Python

Created	@June 28, 2025 4:34 AM
Tags	

Python

🧹 What is Garbage Collection in Python?

Garbage Collection (GC) in Python is the **automatic process of identifying and reclaiming memory occupied by objects that are no longer in use**, i.e., they are **unreachable**. Python provides a **built-in garbage collector** to manage memory by deallocating unused objects and preventing memory leaks.

✅ Why Garbage Collection is Needed

- Python programs allocate memory dynamically.
- When objects (like lists, dicts, and custom class instances) are no longer needed, they should be removed from memory.
- If not cleared, they cause **memory leaks**, leading to:
    - Poor performance
    - Application crashes
    - System resource exhaustion

Python handles this with:

1. **Reference Counting**
2. **Cycle Detection (Generational Garbage Collection)**

🔁 1. Reference Counting (Primary GC Mechanism)

Every Python object has an internal **reference count** (accessible via `sys.getrefcount()`), which tracks how many references point to it.

🔷 How It Works

- When an object is created: **refcount = 1**
- When a new reference is made, **refcount += 1**
- When a reference is deleted: **refcount -= 1**
- When **refcount == 0**: Object is automatically destroyed

🔷 Example

```
import sys

a = [1, 2, 3]
print(sys.getrefcount(a))  # e.g., 2: one from 'a', one from getrefcount()

b = a
```

```
print(sys.getrefcount(a))  # increased by 1 (b refers to same object)

del a
print(sys.getrefcount(b))  # decreased again
```

When the count reaches **zero**, Python deallocates the memory by calling the object's `__del__` method (destructor), if defined.

🔁 2. Generational Garbage Collection (Handles Cycles)

Reference counting **fails with circular references** (e.g., two objects referring to each other but not used elsewhere). Python uses a **cyclic GC algorithm** based on **generational collection** to handle this.

🔷 Generations

Python divides objects into **three generations**:

- **Gen 0**: New objects
- **Gen 1**: Survived one collection
- **Gen 2**: Survived multiple collections

Rationale: Most objects die young, so check Gen 0 frequently.

🔷 Collection Strategy

- Python periodically scans objects in **Gen 0** for unreachable cycles.
- If enough objects survive, it triggers Gen 1 collection, and so on.

```
import gc

gc.collect()  # Manually trigger GC
```

🔷 Cycle Detection Example

```
import gc

class Node:
    def __init__(self):
        self.ref = None

a = Node()
b = Node()
a.ref = b
b.ref = a  # Circular reference

del a
del b

print(gc.collect())  # Returns number of unreachable objects collected
```

Without `gc.collect()`, this cycle may persist if not enough pressure occurs for a GC pass.

🔍 Internals: How Python Tracks Objects

- All objects are tracked using **doubly linked lists** internally.
- GC module maintains a list of **tracked containers** (objects that may participate in cycles).
- Immutable objects like numbers and strings are **not tracked** because they can't participate in cycles.

📦 `gc` Module — Garbage Collector Interface

Python's `gc` module allows introspection and control over the garbage collector.

```
import gc

gc.enable()        # Enable automatic GC (default)
gc.disable()       # Disable GC
gc.collect()       # Force collection
gc.get_count()     # (count0, count1, count2) → objects in each generation
gc.get_stats()     # Get collection statistics
gc.set_threshold(700, 10, 10)  # Customize thresholds for generations
```

⚠️ Caveats & Best Practices

- **Don't rely entirely on GC**: Use `with` statements and destructors (`__del__`) carefully.
- **Avoid circular references** unless necessary.
- **Use weak references** (`weakref`) to avoid strong cycles.
- Clean up resources explicitly using `context managers` or `finally` blocks.

🧠 Summary

Mechanism	Description
Reference Counting	Automatic deallocation when ref count = 0
Generational GC	Detects circular references using three-generation model
`gc` Module	Interface to inspect, trigger, or tweak GC
Circular Ref Problem	Handled by GC, not by reference counting alone

---

🔐 Global Interpreter Lock (GIL) in Python

🧠 What is the GIL?

The **Global Interpreter Lock (GIL)** is a **mutex (mutual exclusion lock)** in **CPython** (the standard Python implementation) that **allows only one thread to execute Python bytecode at a time**, even if the machine has multiple CPU cores.

