**Python Preparation**

**Interpreter in Python**

Python is primarily an **interpreted language**, which means:

* The Python **interpreter** reads your code **line by line**, translates it into machine code, and executes it immediately.

1. **You write Python code**

**Print (“hello world”)**

**2.Lexical Analysis (Tokenization)**

* Python breaks the code into **tokens** (smallest units like keywords, identifiers, symbols).
* For example, print("Hello, world!") becomes:
  + print → function name
  + ( → opening parenthesis
  + "Hello, world!" → string literal
  + ) → closing parenthesis

**3. Parsing**

* The interpreter checks if the syntax is correct.
* It builds a **parse tree** (a structure that represents the grammatical structure of the code).

**4. Bytecode Compilation**

* Python **compiles** the parsed code into **bytecode** (a low-level, platform-independent representation).
* This bytecode is stored in .pyc files (in the \_\_pycache\_\_ folder).
* Example bytecode: LOAD\_NAME, CALL\_FUNCTION, etc.

**5. Execution by Python Virtual Machine (PVM)**

* The **Python Virtual Machine (PVM)** reads the bytecode **line by line** and executes it.
* It translates bytecode into **machine-level instructions** that your computer can run.

**6. Output**

* The result is shown immediately (e.g., Hello, world! is printed on the screen).

**Summary Flow**

Your Code (.py)

↓

Lexical Analysis → Parsing

↓

Bytecode Compilation (.pyc)

↓

Python Virtual Machine (PVM)

↓

Execution (Output)

* This allows for **interactive programming** (like using the Python shell or Jupyter notebooks).
* Errors are caught **at runtime**, which can make debugging easier but also means some errors only show up when that part of the code runs.

Examples of Python interpreters:

* **CPython** (the default and most widely used)
* **PyPy** (uses Just-In-Time compilation for faster execution)
* **Jython** (Python running on the Java platform)
* **IronPython** (Python for .NET)

**Compiler in PythonN**

While Python is interpreted, it also uses **compilation** in a limited way:

* Python source code (.py files) is **compiled into bytecode** (.pyc files), which is a lower-level, platform-independent representation.
* This bytecode is then executed by the **Python Virtual Machine (PVM)**.
* This compilation step is automatic and helps improve performance slightly

**What is the Python Virtual Machine (PVM)?**

The **PVM** is the **runtime engine** of Python. It’s a part of the Python interpreter that takes the compiled **bytecode** and **executes** it.

Think of it like a **translator** that reads Python’s intermediate language (bytecode) and tells your computer what to do, step by step.

**How the PVM Executes Bytecode**

Here’s a simplified breakdown:

**1. Bytecode is Loaded**

* When you run a Python script, Python first compiles it into **bytecode** (e.g., print("Hello") becomes a series of bytecode instructions).
* This bytecode is stored in memory or in .pyc files.

**2. PVM Reads Bytecode Instructions**

* The PVM reads each bytecode instruction **one at a time**.
* Each instruction is like a command: e.g., LOAD\_NAME, CALL\_FUNCTION, RETURN\_VALUE.

**3. Execution Stack**

* The PVM uses a **stack-based architecture**.
* It pushes and pops data (like variables, function calls, results) onto a **stack** to keep track of what’s happening.

**4. Performs Operations**

* For each bytecode instruction, the PVM performs the corresponding **low-level operation**:
  + Fetch a value
  + Call a function
  + Perform a calculation
  + Print output
  + Handle exceptions

**5. Manages Memory**

* The PVM also handles **memory management**, including:
  + Creating and destroying objects
  + Reference counting
  + Garbage collection

**6. Produces Output**

* As the instructions are executed, the PVM produces the final result (e.g., prints to the screen, returns a value, etc.).

**Example**

For this code:

The bytecode might look like:

LOAD\_CONST 2

LOAD\_CONST 3

BINARY\_ADD

STORE\_NAME x

LOAD\_NAME print

LOAD\_NAME x

CALL\_FUNCTION 1

The PVM executes each of these instructions in order, using a stack to manage values and operations.

