Python Decorators to Improve Your Coding

The Problem of Repetitive Patterns

Repetitive patterns are not rare in software development. Logging and/or performance monitoring require similar blocks of code to occur across multiple functions and hence increase the risk of making bugs. Among others, Python offers an elegant solution to this problem – decorators. Python decorators are a powerful tool that allows developers to modify or extend the behavior of functions. Read up to learn about some useful decorators that can help to enhance your Python code.

What Is a Decorator?

Decorator in Python is a way to modify functions without changing its original code. Decorator is essentially a function that takes another function as an argument, modifies its behavior, and returns a new function. Here is a very simple example:

```
def simple_decorator(func):
    def wrapper():
        print("Before the function call")
        func()
        print("After the function call")

    return wrapper

@simple_decorator
def say_hello():
    print("Hello Decorator!")

say_hello()
```

The result of this code execution:

```
Before the function call
Hello Decorator!
After the function call
```

Naturally, decorators can also be used to modify functions that have arguments:

```
def args_decorator(func):
        def wrapper(*args, **kwargs):
                print(f"Calling '{func. name  }' with argsuments: {args} kwargs: {kwargs}")
                result = func(*args, **kwargs)
                print("Goodbye!")
                return result

return wrapper

@args_decorator
def func_with_arg(p):
        return print(p.title())

func_with_args("decorator for function with  arguments ")
```

The output:

```
Calling func_with_args with arguments: "decorator for function with  arguments "
Decorator For Function With Arguments
Goodbye!
```

Decorators can be connected into a sequence (a chain) so you can apply multiple decorators to a single function. The chain works from the innermost to the outermost. Here is an example:

```
def decorator_one(func):
        def wrapper():
                print("Decorator One") func()
        return wrapper

def decorator_two(func):
        def wrapper():
                print("Decorator Two") func()
        return wrapper

@decorator_one
@decorator_two
def say_goodbye():
        print("Goodbye!")

say_goodbye()
```

The output:

```
I am Decorator One
I am Decorator Two
Goodbye!
```

We have a few practical examples of decorators that can help to make code neat and efficient.

Check these scripts:

- decorator-calls-counter.py
- decorator-check-type.py
- decorator-debugger.py
- decorator-time-logger.py

Counting Numbers of Function Calls

This decorator wraps the function and increments a counter stored as wrapper.calls on each call, and then calls the original function.

```
def count_calls(func):
    def wrapper(*args, **kwargs):
        wrapper.calls += 1
        return func(*args, **kwargs)
    wrapper.calls = 0
    return wrapper

@count_calls
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
greet("Bob")
greet("Zena")

print(f"'greet' was called {greet.calls} times.")
```

The output:

```
Hello, Alice!
Hello, Bob!
Hello, Zena!
'greeting()' was called 3 times.
```

Arguments Type Check

Python's type hints are informative, but they do not enforce a type at runtime. This decorator ensures that functions are called with the correct argument types.

```
def type_check(*expected_types):
        def decorator(func):
                def wrapper(*args, **kwargs):
                        for arg, expected in zip(args, expected_types):
                                if not isinstance(arg, expected):
```

```
                raise TypeError(f"Expected {expected}, got {type(arg)}")

            return func(*args, **kwargs)

        return wrapper

    return decorator

@type_check(int, int)
def add(a, b):
    return a + b

# usage
print(add(2, 3))

# this throws an exception
print(add('two', 'three'))
```

Logging and Debugging

Decorators can be very useful when we need a log of function calls to measure time execution or count number of calls. Also, a decorator can log input arguments and return values, helping with debugging the code.

```
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Calling '{func.__name__  }' with args: {args} kwargs: {kwargs}") result = func(*args, **kwargs)
        print(f"'{func. name  }' returns: {result}")
        return result

    return wrapper

@log_decorator
def multiply(x, y):
    return x * y

multiply(21, 2)
```

The output:

```
Calling multiply with args: (21, 2), kwargs: {},
multiply returns: 42
```

Measuring Function's Execution Time

Very often we need to track much time a function takes to do its work. Here is a pythonic solution for this task: the decorator time logger.

```
"""
This script is an example of using decorators to measure function execution time
"""
import time

def log_time_decorator(func):
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        end = time.perf_counter()

        print(f"Call of '{func.__name__()}' took {end - start:.3f} sec")
        return result

    return wrapper


@log_time_decorator
def run_and_count():
    # execute a long counting loop: 10 mln multiplications
    for x in range(1, 10000000):
        r = ((x * 2) * 3) / x

run_and_count()
```