🎯 Why Does the GIL Exist?

The GIL was introduced for **simplicity and safety** in CPython's **memory management**, especially around **reference counting**.

Key Reasons:

1. **Thread Safety of Memory Management**

- Python uses **reference counting** for garbage collection.
- Without the GIL, multiple threads updating reference counts simultaneously would require fine-grained locks, complicated and error-prone.

2. **Simplicity**
   - With GIL, CPython avoids adding locks around every low-level operation.
   - Makes the interpreter easier to maintain.
3. **Performance for Single-threaded Code**
   - GIL speeds up single-threaded execution because there's no need to acquire/release locks constantly.

🔥 Impact on Multithreading

❌ CPU-bound Threads: Bad Performance

Python threads **do not run in parallel** if they're executing Python bytecode, due to the GIL.

- Only **one thread runs at a time**, even on multi-core processors.
- This **limits performance in CPU-bound programs**, like:
  - Image processing
  - Number crunching
  - Matrix multiplications

```
# Example: CPU-bound
import threading

def compute():
    for _ in range(10**7):
        pass

threads = [threading.Thread(target=compute) for _ in range(4)]

for t in threads: t.start()
for t in threads: t.join()
```

Even on a 4-core CPU, the threads run **serially**, not in parallel — the GIL is the reason.

✅ I/O-bound Threads: Good Performance

In **I/O-bound programs** (e.g., file I/O, network calls), threads often **wait** for external events.

- The GIL is **released** during blocking I/O operations.
- So **other threads can run while one is waiting**.

```
# Example: I/O-bound
import threading
import time

def wait_io():
    time.sleep(2)
```

```
threads = [threading.Thread(target=wait_io) for _ in range(4)]

for t in threads: t.start()
for t in threads: t.join()
```

This type of workload **benefits from threading** in Python.

⚙️ Workarounds for the GIL

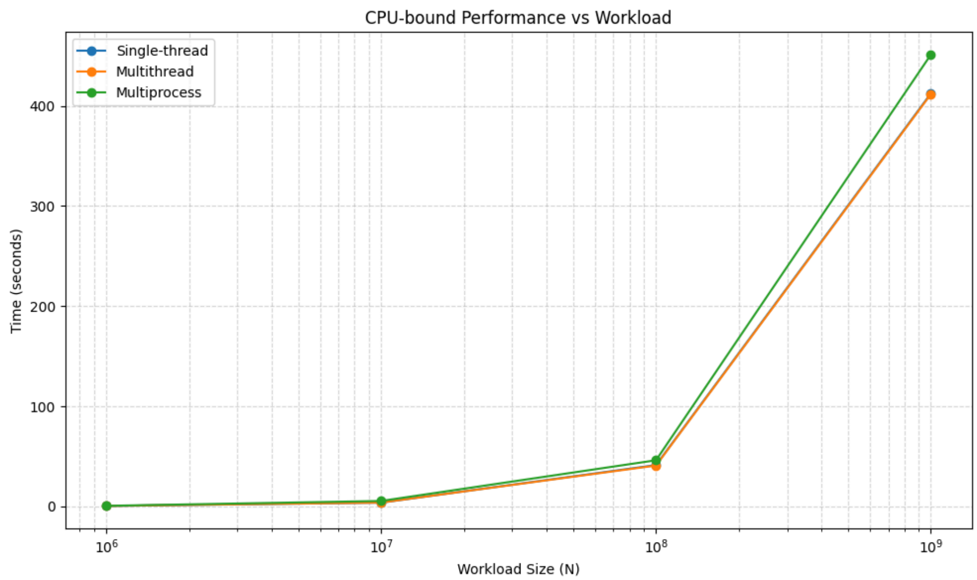
✅ 1. Multiprocessing

- Use `multiprocessing` **module** to create **separate processes**.
- Each process has **its own GIL** and memory space — true parallelism.

```
from multiprocessing import Process

def compute():
    for _ in range(10**7):
        pass

processes = [Process(target=compute) for _ in range(4)]

for p in processes: p.start()
for p in processes: p.join()
```