**Summary**

| **Component** | **Role in Execution** |
| --- | --- |
| **Bytecode** | Intermediate code generated by Python |
| **PVM** | Executes bytecode instructions |
| **Stack** | Temporary storage for operations |
| **Memory Manager** | Handles object creation and cleanup |

**2.What is statically and dynamically typed means**

 Dynamically Typed (Python is this)

* Definition: The type of a variable is determined at runtime, not in advance.
* Example:
* Key Points:
  + You don’t need to declare the type of a variable.
  + Python figures it out when the code runs.
  + This makes Python flexible and concise, but can lead to runtime errors if types are misused.

🔹 Statically Typed (e.g., Java, C++)

* Definition: The type of a variable is declared explicitly and checked at compile time.
* Example in Java:
* Key Points:
  + Type errors are caught early, before the program runs.
  + Code is often more verbose but safer in large systems.

✅ Summary

| Feature | Dynamically Typed (Python) | Statically Typed (Java, C++) |
| --- | --- | --- |
| Type checking | At runtime | At compile time |
| Type declaration | Not required | Required |
| Flexibility | High | Low |
| Error detection time | Later (runtime) | Earlier (compile time) |

**3.How memory management works in python**

Memory management in Python is handled automatically through a combination of reference counting and garbage collection, making it easier for developers to focus on writing code without worrying too much about manual memory allocation and deallocation. Here's a breakdown of how it works:

Core Concepts of Python Memory Management

1. Reference Counting

Every object in Python has a reference count, which tracks how many references point to that object. When the reference count drops to zero (i.e., no references to the object remain), Python automatically deallocates the memory.

2. Garbage Collection (GC)

Python uses a garbage collector to handle cyclic references—situations where objects reference each other, forming a cycle that reference counting alone can't resolve.

* The gc module provides tools to interact with the garbage collector.
* It uses generational garbage collection, dividing objects into generations based on how long they've been in memory.

3. Memory Pools (via PyMalloc)

Python uses a specialized allocator called PyMalloc for small objects (less than 512 bytes). This helps reduce fragmentation and improve performance.

Additional Details

* Immutable vs Mutable Objects: Immutable objects (like integers, strings, tuples) are often cached and reused, which affects memory usage.
* Interning: Python may reuse certain immutable objects (like small integers or strings) to save memory.
* Memory Leaks: While rare, they can occur due to lingering references or global variables. Tools like objgraph or tracemalloc can help detect them.

Useful Tools for Monitoring Memory

* gc — garbage collection interface
* sys.getsizeof() — get size of an object
* tracemalloc — track memory allocations
* memory\_profiler — line-by-line memory usage

**4**.**How Garbage Collection Works in Python**

1. Reference Counting

* Every object in Python has a reference count.
* When you assign an object to a variable, the count increases.
* When a reference is deleted or goes out of scope, the count decreases.
* If the count drops to zero, the memory is immediately freed.

2. Cyclic Garbage Collector

Reference counting alone can't handle cyclic references (e.g., two objects referencing each other). Python’s garbage collector detects and cleans up these cycles.

* Python uses a generational garbage collection strategy:
  + Generation 0: New objects.
  + Generation 1: Survived one collection.
  + Generation 2: Long-lived objects.
* Objects that survive collections are promoted to older generations.
* Older generations are collected less frequently.

When Does Garbage Collection Happen?

* Automatically, when thresholds are exceeded.
* Manually, using gc.collect().
* When reference count hits zero (immediate deallocation).

Tools to Monitor or Control GC

* gc.get\_count() — current object count in each generation.
* gc.set\_threshold() — adjust when GC runs.
* gc.get\_objects() — list of all tracked objects.

Why It Matters

Garbage collection:

* Prevents memory leaks.
* Keeps memory usage efficient.
* Helps manage complex object graphs.

**5.What Happens When Reference Count Becomes Zero**

1. Object is Marked for Deletion  
   Python detects that the object is no longer referenced.
2. Memory is Deallocated  
   The memory used by the object is released back to the Python memory manager (not necessarily to the OS immediately).
3. Finalizer (\_\_del\_\_) is Called (if defined)  
   If the object has a \_\_del\_\_() method (a destructor), it is called before the object is destroyed.
4. Garbage Collector May Step In (for Cycles)  
   If the object is part of a reference cycle, the garbage collector (GC) will detect and clean it up later.

