Python - Fundamentals Of Python Language

Introduction to Python

Q1. What is Python?

Answer:
Python is a high-level, interpreted, object-oriented programming language known for its simplicity and readability.
It allows programmers to express concepts in fewer lines of code than other languages like C++ or Java.
Python supports multiple programming paradigms — such as procedural, object-oriented, and functional programming.

Q2. What are the main features of Python?

Answer:
Python has several important features that make it one of the most popular programming languages today:

1. **Simple and Easy to Learn:**
   The syntax of Python is clean and resembles English, which makes it beginner-friendly.
2. **Interpreted Language:**
   Python code is executed line-by-line by the interpreter — no need for compilation.
3. **High-Level Language:**
   You don't need to manage memory or deal with complex hardware-level details.
4. **Portable:**
   Python programs can run on any operating system (Windows, macOS, Linux) without modification.
5. **Object-Oriented:**
   Supports classes, objects, inheritance, and encapsulation.
6. **Extensive Libraries:**
   Comes with a large set of built-in modules for file handling, math, web access, and more.
7. **Dynamic Typing:**
   No need to declare variable types; Python determines them automatically at runtime.
8. **Open Source:**
   Python is freely available and maintained by a large community of developers.

9. **Embeddable and Extensible:**
   Python code can be integrated with C, C++, and Java easily.

Q3. What is the history and evolution of Python?

Answer:

- Creator: Guido van Rossum
- Origin: Developed at CWI (Centrum Wiskunde & Informatica) in the Netherlands in the late 1980s.
- 1991: First public release — Python 1.0.
    - Included exception handling, functions, and core data types.
- 2000: Release of Python 2.0, introducing list comprehensions and garbage collection.
- 2008: Python 3.0 released — not backward-compatible but included many improvements like Unicode support and better syntax.
- Present: Managed by the Python Software Foundation (PSF) and widely used in Data Science, Machine Learning, Web Development, Automation, IoT, and more.

Q4. What are the advantages of using Python over other languages?

Answer:

Feature	**Python**	**Other Languages**
Ease of Use	Simple, readable syntax	More complex syntax
Speed of Development	Faster — fewer lines of code	Slower — more code required
Libraries	Extensive standard and third-party libraries	Limited libraries
Community Support	Huge global community	Smaller in comparison
Platform Independence	Works on all platforms	Some need recompilation
Applications	AI, ML, Web, Data Science, Automation	Often domain-specific

Q5. How to install Python and set up the development environment?

Answer:
There are multiple ways to install and run Python:

1. **Official Python Installer:**
    - Visit https://www.python.org/downloads/.
    - Download and install the latest version.
    - Check installation:
    - python --version
2. **Anaconda Distribution:**
    - Best for Data Science and Machine Learning.
    - Includes Python, Jupyter Notebook, and many preinstalled libraries.
    - Download from https://www.anaconda.com/.
3. **IDEs (Integrated Development Environments):**
    - **IDLE** – Comes with Python by default.
    - **PyCharm** – Professional IDE for Python projects.
    - **VS Code** – Lightweight editor with Python extensions.

Q6. How to write and execute your first Python program?

Answer:

1. Open your Python IDE or any text editor.
2. Type the following code:
3. # My first Python program
4. print("Hello, Python!")
5. Save the file as hello.py.
6. Run it in the terminal or command prompt:
7. python hello.py

Output:

Hello, Python!

Q7. Where is Python used?

Answer:
Python is widely used in:

- Web Development (Django, Flask)
- Data Science & Machine Learning (NumPy, Pandas, TensorFlow)
- Automation / Scripting
- Game Development
- Internet of Things (IoT)
- Cybersecurity & Ethical Hacking
- Desktop and GUI Applications

2. Programming Style

Q1. What are Python's PEP 8 guidelines?

Answer:
PEP 8 (Python Enhancement Proposal 8) is the official style guide for writing Python code.
It provides a set of rules and best practices that help developers write clean, readable, and consistent code.
Following PEP 8 makes programs easier to understand and maintain, especially when multiple people work on the same project.