🧾 Summary

Feature	Description
What	Global lock in CPython to ensure one thread executes at a time
Why it exists	Simplifies memory management, esp. reference counting
Hurts	CPU-bound multithreading performance
Helps	Single-threaded performance, I/O-bound concurrency
Workarounds	Multiprocessing, C extensions, async I/O, alternative interpreters



Python Functions (map, reduce, filter, functools, itertools)

🧰 Python Functional Tools – Detailed Comparison Table

Function / Tool	Purpose	Syntax	Returns	Sample Use Case	Example Output	Notes
`map()` (built-in)	Transform each element in iterable	`map(func, iterable, ...)`	Iterator	Convert list of strings to integers	`['1', '2'] → [1, 2]`	Supports multiple iterables (zips them together)
`filter()` (built-in)	Select elements based on condition	`filter(func, iterable)`	Iterator	Keep even numbers from a list	`[1, 2, 3] → [2]`	Only keeps items where function returns `True`
`reduce()` (functools)	Reduce iterable to a single value	`reduce(func, iterable, initializer)`	Single value	Calculate product of a list	`[1, 2, 3] → 6`	Needs import from `functools`; similar to left-fold
`zip()` (built-in)	Combine iterables element-wise	`zip(iter1, iter2, ...)`	Iterator	Pair names and scores	`['A'], [90] → [('A', 90)]`	Truncates to shortest iterable; use `zip(*iterables)` to unzip
`partial()` (functools)	Fix some arguments of a function	`partial(func, *args, **kwargs)`	Callable function	Create a `square = power(exp=2)`	`square(4) → 16`	Returns a new function with pre-filled arguments
`lru_cache()` (functools)	Cache function results (memoization)	`@lru_cache(maxsize=None)`	Decorator	Speed up recursive Fibonacci	`fib(40)` cached	Great for dynamic programming problems; used on pure functions

`count()` (itertools)	Infinite counter	`count(start=0, step=1)`	Infinite iterator	Create a counting ID generator	`0, 1, 2, 3...`	Must break manually to prevent infinite loop
`cycle()` (itertools)	Infinite loop through iterable	`cycle(iterable)`	Infinite iterator	Loop over `[A, B]` infinitely	`A, B, A, B...`	Useful for round-robin schedulers
`repeat()` (itertools)	Repeat one element	`repeat(elem, times=None)`	Infinite/fixed iterator	Generate 0s or True values	`0, 0, 0...`	Infinite by default; stop with `times=`
`chain()` (itertools)	Flatten nested iterables	`chain(*iterables)`	Iterator	Flatten `[[1,2], [3]]`	`[1, 2, 3]`	Faster and memory efficient than nested loops
`combinations()`	Get r-length combinations	`combinations(iterable, r)`	Iterator of tuples	Select all 2-item combinations from `[1,2,3]`	`(1,2), (1,3), (2,3)`	No repeat, order doesn't matter
`permutations()`	Get all permutations	`permutations(iterable, r)`	Iterator of tuples	Arrange `[1,2,3]` into 2-item permutations	`(1,2), (2,1), ...`	Order matters
`groupby()`	Group consecutive items by key	`groupby(iterable, key=func)`	Iterator of groups	Group 'aaabbcc' into [('a', ...), ('b', ...)]	`[('a',['a','a']), ('b',['b','b']), ...]`	Items must be pre-sorted for correct grouping

Iterators, Generators, Coroutines

🔁 1. Iterables and Iterators

✅ Definitions

Concept	Description
Iterable	An object that can be **looped over** (e.g., list, string, range)
Iterator	An object that produces values **one at a time** using `__next__()`
Iterable Protocol	An object with `__iter__()` that returns an **iterator**
Iterator Protocol	An object with `__next__()` and `__iter__()` returning self

🔧 Custom Iterator Example

```
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
```

```
            raise StopIteration
        else:
            val = self.current
            self.current += 1
            return val

c = Counter(1, 3)
for i in c:
    print(i)  # 1, 2, 3
```

⚙️ 2. Generators

✅ Definition

Generators are **functions that use `yield` to return values lazily** — one at a time — **without storing the entire sequence in memory**.

