**Python Behind the Seen**

**What Happens When You Run a Python Program?**

**1. Running Python Programs:**

* When you run a Python script (a .py file), Python interprets your code. It doesn’t directly talk to your computer’s hardware like other programming languages do (e.g., C or C++). Instead, it takes your code and translates it into something it can understand internally.

**2. Bytecode:**

* Python doesn’t execute your original code directly. Instead, it first compiles your Python code into bytecode (a lower-level, more efficient version of your code).
* Bytecode is not machine code (which the hardware understands), but it’s Python-specific and is easier to run repeatedly.
* Normally, you don’t see this bytecode because it’s hidden, but when you import a Python file into another file, Python will create a folder called **pycache**. Inside this folder, Python stores the bytecode in a file with the extension .pyc.

**3. Python Virtual Machine (PVM):**

* Python uses something called the Python Virtual Machine (PVM), which is like a small computer program that reads the bytecode and executes it.
* This is why Python is known as an interpreted language — the PVM interprets the bytecode and runs it line by line.
* The PVM is what makes Python run the bytecode on your computer, and it is automatically installed when you download Python.

**4. .pyc Files:**

* A .pyc file is the compiled bytecode version of your Python code.
* These files are stored in the **pycache** folder to help Python run your programs faster in the future, so it doesn’t have to recompile the code every time.

**5.CPython, Jython, etc.:**

* CPython is the standard implementation of Python. When you download Python from the official website, you are downloading CPython.
* Other versions like Jython (for Java), IronPython (for .NET), PyPy (faster version of Python) also exist, but they all work in a similar way — they translate Python code into some intermediate form (like bytecode) and then run it.

**6. Python is not machine code:**

* The bytecode created by Python is not machine code (it cannot run directly on the hardware). It still needs to be interpreted by the PVM (Python Virtual Machine).

**Memory representation**

* always remember that when we create a variable so our data store in memory is firstly data store in memory and then assign any variable.
* All data types are stored in memory. vo variable pa nh jata.
* We never say that this variable is int, string etc.
* always we say the data that store in memory that you are this data type and this variable reference you.

In python not any data type for variable but in memory the reference has data type

**1. What is an Object?**

in Python, everything is an object. Whether it’s a simple integer, a list, or even a function, all data types are objects. This is one of the fundamental principles of Python: everything is an object.

* When you create a variable, Python creates an object for that value in memory.
* a = 10 # 'a' is a reference to an integer object with the value 10
* b = "hi" # 'b' is a reference to a string object with the value "hi"

**2. What is a Reference?**

A reference is like a pointer or a label that points to the location in memory where the object exists.  
- When you assign a value to a variable,  
- that variable is not the object itself,  
- it’s a reference to that object in memory.

* x = 5 # 'x' is a reference to the object 5
* y = x # 'y' is now also a reference to the same object 5

Here, both x and y are references that point to the same object 5 in memory. **Key point:** The variables x and y don't "store" the value directly. They "point"() to the object that holds the value.

**3. Metadata**

metadata is "data about data." It's information that describes or provides more details about other data or objects.

If you have a book, the book itself contains the main data (the story or content). But there’s extra information about the book like:

* The title
* The author
* The publication date
* The number of pages

This extra information is the metadata for the book. It tells you important details about the book but is not part of the story itself.

**In the context of Python objects:**

When we say metadata of an object, we’re talking about details related to the object, like:

* Type of the object (e.g., integer, string, list)
* Size of the object (how much memory it takes up)
* Reference count (how many references point to this object)
* Attributes or methods that belong to the object (functions you can use with the object)

**Garbage collector**

The garbage collector in Python is like a cleanup worker that automatically removes things (objects) from memory when they are no longer needed. This prevents your computer from running out of memory by making sure unused data is thrown away.

**How it works:**

**Reference Counting:** Every object in Python keeps track of how many variables are using it (references). When no variable is using the object anymore (its reference count becomes zero), Python's garbage collector deletes it from memory.

**Cyclic Garbage Collection:** Sometimes, two or more objects point to each other in a loop (a cycle), and no other variable uses them. Python's garbage collector can detect these "cycles" and clean them up to

**1. repr()**

– "Official" or Debug Representation Purpose: The repr() function is used to give an official, unambiguous representation of an object, mainly for debugging purposes.