Main PEP 8 Guidelines:

- Use 4 spaces per indentation level (no tabs).
- Keep line length under 79 characters.
- Use blank lines to separate functions and classes.
- Add spaces around operators (e.g., x = a + b).
- Use meaningful variable names (e.g., student_name instead of sn).
- Write comments to explain the logic of your code.
- Follow naming conventions:
    - Variables and functions → lower_case_with_underscores
    - Classes → CamelCase
    - Constants → UPPER_CASE

Q2. What are indentation, comments, and naming conventions in Python?

Answer:

1. Indentation:
Indentation in Python means giving spaces at the beginning of a line to define code blocks.
Unlike other languages that use {}, Python uses indentation to decide which statements belong together.
If indentation is not correct, Python will show an IndentationError.

```
if 5 > 2:
  print("Five is greater than two")  # Correct indentation
```

---

2. Comments:
Comments are lines that Python **ignores during execution**.
They are used to make the code easier to understand.

- **Single-line comment:** starts with #
- # This is a single-line comment
- **Multi-line comment:** uses triple quotes ''' or """

  """

  This is a

  multi-line comment

  """

3. Naming Conventions:
Naming conventions make code clean and consistent.
Examples:

Type	**Example**	**Convention**
Variable	student_name	lowercase_with_underscores
Function	calculate_area()	lowercase_with_underscores
Constant	MAX_VALUE	UPPERCASE_WITH_UNDERSCORES
Class	StudentDetails	CamelCase

Q3. How to write readable and maintainable code in Python?

Answer:
Writing readable and maintainable code helps in understanding, debugging, and updating programs easily.

Best Practices:

1. Follow PEP 8 rules for style and formatting.
2. Use meaningful names for variables and functions.
3. Keep your code properly indented.
4. Write comments and docstrings to explain logic.
5. Avoid very long lines of code.

6. Keep functions small — each should do one task.
7. Use consistent spacing and structure throughout the program.

Example:

```
# Program to calculate the area of a rectangle


def calculate_area(length, width):
    """Return the area of a rectangle."""
    area = length * width
    return area



# Main part of the program
length = 5
width = 3
print("Area of rectangle:", calculate_area(length, width))
```

Output:

```
Area of rectangle: 15
```

3. Core Python Concepts

Q1. What are data types in Python?

Answer:
In Python, data types define the kind of value a variable can hold.
Python automatically assigns a data type based on the value you store — this is called dynamic typing.

Below are the most commonly used data types:

Data Type	Description	Example
int	Stores whole numbers (no decimals)	x = 10
float	Stores decimal or fractional numbers	pi = 3.14
str (string)	Sequence of characters enclosed in quotes	name = "Aayushi"
list	Ordered, changeable collection	fruits = ['apple', 'banana', 'mango']
tuple	Ordered, unchangeable collection	colors = ('red', 'green', 'blue')
dict (dictionary)	Key–value pairs	student = {'name': 'John', 'age': 20}
set	Unordered collection of unique items	numbers = {1, 2, 3, 3} → {1, 2, 3}

Example Code:

```
# Different data types in Python
x = 10              # int
y = 3.5             # float
name = "Aayushi"       # string
fruits = ['apple', 'banana', 'mango']  # list
colors = ('red', 'green', 'blue')     # tuple
student = {'name': 'John', 'age': 20}  # dictionary
numbers = {1, 2, 3, 3}   # set
print(type(x), type(y), type(name))
```

Q2. What are Python variables and how does memory allocation work?

Answer:
A **variable** is a name given to store a value in memory.
In Python, you don't need to declare the variable type explicitly — the interpreter assigns it automatically.

Example:

```
a = 10        # integer variable

b = 3.14      # float variable

name = "John"   # string variable
```

When a variable is created:

- Python creates an object in memory to store the value.
- The variable name becomes a reference (or label) to that object.
- Python's memory manager handles allocation and garbage collection automatically.

