**🧠 Function Copying, Closures & Decorators, in Python — Explained Like You’re Chatting Over Coffee**

If you’ve ever found yourself scratching your head while trying to wrap your brain around ***closures***, ***decorators,*** or why Python lets you pass functions around like trading cards, you’re not alone.

These concepts are cornerstones of Python’s flexibility. They can seem mysterious at first, but once you get them, you’ll feel like you've unlocked a new superpower.

In this article, we’ll demystify:  
- ✅ What closures are?  
- ✅ How function copying works!!!  
- ✅ How decorators work under the hood!!!

**# 📦 First up: Functions are first-class citizens in Python**

In Python, functions are ***first-class citizens***  — meaning they can be *assigned to variables, passed as arguments, store them in data structures*, *returned from other functions and yes* **copy** *them.*

* Yep, just like strings, integers, or lists, if we look at the below code.

def greet(name):  
 return f'Hello, {name}!'  
  
say\_hello = greet   
'''  
 - Assigning the function name 'greet' to a variable 'say\_hello'  
 - Note no parentheses () at the end of the function name  
'''  
  
print(say\_hello('Anshu'))

What just happened here? We actually didn’t “**copy**” the function, but rather **created another reference** to the same object.

A black screen with white text

AI-generated content may be incorrect.

*Python functions are* ***objects****. Assigning a function to another variable doesn’t duplicate it — it just references the same object in memory.*

**📋 When Would You Want to Copy a Function?**

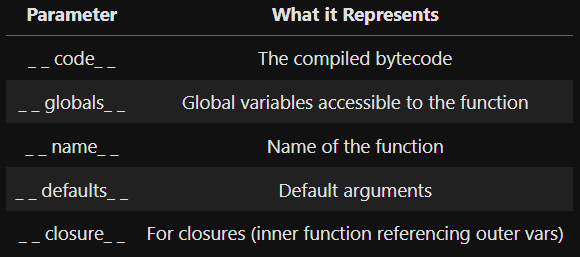
* You want to **modify** a copy without affecting the original (e.g., for monkey-patching or decorators).
* You need to clone behavior across modules or test environments.
* You’re dynamically generating or modifying functions.

**Types Copy Example**

If you *really* need to copy a function:

import types  
  
def greet(name):  
 return f'Hello, {name}!'  
  
clone = types.FunctionType(  
 greet.\_\_code\_\_,   
 greet.\_\_globals\_\_,   
 name=greet.\_\_name\_\_,   
 argdefs=greet.\_\_defaults\_\_,   
 closure=greet.\_\_closure\_\_  
)  
  
print(clone("Anshu")) # Hello, Anshu!  
print(f'{clone is greet = }') # Clone is greet = False

**🧠 What’s happening here?**



**🏁 Summary**

* Functions in Python are objects — copying them means dealing with references.
* You can use types.FunctionType to create a shallow copy.
* Full deep copies are rare and usually unnecessary.
* When in doubt, prefer **clean redefinition or wrapping** over copying.

**🔐 Closures: Functions with Memory**

Imagine you’re at your favorite coffee shop. You order your usual — an *Espresso Macchiato*. The barista remembers your order and smiles: “The usual?” That memory is exactly what a **closure** is.

*A* ***closure*** *is a function object that “remembers” the variables/values from the enclosing scope where it was created, even after that scope has finished execution / is no longer available.*

**Core Concepts of Closures**

1. **Nested Function**: A closure always involves a function defined inside another function.
2. **Free Variables**: The inner function refers to variables from the outer function.
3. **Returning Functions**: The outer function returns the inner function.
4. **State Preservation**: The returned function “remembers” the environment in which it was created.

def make\_multiplier(factor): # Outer Function  
 def multiply(number): # Inner Function   
 return number \* factor # 'factor' from outer function is remembered  
 return multiply # Inner function being returned   
  
double = make\_multiplier(2) # double is multiply function with factor==>2  
triple = make\_multiplier(3) # triple is multiply function with factor==>3  
  
print(double(5)) # 10 # returns 10 as number==>5 \* factor==> 2 = 10  
print(triple(5)) # 15 # returns 15 as number==>5 \* factor==> 3 = 15

The key concept here is a **closure**. Even though the make\_multiplier function has *finished executing*, the returned multiply function which still remembers the value of factor from its enclosing scope, that was active at the time of creation, even after the outer function has finished executing.

