# Python

Note: Python is an object-oriented programming language and fully supports Object-Oriented Programming (OOP) concepts.

1. **Is Python a compiled language or an interpreted language?**

---

Python is actually both compiled and interpreted, depending on which implementation we are talking about. It's important to note that whether a language is compiled or interpreted is not defined by the language itself, but rather by its implementation.
For example, in the most common implementation, **CPython**, the process works in two stages:

- **Compilation stage** – When I run a `.py` file, Python first compiles the source code into an intermediate representation called **bytecode** ( `.pyc` files). This bytecode isn't machine code, but it's a lower-level form that the Python Virtual Machine can understand.
- **Interpretation stage** – The compiled bytecode is then executed by the **Python Virtual Machine (PVM)**, which acts as an interpreter. This is why Python is generally referred to as an interpreted language.
  Other implementations behave differently. For example, **PyPy** uses **Just-In-Time (JIT) compilation**, where bytecode is compiled into machine code at runtime for faster execution. So, in short: Python is usually seen as an interpreted language, but under the hood it goes through both compilation and interpretation depending on the implementation.

2. **How can you concatenate two lists in Python?**

---

We can concatenate two lists in Python using the +operator or the **extend()** method.

- **Using the + operator:** This creates a new list by joining two lists together.

```
a = [1, 2, 3]
b = [4, 5, 6]
res = a + b
print(res) // Output: [1, 2, 3, 4, 5, 6]
```

- **Using the extend() method:** This adds all the elements of the second list to the first list in-place.

```
a = [1, 2, 3]
b = [4, 5, 6]
a.extend(b)
print(a) // Output: [1, 2, 3, 4, 5, 6]
```

3. **Difference between for loop and while loop in Python**

---

- **For loop:** Used when we know how many times to repeat, often with lists, tuples, sets, or dictionaries.
- **While loop:** Used when we only have an end condition and don't know exactly how many times it will repeat.

```
for i in range(5):
    print(i) // 0 1 2 3 4

c = 0
while c < 5:
    print(c)
    c += 1 // 0 1 2 3 4
```

4. **How do you floor a number in Python?**

---

To floor a number in Python, you can use the math.floor() function, which returns the largest integer less than or equal to the given number.

- floor() method in Python returns the floor of x i.e., the largest integer not greater than x.

- Also, The method ceil(x) in Python returns a ceiling value of x i.e., the smallest integer greater than or equal to x.

```
import math

n = 3.7
F_num = math.floor(n)

print(F_num) // Output: 3
```

5. **What is the difference between / and // in Python?**

---

/ represents precise division (result is a floating point number) whereas // represents floor division (result is an integer). For Example:

```
print(5//2) // Output: 2
print(5/2) // Output: 2.5
```

6. **Is Indentation Required in Python?**

---

Yes, indentation is required in Python. Indentations make the code easy to read for developers in all programming languages but in Python, it is very important to indent the code in a specific order.

7. **Can we Pass a function as an argument in Python?**

---

Yes, Several arguments can be passed to a function, including objects, variables (of the same or distinct data types) and functions. Functions can be passed as parameters to other functions because they are objects. Higher-order functions are functions that can take other functions as arguments.

```
def add(x, y):
    return x + y
def apply_func(func, a, b):
    return func(a, b)
```

8. **What is a dynamically typed language?**

---

A dynamically typed language is one where the type of a variable is determined at **runtime**, not at compile time. In other words, I don't need to explicitly declare the data type — Python automatically infers it based on the value assigned. For example:

```
x = 10      # integer
x = "Hello" # now a string
```

- Here, the same variable `x` can hold different types at different points in the program.
- Languages like **Python** and **JavaScript** are dynamically typed, while **C, C++, and Java** are statically typed.
  The main advantage is **ease and speed of development** since I don't have to worry about type declarations. The trade-off is that dynamically typed languages may have more **runtime errors** and are generally **slower to execute** compared to statically typed languages, which perform type checking at compile time.

9. **What is pass in Python?**

---

- The **pass** statement is a **placeholder that does nothing.**
- It is used when a statement is syntactically required but no code needs to run.

```
def fun():
    pass  # Placeholder, no functionality yet
fun() // Output:
```

Here, fun() does nothing, but the code stays syntactically correct.

10. **How are arguments passed by value or by reference in Python?**

---

Python doesn't strictly follow "pass by value" or "pass by reference." Instead, it uses a model called **"Pass by Object Reference"** (sometimes also explained as

*call by sharing*). What this means is:

- When we pass an object to a function, Python passes a **reference to that object**, not the actual object itself.
- Now, whether it *behaves like pass by value* or *pass by reference* depends on whether the object is **mutable** or **immutable**.
  **Immutable objects (like int, str, tuple):**
  If I try to modify them inside the function, a **new object is created** and the original remains unchanged. This feels like *pass by value*.
  **Mutable objects (like list, dict, set):**
  If I modify them inside the function, the **same object gets modified**, so the change reflects outside. This feels like *pass by reference*.

```
def call_by_val(x):
    x = x * 2
    return x

def call_by_ref(b):
    b.append("D")
    return b

a = ["E"]
num = 6

updated_num = call_by_val(num)
updated_list = call_by_ref(a)

print("Updated after call_by_val:", updated_num) // Output: Updated
after call_by_val: 12
print("Updated after call_by_ref:", updated_list) // Output:
Updated after call_by_ref: ['E', 'D']
```