Example:-

import sys

a = [1, 2, 3]

print(sys.getrefcount(a))  # e.g., 2 (one from 'a', one from getrefcount argument)

del a  # Reference count drops to 0 → memory is freed

Important Notes

* Immediate deallocation happens only in CPython (the standard Python implementation).
* Other implementations like PyPy may use different memory management strategies (e.g., tracing garbage collection).
* Objects with \_\_del\_\_() can delay garbage collection if they are part of a cycle.

**5**.**How Python Stores an Object in Memory**

1. Object Creation

When you write something like:

Python creates an integer object with the value 42.

2. Memory Allocation

Python uses a memory manager to allocate space for the object in the heap memory (a region of memory used for dynamic allocation).

* The object is represented internally as a PyObject (in CPython).
* It includes metadata like:
  + Type (e.g., int, str)
  + Reference count
  + Value/data

3. Reference Binding

The variable name x is stored in a namespace (a dictionary) and points to the memory address of the object.

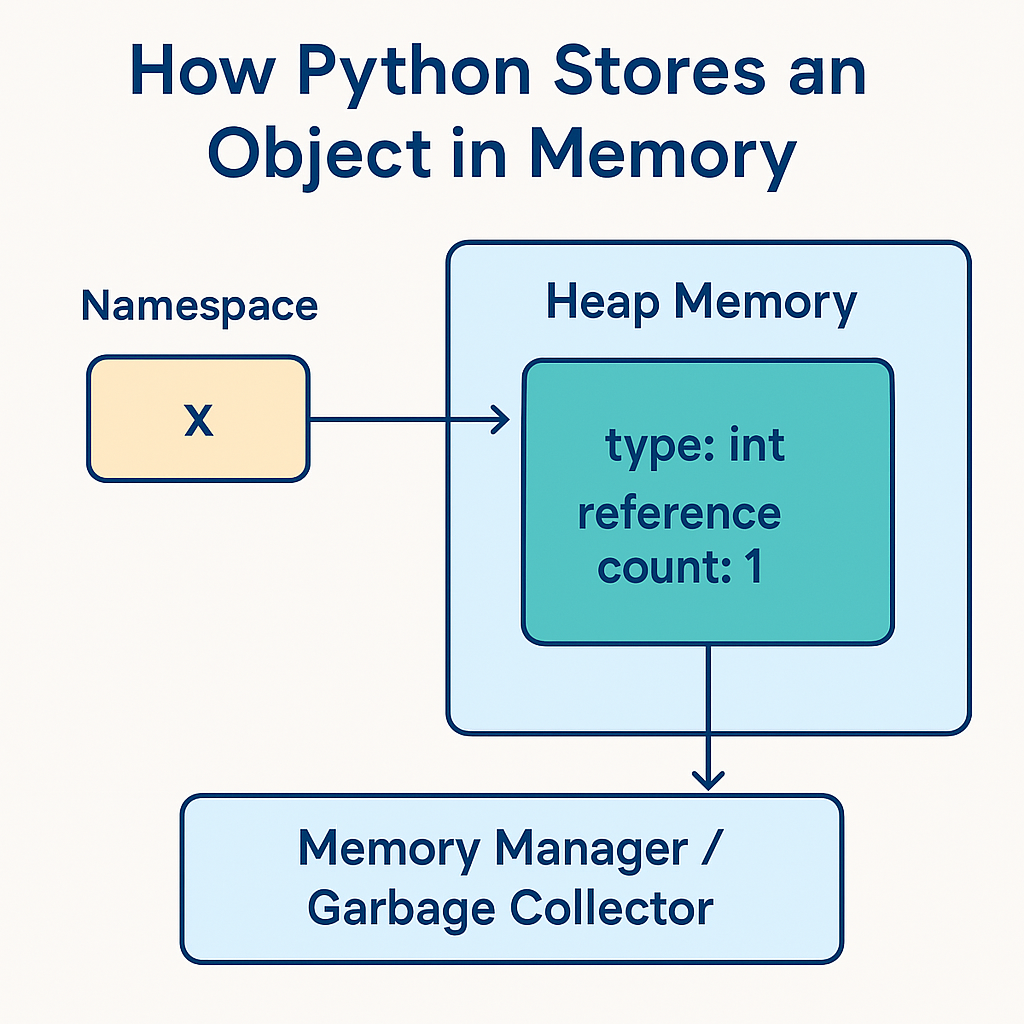
You can check this with:

4. Interning (for Some Objects)

Python reuses memory for certain immutable objects like small integers and short strings to save space.

