state**
- To use a class as a decorator, we need to **implement** the `__call__` method
- The `__call__` method allows class **instances** to be called **as functions**

Example: __call__ method

```
class Fibonacci:
    def __init__(self):
        self.cache = {}

    def __call__(self, n):
        if n not in self.cache:
            if n == 0:
                self.cache[0] = 0
            elif n == 1:
                self.cache[1] = 1
            else:
                self.cache[n] = self(n-1) + self(n-2)
        return self.cache[n]
```

Example: __call__ method

```
fib = Fibonacci()

for i in range(5):
    print(fib(i))

print(fib.cache)

# 0
# 1
# 1
# 2
# 3
# {0: 0, 1: 1, 2: 1, 3: 2, 4: 3}
```

Example: Class Decorator

```
class func_logger:

    _logfile = 'out.log'

    def __init__(self, func):
        self.func = func

    def __call__(self, *args):
        log_string = self.func.__name__ + " was called"
        with open(self._logfile, 'a') as opened_file:
            opened_file.write(log_string + '\n')
        return self.func(*args)
```

Example: Class Decorator

```
@func_logger
def say_hi(name):
    print(f"Hi, {name}")

@func_logger
def say_bye(name):
    print(f"Bye, {name}")

say_hi("Peter")
say_bye("Peter")
```

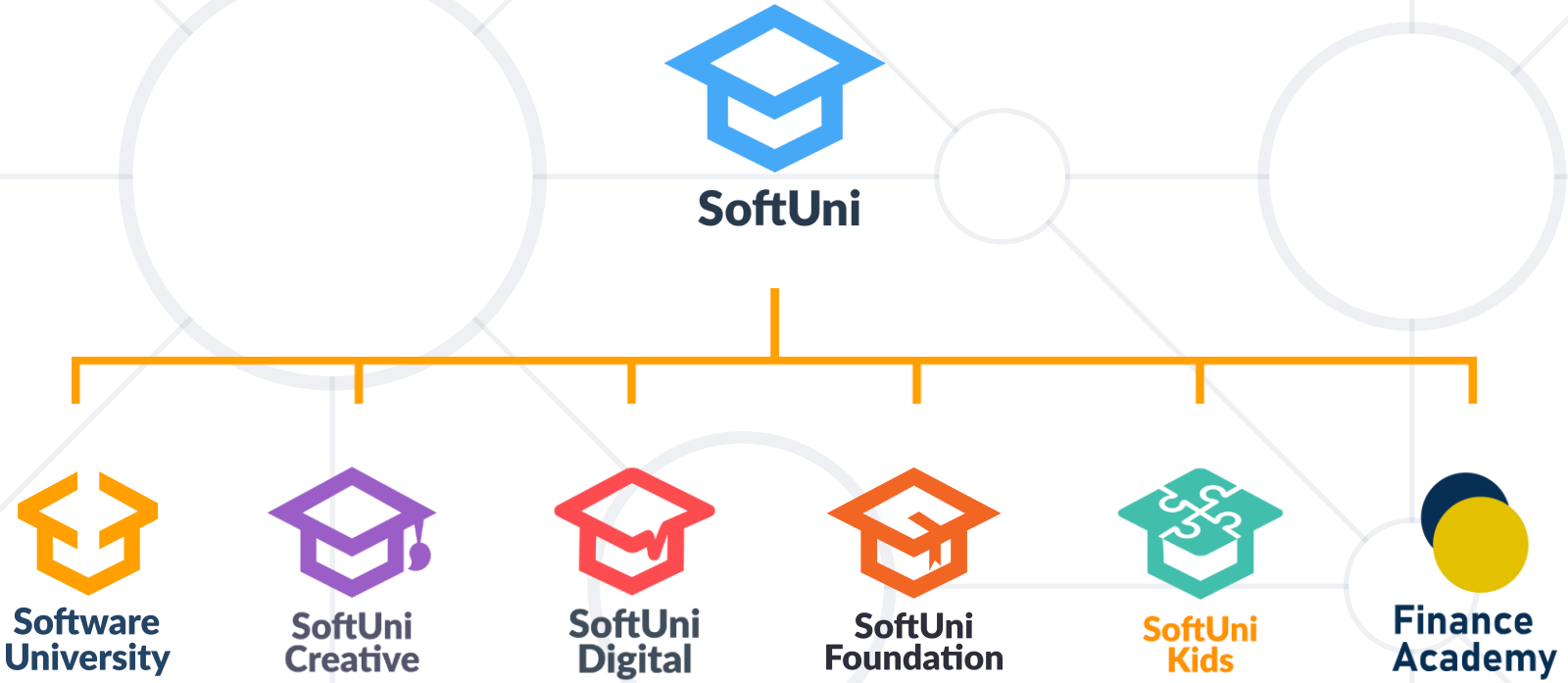
out.log

```
say_hi was called
say_bye was called
```

Summary

- Functions can return other functions
- When a nested function accesses the outer scope of the enclosing function, it is called a closure
- Decorators wrap another function to extend the behavior of the wrapped function

Questions?



SoftUni Diamond Partners

Software University

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License

- This course (slides, examples, demos, exercises, homework, documents, videos, and other assets) is **copyrighted content**
- Unauthorized copy, reproduction, or use is illegal
- © SoftUni – https://about.softuni.bg
- © Software University – https://softuni.bg