So double remembers factor = 2, and triple remembers factor = 3.

That’s a closure. A function with an inner function that captures variables from the outer function.

Still, want proof that double actually *closed over* a variable?

A black background with blue and green text

AI-generated content may be incorrect.

**Why Closures Are Useful**

* **Data Encapsulation**: Closures provide a way to hide data from the global scope while still making it accessible to specific functions.
* **Creating Function Factories:** As shown in your original example, closures enable creating customized functions.
* **Implementing Decorators:** Closures are fundamental to Python decorators (More on this in next section).

**🎁 Decorators: Fancy Closures That Wrap Functions**

Now that we know what closures are, decorators will feel like a natural extension. ***Decorators*** are a powerful feature in Python that allow you to modify the behavior of functions or classes without permanently changing their source code. Let’s explore them in detail.

**Core Concept**

A decorator is a function that takes another function as an argument, adds some functionality, and returns another function. All of this without altering the source code of the original function.  
Think of it like putting a present inside a gift box. The gift (original function) is still there — but now it’s wrapped in something new.

**Basic Structure**

def decorator\_function(original\_function):  
 def wrapper\_function(\*args, \*\*kwargs):  
 # Code to be executed before the original function  
 result = original\_function(\*args, \*\*kwargs)  
 # Code to be executed after the original function  
 return result  
 return wrapper\_function

**Simple Example**

Here’s a basic example of using decorator, that logs when a function is called. Here we use @log\_function\_call to decorate the greet function to modify the functionality .

def log\_function\_call(func):  
 def wrapper(\*args, \*\*kwargs):  
 print(f"Calling function: {func.\_\_name\_\_}")  
 result = func(\*args, \*\*kwargs)  
 print(f"Function {func.\_\_name\_\_} completed")  
 return result  
 return wrapper  
  
@log\_function\_call  
def greet(name):  
 print(f"Hello, {name}!")  
  
greet("Anshu")

A black background with white text

AI-generated content may be incorrect.

**The @ Syntax**

The @decorator\_name syntax is just syntactic sugar. The below two code snippets are equivalent:

# Using @ syntax  
@log\_function\_call  
def greet(name):  
 print(f"Hello, {name}!")  
  
greet('Anshu')  
print('----------------')  
  
# Without @ syntax (manual decoration)  
def greet(name):  
 print(f"Hello, {name}!")  
greet = log\_function\_call(greet)  
greet('Anshu')

A black screen with white text

AI-generated content may be incorrect.

**Decorators with Arguments**

You can also create decorators that accept arguments:

def repeat(num\_times):  
 def decorator\_repeat(func):  
 def wrapper(\*args, \*\*kwargs):  
 results = []  
 for \_ in range(num\_times):  
 results.append(func(\*args, \*\*kwargs))  
 return results  
 return wrapper  
 return decorator\_repeat  
  
@repeat(3)  
def say\_hello(name):  
 return f"Hello {name}"  
  
print(say\_hello("World"))



**Stacking Decorators**

You can use multiple decorators for a single function. They are applied from bottom to top:

def bold(func):  
 def wrapper(\*args, \*\*kwargs):  
 return f"<b>{func(\*args, \*\*kwargs)}</b>"  
 return wrapper  
  
def italic(func):  
 def wrapper(\*args, \*\*kwargs):  
 return f"<i>{func(\*args, \*\*kwargs)}</i>"  
 return wrapper  
  
