Module : Python Fundamentals

❖ Python Fundamentals

1. Introduction to Python and its Features (simple, high-level, interpreted language).

- Python is a high level, interpreted programming language.
- It was created by Guido van Rossum and first released in 1991.

➢ Features of Python

1. Simple and Easy to Learn

- Python has a clean and straightforward syntax.
- Writing Python code is often similar to writing english

2. High-Level Language

- Python handles complex details like memory management automatically.
- You can focus on problem-solving instead of worrying about system-level programming.

3. Interpreted Language

- Python code is executed line-by-line by the interpreter.
- This makes it easy to test and debug your programs.

4.object -oriented

- Python is object-oriented but supports both functional and object-oriented programming.

5. Platform-independent

- Python is platform-independent.
- You don’t need to write a them separately for each platform.

6. Open-Source

- It is free and its source code is freely available to the public.
- You can download python from the official python website.

7. Large Standard library

- The standard library is large and has many packages and modules with common and important functionality.

8. Dynamically-Typed language

- You don’t need to declare data type while defining a variable
- This is easy for programmers but can everything in python is an object.

9. GUI support

- You can use python to create GUI (Graphical user interfaces)

2. History of Python.

- Python is a high-level programming language developed by Guido van Rossum in the late 1980s.
- He started working on it in 1989 during his time at the Centrum Wiskunde & Informatica (CWI) in the Netherlands. The first official version, Python 0.9.0, was released in 1991.
- Guido Van Rossum created Python as a successor to the ABC programming language, aiming to fix some of its issues while adding new features like exception handling and extensibility.
- The language was designed to be easy to read, simple to write, and powerful for both beginners and experienced developers.

3. Advantages of using Python over other programming languages.

1. Easy to Learn and Use

- Python's syntax is simple and readable, making it an excellent choice for beginners
- Its structure resembles plain English, reducing complexity compared to languages like C++ or Java.

2. Cross-Platform Compatibility

- Python runs seamlessly on Windows, Mac, and Linux without modification.

3. Extensive Libraries and Frameworks

- Python has powerful libraries like NumPy, Pandas, TensorFlow, and Django, speeding up development.
- Whether it's AI, machine learning, web development, or automation, Python has a solution.

4. Strong Community Support

- Python boasts a large, active developer community for support and troubleshooting.
- Continuous updates ensure Python stays relevant and secure for modern applications.

5. High Demand and career Growth

- Python is widely used in industries like finance, healthcare, education, and cybersecurity.
- Many companies, including Google, Facebook, and Netflix, rely on Python for various tasks.

4. Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).

➢ Installing Python

- Visit the official Python website and download the latest version. Run the installer and check the box for "Add Python to PATH" before clicking Install Now. Once installed, verify by running python --version in the Command Prompt (Windows) or Terminal (Mac/Linux).

➢ Setting Up Development Environments

◇ VS Code (Lightweight & Versatile)

- Get VS Code . Install the Python extension from the Marketplace. Set up the Python interpreter under Settings > Python: Interpreter. Open a new .py file and start coding!

5.Writing and executing your first Python program.

Step 1: Write a Simple Python Program

Let's start with a classic **"Hello, World!"** program:

Code:

```
Print ("Hello, World!")
```

This program simply prints "Hello, World!" to the screen.

Step 2: Save the File

- Open any text editor (like Notepad, VS Code, or PyCharm). Type the code above. Save the file with a .py extension, like hello.py.

Step 3: Running in an IDE (PyCharm or VS Code)

- Open PyCharm or VS Code and create a new Python file. Write the code inside the editor. Click Run or press Ctrl + Shift + F10 (PyCharm) / F5 (VS Code).

❖ Programming Style

1.Understanding Python's PEP 8 Guidelines

- PEP 8 (Python Enhancement Proposal 8) is the official style guide for writing Python code. It helps ensure that Python code is readable, consistent, and maintainable, especially when working in teams or on large projects.

2. Indentation, comments, and naming conventions in Python.

➢ Indentation in Python:

Indentation defines code blocks (like in loops, functions, classes). Python does not use **{}** brackets like other languages—indentation is mandatory.

- Use 4 spaces per indentation level .

➢ Comments in Python

- Comments explain why something is done, not what—that should be clear from the code.

➢ Naming Conventions in Python

Type	Convention	Example
Variable	lower_case_with_underscores	user_name
Function	lower_case_with_underscores	def_total()
Class	CapWords (PascalCase)	StudentDetails
Constant	ALL_CAPS	MAX_SIZE = 100

3. Writing readable and maintainable code.

- Readable Code: Code that is easy to read and understand.
- Maintainable Code: Code that is easy to fix, improve, or update.