🧠 Key Features

- Use of `yield` to emit a value
- Suspends state, resumes where left off
- Supports **lazy evaluation**

🧪 Example: Generator Function

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

gen = countdown(3)
for i in gen:
    print(i)  # 3, 2, 1
```

🔄 `yield from` — Delegates to a Sub-generator

```
def wrapper():
    yield from range(3)
    yield from ['a', 'b']

for val in wrapper():
    print(val)  # 0, 1, 2, 'a', 'b'
```

⏳ Lazy Evaluation

Generators are **lazy** — they don't compute values until needed:

```
squares = (x*x for x in range(10))
next(squares)  # 0
```

🔄 3. Coroutines

Coroutines are **generators that can receive data**, and more generally, `async def` **functions that suspend and resume** based on I/O or task scheduling.

🕹️ Generator-based Coroutines (Legacy, pre-3.5)

```
def grep(pattern):
    print("Looking for", pattern)
    while True:
        line = yield
        if pattern in line:
            print(line)

g = grep("python")
next(g)
g.send("hello python")  # Prints: hello python
```

- Requires manual use of `.send()`, `.throw()`, etc.
- Mostly replaced by `async def` / `await`

🚀 Modern Coroutines with `asyncio`

✅ Keywords

Keyword	Purpose
`async def`	Declares a coroutine function
`await`	Suspends coroutine until awaited task completes
`asyncio.run()`	Starts the event loop

🧪 Example: Basic Async Function

```
import asyncio

async def say_hello():
    print("Hello")
    await asyncio.sleep(1)
    print("World")

asyncio.run(say_hello())
```

🕸️ Event Loop & Scheduling

✅ Task Creation Methods

Function	Purpose
`asyncio.run()`	Start main coroutine & event loop
`asyncio.create_task()`	Schedule coroutine execution

`asyncio.gather()`	Run multiple coroutines concurrently
`asyncio.ensure_future()`	Wrap coroutine for manual management

🧪 Concurrent Example

```
import asyncio

async def fetch(id):
    print(f"Start {id}")
    await asyncio.sleep(1)
    print(f"Done {id}")
    return id

async def main():
    results = await asyncio.gather(fetch(1), fetch(2), fetch(3))
    print(results)

asyncio.run(main())
```

Output:

```
Start 1
Start 2
Start 3
Done 1
Done 2
Done 3
[1, 2, 3]
```

⚠️ `ensure_future` vs `create_task`

```
task = asyncio.ensure_future(fetch(1))  # Lower-level, backward-compatible
task = asyncio.create_task(fetch(1))    # Preferred from Python 3.7+
```

📌 Summary Table

Concept	Syntax	Returns	Notes
Iterable	`__iter__()`	Iterator	E.g., list, set, string
Iterator	`__next__()`	Value/StopIter	Produces next item
Generator	`def + yield`	Generator	Lazily produces sequence
yield from	`yield from iterable`	Delegates	Delegates sub-iteration
Coroutine (legacy)	`def + yield` and `.send()`	Generator	Old-style coroutine
Coroutine (modern)	`async def`, `await`	Awaitable	Schedules async I/O and concurrency
Event loop	`asyncio.run(main())`	Starts loop	Required for `await` to run
gather()	`asyncio.gather(coro...)`	Awaitable	Run multiple tasks concurrently
create_task()	`asyncio.create_task(coro)`	Task	Schedule coroutine now
ensure_future()	`asyncio.ensure_future(coro)`	Future/Task	Backward-compatible task wrapper

🔷 1. What Are Decorators?

A **decorator** in Python is a **design pattern** that allows you to **modify or extend** the behavior of a **function or method** without changing its source code. Commonly used with `@decorator_name` syntax.

🔹 Simple Example

```
def my_decorator(func):
    def wrapper():
        print("Before call")
        func()
        print("After call")
    return wrapper

@my_decorator
def greet():
    print("Hello")

greet()
```

🔷 2. Pattern for Writing Decorators

✅ Use `functools.wraps` (Good Practice)

Preserves original function's metadata like `__name__` and `__doc__`.

