Decorators

Functions are first class objects

In Python, functions are first-class objects. This means that functions can be passed around and used as arguments, just like any other object (string, int, float, list, and so on).

```
def say_hello(name):
    return f"Hello {name}"

def greet(name):
    return f"Hi {name}, You are in our Python Course!"

def greet_student(greeter_func):
    return greeter_func("Student")
```

```
greet_student(say_hello)
greet_student(greet)
```

Explanation - Notes

- greet_student(say_hello) refers to two functions, but in different ways: greet() and say_hello.
- The say_hello function is named without parentheses. This means that only a reference to the function is passed.
- The function is not executed. The greet_student() function, on the other hand, is written with parentheses, so it will be called as usual.

- Is a design pattern in Python that allows a user to add new functionality to an existing
  object without modifying its structure.
- Decorators are usually called before the definition of a function you want to decorate.

Simple Decorators

```
def my_decorator(func):
  def wrapper():
    print("Before the function is called.")
    func()
    print("After the function is called.")
  return wrapper
def say_hi():
  print("hi!")
say_hi = my_decorator(say_hi)
```

Explanation

- Say_hi is pointing to wrapper function in my_decorator, Why?
- Return wrapper as a function when you call my_decorator(say_hi), So?
- Wrapper() has a reference to the original say_hi() as func,OK?
- Calls that function between the two calls to print().

Conclusion

Decorators wrap a function, modifying its behavior.

Better way:

Python allows you to use decorators in a simpler way with the @ symbol, sometimes called the “pie” syntax.

```
def my_decorator(func):
  def wrapper():
      print("Before the function is called.")
      func()
      print("After the function is called.")
  return wrapper
@my_decorator
def say_hi():
  print("hi!")
```

Real World Examples

General Formula

```
import functools

def decorator(func):
    @functools.wraps(func)
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```

Timing Functions

creating a @timer decorator. It will measure the <font color='blue'> time a function takes to execute and print the duration to the console.

Steps (Logic)

- We need to find the excution time = end_time - start_time
- Get start time - before running func
- Get end time - after running func
- Get end - start value
- Use time package
- from time package use perf_counter func

```
import functools
import time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        # befor
        start_time = time.perf_counter()    # 1
        # calling function
        value = func(*args, **kwargs)
        # after
        end_time = time.perf_counter()      # 2
        run_time = end_time - start_time    # 3
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer
```

```
@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])
```

```
waste_some_time(1)
```

Note

- The @timer decorator is great if you just want to get an idea about the runtime of your functions
- If you want to do more precise measurements of code, you should instead consider the timeit module in the standard library
- It temporarily disables garbage collection and runs multiple trials to strip out noise from quick function calls