➢ Simple Rules to Write Good Code

Rule	Why?
Use good names	Easy to understand
Use 4 spaces for indentation	Keeps code clean and structured
Write comments	Explains your thinking
Use functions	Reuse code, avoid repeating
Don’t hardcode numbers	Easy to update

- # Core Python Concepts

1. Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

1.Integer (int)

- Integers are whole numbers.
- They can be positive or negative, but do not have a decimal point.
- Used for counting, indexing, calculations, etc.
- Example values: 0, 7, -10, 200

2. Float (float)

- Floats are decimal numbers or real numbers.
- They represent numbers with fractional parts.
- Useful for precise measurements like height, temperature, etc.
- Example values: 3.14, -5.6, 0.0

3. String (str)

- A **string** is a sequence of characters (letters, numbers, symbols).
- Strings are always enclosed in quotes: either single ' ' or double " " quotes.
- Used to store textual data like names, messages, etc.
- Example values: "Hello", '123', "@python"

4. List (list)

- A list is an ordered collection of items.
- Lists can store multiple values of any type (including other lists).
- Lists are mutable, meaning we can change, add, or remove items after creating them.
- Defined using square brackets [ ].
- Example: [1, 2, 3], ["apple", "banana"]

5. Tuple (tuple)

- A tuple is similar to a list (it can store a collection of items).
- But tuples are immutable, meaning they cannot be changed once created.
- Useful when data should not be modified.
- Defined using round brackets ( ).
- Example: (10, 20, 30), ("red", "green")

6. Dictionary (dict)

- A dictionary stores data in key-value pairs.
- Each key is linked to a value using a colon : — like a real-world dictionary.
- Keys must be unique, but values can be duplicated.
- Useful for storing related data (e.g., name and age).
- Defined using curly braces { }.
- Example: {"name": "Niyati", "age": 21}

7. Set (set)

- A set is a collection of unordered and unique items.
- It does not allow duplicate values.
- Sets are useful when we want to store items without repetitions.
- Also defined using curly braces { }, but unlike dictionaries, sets have only values, not key-value pairs.
- Example: {1, 2, 3}, {“apple”, “banana”}

2.Python variables and memory allocation.

➢ Introduction to Variables in Python

In Python, a variable is a name that refers to a value stored in memory. Variables allow programmers to store, retrieve, and manipulate data within a program. Unlike some programming languages, Python does not require the variable's type to be declared explicitly. Python determines the data type automatically at runtime based on the value assigned.

➢ Memory Allocation in Python

When a variable is created and assigned a value, Python performs the following steps:

1. Creates an object in memory to store the value.
2. Assigns the object a unique ID (memory address).
3. Links the variable name to that memory location through reference binding.

Python uses automatic memory management, which means the programmer does not need to allocate or free memory manually.

3.Python operators: arithmetic, comparison, logical, bitwise.

➢ Arithmetic Operators

Arithmetic operators are used to perform mathematical calculations.

Operator	Name	Description	Example
+	Addition	Adds two values	a + b
-	Subtraction	Subtracts right operand from left	a - b
*	Multiplication	Multiplies both operands	a * b
/	Division	Divides left operand by right	a / b
//	Floor Division	Divides and returns integer result	a // b
%	Modulus	Returns remainder	a % b
**	Exponentiation	Raises left operand to power of right	a ** b

➢ Logical Operators

Logical operators are used to combine conditional statements. They return True or False.

Operator	Description	Example
and	Returns True if both conditions are true	(a > 10 and b < 5)
or	Returns True if at least one is true	(a > 10 or b < 5)
not	Reverses the result	not(a > 10)

➢ Comparison Operators

Comparison operators are used to compare two values and return a Boolean result (True or False).

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

➢ Bitwise Operators

Bitwise operators work on binary (bit) level. They perform operations on the bitwise representation of integers.

Operator	Name	Description	Example
&	AND	Bits that are 1 in both	a & b
`	`	OR	Bits that are 1 in either
^	XOR	Bits that are 1 in only one	a ^ b
~	NOT	Inverts all bits	~a
<<	Left Shift	Shifts bits left by given number of places	a << 2
>>	Right Shift	Shifts bits right by given number of places	a >> 2

Conditional Statements

1.Introduction to conditional statements: if, else, elif.

- Conditional statements allows your Python program to make decisions based on certain conditions.

1) If statement:

The if statement checks a condition. If it is True, the block of code under it runs.

2) Else statement:

The else statement runs when the if condition is False.