Example (to show reference):

```
x = 5

y = x

print(id(x), id(y))   # Both point to the same memory location
```

If both variables have the same value, they may reference the same object in memory until one of them changes.

Q3. What are Python operators?

Answer:
Operators are special symbols in Python that perform operations on variables and values.

Arithmetic Operators

Used for basic mathematical operations:

Operator	Description	Example	Result
+	Addition	10 + 5	15
-	Subtraction	10 - 5	5

Operator	Description	Example	Result
*	Multiplication	10 * 5	50
/	Division	10 / 5	2.0
//	Floor Division	10 // 3	3
%	Modulus (remainder)	10 % 3	1
**	Exponentiation	2 ** 3	8

Comparison (Relational) Operators

Used to compare values; returns **True** or **False**:

Operator	**Meaning**	**Example**	**Result**
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	5 > 3	True
<	Less than	2 < 5	True
>=	Greater than or equal to	5 >= 5	True
<=	Less than or equal to	3 <= 5	True

Logical Operators

Used to combine conditional statements:

Operator	**Description**	**Example**	**Result**
and	Returns True if both conditions are True	(5 > 2 and 4 > 1)	True
or	Returns True if at least one condition is True	(5 > 10 or 4 > 1)	True
not	Reverses the result	not(5 > 2)	False

Bitwise Operators

Work at the **binary level** on integers:

Operator	Description	Example	Result
&	Bitwise AND	5 & 3	1
`	`	Bitwise OR	` 5
^	Bitwise XOR	5 ^ 3	6
~	Bitwise NOT	~5	-6
<<	Left Shift	5 << 1	10
>>	Right Shift	5 >> 1	2

4. Conditional Statements

Q1. What are conditional statements in Python?

Answer:
Conditional statements are used to make decisions in a Python program.
They allow the program to execute different blocks of code based on certain conditions.

Python provides three main conditional statements:

1. if statement
2. if–else statement
3. if–elif–else ladder

Q2. Explain the use of if, else, and elif statements with examples.

Answer:

if statement

Used to execute a block of code only if a condition is True.

```
x = 10
if x > 5:
    print("x is greater than 5")
```

Output:
x is greater than 5

if–else statement

Used when there are two possibilities — one if the condition is True, and another if it's False.

```
x = 4
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

Output:
x is not greater than 5

if–elif–else ladder

Used when there are **multiple conditions** to check.

```
marks = 85

if marks >= 90:
   print("Grade A")
elif marks >= 75:
   print("Grade B")
elif marks >= 50:
   print("Grade C")
else:
   print("Fail")
```

Output:
Grade B

Q3. What are nested if-else statements?

Answer:
A **nested if-else** means placing one if-else **inside another if-else**.
It is used when a decision depends on another condition.

Example:

```
age = 20

weight = 50

if age >= 18:
   if weight >= 45:
      print("Eligible to donate blood")
   else:
      print("Not eligible due to low weight")
else:
   print("Not eligible due to age")
```

Output:
Eligible to donate blood

5. Looping (For, While)

Q1. What are loops in Python?

Answer:
Loops are used to execute a block of code repeatedly as long as a condition is True. They help reduce repetition and make programs more efficient.

Python has two main types of loops:

1. for loop
2. while loop

Q2. How do for and while loops work in Python?

for loop

Used to **iterate over a sequence** (like a list, tuple, string, or range).

Example:

```
fruits = ['apple', 'banana', 'mango']

for fruit in fruits:
  print(fruit)
```

Output:

apple

banana

mango

You can also use range() to run the loop a fixed number of times:

```
for i in range(1, 6):
  print(i)
```

Output:

1

2

3

4

5

while loop

Used to execute a block of code **as long as a condition is True**.

Example:

