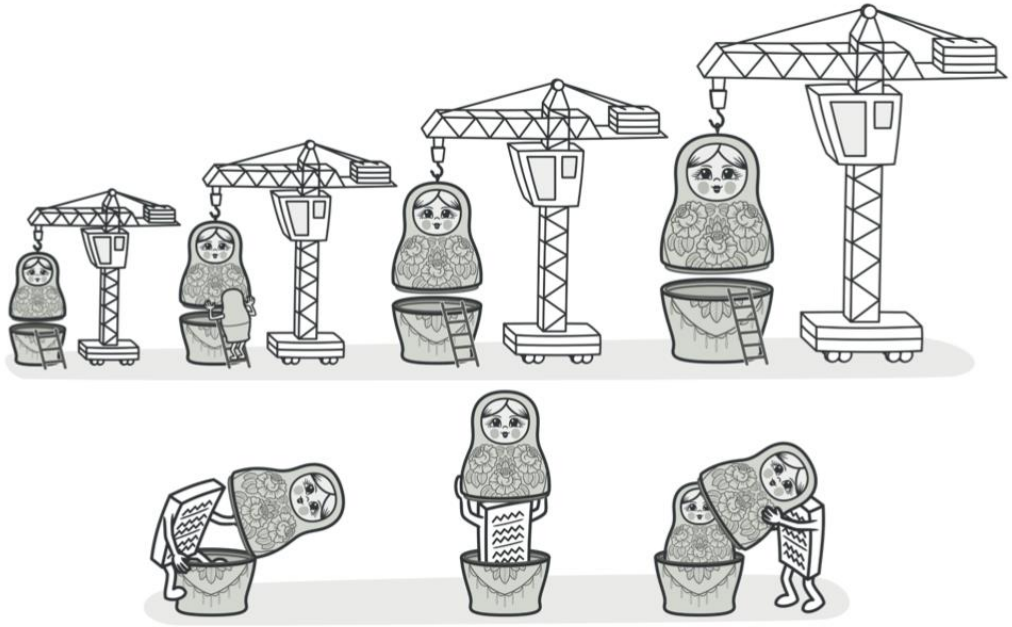
Lecture 10: Design Patterns

Decorator

Decorator Design Pattern

Also known as: Wrapper

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



Tools we use

Pass Function as Argument

We can pass a function as an argument to another function in Python. For Example,

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

def calculate(func, x, y):
    return func(x, y)

result = calculate(add, 4, 6)
print(result)  # prints 10
```

Tools we use

In the above example, the calculate() function takes a function as its argument. While calling calculate(), we are passing the add() function as the argument.

In the calculate() function, arguments: func, x, y become add, 4, and 6 respectively.

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

def calculate(func, x, y):
    return func(x, y)

result = calculate(add, 4, 6)
print(result)  # prints 10
```

Tools we use

In Python, we can also return a function as a return value.

Tools we use

In the above example, the return hello statement returns the inner hello() function. This function is now assigned to the greet variable.

That's why, when we call greet() as a function, we get the output.

```
def greeting(name):
    def hello():
        return "Hello, " + name + "!"
    return hello

greet = greeting("Atlantis")
print(greet())  # prints "Hello, Atlantis!"

# Output: Hello, Atlantis!
```

Python Decorator

A Python decorator is a function that takes in a function and returns it by adding some functionality.

In fact, any object which implements the special __call__() method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.

Basically, a decorator takes in a function, adds some functionality and returns it.

Decorator

Function returns other function

```
def make_pretty(func):
    # define the inner function
    def inner():
        # add some additional behavior to decorated function
        print("I got decorated")

        # call original function
        func()
    # return the inner function
    return inner

# define ordinary function
def ordinary():
    print("I am ordinary")
```

Python decorator

```
def make_pretty(func):

    def inner():
        print("I got decorated")
        func()
    return inner

@make_pretty
def ordinary():
    print("I am ordinary")

ordinary()
```

Decorating functions with Parameters

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like:

```
def divide(a, b):
    return a/b
```

This function has two parameters, a and b. We know it will give an error if we pass in b as 0.

Now let's make a decorator to check for this case that will cause the error.

Decorating Functions as Parameters

```
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a, b)
    return inner

@smart_divide
def divide(a, b):
    print(a/b)

divide(2,5)

divide(2,0)
```

Decorating functions with parameters

Here, when we call the divide() function with the arguments (2,5), the inner() function defined in the smart_divide() decorator is called instead.

This inner() function calls the original divide() function with the arguments 2 and 5 and returns the result, which is 0.4.

Similarly, When we call the divide() function with the arguments (2,0), the inner() function checks that b is equal to 0 and prints an error message before returning None.

Chaining Decorators

Multiple decorators can be chained in Python.

To chain decorators in Python, we can apply multiple decorators to a single function by placing them one after the other, with the most inner decorator being applied first.

Chaining decorators

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 15)
        func(*args, **kwargs)
        print("*" * 15)
    return inner


def percent(func):
    def inner(*args, **kwargs):
        print("%" * 15)
        func(*args, **kwargs)
        print("%" * 15)
    return inner


@star
@percent
def printer(msg):
    print(msg)

printer("Hello")
```

In-Class Example

We have an external class that can send a slack notification:

```
class SlackClient:
    def __init__(self):
        ...
    def notify_via_slack(self, msg: str):
        ...
```

In-Class Example

We want to notify slack each time we push to production. That is, whenever the function push_to_prod() is called, we want the SlackClient to notify.

push_to_prod is called all over the codebase, so we want to just decorate it rather than change everywhere it's used!

Design a decorator for notifying and add it to the push_to_prod function using the @ notation.