3) Elif statement:

The elif allows you to check multiple conditions.

➢ Basic rules:

- Use colon: after if , elif and else.
- You can have multiple elif, but only one else.

2. Nested if-else conditions.

- Nested if…else condition means using one if…else inside another if…else. This is useful when decisions depend on multiple levels of conditions.
- You can have **if** inside **if** or **if** inside **else**.

❖ Looping (For, While)

1.Introduction to for and while loops

➢ For Loop

A **for** loop is used to iterate over a sequence (such as a list, tuple, dictionary, string, or range). It executes a block of code for each item in the sequence.

Example : Using range()

```
for i in range (5):
    print(i)
```

This prints numbers from 0 to 4.

➢ While Loop

A **while** loop continues to execute a block of code **as long as** a specified condition is **true**.

Example :

```
i = 0
while count < 5:
    print(i)
    i += 1
```

This prints numbers from 0 to 4.

2. How loops work in Python.

- Loops in Python allow you to execute a block of code multiple times without writing it repeatedly. They help automate repetitive tasks, making programs more efficient.

How Loops Work

Python has two main types of loops: **for loops** and **while loops**.

For Loop

A **for loop** iterates over a sequence (like a list, tuple, string, or range) and executes the loop body for each element.

Example of a for loop:

```
for i in range (5):
    print(i)
```

- For each number `i` in the sequence `0,1,2,3,4` Do the following block of code.

- The `print(i)` line runs once per loop cycle, printing the current value of `i`.

Output:

```
0
1
2
3
4
5
```

While Loop

A while loop executes as long as a condition remains True.

Example:

```
count = 0
while count < 3:
    print(count)
    count += 1
```

- Check if count < 3 → yes, so print 0.
- Add 1 to count → now it's 1.
- Check again → still less than 3, so print 1.
- Repeat until count is no longer less than 3.

Output:

```
0
1
2
```

3.Using loops with collections (lists, tuples, etc.)

- Using Loop with List

```
numbers = [1, 2, 3, 4]
for i in numbers:
    print(i)
```

output:

```
1
2
3
4
```

- Using Loop with Tuple

```
colors = ("red", "green", "blue")
for i in colors:
    print(i)
```

output:

```
red
green
blue
```

- Using Loop with Set

```
items = {10, 20, 30}
for i in items:
    print(i)
```

Order may change because sets are unordered.

- Using Loop with Dictionary

```
student = {"name": "Niyati", "age": 20}
for key, value in student.items():
    print(key, value)
```

output:

```
name Niyati
age 20
```

❖ Generators and Iterators

1.Understanding how generators work in Python.

How Generators Work

- When a generator function is called, it does not execute immediately.
- It returns a generator object.
- Each time next() is called, the function runs until it reaches yield, returns that value, and then pauses.
- On the next call, execution continues from where it stopped.

```
def numbers():
    yield 1
    yield 2
    yield 3
gen = numbers()
print(next(gen))
print(next(gen))
print(next(gen))
```

output:

```
1
2
3
```

2.Difference between yield and return

Return:

- Ends the function execution completely
- Sends one final value back to the caller
- Function cannot continue after return

Example:

```
def add(a, b):
  return a + b
```

yield:

- Pauses the function instead of ending it
- Sends values one at a time
- Function state is saved and resumes from the next line

Example:

```
def count():
  yield 1
  yield 2
```

3.Understanding iterators.

An iterator is an object that allows you to traverse elements one by one.
In Python, an iterator follows the iterator protocol, which requires two methods:

- __iter__() → returns the iterator object
- __next__() → returns the next value

When no items are left, __next__() raises StopIteration.

❖ Functions and methods

1. Defining and calling functions in Python.

➢ A function in Python is a reusable block of code that performs a specific task.

Types of Functions

- **Built-in functions**: print(), len(), input(), etc.
- **User-defined functions**: You create these using def.

Example:

```
def add(a, b):
    result = a + b
    print("Sum is:", result)
add(5, 3)              # calling function with passing argument
```

output:

sum is 8

2.Function arguments (positional, keyword, default).

➢ Positional Arguments

➢ Keyword Arguments

➢ Default Arguments

Positional Arguments

These are passed in the order in which parameters are defined.

Example:

```
def info(name, age):
    print("Name:", name)
    print("Age:", age)
info("Niyati", 18)   # Positional: 'Niyati' → name, 18 → age
```

- Order matters: info(21, "Niyati") will give wrong output.

Keyword Arguments

You can pass arguments using the parameter name.

