PYTHON INTERVIEW QUESTIONS

BEGINNER

1. If you have data with names of products and their prices, which Python data type will you use to store it and why?

Answer:

In Python, we can use a **dictionary** to store this kind of data. The **product name** can be the **key**, and its **price** can be the **value**.

Example:

```
product_prices = {
    "Laptop": 55000,
    "Headphones": 1500,
    "Monitor": 8000
}
```

This structure allows us to quickly find or update the price of any product using its name. Dictionaries are efficient for lookup and are ideal for storing such key-value paired data.

2. What are the main benefits of Python

Answer:

- **Clean & English-like Syntax**
  Easy to read and write. No {} or ;. Great for beginners.
- **Multi-Paradigm Support**
  Supports object-oriented, functional, procedural, and imperative styles.
- **Massive Standard Library & PyPI Ecosystem**
  "Batteries-included" — tools for web, data, ML, automation, etc. Has 300,000+ packages on PyPI.
- **Dynamic & Interpreted**
  No need to declare variable types. Runs line-by-line — fast testing & scripting.
- **Leader in Data Science & AI**
  Top choice for ML, AI, and analytics (NumPy, Pandas, TensorFlow, etc.)
- **Cross-Platform & Portable**
  Write once, run anywhere (Windows, Mac, Linux).
- **Strong Community & Documentation**
  Huge support base, tutorials, and open-source contributions.

3. What is the output of following code in Python?

```
>>> name = 'Codebasics'
>>> print(name[:4] + name[4:])
```

Answer:
Codebasics

This is an example of **Slicing**. Since we are slicing at the same index, the first name[:4] gives the substring from the beginning up to the 4th location (excluding the 4th character).
The name[4:] gives the rest of the substring from the 4th character onwards.
So we get the full name as output by combining both parts.

4. **You have a list of top 5 MAANG companies and want to print each company with its position in the list.**

   ```
   companies = ["Meta", "Amazon", "Apple", "Netflix", "Google"]
   ```

   What is the easiest way to print the position and the company name (like "0 Meta", "1 Amazon", etc.) without using a separate counter variable?

Answer:

Use the enumerate() function:

```
companies = ["Meta", "Amazon", "Apple", "Netflix", "Google"]

for i, company in enumerate(companies):

    print(i, company)
```

Output:

```
0 Meta

1 Amazon

2 Apple

3 Netflix

4 Google
```

5. **You ask the user to enter their age using input(), but sometimes they type words like "twenty" instead of numbers.**
   This causes the program to crash with a ValueError.

   How will you handle this error in Python so that the program shows a friendly message like "Please enter a valid number!" **instead of crashing?**

Answer:

Use a try-except block to catch the ValueError and show a friendly message.

```
try:

    age = int(input("Enter your age: "))

    print("You are", age, "years old.")

except ValueError:

    print("Please enter a valid number!")
```

6. In Python, you have two lists. One contains existing data, and the other contains new items to be added. You want to add all new items to the existing list.

   How will you decide whether to use append() or extend()?
   Also, explain the difference between the two with a suitable example.

Answer:

- Use extend() when you want to **add all items from another list individually**.
- Use append() if you want to **add the entire list as a single element**.

```
a = [1, 2]
b = [3, 4]

# Using append
a1 = [1, 2]
a1.append(b)
print(a1)  # [1, 2, [3, 4]]

# Using extend
a2 = [1, 2]
a2.extend(b)
print(a2)  # [1, 2, 3, 4]
```

7. What is a Module in Python?

Answer:

A **module** in Python is a file containing Python code, typically with a .py extension. It can include **functions, classes, variables**, and even **executable code**.

Modules help organize code into **separate, manageable parts**, making it reusable and easier to maintain. You can use a module in another Python script by **importing it** using the import statement.

Example:

```
# mymodule.py

def greet():

    return "Hello from the module!"
```

```
# main.py

import mymodule

print(mymodule.greet())  # Output: Hello from the module!
```

8. What is the use of // operator in Python?

Answer:

The // operator in Python is used for **floor division**. It divides the left operand by the right operand and returns the **largest whole number less than or equal to the result**. In other words, it **removes the decimal part** and **rounds down** to the nearest integer.

a // b gives the largest integer that is ≤ (a ÷ b), even if the division result is negative.

- For **positive numbers**, // simply drops the decimal (like truncating).

7 // 3 → 2

7 ÷ 3 = 2.33

Floor division // drops the decimal → **2**

This is the **largest whole number ≤ 2.33**

- For **negative numbers**, it **rounds further down** to the next lower integer (more negative).

