Programming technologies

Decorators

Decorator

A decorator 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.

In Python, functions are **first class objects** which means that functions in Python can be used or passed as arguments.

Properties of first class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

Functions - First Class Objects

```
def increment(number):
    return number + 1

add_one = increment
add_one(5)
```

```
def increment(number):
    return number + 1

def decrement(number):
    return number - 1

def external_func(func):
    # storing the function in a
variable
    result_val = func(1)
    print (result_val)

external_func(increment)
external_func(decrement)
```

Inner Functions

It’s possible to define functions inside other functions. Such functions are called inner functions. Here’s an example of a function with two inner functions:

```
def parent():
    print("Printing from the parent() function")

    def first_child():
        print("Printing from the first_child() function")

    def second_child():
        print("Printing from the second_child() function")

    second_child()
    first_child()
```

Returning Functions From Functions

Python also allows you to use functions as return values. The following example returns one of the inner functions from the outer parent() function:

```
def parent(num):
    def first_child():
        return "Hi, I am Emma"

    def second_child():
        return "Call me Liam"

    if num == 1:
        return first_child
    else:
        return second_child

first_child = parent(1)
second_child = parent(2)

print(first_child())
print(second_child())
```

Decorators

As stated above the decorators are used to modify the behaviour of function or class. In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

```
def logging(func):
    def wrapper():
        print("Starting function...")
        func()
        print("Ending function...")
    return wrapper

def tiny_func():
    print("I am tiny func")

tiny_func = logging(tiny_func)

tiny_func()
```

Decorating Functions With Arguments

The inner function takes the argument as ***args** and ****kwargs** which means that a tuple of positional arguments or a dictionary of keyword arguments can be passed of any length. This makes it a general decorator that can decorate a function having any number of arguments.

```
def logging(func):
    def wrapper(*args, **kwargs):
        print("Starting function...")
        inner_return_val = func(*args, **kwargs)
        print("Ending function...")
        return inner_return_val
    return wrapper

def multiply(val1, val2 = 0):
    return val1 * val2

multiply = logging(multiply)

result = multiply(3, 4)
print(result)
```

Syntactic Sugar

Python allows the using decorators in a simpler way with the @ symbol, sometimes called the “pie” syntax. The following example does the exact same thing as the first decorator example:

```
def logging(func):
    def wrapper(*args, **kwargs):
        print("Starting function...")
        inner_return_val = func(*args, **kwargs)
        print("Ending function...")
        return inner_return_val
    return wrapper


@logging
def multiply_2(val1, val2 = 0):
    return val1 * val2

multiply(5, 6)
```

Example

To evaluate the performance of the programming code, often the **perf_counter()** method of the module **time** uses. Method return the float representation of current time and very useful for resolving such tasks.

```
from time import perf_counter

def timeit(func):
    def wrapper(*args, **kwargs):
        start = perf_counter()
        inner_return_val = func(*args, **kwargs)
        print(perf_counter() - start)
        return inner_return_val
    return wrapper

@timeit
def light_function():
    return [_ for _ in range(1000)]

@timeit
def heavy_function():
    return [_ for _ in range(100000000)]

light_function()
heavy_function()
```

Chain of decoratore

```
from time import perf_counter

def timeit(func):
    def wrapper(*args, **kwargs):
        start = perf_counter()
        inner_return_val = func(*args, **kwargs)
        print(perf_counter() - start)
        return inner_return_val
    return wrapper
```

```
@logging
@timeit
def super_decorated_multiplycation(val1, val2 = 0):
    return val1 * val2

super_decorated_multiplycation(45)
```

```
def logging(func):
    def wrapper(*args, **kwargs):
        print("Starting function...")
        inner_return_val = func(*args,
**kwargs)
        print("Ending function...")
        return inner_return_val
    return wrapper
```

Use Cases of Decorators

Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated. Using decorators in Python also ensures that your code is DRY(Don't Repeat Yourself). Decorators have several use cases such as:

- Authorization in Python frameworks such as Flask and Django
- Logging
- Measuring execution time
- Synchronization

Example

```python
import time

def multiply(val1, val2 = 1):
    time.sleep(1)
    return val1 * val2

def adding(val1, val2 = 0):
    time.sleep(2)
    return val1 + val2

def substraction(val1, val2 = 0):
    time.sleep(3)
    return val1 - val2

def division(val1, val2 = 1):
    time.sleep(15)
    return val1 / val2

start = time.time()
print("Start function execution")
print(multiply(5, 3))
print("Finish function execution")
print(time.time() - start)

start = time.time()
print("Start function execution")
print(adding(5, 3))
print("Finish function execution")
print(time.time() - start)

start = time.time()
print("Start function execution")
print(substraction(5, 3))
print("Finish function execution")
print(time.time() - start)

start = time.time()
print("Start function execution")
print(division(5, 3))
print("Finish function execution")
print(time.time() - start)
```

```python
import time

def logging_time_decorator(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        print("Start function execution")
        result = func(*args, **kwargs)
        print("Finish function execution")
        print(time.time() - start)
        return result
    return wrapper

def multiply(val1, val2 = 1):
    time.sleep(1)
    return val1 * val2

def adding(val1, val2 = 0):
    time.sleep(2)
    return val1 + val2

def substraction(val1, val2 = 0):
    time.sleep(3)
    return val1 - val2

def division(val1, val2 = 1):
    time.sleep(15)
    return val1 / val2


multiply = logging_time_decorator(multiply)
adding = logging_time_decorator(adding)
substraction =
logging_time_decorator(substraction)
division = logging_time_decorator(division)


print(multiply(5))
print(adding(5))
print(substraction(5))
print(division(5))
```

```python
import time

def logging_time_decorator(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        print("Start function execution")
        result = func(*args, **kwargs)
        print("Finish function execution")
        print(time.time() - start)
        return result
    return wrapper

@logging_time_decorator
def multiply(val1, val2 = 1):
    time.sleep(1)
    return val1 * val2

@logging_time_decorator
def adding(val1, val2 = 0):
    time.sleep(2)
    return val1 + val2

@logging_time_decorator
def substraction(val1, val2 = 0):
    time.sleep(3)
    return val1 - val2

@logging_time_decorator
def division(val1, val2 = 1):
    time.sleep(15)
    return val1 / val2

print(multiply(5, 3))
print(adding(5, 3))
print(substraction(5, 3))
print(division(5, 3))
```

```python
import time

def logging_decorator(func):
    def wrapper(*args, **kwargs):
        print("Start function execution")
        result = func(*args, **kwargs)
        print("Finish function execution")
        return result
    return wrapper

def time_decorator(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(time.time() - start)
        return result
    return wrapper

@logging_decorator
@time_decorator
def multiply(val1, val2 = 1):
    time.sleep(1)
    return val1 * val2

@logging_decorator
@time_decorator
def adding(val1, val2 = 0):
    time.sleep(2)
    return val1 + val2

@logging_decorator
@time_decorator
def substraction(val1, val2 = 0):
    time.sleep(3)
    return val1 - val2

@logging_decorator
@time_decorator
def division(val1, val2 = 1):
    time.sleep(15)
    return val1 / val2

print(multiply(5, 3))
print(adding(5, 3))
print(substraction(5, 3))
print(division(5, 3))
```

Useful links

https://wiki.python.org/moin/PythonDecoratorLibrary

https://realpython.com/primer-on-python-decorators/

https://www.geeksforgeeks.org/decorators-in-python/

https://www.datacamp.com/tutorial/decorators-python