🧠 Step 1: When variables are created

When your Python program runs, each variable or object (like `Car`, `my_tesla`, etc.) is stored in **memory (RAM)**.
Python keeps a **reference count** — meaning it tracks *how many names* are pointing to that object.

Example:

```
my_tesla = ElectricCar("Tesla", "Model S", "85 kWh")
```

Here:

- A new `ElectricCar` object is created in memory.
- The variable name `my_tesla` points to that object.
- Reference count = 1.

⚙️ Step 2: During execution

If you do:

```
other_car = my_tesla
```

Now **two variables** point to the same object → reference count = 2.

When you do:

```
del my_tesla
```

Now reference count = 1 (`other_car` still points to it).

Python's **garbage collector (GC)** automatically deletes objects when:

> reference count drops to 0 (no variable is pointing to that object).

So as long as something references the object, it stays in memory.

🧹 Step 3: Garbage collection

Garbage collection (GC) is triggered **automatically**, but not instantly.

It can happen:

1. **When an object's reference count = 0** → immediate cleanup.
2. **Periodically** — Python also checks for *cyclic references* (like objects referring to each other).
3. **At program shutdown** — final cleanup.

🧾 Step 4: At program termination

When your Python program finishes execution:

- **All namespaces** (like variables, objects, classes, etc.) are destroyed.
- **Reference counts drop to 0**.
- **The garbage collector** does a **final cleanup pass**.
- **Memory is released back to the OS**.

So, to directly answer your question:

> **The garbage collector clears memory *after* program execution ends — during the interpreter's shutdown phase.**

That means:

- Before your program *ends*, variables still exist in memory.
- **As soon as execution ends**, Python starts cleaning up automatically (you don't need to do anything).

💡 Analogy

Think of it like a movie theatre:

- While the movie is playing (program running), people (variables/objects) are in seats (memory).
- When the movie ends, everyone leaves — but cleanup staff (garbage collector) comes in **after** to ensure no one's left and everything's cleared.

🧪 Bonus tip

You can even test this manually:

```
import gc

print(gc.get_stats())    # shows garbage collector stats
gc.collect()             # manually trigger garbage collection
```

You rarely need to do this — Python handles it for you automatically.

✅ **Summary:**

Phase	Variables exist?	Garbage collector active?
During program execution	✅ Yes	✅ Runs as needed
Just before program ends	✅ Still there	🔄 May run
After program ends	❌ All destroyed	✅ Final cleanup

🧩 Example: Understanding Object Deletion and `__del__`

```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
        print(f"✅ Car created: {self.brand} {self.model}")

    def __del__(self):
        print(f"🗑️ Car destroyed: {self.brand} {self.model}")


print("Program started...")

car1 = Car("Tesla", "Model S")
car2 = Car("Tata", "Safari")

print("Both cars created and alive in memory!")

# Delete one object manually
del car1
print("Deleted car1 manually")
```

```
# Program ending — remaining objects will be cleaned up automatically
print("Program ending...")
```

🧠 Output (typical):

```
Program started...
✅ Car created: Tesla Model S
✅ Car created: Tata Safari
Both cars created and alive in memory!
Deleted car1 manually
🗑️ Car destroyed: Tesla Model S
Program ending...
🗑️ Car destroyed: Tata Safari
```

🔍 Explanation:

- The `__del__()` method is called automatically when an object is about to be destroyed.
- When you do `del car1`, Python deletes the name `car1`, and if nothing else points to it, it **calls `__del__()`**.
- When the program ends, the interpreter automatically cleans up all remaining objects (`car2` in this case).

💡 Important Notes:

- `__del__` is also called a **destructor**.
- You **should not rely** on it for important cleanup (like saving data), because:
  - The timing of garbage collection can vary.
  - In some cases (like circular references), it might not trigger immediately.
- Instead, for predictable cleanup, use **context managers** (`with` statements).

⚙️ Optional Experiment

Try running this code:

```
import time


car = Car("Ford", "Mustang")
print("Waiting before deletion...")
time.sleep(3)
del car
print("Done!")
```

You'll see that the object stays alive during the `sleep()` — and only when `del car` executes, the `__del__` message appears.

🧩 Step 1: What is a circular reference?

A **circular reference** happens when **two (or more) objects reference each other**, forming a loop — so their reference counts never reach zero naturally.

Example:

```
A → B

B → A
```

Even if you `del A` and `del B`, both still have one internal reference left — *each other*.

That's why Python's **garbage collector** has a special cycle detector to handle this.

🧠 Step 2: Let's see this in code

```
import gc


class Person:

    def __init__(self, name):

        self.name = name

        self.friend = None

        print(f"✅ Person created: {self.name}")


    def __del__(self):

        print(f"🗑️ Person destroyed: {self.name}")



print("🟢 Program started...")
```

```
# Disable automatic garbage collection for demo clarity
gc.disable()

# Create two people
p1 = Person("Alice")
p2 = Person("Bob")

# Create circular reference
p1.friend = p2
p2.friend = p1

# Delete both variables
del p1
del p2

print("🟡 Deleted both variables manually.")
print("🟡 But notice: __del__ not called yet (because of circular reference)!")

# Manually trigger garbage collection
print("🔵 Now forcing garbage collection...")
gc.collect()
```

```
print("🔴 Program ending...")
```

🧾 Example Output

```
🟢 Program started...

✅ Person created: Alice

✅ Person created: Bob

🟡 Deleted both variables manually.

🟡 But notice: __del__ not called yet (because of circular reference)!

🔵 Now forcing garbage collection...

🗑️ Person destroyed: Bob

🗑️ Person destroyed: Alice

🔴 Program ending...
```

🔍 Explanation:

- Both `Person` objects reference each other (`p1.friend = p2`, `p2.friend = p1`).
- When you `del p1` and `del p2`, the **names** are deleted — but the objects still point to each other → reference count ≠ 0.
- So, the `__del__()` destructor **does not run immediately**.
- When you manually call `gc.collect()`, Python's **cycle detector** finds this circular link, breaks it, and destroys both objects safely.

💡 Key Takeaways

Concept	Meaning
Reference count	Tracks how many names/objects point to another object
Circular reference	Two objects referencing each other, preventing auto-deletion
`gc.collect()`	Manually runs garbage collection cycle detector
`__del__()`	Destructor called only when the object is *really* destroyed

🧩 Bonus tip

You can check how many objects the GC tracked:

```
import gc

print(gc.get_count())  # Shows GC generations info
```