The Power of Decorators

@Matt Dinh - Becamex IDC

1. Python Decorator Basics

decorators.py

```
>>> def my_decorator(func):
...     def wrapper():
...         print("Before the function is called.")
...         func()
...         print("After the function is called.")
...     return wrapper

>>> def say_hello():
...     print("Hello!")

>>> greeting = my_decorator(say_hello)
>>> greeting()
Before the function is called.
Hello!
After the function is called.
```

Discover the fundamentals of decorators as higher-order functions.

In basic terms, a decorator is a callable that takes a callable as input and returns another callable.

Python's decorators allow you to extend and modify the behaviour of a callable (functions, methods, classes) without permanently modifying the callable itself.

@Matt Dinh - Becamex IDC

2. Decorators Can Modify Behavior

decorators.py

```
>>> import time
>>> def timing_decorator(func):
...     def wrapper():
...         start_time = time.time()
...         func()
...         end_time = time.time()
...         print(f"Execution took {end_time - start_time} seconds.")
...     return wrapper

>>> @timing_decorator
... def slow_function():
...     time.sleep(2)
...     print("Slow function has finished.")

>>> slow_function()
Slow function has finished.
Execution took 2.0032544136047363 seconds.
```

Decorators can be used to change the behavior of functions, adding functionalities like logging, memoization, and access control in a clean and reusable way.

This decorator measures the execution time of the slow_function, showcasing the additional functionality provided by decorators.

Use the @ syntax is syntactic sugar and a shortcut for this commonly used pattern.

@Matt Dinh - Becamex IDC

3. Applying Multiple Decorators to a Function

decorators.py

```
>>> def bold_decorator(func):
...     def wrapper():
...         return f"<b>{func()}</b>"
...     return wrapper

>>> def italic_decorator(func):
...     def wrapper():
...         return f"<i>{func()}</i>"
...     return wrapper

>>> @bold_decorator
... @italic_decorator
... def hello():
...     return "Hello, world!"

>>> hello()
<b><i>Hello, world!</i></b>
```

Stack decorators to achieve more complex functionality. Apply multiple decorators in the correct order to enhance your functions even further!

@Matt Dinh - Becamex IDC

4. Decorating Functions that accept arguments

decorators.py

```
>>> def print_args_decorator(func):
...     def wrapper(*args, **kwargs):
...         print(f"Positional arguments: {args}")
...         print(f"Keyword arguments: {kwargs}")
...         return func(*args, **kwargs)
...     return wrapper

>>> @print_args_decorator
... def my_function(a, b, c=0, d=0):
...     return a + b + c + d

>>> my_function(1, 2, c=3, d=4)
Positional arguments: (1, 2)
Keyword arguments: {'c': 3, 'd': 4}
10
```

How do you decorate a function that takes arbitrary arguments?

This is where Python's *args and **kwargs feature for dealing with variable numbers of arguments comes in handy.

@Matt Dinh - Becamex IDC

5. How to write "debuggable" decorators?

decorators.py

```
>>> import functools

>>> def logging_decorator(func):
...     @functools.wraps(func)
...     def wrapper(*args, **kwargs):
...         print(f"Calling function '{func.__name__}'  \
                    with args: {args} and kwargs: {kwargs}")
...         result = func(*args, **kwargs)
...         return result
...     return wrapper

>>> @logging_decorator
... def greet(name, greeting="Hello"):
...     """Greets a person with the given greeting."""
...     return f"{greeting}, {name}!"

>>> greet("John", greeting="Hi")
Calling function 'greet' with args: ('John',) \
and kwargs: {'greeting': 'Hi'}
'Hi, John!'

>>> print(f"Function name: {greet.__name__}")
Function name: greet
>>> print(f"Function docstring: {greet.__doc__}")
Function docstring: Greets a person with the given greeting.
```

When you use a decorator, really what you're doing is replacing one function with another.

One downside of this process is that it hides some of the metadata attached to the original (undecorated) function.

The @functools.wraps(func) decorator is used to preserve the metadata (such as the function's name, docstring, and module) of the original function when creating a decorator.

When we call the decorated greet function, the logging information is printed, and the metadata (function name and docstring) is preserved.

@Matt Dinh - Becamex IDC

Thanks for your supporting!

Follow me for more amazing content.

@Matt Dinh - Becamex IDC