```
count = 1
while count <= 5:
  print(count)
  count += 1
```

Output:

1

2

3

4

5

Q3. How are loops used with collections like lists and tuples?

Answer:
Loops can be used to iterate through elements of collections like lists, tuples, sets, and dictionaries.

Examples:

- **List**
- colors = ['red', 'green', 'blue']
- for c in colors:
- print(c)
- **Tuple**
- numbers = (1, 2, 3)
- for n in numbers:
- print(n)
- **Dictionary**
- student = {'name': 'Aayushi', 'age': 20, 'marks': 85}

- for key, value in student.items():
- print(key, ":", value)

Output:

name : Aayushi

age : 20

marks : 85

6. Generators and Iterators

Q1. What are generators in Python and how do they work?

Answer:
A generator in Python is a special type of function that produces values one at a time instead of returning them all at once.
They are used for memory-efficient data handling, especially with large datasets.

Generators use the yield keyword instead of return.
When a generator function is called, it doesn't run immediately; it returns a generator object that can be iterated using a for loop or the next() function.

Example:

```
def generate_numbers():
  for i in range(1, 6):
    yield i  # returns one value at a time


for num in generate_numbers():
  print(num)
```

Output:

1

2

3

4

5

How it works:

- The function pauses each time it hits a yield statement.
- When the next value is requested, execution resumes from where it left off.

Q2. What is the difference between yield and return?

Feature	yield	return
Keyword use	Used in **generator functions**	Used in **normal functions**
Returns	Returns a **generator object**	Returns a **single value**
Behavior	Pauses the function and saves its state	Ends the function completely
Memory use	**Memory efficient** (values generated one by one)	**Stores all values** before returning
Example	yield i	return i

Example:

```
def normal_func():
  return [1, 2, 3]


def generator_func():
  for i in range(1, 4):
    yield i
```

Q3. What are iterators in Python and how to create custom iterators?

Answer:
An **iterator** is an object that allows sequential access to elements in a collection (like lists or tuples) **one item at a time**.
Python uses two special methods to make an object an iterator:

- __iter__() — returns the iterator object itself.
- __next__() — returns the next item in the sequence.

Example (custom iterator):

```
class CountDown:
  def __init__(self, start):
    self.current = start


  def __iter__(self):
    return self
```

```
    def __next__(self):
        if self.current <= 0:
            raise StopIteration
        num = self.current
        self.current -= 1
        return num

for value in CountDown(5):
    print(value)
```

Output:

```
5
4
3
2
1
```

7. Functions and Methods

Q1. What is a function and how do you define and call it in Python?

Answer:
A function is a block of reusable code that performs a specific task.
It helps in avoiding code repetition and improves readability.

Defining a function:

```
def greet():
  print("Hello, Python!")
```

Calling a function:

```
greet()
```

Output:

```
Hello, Python!
```

Q2. What are function arguments in Python?

Answer:
Python functions can accept different types of arguments (parameters) to pass data.

1. Positional Arguments:
   Passed in order of position.
2. def add(a, b):
3. print(a + b)
4. add(5, 3) # Output: 8
5. Keyword Arguments:
   Passed using parameter names.
6. def greet(name, msg):
7. print(msg, name)
8. greet(name="Aayushi", msg="Hello") # Output: Hello Aayushi
9. Default Arguments:
   Have a default value if no value is provided.
10. def greet(name="User"):
11. print("Hello", name)
12. greet() # Output: Hello User

Q3. What is the scope of variables in Python?

Answer:
The **scope** of a variable defines **where it can be accessed** in the program.

Type	Description	Example
Local Variable	Declared inside a function; accessible only there	Defined inside a function
Global Variable	Declared outside all functions; accessible throughout the program	Declared at the top level

Example:

```
x = 10  # global variable

def show():

   y = 5  # local variable

   print("Inside function:", x, y)

show()