-7 // 3 → -3

-7 ÷ 3 = -2.33

Floor division rounds **down** to the next lower integer → **-3**

This is the **largest whole number ≤ -2.33**

9. What is the use of the zip() function in Python? Explain with an example.

Answer:

The zip() function in Python is used to **combine elements from two or more iterables** (like lists, tuples, etc.) into **pairs**. It returns an iterator of tuples, where each tuple contains one element from each iterable, matched by their positions (index).

```
list_1 = ['a', 'b', 'c']

list_2 = ['1', '2', '3']

for a, b in zip(list_1, list_2):

    print(a, b)
```

Output:

```
a 1

b 2

c 3
```

- zip() stops when the shortest iterable is exhausted.
- It is useful for **pairing data from multiple sources**.
- The result can be converted to a list or tuple if needed:

```
list(zip(list_1, list_2))  # [('a', '1'), ('b', '2'), ('c', '3')]
```

10. You're building a library app. You need to represent a book with its title and author. Create a class Book and an object representing "Harry Potter" by "J.K. Rowling".

Answer:

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

book1 = Book("Harry Potter", "J.K. Rowling")
print(book1.title, "-", book1.author)
```

11. What is the role of the __init__() method in Python classes? How does the self keyword work, and why is it important in defining instance methods and variables?

Answer:

- __init__() is a special method that is **automatically called when a new object is created**. This method is known as a **Constructor**.
- self refers to the **current object**. It's used to access attributes and methods inside the class.

12. Explain how file handling is done in Python. Describe the different file modes available and how they affect file operations.

Answer:

In Python, files are handled using the built-in open() function, which allows you to **read from**, **write to**, or **append to** files.

```
file = open("filename.txt", mode)
```

Mode	Description
'r'	Read-only mode (default)

Mode	Description
'w'	Write mode (overwrites file)
'a'	Append mode
'x'	Create a new file (error if exists)
'b'	Binary mode (e.g., 'rb', 'wb')
'+'	Read and write mode (e.g., 'r+')

Reading a File:

```
with open("example.txt", "r") as file:

    content = file.read()

    print(content)
```

with ensures the file is **automatically closed** after the block is executed — even if an error occurs.

Writing to a File:

```
with open("output.txt", "w") as file:

    file.write("This is a test message.")
```

- If the file doesn't exist, it will be created.
- If it exists, its contents will be overwritten.

Why Use with Statement?

- Automatically handles opening and closing the file.
- Prevents **resource leaks** and makes code cleaner and safer.

INTERMEDIATE

1. How will you specify source code encoding in a Python source file?

Answer:

By default, every source code file in Python is in UTF-8 encoding. But we can also specify our own encoding for source files. This can be done by adding the following line after the #! line in the source file:

```
# -*- coding: <encoding-name> -*-
```

2. What is the use of PEP 8 in Python?

Answer:

PEP 8 is the official style guide for writing Python code. It outlines coding standards related to indentation, formatting, line length, imports, whitespace, and more. The primary goal of PEP 8 is to promote code consistency and readability across Python projects. By following PEP 8, developers make their code more understandable and maintainable for others, fostering collaboration and best practices in software development.

3. How does memory management work in Python?

Answer:

Python uses a **private heap space** to store all objects and data structures. This memory is managed internally by the Python memory manager. In the case of **CPython**, a built-in memory manager is responsible for allocating and managing this heap space.

The memory manager handles multiple tasks such as **segmentation, caching, sharing, and pre-allocation**. Additionally, Python has an automatic **garbage collector** that reclaims memory occupied by unused objects using a technique called **reference counting**.

4. How can you retrieve data from a MySQL database using a Python script?
   Explain the steps involved and mention the functions used for querying and fetching data.

Answer:

To retrieve data from a MySQL database in Python, you can follow these steps using the mysql.connector module:

1. Import the MySQL connector module:

```
import mysql.connector
```

2. Establish a connection to the database:

```
conn = mysql.connector.connect(

    host="localhost",

    user="your_username",

    password="your_password",

    database="your_database"

)
```

3. Create a cursor object:

```
cursor = conn.cursor()
```

4. Execute a SQL query:

```
cursor.execute("SELECT * FROM your_table")
```

5. Fetch the results:

```
results = cursor.fetchall()  # Use fetchone() if you want just one record
for row in results:
    print(row)
```

6. Close the connection:

```
conn.close()
```

5. Is Python an Object-Oriented or Functional Programming Language?

Answer:

Python is a **multi-paradigm programming language**, which means it supports:

- Object-Oriented Programming – using classes and objects.