```
import functools

def decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper
```

✅ Decorator with Arguments

When you want the decorator to **accept parameters**, you need **three layers**:

```
def repeat(n):  # Outer layer (decorator args)
    def decorator(func):  # Middle layer (actual decorator)
        @functools.wraps(func)
        def wrapper(*args, **kwargs):  # Inner layer (wrapper)
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator
```

🔷 3. Different Use Cases

Use Case	Description	Example
Logging	Track function calls	`@log`
Validation	Ensure input meets requirements	`@validate_non_empty`
Authorization	Access control for users	`@require_admin`
Memoization	Cache function results	`@lru_cache`
Profiling	Measure time taken by function	`@timing`
Retry	Re-run failed functions	`@retry(times=3)`
Custom Behavior	Any custom wrap logic	`@my_decorator`

🔷 1. What Are Dunder Methods?

Dunder methods (short for **Double Underscore**) are **special methods** with names like `__init__`, `__str__`, `__len__`, etc. They let you define how your object behaves in **core Python operations**—like instantiation, printing, calling, attribute access, and arithmetic.

They form the core of Python's **Data Model**.

🔷 2. Python Data Model: Lifecycle of an Object

🧬 Object Creation: `__new__()` vs `__init__()`

`__new__(cls, *args, **kwargs)`

- **Static method**.
- Responsible for **creating** a new instance (returns the instance).
- Rarely overridden (except in immutable types like `int`, `str`, `tuple`).

`__init__(self, *args, **kwargs)`

- Initializes attributes of the object.
- Called **after** the object is created.

📌 Example:

```
class Demo:
    def __new__(cls, *args, **kwargs):
        print("Creating instance")
        return super().__new__(cls)

    def __init__(self, name):
        print("Initializing instance")
        self.name = name

obj = Demo("ChatGPT")
```

Output:

```
Creating instance
Initializing instance
```

🔷 3. Attribute Access and Control

Python lets you customize how attributes are accessed, set, and handled when missing:

🔷 `__getattribute__(self, name)`

- Called **every time** an attribute is accessed.
- Must be careful to avoid infinite recursion.

🔷 `__getattr__(self, name)`

- Called **only if the attribute doesn't exist**.

🔷 `__setattr__(self, name, value)`

- Called every time an attribute is assigned.

📌 Example:

```
class AttrDemo:
    def __init__(self):
        self.x = 10

    def __getattribute__(self, name):
        print(f"Accessing {name}")
        return super().__getattribute__(name)

    def __getattr__(self, name):
        print(f"{name} not found")
        return None

    def __setattr__(self, name, value):
        print(f"Setting {name} = {value}")
        super().__setattr__(name, value)

obj = AttrDemo()
print(obj.x)
print(obj.y)  # y doesn't exist
```

🔷 4. Callables and Invocation: `__call__()`

If a class defines `__call__()`, **its instances can be called like functions**.

📌 Example:

```
class Greeter:
    def __init__(self, name):
        self.name = name

    def __call__(self):
        print(f"Hello, {self.name}!")
```

```
g = Greeter("Alice")
g()  # behaves like a function
```

🔷 5. Object Slimming: `__slots__`

By default, Python uses a `__dict__` to store attributes. This is flexible but consumes memory.

Using `__slots__` tells Python to use **a fixed layout** for attributes → faster access, **less memory**.

📌 Example:

```
class Slim:
    __slots__ = ('x', 'y')  # Only allow x and y

    def __init__(self, x, y):
        self.x = x
        self.y = y

s = Slim(1, 2)
# s.z = 3  # ❌ Error: 'Slim' object has no attribute 'z'
```

✅ When to use:

- You have **many objects** of a class.
- You want to **save memory** and prevent new attributes.

🔷 Summary Table

Concept	Method	Purpose
Object Creation	`__new__()`	Creates instance (rarely used)
Object Initialization	`__init__()`	Initializes the instance
Attribute Access	`__getattribute__()`	Always triggered when accessing any attr
Fallback Access	`__getattr__()`	Triggered when attribute doesn't exist
Attribute Setting	`__setattr__()`	Called on any assignment
Callable Instances	`__call__()`	Makes object behave like a function
Memory Optimization	`__slots__`	Restrict attributes, reduce memory