11. **What is a lambda function?**

---

A **lambda function** in Python is simply an **anonymous function** — meaning it doesn't have a name like normal functions defined with `def`. It can take any number of arguments but is restricted to a **single expression**, which is implicitly returned. For example:

```
s1 = 'GeeksforGeeks'

s2 = lambda func: func.upper()
print(s2(s1)) // Output: GEEKSFORGEEKS
```

12. **What is List Comprehension? Give an Example.**

---

List comprehension is a way to create lists using a concise syntax. It allows us to generate a new list by applying an **expression** to each item in an existing **iterable** (such as a **list** or **range**). For example:

```
a = [2,3,4,5]
res = [val ** 2 for val in a]
print(res) // Output: [4, 9, 16, 25]
```

13. **What are * args and** kwargs?**

---

`*args` and `**kwargs` are special syntaxes in Python that allow us to pass a **variable number of arguments** to a function.

- `*args` **(non-keyword arguments):** It lets us pass any number of positional arguments to a function. Inside the function, they’re received as a **tuple**.

```
def fun(*argv):
    for arg in argv:
        print(arg)

fun('Hello', 'Welcome')

// Output:
Hello
Welcome
```

Here, all the arguments are collected into a tuple `argv`.

- `kwargs` (keyword arguments): **It allows us to pass any number of** keyword arguments **to a function. Inside the function, they’re received as a**

dictionary**.

```
def fun(**kwargs):
    for k, val in kwargs.items():
        print(f"{k} == {val}")

fun(s1='Geeks', s2='for')

// Output:
s1 == Geeks
s2 == for
```

14. **What is the difference between a Set and Dictionary?**

---

- A Python Set is an unordered collection data type that is iterable, mutable and has no duplicate elements. Python's set class represents the mathematical notion of a set.
    - **Syntax**: Defined using curly braces {} or the set() function. For example: `my_set = {1, 2, 3}`
- Dictionary in Python is an ordered collection of data values, used to store data values like a map, which, unlike other Data Types that hold only a single value as an element, Dictionary holds **key:value** pair.
    - **Syntax**: Defined using curly braces {} with key-value pairs. For example: `my_dict = {"a": 1, "b": 2, "c": 3}`

15. **What is the difference between a Mutable datatype and an Immutable data type?**

---

- Mutable data types can be edited i.e., they can change at runtime. **Eg** – List, Dictionary, etc.
- Immutable data types can not be edited i.e., they can not change at runtime. **Eg** – String, Tuple, etc.

16. **What is docstring in Python?**

---

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes and methods.

- **Declaring Docstrings:** The docstrings are declared using ”’triple single quotes”’ or “”””triple double quotes””” just below the class, method, or function declaration. All functions should have a docstring.
- **Accessing Docstrings:** The docstrings can be accessed using the **doc** method of the object or using the help function.

17. **How is Exceptional handling done in Python?**

---

There are 3 main keywords i.e. try, except and finally which are used to catch exceptions:

- try: A block of code that is monitored for errors.
- except: Executes when an error occurs in the try block.
- finally: Executes after the try and except blocks, regardless of whether an error occurred. It’s used for cleanup tasks.

18. **What are Modules and Packages in Python?**

---

A module is a single file that contains Python code (functions, variables, classes) which can be reused in other programs. You can think of it as a code library. For example: **math** is a built-in module that provides math functions like sqrt(), pi, etc.

```
import math
print(math.sqrt(16)) // Output: 4.0
```

package is a collection of related modules stored in a directory. It helps in organizing and grouping modules together for easier management.

19. **Differentiate between List and Tuple?**

---

Yes. Both Lists and Tuples are sequence data types in Python, but they have some key differences:

- **Mutability:**
  - **List** → Mutable, meaning elements can be modified, added, or removed.
  - **Tuple** → Immutable, once created its elements cannot be changed.
- **Memory consumption:**
  - **List** → Consumes more memory because it needs to support dynamic operations like insertions and deletions.
  - **Tuple** → Consumes less memory, making it more memory-efficient.
- **Performance:**
  - **List** → Slightly slower in iteration and access compared to tuples.
  - **Tuple** → Faster iteration and access due to immutability.

20. **What are Decorators?**

---

A decorator is essentially a function that takes another function as an argument and returns a new function with enhanced functionality.

21. **What are Iterators and Generators in Python?**

---

An **iterator** in Python is an object that allows us to iterate over a sequence of elements, like lists, tuples, or dictionaries. A **generator** is a simpler way to create iterators. Instead of implementing `__iter__()` and `__next__()` manually, I can write a normal function with a `yield` statement.