```
class Dog:
    def bark(self):
        print("Woof!")
```

- Functional Programming – using functions like map(), filter(), lambda, and avoiding state changes.

```
nums = [1, 2, 3]
squared = list(map(lambda x: x**2, nums))
```

- Procedural Programming – writing step-by-step instructions using functions and control flow.

```
def greet(name):
    print("Hello", name)
```

- Imperative Programming – Focuses on giving explicit commands to change program state.

```
total = 0
for i in range(5):
    total += i
print(total)
```

6. What are Magic Functions in Python?

Answer:

Magic functions, also called **dunder methods** (short for **double underscore**), are **special built-in methods** in Python that start and end with **double underscores**, like __init__, __str__, __len__, etc.

They are automatically called by Python to perform **built-in operations**.

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

    def __str__(self):
        return f"This is {self.name}"

my_dog = Dog("Buddy")
print(my_dog)  # Output: This is Buddy
```

__init__ → Called when you create an object
__str__ → Called when you use print() on the object

7. What is a Lambda Expression in Python?

Answer:

A **lambda expression** (or **lambda function**) in Python is a way to write **small, anonymous (nameless) functions** in a single line.

It's often used when you **only need a function temporarily**, especially inside functions like map(), filter(), or sorted().

Syntax:

lambda arguments: expression

- lambda → keyword to define the function
- arguments → like parameters of a normal function
- expression → the result you want to return

Example 1: lambda

```
square = lambda x: x * x

print(square(5))  # Output: 25
```

Example 2: With map()

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x**2, nums))
print(squares)  # Output: [1, 4, 9, 16]
```

8. What is inheritance in Python and why is it useful? Explain how method overriding works in the context of inheritance. Create a base class Shape and a subclass Rectangle that overrides the area() method.

Answer:

- **Inheritance** allows one class (child) to inherit attributes and methods from another (parent).
- **Method overriding** occurs when a child class **redefines a method** from the parent class to give it a new behavior.

```
class Shape:
    def area(self):
        print("Area is undefined")


class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width
    def area(self):  # Overriding
        return self.length * self.width


r = Rectangle(5, 3)
print(r.area())  # Output: 15
```

9. What is the purpose of the super() function in Python?

Answer:

- super() allows access to methods from a **parent class**, especially helpful in inheritance to avoid rewriting code.
- Often used to call the **parent class constructor**.

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

```
class Employee(Person):

    def __init__(self, name, emp_id):

        super().__init__(name)  # Call parent constructor

        self.emp_id = emp_id



e = Employee("Ravi", 101)

print(e.name, e.emp_id)  # Output: Ravi 101
```

10. Explain how string immutability in Python affects string operations. How does Python handle operations like concatenation, slicing, or replacement if strings cannot be changed? Discuss the advantages and disadvantages of string immutability with examples.

Answer:

What is String Immutability?

In Python, **strings are immutable**, which means that once a string is created, its contents **cannot be modified**. You cannot change a character at a specific index — any operation that seems to modify a string actually returns a **new string**.

How It Affects Operations:

- **Concatenation:**
  "a" + "b" creates a **new string** "ab", not altering "a" or "b".
- **Slicing:**
  "hello"[0:2] gives "he" — again, a **new string**, not a sliced version of the original one in-place.
- **Replacement:**
  "hi there".replace("hi", "hello") gives "hello there", leaving the original string unchanged.

Advantages of Immutability:

- **Safety:** Immutable objects can be used as keys in dictionaries or elements in sets.
- **Predictability:** Since strings don't change, functions relying on them behave consistently.

Disadvantages:

- **Performance Overhead:** Frequent concatenation in loops can be inefficient because each operation creates a new string.
- **Memory Usage:** Each "modification" results in a new object, which can increase memory use in large-scale applications.

ADVANCED

1. **You're building a student ranking system.**

   You have a list of student names along with their scores, like this:

```
students = [
    ("Alice", 88),
    ("Bob", 75),
    ("Charlie", 95),
    ("David", 82)
]
```

   You want to sort this list based on the students' scores in descending order, but you don't want to write a full function just for that.

   How can you do this using a lambda expression?

Answer:

Use the sorted() function with a lambda as the key:

```
sorted_students = sorted(students, key=lambda x: x[1], reverse=True)
print(sorted_students)
```

Output:

