Design Patterns: Decorator

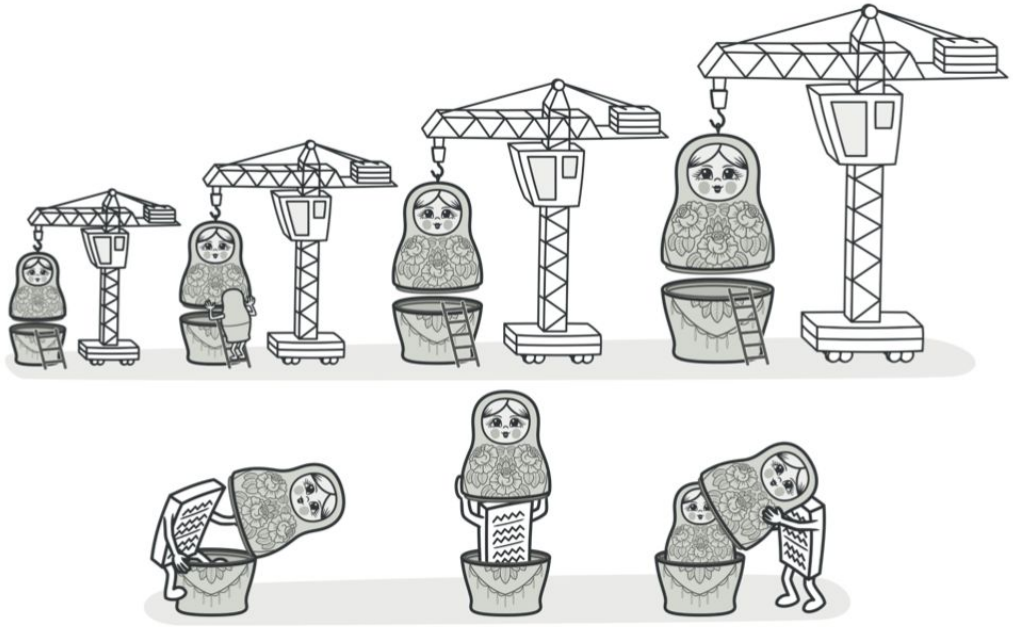
...

Decorator Design Pattern

Also known as: Wrapper

Decorator is a structural design pattern

Attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



Advanced Python: *args and *kwargs

In Python, we can pass a variable number of arguments to a function using special symbols. There are two special symbols:

`*args` (Non Keyword Arguments)
`**kwargs` (Keyword Arguments)

Use these when we are unsure about the number of arguments to pass in the functions.

Advanced Python: *args and *kwargs

- `*args` allows us to pass a variable number of **non-keyword** arguments to a function
- In the function definition, we use an asterisk * before the parameter name.
- The arguments are passed as a **tuple** and these passed arguments are accessed as a tuple inside the function with same name as the parameter excluding asterisk *
- Cannot use `*args` for keyword args

Advanced Python: *args and *kwargs

- `**kwargs` allows us to pass a variable length of keyword arguments to the function.
- In the function definition, we use the double asterisk `**` before the parameter name to denote this type of argument.
- The arguments are passed as a dictionary and these arguments make a dictionary inside function with name same as the parameter excluding double asterisk `**`.

Advanced Python: Python Typing

```
def greeting(name: str) -> str:
    response: str = 'Hello ' + name
    return response
```

Passing Functions as Arguments

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
```

Passing Functions as Arguments

- In the 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, 6`

```
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
```

Returning Functions

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

Returning Functions

- In the example:
- `return hello`
  - 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, adds some extra behavior to it, and returns the enhanced function.
- 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

The decorator returns a new function which “wraps” the original 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")
```

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)
```

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
- The most **inner** decorator gets applied first.

In-Class Exercise

- Implement the decorators in decorator_exercise.py in the class github
- Fine to work in small groups, as usual
- Hint: you will need args or kwargs ;)
- Show me when done!