22. **How is memory management done in Python?**

---

In Python, memory management is handled automatically by the interpreter. All objects and data structures are stored in a **private heap space** that's managed internally by Python — as developers, we don't access this space directly. Additionally, memory allocation for objects is managed by **Python's memory manager**, which optimizes allocation to reduce fragmentation. Memory management in Python relies on two key mechanisms:

- **Reference Counting:**
  - Every object keeps track of how many references point to it.

    - When the reference count drops to zero, meaning no variable is using the object anymore, the memory occupied by that object is freed.
- **Garbage Collection:**
    - While reference counting handles most cases, it cannot clean up **circular references** (when objects reference each other).
    - To address this, Python includes a **garbage collector** that identifies and removes such cycles, ensuring memory is reclaimed properly.

23. **What is slicing in Python?**

---

Python Slicing is a string operation for extracting a part of the string, or some part of a list. With this operator, one can specify where to start the slicing, where to end and specify the step. List slicing returns a new list from the existing list.
**Syntax:** `substring = s[start : end : step]`

24. **What is PIP?**

---

PIP is an acronym for Python Installer Package which provides an interface to install various Python modules. It is a command-line tool that can search for packages over the internet and install them without any user interaction.

25. **What are Pickling and Unpickling?**

---

- Pickling is the process of converting a Python object into a **byte stream** using the `pickle` module, and Unpickling is the reverse process — converting that byte stream back into the original Python object. The functions commonly used are `pickle.dump()` for pickling and `pickle.load()` for unpickling.
- We use pickling mainly for two reasons:
    - **Persistence** → to save objects (like dictionaries, lists, or even ML models) to a file and load them later.
    - **Data Transfer** → since the object is converted into a byte stream, it can be easily sent over a network or between processes.

26. **What is the difference between @classmethod, @staticmethod and instance methods in Python?**

In Python, methods inside a class can be of three types, depending on how they access data:

- Instance Method → Operates on an instance of the class, has access to instance attributes, and takes `self` as the first parameter. For example:

```
def method(self):
    return self.attribute
```

- Class Method → Operates on the class itself rather than individual objects, takes `cls` as the first parameter, and is defined with `@classmethod`. For example:

```
@classmethod
def method(cls):
    return cls.class_attribute
```

- Static Method → Does not operate on the instance or the class, takes no `self` or `cls` parameter, and is defined with `@staticmethod`. It behaves like a normal function placed inside the class for logical grouping. For example:

```
@staticmethod
def method():
    return "Utility function"
```

In summary, instance methods work with object data, class methods work with class-level data, and static methods are independent utility functions inside a class.

27. **What is init() in Python and how does self play a role in it?**

---

- In Python, the `__init__()` method is a special method that works like a constructor in other object-oriented languages. It's automatically called when a new object of a class is created, and its main purpose is to initialize the attributes of that object with the provided values.
- One important point is that `__init__()` does not handle memory allocation — that part is taken care of by the `__new__()` method, which gets called before

`__init__()`.

- Now, the `self` parameter is crucial here. `self` refers to the specific instance of the class that's being created. By using `self`, we can attach data (attributes) and behavior (methods) to that particular object. That's why `self` must always be the first parameter in instance methods, including `__init__()`.
  For example:

```
class MyClass:
    def __init__(self, value):
        self.value = value   # initialize object attribute

    def display(self):
        print(f"Value: {self.value}")

obj = MyClass(10)
obj.display() // Output: Value: 10
```

Here, `self.value = value` means that the attribute `value` belongs to the object `obj`, not to the class itself. So `self` is what ties the data to each individual object.

28. **Python Global Interpreter Lock (GIL)?**

---

- The Global Interpreter Lock, or GIL, is a mutex in CPython that ensures only one thread executes Python bytecode at a time, even on multi-core processors. This means that, although we can create multiple threads, only one thread can be active in the interpreter at any given moment.
- Because of this, Python's multithreading does not give true parallelism for CPU-bound tasks. In fact, both single-threaded and multi-threaded Python programs often have similar performance when dealing with CPU-heavy computations due to the GIL.
- However, the GIL does not prevent concurrency in I/O-bound tasks — such as file handling, network requests, or database operations — where multithreading can still be beneficial. For CPU-intensive tasks that require real parallelism, we typically use multiprocessing or leverage extensions written in C that can release the GIL.

- So in summary, the GIL simplifies memory management in CPython but is also a limitation when it comes to multi-core parallelism in Python.

29. **What are Function Annotations in Python?**

---

- Function annotations in Python are a way to add **metadata** to function parameters and return values. They allow us to specify the expected input types and the return type of a function, but they don’t enforce type checking by themselves.
- Annotations are just stored in a special attribute called `__annotations__` and have no effect during runtime. Python itself doesn’t attach any meaning to them — instead, they can be interpreted by third-party tools and libraries such as `mypy` for static type checking. For example:

```
def greet(name: str, age: int) -> str:
    return f"Hello {name}, you are {age} years old."

print(greet.__annotations__)
# Output: {'name': <class 'str'>, 'age': <class 'int'>, 'return':
<class 'str'>}
```