Examples:

```
def info(name, age):
    print("Name:", name)
    print("Age:", age)
info(age=21, name="Niyati")     # Order doesn't matter now
```

- Clear and readable, especially for functions with many parameters.

Default Arguments

You can assign default values to parameters. If no value is provided, the default is used.

Example:

```
def greet(name="Guest"):
    print("Hello", name)
greet("Niyati")   # Output: Hello Niyati
greet()         # Output: Hello Guest
```

3.Scope of variables in Python.

- In Python, the scope of a variable means the region of the program where the variable is recognized and can be used.
- Python has four types of variable scopes:

1. Local Scope

- A variable declared inside a function is said to be in the local scope.
- It can only be used within that function.
- The variable disappears once the function finishes.

2. Global Scope

- A variable declared outside any function is called a global variable.
- It can be accessed throughout the program, in any function.

- To modify a global variable inside a function, you must use the global keyword.

4. Built-in Scope

- Python has some built-in functions and keywords like print(), len(), etc.
- These are always available and are part of Python's built-in scope.

4.Built-in methods for strings, lists, etc.

1. String Methods

Strings are sequences of characters. Some common built-in methods:

Method	Description
lower()	Converts string to lowercase
upper()	Converts string to uppercase
strip()	Removes leading/trailing whitespace
replace(old, new)	Replaces part of the string
split()	Splits string into list

Method	Description
find(sub)	Finds position of substring
len()	Returns length of string
isdigit()	Checks if string contains only digits
startswith(), endswith()	Checks beginning or end of string

2. List Methods

Lists are used to store multiple items. Common methods include:

Method	Description
append(item)	Adds item at the end
insert(index, item)	Inserts item at specified index
remove(item)	Removes first matching item
pop(index)	Removes item at index (default: last)

Method	Description
sort()	Sorts the list (ascending by default)
reverse()	Reverses the list order
extend(list2)	Adds elements from another list
index(item)	Returns index of item
count(item)	Counts occurrences of item

3. Dictionary Methods

Dictionaries store key-value pairs.

Method	Description
keys()	Returns all keys
values()	Returns all values
items()	Returns key-value pairs as tuples
get(key)	Returns value for key (safe)
pop(key)	Removes key-value pair
update(dict2)	Updates dictionary with another

4. Tuple Methods

Tuples are immutable (unchangeable) sequences.

Method	Description
count(item)	Counts occurrences of item
index(item)	Finds index of item

5. Set Methods

Sets store unique, unordered items.

Method	Description
add(item)	Adds item to set
remove(item)	Removes item
discard(item)	Removes item (no error if not found)
union(set2)	Combines two sets
intersection(set2)	Common elements in both sets

Method	Description
difference(set2)	Items in set1 not in set2

❖ Control Statements (Break, Continue, Pass)

1. Understanding the role of break, continue, and pass in Python loops.

➢ break Statement

- The break statement is used to immediately terminate the loop in which it appears.
- Once break is executed, control moves to the next statement outside the loop.
- It is commonly used when a specific condition is met and there is no need to continue looping further.

➢ continue Statement

- The continue statement is used to skip the current iteration of the loop.
- After executing continue, the loop does not terminate, but jumps to the next iteration.
- It is useful when you want to skip some specific conditions but still continue looping.

➢ pass Statement

- The pass statement is a null operation — it does nothing when executed.
- It acts as a placeholder where code is syntactically required but no action is needed.
- It is often used in empty control structures, functions, classes, or loops during development.

❖ String Manipulation

1.Understanding how to access and manipulate strings

A string in Python is a sequence of characters, enclosed in single ' ' or double quotes " ".

Example:

text = "Python"

1. Accessing Characters in a String

You can access individual characters using indexing.

Index starts at 0

Negative index -1 refers to the last character

Example:

```
text = "Python"

text[0]  → 'P'

text[-1] → 'n'
```

Index must be within the length of the string, or it will raise an IndexError.

2. Slicing Strings

You can extract parts of a string using slicing:

Syntax:

string[start:stop:step]

Term	Meaning
start	Starting index (included)
stop	Ending index (excluded)
step	Interval (optional)

Example:

```
text = "Python Programming"

text[0:6]    → 'Python'

text[:6]     → 'Python' (start is 0 by default)

text[7:]     → 'Programming' (till end)

text[::-1]   → reverse the string
```

---

3. Manipulating Strings Using Built-in Methods

Python provides many built-in methods to work with strings:

Method	Description
lower()	Converts to lowercase
upper()	Converts to uppercase
strip()	Removes leading and trailing spaces
replace(old, new)	Replaces one substring with another

split()	Splits the string into a list
join(list)	Joins list items into a string
find(sub)	Returns the index of first match
count(sub)	Counts how many times a substring appears

2.Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.).