**Goal:** The idea is that when you call repr(), it shows a string that can ideally be used to recreate the object.

**When to use:** Use repr() when you want a precise or detailed representation of the object, often for debugging.

**repr() output**

print(repr(now)) # Output: datetime.datetime(2024, 10, 5, 12, 0, 0) repr(now) gives you the detailed, official representation (you could recreate the datetime object from this).

**2.str()**

– "Human-Friendly" or Informal Representation Purpose: The str() function is used to give a user-friendly, readable representation of an object.

**Goal:** It’s designed to return a string that’s easy for humans to read.

**When to use:** Use str() when you want to display or print an object in a simple, user-friendly way.

**str(now)** and **print(now)** give you the simple, human-readable version.

**Dictionary**

* we can acess value by get() method

*ex:* **chai\_dic** = [ 'lemon' : 'khati' , 'gengared' : 'karvi'] print(chai\_dic.get('lemon')) *#output: khati*

when we use *POP()* Method for dic so we need to pass key that we wnat to pop like *EX :* chai\_dic.pop('lemon) #khati But we don't wnat to pass key so we use *POPITEM()* Method for pop the complete 1 item in last

**1. What is an Iterator?**

An iterator is an object in Python that allows you to loop over a collection (like a list) one element at a time. It remembers where you are in the process of going through the collection.

**2. Iteration with a List**

When you use a loop (like a for loop) to go over a list, Python doesn’t look at the entire list all at once. Instead, it uses an iterator to go through it one element at a time.

Here’s what happens behind the scenes:

* **Memory:** The entire list is stored in memory as a collection of elements (let's say [10, 20, 30]).
* When the iteration starts (e.g., for item in my\_list:), Python creates an iterator object using iter(my\_list).
* The iterator has a method called **next**(), which retrieves the next element from the list.
* On the first call, **next**() points to the first element (10), returns it, and moves to the next.
* On the second call, it returns 20 and moves to the next.
* This continues until all elements are exhausted. When there are no more elements, it raises a **StopIteration** exception to end the loop.

**3. Iteration with a File**

Iterating over a file is similar, but files work differently in memory.

* **Memory:** Python doesn't load the entire file into memory (especially for large files). Instead, it reads the file line by line.
* When you use a loop (e.g., for line in file:), Python opens the file and creates an iterator.
* This iterator reads the file one line at a time using the **next**() method.
* The memory usage is low because only one line is kept in memory at a time.

**How It Works in Memory:**

* The file isn't fully loaded into memory.
* Python reads one line, processes it, and then moves to the next line when you use next() or iterate with a loop.
* It efficiently handles large files by reading one chunk at a time.

**Summary:**

* In lists, Python loads the entire list into memory, and the iterator moves from one element to the next.
* In files, Python uses an iterator to read one line at a time, minimizing memory usage.
* Iterators keep track of the current position and retrieve the next element or line using the **next**() method.

**1. Iterable:**

* An iterable is anything you can loop over (like a list, string, or file).
* It means the object has multiple elements and you can access them one by one.
* *Example:* Lists ([1, 2, 3]), strings ("hello"), and files are all iterables.

**2. Iterator:**

* An iterator is an object that does the actual work of going through each element in an iterable, one at a time.
* It remembers the current position (where it is in the loop) and moves to the next element when asked (using next()).
* You can create an iterator from an iterable using iter().

**1. File Objects:**

* When you open a file (e.g., f = open("my.py")), the file object (f) is already an iterator.
* This means it can be used directly in a loop without needing to create a separate iterator.
* If you call iter(f), it returns the file object itself.
* So, iter(f) is f is True because the file is already prepared for iteration.

**2. List Objects:**

* A list (e.g., listNum = [1, 2, 3]) is not an iterator but an iterable.
* An iterable can be looped over (like in a for loop), but it doesn't know how to go one item at a time by itself.
* You have to create an iterator using iter(listNum) to loop through the list element by element.
* iter(listNum) is listNum is False because iter(listNum) creates a new iterator object, separate from the list itself.

**Why the Difference?**

**File objects** are already iterators because they are designed to be read sequentially (e.g., line by line). The file object itself manages the state (which line to read next), so you don’t need to create a separate iterator for it.

**Lists,** on the other hand, are iterables but not iterators. You need to create an iterator using iter(listNum) so that you can traverse the list element by element.

**Key Takeaways:**

File objects are already iterators: iter(f) is f. Lists are not iterators, but they are iterables: iter(listNum) creates a separate iterator for the list.