5. Garbage Collection

When no variable references the object anymore, Python's garbage collector will eventually reclaim the memory.



Visual Summary

| Component | Description |
| --- | --- |
| Heap | Where objects live |
| Namespace | Maps variable names to object references |
| PyObject | Internal structure holding type, ref count, and data |
| Memory Manager | Allocates and deallocates memory |
| Garbage Collector | Cleans up unused objects |

**7.whta is circular reference and how python handles scope.?**

A circular reference occurs when two or more objects reference each other, forming a loop that prevents their reference count from ever reaching zero.

Example:

class A:

    def \_\_init\_\_(self):

        self.b = None

class B:

    def \_\_init\_\_(self):

        self.a = None

a = A()

b = B()

a.b = b

b.a = a  # Circular reference: a → b → a

Even if you del a and del b, the objects still reference each other, so their reference counts don’t drop to zero. Python’s garbage collector detects these cycles and cleans them up using cyclic garbage collection.

You can manually trigger it with:

import gc

gc.collect()

**How Python Handles Scope**

Python uses the LEGB Rule to resolve variable scope:

LEGB = Local → Enclosing → Global → Built-in

1. Local: Names defined inside a function.
2. Enclosing: Names in enclosing functions (for nested functions).
3. **Global**: Names defined at the top-level of a module.
4. **Built-in**: Names in Python’s built-in namespace (e.g., len, print).

Example:

x = "global"

def outer():

    x = "enclosing"

    def inner():

        x = "local"

        print(x)

    inner()

outer()  # Prints "local"

Python searches for x starting from the **local scope**, then **enclosing**, then **global**, and finally **built-in**.

**What Is a Module?**

A **module** is a single Python file (.py) that contains Python code—functions, classes, variables, and runnable code.

**Example:**

**# math\_utils.py**

**def add(a, b):**

**return a + b**

You can import and use this module in another file:

import math\_utils

print(math\_utils.add(2, 3))  # Output: 5

**🔹 Built-in Modules:**

Python comes with many built-in modules like:

* math — mathematical functions
* os — operating system interaction
* sys — system-specific parameters

**What Is a Library?**

A **library** is a collection of modules packaged together to provide specific functionality.

**✅ Example:**

**import numpy as np**

**arr = np.array([1, 2, 3])**

The **NumPy** library contains many modules for numerical computing.

**🔹 Popular Python Libraries:**

* **NumPy** — numerical computing
* **Pandas** — data analysis
* **Matplotlib** — data visualization
* **Requests** — HTTP requests
* **Scikit-learn** — machine learning

**🔄 Summary**

| **Term** | **Description** | **Example** |
| --- | --- | --- |
| **Module** | A single .py file with Python code | math.py |
| **Library** | A collection of modules for a purpose | NumPy, Pandas |

**Step-by-Step: How Python Executes a Module**

**1. Locate the Module**

When you run a script or import a module, Python searches for it in the following order:

* The current directory
* Directories listed in sys.path
* Installed packages (site-packages)

import sys

print(sys.path)  # Shows where Python looks for modules

**2. Compile to Bytecode**

Python compiles the .py file into **bytecode** (a low-level, platform-independent representation).

* This bytecode is stored in .pyc files inside the \_\_pycache\_\_ folder.
* Compilation happens only once unless the source code changes.

**3. Load Bytecode into the Python Virtual Machine (PVM)**

The bytecode is loaded into the **Python Virtual Machine**, which interprets and executes it line by line.

**4. Create a Module Object**

Python creates a **module object** (an instance of types.ModuleType) and assigns it to a name in sys.modules.

import mymodule

print(type(mymodule))  # <class 'module'>

**5. Execute the Module Code**

Python executes the top-level code in the module **only once** (during the first import). This includes:

* Function and class definitions
* Variable assignments
* Any executable statements

**6. Store in sys.modules**

The module is cached in sys.modules so that future imports don’t re-execute the code—they just reuse the existing module object.

import sys

print(sys.modules['mymodule'])  # Shows the cached module

**7. Namespace Creation**

A **namespace** (a dictionary) is created for the module to store all its variables, functions, and classes.

**Re-import Behavior**

If you import the same module again, Python **does not re-execute** it unless you explicitly reload it:

import importlib

importlib.reload(mymodule)