Basic Operations on Strings in Python

Python provides powerful and easy-to-use operations to work with strings. The most common ones are:

1. Concatenation (+)

Concatenation means joining two or more strings using the + operator.

Example:

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
```

Output: "Hello World"

2. Repetition (*)

Repetition means repeating a string multiple times using the * operator.

Example:

```
word = "Hi"
print(word * 3)
```

Output: "HiHiHi"

3. Common String Methods

Python has many built-in methods to manipulate strings.

Method	Description	Example	Output
upper()	Converts string to uppercase	"hello".upper()	'HELLO'
lower()	Converts string to lowercase	"HELLO".lower()	'hello'
title()	Capitalizes first letter of each word	"python programming".title()	'Python Programming'
strip()	Removes spaces from both ends	" hello ".strip()	'hello'
replace()	Replaces part of a string	"apple".replace("a", "A")	'Apple'
split()	Splits string into a list	"a,b,c".split(",")	['a', 'b', 'c']
count()	Counts occurrences of a character/word	"hello".count("l")	2
find()	Finds index of the first match	"hello".find("e")	1

3.String Slicing

String Slicing in Python (Theory)

Slicing means extracting a part (substring) of a string using its index positions.

Python allows slicing using this syntax:

string[start : end : step]

Parts of the Slicing Syntax:

Part	Meaning
start	Index to start slicing (included)
end	Index to stop slicing (excluded)
step	Interval between characters (optional)

Indexing in Python Strings

- Indexing starts at 0
- Negative indexing allows access from the end (-1 is the last character)

Example:

text = "Python"

Indexes:	P	y	t	h	o	n
	0	1	2	3	4	5
	-6	-5	-4	-3	-2	-1

Expression	Result	Explanation
text[0:4]	'Pyth'	From index 0 to 3
text[2:]	'thon'	From index 2 to the end
text[:3]	'Pyt'	From start to index 2
text[::2]	'Pto'	Every 2nd character
text[::-1]	'nohtyP'	Reverses the string
text[-3:-1]	'ho'	From index -3 to -2

❖ Advanced Python (map(), reduce(), filter(), Closures and Decorators)

1.How functional programming works in Python

Functional programming is a programming style in which programs are written using functions and immutable data. Python supports functional programming concepts along with other programming paradigms.

Key Points:

1. Functions are treated as first-class objects, meaning they can be passed as arguments and returned from other functions.
2. Pure functions always return the same output for the same input and do not change external data.
3. Lambda functions are small anonymous functions written in a single line.
4. Higher-order functions like map(), filter(), and reduce() work with other functions.
5. Data is generally not modified directly; new data is created instead.

Example:

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

2.Using map(), reduce(), and filter() functions for processing data.

Python provides built-in functional programming tools such as map(), filter(), and reduce() to process collections of data efficiently.

1. map() Function

The map() function applies a given function to each element of an iterable and returns a new iterable.

Syntax:

```
map(function, iterable)
```

Example:

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

output:

```
[2, 4, 6, 8]
```

2. filter() Function

The filter() function selects elements from an iterable that satisfy a condition.

Syntax:

```
filter(function, iterable)
```

Example:

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

output:

```
[2, 4]
```

3. reduce() Function

The reduce() function applies a function **cumulatively** to the elements of an iterable to reduce it to a single value. It is available in the functools module.

Syntax:

```
from functools import reduce
reduce(function, iterable)
```

Example:

```
from functools import reduce

numbers = [1, 2, 3, 4]
result = reduce(lambda a, b: a + b, numbers)
```

Output:

```
10
```

3.Introduction to closures and decorators.

Closures in Python

A closure is a function that remembers the variables from its outer function, even after the outer function has finished executing.

Definition:

A closure occurs when a nested function accesses variables defined in its enclosing scope.

Example:

```
def outer(x):
    def inner():
        return x
    return inner

func = outer(10)
print(func())
```

output:

```
10
```

Decorators in Python

A decorator is a function that modifies the behavior of another function without changing its actual code. Decorators use the concept of closures.

Definition:

A decorator is a function that takes another function as an argument and extends its functionality.

Example:

```
def my_decorator(func):
    def wrapper():
        print("Before function execution")
        func()
        print("After function execution")
    return wrapper

@my_decorator
def say_hello():
    print("Hello")
say_hello()
```

Output:

```
Before function execution

Hello

After function execution
```