@bold  
@italic  
def format\_text(text):  
 return text  
  
print(format\_text("Hello")) # <b><i>Hello</i></b>

**Preserving Function Metadata**

When using decorators, metadata of the original function (like name, docstring) gets lost:

def my\_decorator(func):  
 def wrapper(\*args, \*\*kwargs):  
 return func(\*args, \*\*kwargs)  
 return wrapper  
  
@my\_decorator  
def example():  
 """This is the docstring"""  
 pass  
  
print(example.\_\_name\_\_) # Outputs: wrapper (not example)  
print(example.\_\_doc\_\_) # Outputs: None (not the docstring)  
You can preserve metadata using the functools.wraps decorator:  
from functools import wraps  
  
def my\_decorator(func):  
 @wraps(func) # This preserves metadata  
 def wrapper(\*args, \*\*kwargs):  
 return func(\*args, \*\*kwargs)  
 return wrapper  
  
@my\_decorator  
def example():  
 """This is the docstring"""  
 pass  
  
print(example.\_\_name\_\_) # Outputs: example  
print(example.\_\_doc\_\_) # Outputs: This is the docstring

**Practical Use Cases**

1. **Timing Functions**:

import time  
from functools import wraps  
  
def measure\_time(func):  
 @wraps(func)  
 def wrapper(\*args, \*\*kwargs):  
 start = time.time()  
 result = func(\*args, \*\*kwargs)  
 end = time.time()  
 print(f"{func.\_\_name\_\_} took {end - start:.2f} seconds")  
 return result  
 return wrapper  
  
@measure\_time  
def slow\_function():  
 time.sleep(1)  
 return "Done"  
  
slow\_function()

A black background with white text

AI-generated content may be incorrect.

2. **Caching Return Values**:

import time  
from functools import wraps  
  
def measure\_time(func):  
 call\_depth = 0 # shared state  
  
 @wraps(func)  
 def wrapper(\*args, \*\*kwargs):  
 nonlocal call\_depth  
 is\_outermost\_call = call\_depth == 0  
  
 if is\_outermost\_call:  
 start = time.time()  
  
 call\_depth += 1  
 result = func(\*args, \*\*kwargs)  
 call\_depth -= 1  
  
 if is\_outermost\_call:  
 end = time.time()  
 print(f"[Timing] {func.\_\_name\_\_}({args}) took {end - start:.6f} seconds")  
  
 return result  
  
 return wrapper  
   
def memoize(func):  
 cache = {}  
 @wraps(func)  
 def wrapper(\*args):  
 if args in cache:  
 return cache[args]  
 result = func(\*args)  
 cache[args] = result  
 return result  
 return wrapper

@measure\_time  
def fibonacci(n):  
 if n <= 1:  
 return n  
 return fibonacci(n - 1) + fibonacci(n - 2)  
  
print(fibonacci(35))



@measure\_time  
@memoize  
def fibonacci(n):  
 if n <= 1:  
 return n  
 return fibonacci(n - 1) + fibonacci(n - 2)  
  
print(fibonacci(35)) # Only one timing line!  
print(fibonacci(35)) # Instant return from cache, no timing printed

**Built-in Decorators**

Python has several built-in decorators:

1. **@property**: Transforms a method into a getter for a property
2. **@classmethod**: Converts a method to a class method
3. **@staticmethod**: Converts a method to a static method
4. **@abstractmethod**: Indicates that the method must be implemented by subclasses

Decorators offer a clean and reusable way to modify or enhance the behavior of functions and classes, making your code more modular and following the DRY (Don’t Repeat Yourself) principle.

**🙌 Final Thoughts**

We’ve now covered:

* ✅ Function Copying (deep dive into internals)
* ✅ Closures (functions with memory)
* ✅ Decorators (closures in a fancy outfit)

And we did it with real-world metaphors, detailed code, and just the right amount of nerdy charm.

Got feedback? Want to go deeper into functools, contextlib, generators, classes? Drop a comment — I’m always up for more Python conversations!

— **Anshu**