```
[('Charlie', 95), ('Alice', 88), ('David', 82), ('Bob', 75)]
```

sorted(students, ...)

- This uses Python's built-in sorted() function to sort the list.
- It does **not** change the original list — it returns a **new sorted list**.

key=lambda x: x[1]

- This tells Python **how** to sort the items.
- Each x is a tuple like ("Alice", 88)
- x[1] means **"look at the second item"** in the tuple → the score.
- So it's telling Python: **"Sort by the score (not by name)"**

reverse=True

- This means **sort in descending order** (from high to low).

2. What is a Metaclass in Python?

Answer:

A **metaclass** in Python is a **class of a class** — just like classes create objects, **metaclasses create classes**.

Objects are made from classes and Classes are made from metaclasses.

In Python, **type is the default metaclass**:

```
class Dog:

    pass

print(type(Dog))  # <class 'type'>
```

This means Python internally used type to *create* the Dog class.

3. What is the Global Interpreter Lock (GIL) in Python, and how does it impact multithreading?

Answer:

The Global Interpreter Lock (GIL) in Python is a mutex (mutual exclusion lock) that ensures only **one thread executes Python bytecode at a time**, even on multi-core processors. This lock exists in the **CPython** interpreter to simplify memory management, especially reference counting used for garbage collection.

Impact on Multithreading:

- **CPU-bound tasks** (e.g., image processing, large computations):
  Performance is limited because threads cannot run in true parallel. Use multiprocessing instead of threading.
- **I/O-bound tasks** (e.g., file reading, API calls):
  Works well, as threads spend time waiting for external resources.

4. **What are the different debugging techniques used in Python?**

Answer:

Using print() Statements

Simple but effective for small scripts.

Use when:

- Tracking variable values
- Checking control flow

```
def divide(a, b):

    print(f"Dividing {a} by {b}")

    return a / b

print(divide(10, 2))
```

Using the pdb Module (Python Debugger)

A built-in interactive debugger for line-by-line control.

```
import pdb

def add(x, y):

    pdb.set_trace()

    return x + y

add(2, 3)
```

Using IDE Debuggers (VSCode, PyCharm, etc.)

Benefits:

- Breakpoints
- Watch variables
- Step through code visually
- See call stacks

Logging Instead of Printing

More scalable and flexible than print(), especially in production.

```
import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug("Debug message")

logging.info("Info message")
```

Unit Testing for Debugging

Write small tests to isolate bugs using unittest or pytest.

```
import unittest

def square(x):

    return x * x

class TestSquare(unittest.TestCase):

    def test_positive(self):

        self.assertEqual(square(4), 16)



unittest.main()
```

Exception Handling and Tracebacks

Catch and understand errors gracefully:

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print("Error:", e)
```

Using assert Statements

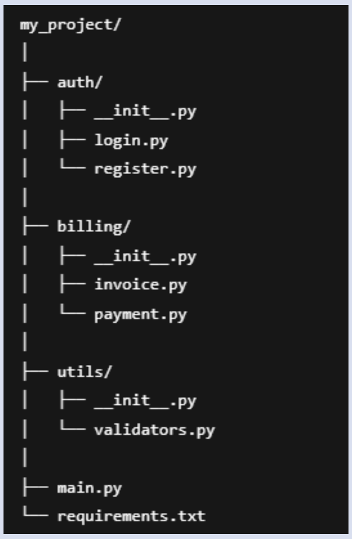
Validate assumptions during development:

```
x = 5
assert x > 0, "x must be positive"
```

Third-party Debuggers

- ipdb – IPython version of pdb
- pudb – Text-based UI debugger
- py-spy – Profiling + debugging slow code

5. You're working on a large-scale Python project that is split across multiple teams. The project has the following structure:



```
my_project/
|
├─ auth/
|   ├─ __init__.py
|   ├─ login.py
|   └─ register.py
|
├─ billing/
|   ├─ __init__.py
|   ├─ invoice.py
|   └─ payment.py
|
├─ utils/
|   ├─ __init__.py
|   └─ validators.py
|
├─ main.py
└─ requirements.txt
```

How would you convert this into a distributable Python package?

Answer:

Convert to a package:

- ➢ Add a setup.py, pyproject.toml, and MANIFEST.in (for non-Python files)
- ➢ Add a README.md and LICENSE
- ➢ Add __init__.py in each submodule (already present)

6. You are given a nested list of employee records, where each record is a tuple containing (name, department, salary). You are tasked to:

a) Extract the names of employees in the "Tech" department who earn more than 75,000.
b) Convert all such names to uppercase.
c) Implement this using a single list comprehension.

