Lecture 12: Decorators

Swakkhar Shatabda

B.Sc. in Data Science
Department of Computer Science and Engineering
United International University

March 31, 2024

Decorators

- A decorator is a callable that takes another function as argument (the decorated function).
- Decorators have the power to replace the decorated function with a different one.

```
def decorator(func):
    def wrapper():
        print("Something is happening before the target function is called
        func()
        print("Something is happening after the target function is called.
    return wrapper

def target():
    print("I am a target function!")

target = decorator(target)
target()
```

Decorators

```
Something is happening before the target function is called.
I am a target function!
Something is happening after the target function is called.
```

- In effect, the name target now points to the wrapper() inner function. Remember that you return wrapper as a function when you call decorator(target)

```
>>> target
<function decorator.<locals>.wrapper at 0x100f144c0>
```

Decorator

- A decorator wraps a function, modifying its behavior.

```
from datetime import datetime

def not_during_the_night(func):
    def wrapper():
        if 7 <= datetime.now().hour < 22:
            func()
        else:
            pass  # Hush, the neighbors are asleep
    return wrapper

def target():
    print("Play Music!")

target = not_during_the_night(target)
target()
```

Syntactic Sugar

```
def decorator(func):
    def wrapper():
        print("""Something is happening before the target
                    function is called.""")
        func()
        print("""Something is happening after the target
                    function is called.""")
    return wrapper

@decorator
def target():
    print("I am a target function!")

target()
```

Functions With Arguments

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
@do_twice
def target():
    print("Whee!")
target()
@do_twice
def greet(name):
    print(f"Hello {name}")
greet("Hello")
```

```
Whee!
Whee!
Traceback (most recent call last):
  File "/Users/swakkhar/Desktop/decorator.py", line 15, in <mo
```

Solution

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice

@do_twice
def target():
    print("Whee!")
target()

@do_twice
def greet(name):
    print(f"Hello {name}")
greet("Hello")
```

Returning Values From Decorated Functions

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice

@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"

print(return_greeting("Swakkhar"))
```

```
Creating greeting
Creating greeting
None
```

Returning Values

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice

@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"

print(return_greeting("Swakkhar"))
```

```
Creating greeting
Creating greeting
Hi Swakkhar
```

Identity of the target

```
help(return_greeting)
```

```
Help on function wrapper_do_twice in module __main__:

wrapper_do_twice(*args, **kwargs)
```

```
import functools

def do_twice(func):
    @functools.wraps(func)
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice
```

Usecase 1: Timing of Functions

```
import functools
import time
def timer(func):
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()
        value = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Finished {func.__name__}() in {run_time:.4f} secs")
        return value
    return wrapper_timer
@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([number**2 for number in range(10_000)])
waste_some_time(1)
waste_some_time(999)
```

Usecase 2: Debugging

```
import functools
def debug(func):
    @functools.wraps(func)
    def wrapper_debug(*args, **kwargs):
        args_repr = [repr(a) for a in args]
        kwargs_repr = [f"{k}={repr(v)}" for k, v in kwargs.items()]
        signature = ", ".join(args_repr + kwargs_repr)
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__}() returned {repr(value)}")
        return value
    return wrapper_debug
@debug
def fibonacci(n):
    if n==1 or n==0:
        return n
    else:
        return fibonacci(n-1)+fibonacci(n-2)
print(fibonacci(5))
```

Register of functions

```
PLUGINS = dict()
def register(func):
    """Register a function as a plug-in"""
    PLUGINS[func.__name__] = func
    return func
@register
def say_hello(name):
    return f"Hello {name}"
@register
def be_awesome(name):
    return f"Yo {name}, together we're the awesomest!"

print(PLUGINS)
```

```
{'say_hello': <function say_hello at 0x104aad750>, 'be_awesome
```