print("Outside function:", x)
```

Output:

Inside function: 10 5

Outside function: 10

Q4. What are built-in methods in Python?

Answer:
Methods are functions that belong to specific data types or objects.
Python provides many built-in methods for strings, lists, and other collections.

String Methods:

Method	Description	Example
upper()	Converts to uppercase	"hello".upper() → 'HELLO'
lower()	Converts to lowercase	"HELLO".lower() → 'hello'
strip()	Removes spaces	" hi ".strip() → 'hi'
replace()	Replaces part of a string	"good".replace("good", "bad") → 'bad'

List Methods:

Method	Description	Example
append()	Adds an item	[1, 2].append(3) → [1, 2, 3]
remove()	Removes an item	[1, 2, 3].remove(2) → [1, 3]
sort()	Sorts items	[3, 1, 2].sort() → [1, 2, 3]
pop()	Removes last element	[1, 2, 3].pop() → 3

8. Control Statements (Break, Continue, Pass)

Theory:

- **Control statements** are used to alter the normal flow of a loop.
  They allow you to skip iterations, stop loops early, or write placeholder code.

1. break Statement

- The break statement **terminates the loop immediately**, even if the loop condition is still true.
- It is often used when a specific condition is met and no further looping is required.

Example:

```
for fruit in ['apple', 'banana', 'mango']:
  if fruit == 'banana':
    break
  print(fruit)
```

Output:

apple

2. continue Statement

- The continue statement **skips the current iteration** of the loop and moves to the next one.
- It is useful when certain conditions should be ignored during loop execution.

Example:

```
for fruit in ['apple', 'banana', 'mango']:
  if fruit == 'banana':
    continue
  print(fruit)
```

Output:

apple

mango

3. pass Statement

- The pass statement **does nothing** — it's used as a placeholder for future code.
- It helps to maintain syntactic correctness without executing any operation.

Example:

```
for fruit in ['apple', 'banana', 'mango']:
  if fruit == 'banana':
    pass
  print(fruit)
```

Output:

apple

banana

mango

9. String Manipulation

Theory:

- A **string** in Python is a sequence of characters enclosed in single, double, or triple quotes.
- Strings are **immutable**, meaning they cannot be changed after creation.

Basic String Operations:

Operation	Description	Example	Output
Concatenation	Combine strings	"Hello" + "World"	HelloWorld
Repetition	Repeat a string	"Hi" * 3	HiHiHi
Length	Find string length	len("Python")	6

Common String Methods:

Method	Description	Example	Output
upper()	Converts to uppercase	"python".upper()	PYTHON
lower()	Converts to lowercase	"HELLO".lower()	hello
strip()	Removes spaces	" hi ".strip()	hi
replace()	Replaces part of a string	"good".replace("good", "bad")	bad
split()	Splits string into list	"a,b,c".split(",")	['a', 'b', 'c']

String Slicing:

- Slicing extracts a portion of a string using index ranges.

Syntax:

string[start:end:step]

Example:

```
text = "Python"

print(text[1:4])   # yth

print(text[:3])    # Pyt

print(text[::-1])  # nohtyP
```

10. Advanced Python (map(), reduce(), filter(), Closures, Decorators)

1. map() Function

- Applies a function to each element in a sequence.
- Returns an iterator with the results.

Example:

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))
print(squared)
```

Output:

```
[1, 4, 9, 16]
```

2. reduce() Function

- Applies a function **cumulatively** to all elements in a list.
- Used to get a single result (e.g., sum or product).
- Defined in the functools module.

Example:

```
from functools import reduce

numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)
```

Output:

```
24
```

3. filter() Function

- Filters elements from a list based on a condition.
- Returns only the elements that satisfy the condition.

Example:

```
numbers = [1, 2, 3, 4, 5, 6]
even = list(filter(lambda x: x % 2 == 0, numbers))
print(even)
```

Output:

[2, 4, 6]

4. Closures

- A closure is a **function inside another function** that remembers the outer function's variables even after it has finished executing.

Example:

```
def outer_func(x):
    def inner_func(y):
        return x + y
    return inner_func

add_five = outer_func(5)
print(add_five(10))  # Output: 15
```

5. Decorators

- A decorator **modifies the behavior** of another function without changing its actual code.
- Commonly used for logging, authentication, or timing.

Example:

```
def decorator(func):
    def wrapper():
        print("Before function")
        func()
        print("After function")
    return wrapper
@decorator
```

```
def greet():
    print("Hello!")
greet()
```

Output:

```
Before function
Hello!
After function
```