Additionally, explain the trade-offs between using a list comprehension vs. traditional for-loops in this context.

```
employees = [

    ("Alice", "Tech", 80000),

    ("Bob", "HR", 50000),

    ("Charlie", "Tech", 72000),

    ("Diana", "Tech", 88000),

    ("Eve", "Sales", 60000),

]
```

Answer:

```
tech_high_earners = [name.upper() for name, dept, sal in employees if dept == "Tech" and sal > 75000]
```

Output:

```
['ALICE', 'DIANA']
```

- ➢ Use list comprehensions for **simple transformations + filtering**
- ➢ Switch to loops when logic becomes **multi-step or nested**

7. In a large enterprise application, you are asked to implement the following features:

- ✓ Automatically log access whenever certain functions are called (e.g., update_profile, process_payment)
- ✓ Add retry logic to functions that may intermittently fail (e.g., network calls)
- ✓ Apply the same logging and retry logic without modifying the original function code

How would Python decorators help in solving this?
What are the advantages of using decorators for such functionality over traditional function wrappers or class inheritance? Also explain how decorators can be stacked.

Answer:

Role of Decorators

- Decorators allow attaching additional behavior (like logging or retrying) **around a function**, without modifying its internal logic.
- They enable **separation of concerns**—function logic stays clean, while cross-cutting concerns like logging or error handling are added externally.

Advantages over Traditional Techniques

Technique	Limitation	Decorator Advantage
Manual Wrappers	Repetitive and error-prone	Decorators abstract and reuse logic
Inheritance	Only works with classes, not functions	Decorators work for any callable
Inline logic	Pollutes business logic with unrelated functionality	Decorators keep code modular and clean

Decorator Stacking

- Decorators can be stacked like:

```
@retry

@log_access

def sensitive_function(): ...
```

They execute **inside-out**: first log_access, then retry.

8. **You are building a data pipeline that processes large batches of records from a database. The dataset is so large that loading it all into memory would be inefficient and could crash your system.**

a) **How can Python generators help you process this data efficiently?**
b) **Demonstrate your approach using a simple example that shows how to yield one record at a time.**
c) **Briefly explain the memory and performance benefits of using generators in this case.**

Answer:

Generators allow processing **one item at a time** without loading the entire dataset into memory.

```
def fetch_records(records):

    for record in records:

        yield record
```

Now, you can use it like this:

```
for r in fetch_records(huge_list):
```

```
process(r)
```

Benefits:

- **Memory-efficient**: Only one record is in memory at a time.
- **Faster start-up**: You don't wait for the entire dataset to load.
- **Scalable**: Works well even with millions of records.

9. You are designing a payment processing system for an e-commerce platform that supports multiple payment methods (Credit Card, PayPal, UPI, etc.). You want to ensure that all payment methods implement certain core behaviors, but the implementation can vary.
    a) How would you use abstract classes in Python to enforce a consistent interface across different payment methods?
    b) What are the benefits of using abstraction in this context?

Answer:

a) Using Abstract Classes for Consistency

To ensure that all payment methods implement required core behaviors (like authorize(), pay(), refund()):

- Create an **abstract base class** called PaymentMethod using Python's abc module.
- Declare **abstract methods** for authorize(), pay(), and refund() in the base class.
- Each specific payment class (like CreditCardPayment, PayPalPayment, UPIPayment) would **inherit** from PaymentMethod and provide their own **implementations** of these methods.

This ensures that **every subclass** follows the **same structure**, preventing incomplete or inconsistent implementations.

b) Benefits of Using Abstraction Here

- **Code Consistency**: All payment methods follow a unified interface, making the system easy to understand and extend.
- **Enforces Design Contracts**: Developers cannot skip implementing required methods — Python will raise a TypeError if they do.
- **Scalability**: New payment methods can be added without changing existing code — just create a new subclass.
- **Polymorphism**: You can treat all payment methods the same way in client code (e.g., loop through and call pay() on any payment method).

10. What are the SOLID Principles?

Answer:

SOLID principles help you write clean, scalable, and maintainable code by promoting good object-oriented design practices.

Principle	Stands for	Description
S	Single Responsibility Principle	A class should have only one reason to change
O	Open/Closed Principle	Software entities should be open for extension, but closed for modification
L	Liskov Substitution Principle	Subclasses should be substitutable for their base classes
I	Interface Segregation Principle	Clients should not be forced to depend on methods they do not use
D	Dependency Inversion Principle	High-level modules should **not depend on low-level modules**; both should depend on **abstractions**.