L3: Decorator

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

January 14th, 2020

Architecture Research Group

SSE, TGGS

Function as a First-Class Citizen

- What do I mean?

- Essentially, functions can be passed around as if they are values

- Decorator: Functions that takes another function F and produces a function that extends the behavior of F

Let’s Start with Some Basic

Let's Conceptualize a Function

- Takes input argument
- Computes a value based ***only*** on given arguments
- Functions may have side effects:
  - No side effect

```
def triple(x):
    return x*3
```

  - With side effect

```
def say_hello(name):
    pring("Hello, {}".format(name))
```

- **Q:** What are other examples of side effects?

What Can We Do with Functions?

- Many languages allows functions to be passed around
  - Example:

```
def say_hello(name):
    print(“Hello, {}”.format(name))
def say_bye(name):
    print(“Bye {}”.format(name))
def talk_to_kanat(reaction_func):
    reaction_function(“Kanat”)
```

  - Difference between talk_to_kanat(say_hello) vs. talk_to_kanat(bye)
  - This is similar to function pointer in C

Function Inside another Function

- Example:

```
def foo(name):
        def bar(input1):
                print (f”{name}: {input1}”)
        def moo(input2):
                print(name * input2)
```

- Visibility between functions
  - The variable name is visible inside foo, bar and moo
  - How about moo outside of foo?
    - Try it out and tell me the answer

Returning a Function

- Example

```
def foo(multiplier, absolute = False)
        def mult(x):
                return x*multiplier
        def abs_mult(x):
                return abs(x*multiplier)
        if absolute:
                return abs_mult
        else:
                return mult
```

Decorator and Its Usage

Wrapping Functions inside Functions

- Decorator takes a function (F) as an input, and produce a function F' that extends F
- Let's go through an example

```
def tracer(func):
        def perform_trace():
                print("Before func is called")
                func()
                print("After func is called")
        return perform_trace
def say_whatisgoingon():
        print("What is going onnnnnn!?!?!?")
say_whatisgoingon = tracer(say_whatisgoingon)
```

Using Decorator

- Decorator takes a function (F) as an input, and produce a function F' that extends F
- Let's go through an example

```
def tracer(func):
        def perform_trace():
                print("Before func is called")
                func()
                print("After func is called")
        return perform_trace
def say_whatisgoingon():
        print("What is going onnnnnn!?!?!?")
say_whatisgoingon = tracer(say_whatisgoingon)
```

Using Decorator

- Decorator takes a function (F) as an input, and produce a function F' that extends F
- Let's go through an example

```
def tracer(func):
        def perform_trace():
                print("Before func is called")
                func()
                print("After func is called")
        return perform_trace
def say_whatisgoingon():
        print("What is going onnnnnn!?!?!?")
say_whatisgoingon = tracer(say_whatisgoingon)
```

Using Decorator

- Decorator takes a function (F) as an input, and produce a function F' that extends F
- Let's go through an example

```
def tracer(func):
        def perform_trace():
                print("Before func is called")
                func()
                print("After func is called")
        return perform_trace
def say_whatisgoingon():
        print("What is going onnnnnn!?!?!?")
say_whatisgoingon = tracer(say_whatisgoingon)
```

Using Decorator

- Decorator takes a function (F) as an input, and produce a function F' that extends F
- Let's go through an example

```
def tracer(func):
        def perform_trace():
                print("Before func is called")
                func()
                print("After func is called")
        return perform_trace
def say_whatisgoingon():
        print("What is going onnnnnn!?!?!?")
say_whatisgoingon = tracer(say_whatisgoingon)
```

Using Decorator

- Decorator takes a function (F) as an input, and produce a function F' that extends F
- Let's go through an example

```
def tracer(func):
        def perform_trace():
                print("Before func is called")
                func()
                print("After func is called")
        return perform_trace
def say_whatisgoingon():
        print("What is going onnnnnn!?!?!?")
say_whatisgoingon = tracer(say_whatisgoingon)
```

Confused???

Let’s Do a More Complex Example

Using Decorator – Example 2

- Let's use a decorator to report how long a certain function call takes
- Assume the same say_whatisgoingon function

```
def tracer(func):
        def perform_trace(*args):
                from time import time, sleep
                func_name = func.__name__
                print(f"[LOG] Call: {func_name}{args}")
                start = time()
                val = func(*args)
                done = time()
                print(f"[LOG] Call: return in {(done-start):.3}s')
                return val
        return perform_trace
```

Using Decorator – Example 2

- Let's use a decorator to report how long a certain function call takes
- Assume the same say_whatisgoingon function

```
def tracer(func):
        def perform_trace(*args):
                from time import time, sleep
                func_name = func.__name__
                print(f"[LOG] Call: {func_name}{args}")
                start = time()
                val = func(*args)
                done = time()
                print(f"[LOG] Call: return in {(done-start):.3}s')
                return val
        return perform_trace
```

Using Decorator – Example 2

- With example 2, you can call say_whatisgoingon = tracer(say_whatisgoingon)
- You can also use the tracer to time other functions
  - Why?

```
def tracer(func):
    def perform_trace(*args):
        from time import time, sleep
        func_name = func.__name__
        print(f"[LOG] Call: {func_name}{args}")
        start = time()
        val = func(*args)
        done = time()
        print(f"[LOG] Call: return in {(done-start):.3}s')
        return val
    return perform_trace
```

Using Decorator with @ in python

- You can use the symbol @ to decorate a function

```
@decorator_name
def func():
        do_something

is the same as

def func():
        do_something
func = decorator_name(func)
```

Using Decorator – Example 3

- What does this do?

```
def repeat_shell(func):
        import functools
        @functools.wraps(func)
        def repeated_func(*args, **kwargs):
                results = []
                for _ in range(2):
                        results.append(func(*args, **kwargs))
                return tuple(results)
        return repeated_func
return repeat_shell
```

Before We Leave Today

In-class Exercise

- Write the annotation @repeat(n=…) that cause the wrapped function to repeat n times
- Write the annotation @track_calls(history=…) that causes, for each call to the function, its parameter tuple to be appended to the history list
  - I.e., track the series of {func,arguments} in the history lists