How Closure works:
When the outer function is called and returns the inner function, the inner function "closes over" the environment of the outer function. This means it retains a reference to the values of the variables
from the outer function's scope, even though the outer function's execution has completed and its local
variables would normally be destroyed. When the returned inner function is later called, it can still
access and use those "remembered" variables.

When you create a **closure** in Python (that “inner function that remembers” variables from the outer function), it *feels* like those outer variables should be gone — because the outer function has finished executing.
Yet somehow, the inner function **still remembers them**.

So the question is:

> “If the outer function is destroyed after execution, how can the inner function still access those variables?”

💡 The real reason: Cell objects & closures

When Python compiles a nested function, it detects that the inner function uses variables from the enclosing scope (but not global).
These variables are **not destroyed** when the outer function finishes — instead, Python stores them in special “cell” objects that the inner function carries around with it.

That’s the key:

> Those variables are moved into a hidden data structure attached to the inner function — a **closure cell**.

So even though the outer function’s *stack frame* disappears, the *values that were needed* live on inside the inner function’s closure.

⚙️ Let’s see it in action

```
def outer():
    x = 10
    def inner():
        print(x)
```

```
    return inner

f = outer()   # outer runs, returns inner
f()           # prints 10
```

Now check the hidden structure:

```
print(f.__closure__)
print(f.__closure__[0].cell_contents)
```

Output:

```
(<cell at 0x...: int object at 0x...>,)
10
```

🟢 The `__closure__` attribute shows the "cells" that hold those remembered variables.
Each cell contains the actual object (like `10`) that the inner function uses.

🔬 What really gets destroyed?

- The *stack frame* of `outer()` (its temporary call frame) is destroyed.
- But the *objects* (like `x`) that the inner function references are **kept alive**, because the inner function's closure holds references to them.

Python's garbage collector sees that the inner function still refers to those objects, so it doesn't free them.

🧩 In simple words

Think of it like this:

> When `outer()` returns `inner`, Python says —
> "Hey, inner will still need `x` later, so let's pack `x` into its backpack." 🎒

Now even though `outer()` itself is gone, that backpack travels with `inner()` everywhere.
So when you call `f()`, Python opens the backpack, finds `x = 10`, and uses it.

🧩 Step 1: Define the function

```
def outer():
    x = 10
    def inner():
        print(x)
    return inner
```

When Python reads this, it **doesn't execute** anything yet — it just creates a function object `outer`.

You can check that:

```
print(outer)
```

Output:

```
<function outer at 0x7f...>
```

🧩 Step 2: Call `outer()`

```
f = outer()
```

Now this is where magic begins.
Let's break down what happens *inside memory.*

🧠 Inside memory when `outer()` runs:

Object	Location (id)	Contents	Notes

`x`	e.g. `0x1001`	`10`	Local variable in `outer()`
`inner`	e.g. `0x1002`	`<function inner>`	Has a reference to `x`

So far:

```
outer frame (temporarily active)
 ├── x → [10]
 └── inner() → references x
```

🧩 Step 3: `outer()` finishes and returns `inner`

When `return inner` runs, the **stack frame of `outer()`** will soon be destroyed (normally this would also destroy its local variables, including `x`).

But — before destruction — Python notices:

> "Wait, this inner function still needs `x`. I can't destroy it yet."

So Python **creates a closure cell** to store that value safely and attaches it to `inner`.

💾 Memory snapshot right after return:

```
outer() has returned – stack frame gone ❌
but...

inner (f) → __closure__ → [ cell → x = 10 ]
```

✅ The variable `x` is now inside a hidden "cell" object that stays alive as long as `f` (the inner function) exists.

🧩 Step 4: Verify it in code

```
def outer():
    x = 10
    def inner():
        print(x)
    return inner

f = outer()
print(f.__closure__)                # Look inside closure
print(f.__closure__[0].cell_contents)
```

Output:

```
(<cell at 0x7f123abc: int object at 0x7f9876>,)
10
```

🧠 Step 5: Call `f()`

```
f()
```

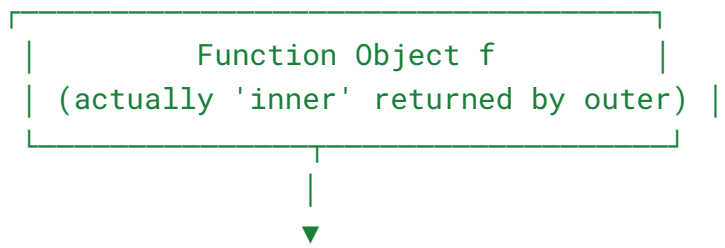
Now Python executes `inner()` — it looks inside the closure backpack 🎒:

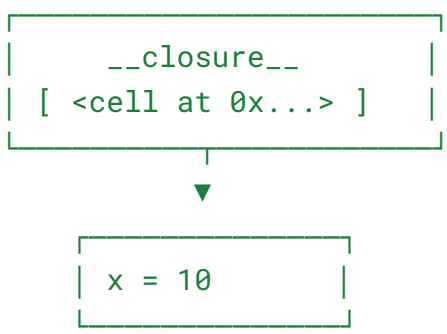
```
closure cell → x = 10
```

It prints:

```
10
```

🧩 Step 6: Visual summary





Even though `outer()` is gone,
the value of `x` **still lives inside** that `cell`,
kept alive by a reference from the inner function.

🔍 Step 7: Proof of memory identity

```
def outer():
    x = [1, 2, 3]
    def inner():
        print("x id inside inner:", id(x))
    print("x id inside outer:", id(x))
    return inner

f = outer()
print("closure id:", id(f.__closure__[0].cell_contents))
f()
```

Output (same IDs):

```
x id inside outer: 140704011453184
closure id:        140704011453184
x id inside inner: 140704011453184
```

✅ The *same memory address* — meaning the variable `x` didn't vanish; it just moved into the closure cell.
