**Module – 1 : Python - Fundamentals of Python Language**

**1) Introduction to Python and its Features.**

* Python is interpreted object-oriented, high-level programming language with dynamic semantics.
* Python known for its simplicity and readability.
* Python supports modules and packages, which encourages program modularity and code reuse.
* Python allowing access to many external libraries.
* **Features :**
* **Easy to Learn and Use:** Python has a simple syntax similar to English, making it beginner-friendly.
* **Dynamically Typed:** You don’t need to declare variable types explicitly.
* **Highly portable:** Runs almost anywhere - high end servers and workstations, cross platform.
* **Reduced development time:** Code is 2-10x shorter than C, C++, Java.

**2) History and Evolution of python.**

* Python was conceived in the late 1980s by Guido van Rossum at the Centrum Wiskunde & Informatica (CWI) in the Netherlands.
* Guido aimed to create a language that was easy to read and powerful enough for advanced programming.
* 1989 – ABC Language.
* 1991 – Python (0.9.0)
* 1994 – Python (1.0.0)
* 2000 – Python (2.0.0)
* 2008 – Python (3.0.0) but Python 2 is running parallel.
* 2020 – Python (3.0.0) and Python 2 is retired.
* **Evolution :**
* **Open Source:** Python has always been freely available and open source, encouraging community contributions.
* **Wide Adoption:** Used in web development, data science, automation, AI, and more.
* **Rich Ecosystem:** Thousands of libraries and frameworks have been developed for Python.

**3) Advantages of using Python over other programming languages.**

* **Simple and Readable Syntax:**

Python’s syntax is clear and close to English, making code easy to write and understand.

* **Versatile and Multi-Paradigm:**

Supports procedural, object-oriented, and functional programming styles.

* **Extensive Standard Library:**

Comes with a rich set of modules for tasks like file I/O, networking, web development, and more.

* **Platform Independent:**

Python runs on Windows, macOS, Linux, and many other platforms without modification.

* **Large Community and Support:**

Huge global community, lots of tutorials, forums, and third-party packages.

* **Rapid Development:**

Ideal for prototyping and quick development due to concise code and dynamic typing.

* **Integration Capabilities:**

Easily integrates with other languages (C, C++, Java) and technologies (web, databases).

* **Strong Support for Data Science and AI:**

Popular libraries like NumPy, pandas, TensorFlow, and scikit-learn make Python the top choice for data analysis and machine learning.

* **Open Source:**

Free to use and distribute, with a permissive license.

* **Great for Automation:**

Widely used for scripting and automating repetitive tasks.

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

* **Install Python:**

**Download:** Go to the official Python website and download the latest version for Windows.

**Install**: Run the installer and check the box "Add Python to PATH" before clicking "Install Now".

* **Install VS Code (Recommended Editor):**

**Download:** Go to Visual Studio Code and download the installer for Windows.

* **Install:** Run the installer and follow the setup instructions.
* **Install Python Extension in VS Code:**

Open VS Code.

Go to Extensions (Ctrl+Shift+X).

Search for "Python" and install the official extension by Microsoft.

* **Verify Installation:**

**Open Command Prompt and type:**

python --version

You should see the installed Python version.

* **Create and Run a Python File in VS Code:**

Open VS Code and create a new .py file.

Write your Python code (e.g., print("Hello, Python!")).

Right-click in the editor and select Run Python File in Terminal.

**5) Writing and executing your first Python program.**

* print(“Hello World!”) #**output** ------ Hello World!
* A = 24

B = 6

print(“A+B : ”,A+B) #**output** ------ 30

**6) Understanding Python’s PEP 8 guidelines.**

* PEP 8 is the official style guide for Python code. It provides conventions for writing readable and consistent Python code.
* **Indentation:** Use 4 spaces per indentation level.
* **Maximum Line Length:** Limit lines to 79 characters.
* **Blank Lines:** Use blank lines to separate functions, classes, and blocks of code.
* **Imports:** Imports should be on separate lines and at the top of the file.
* **Spaces:**

No extra spaces inside parentheses, brackets, or braces.

Use a single space after commas, colons, and semicolons.

No spaces before a comma, semicolon, or colon.

* **Comments:** Use comments to explain code, but keep them concise and relevant.
* **String Quotes:** Use single or double quotes consistently.

**7) Indentation, comments, and naming conventions in Python.**

* **Indentation :**

Python uses indentation (spaces) to define code of block.

Use 4 space per indentation level.

No curly braces are used for blocks.

* **Comments :**

Single Line : #

Multi Line : “”” \_\_\_\_\_\_\_\_\_\_\_ “””

* **Naming Conventions:**

Variables and functions: lower\_case\_with\_underscores

Classes: CapWords

Constants: ALL\_CAPS\_WITH\_UNDERSCORES

**8) Writing readable and maintainable code.**

* **Use descriptive variable names:**

Instead of a and b, use names like greeting and target.

* **Add comments if necessary:**

Briefly explain the purpose of the code, especially if it’s not obvious.

* **Consistent formatting:**

Follow consistent indentation and spacing.

**9) Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.**

* **Integer :** Whole number

**Ex:** a = 10

* **Float :** Numbers with decimal

**Ex:** a = 10.25

* **String :** Sequence of characters

**Ex:** a = “Hello! Python is very easy language.”

* **List :** Ordered, mutable collection

**Ex:** a = [1, 2, 5, 9, 6, 7]

* **Tuple :** Ordered, immutable collection

**Ex:** a = (1, 2, 5, 9, 6, 7)

* **Set :** Unordered collection of unique elements

**Ex:** a = {1, 2, 5, 9, 6, 7}

* **Dictionary :** Key-Value pair

**Ex:** a = { “name”: “Alice”, “Age”: 25}

**10) Python variables and memory allocation.**

* Variables are names that reference objects in memory.
* When you assign a value to a variable, Python creates an object in memory and the variable points to it.
* Python uses dynamic typing, so you don’t need to declare the type.
* If you assign one variable to another, both reference the same object (for mutable types, changes affect both).

**11) Python operators: arithmetic, comparison, logical, bitwise.**

* **Arithmetic Operators :** basic mathematical operation.

|  |  |  |  |
| --- | --- | --- | --- |
| **+** | Addition | 5 + 3 | 8 |
| **-** | Subtraction | 5 – 3 | 2 |
| **\*** | Multiplication | 5 \* 3 | 15 |
| **/** | Division | 5 / 3 | 1.67 |
| **%** | Modulus | 5 % 3 | 2 |
| **\*\*** | Exponentiation | 5 \*\* 3 | 125 |
| **//** | Floor division | 5 // 2 | 2 |

* **Comparison Operators :** Used to compare values. Returns true and false.

|  |  |  |  |
| --- | --- | --- | --- |
| **==** | Equal to | 5 == 5 | True |
| **!=** | Not equal to | 5 != 3 | True |
| **>** | Greater than | 5 > 3 | True |
| **<** | Less than | 5 < 3 | False |
| **>=** | Greater or equal | 5 >= 3 | True |
| **<=** | Less or equal | 5 <= 3 | False |

* **Logical Operators :** Used to combine conditional statements.

|  |  |  |  |
| --- | --- | --- | --- |
| **and** | True if both true | True and False | False |
| **or** | True if one is true | True or False | True |
| **not** | Inverts the result | not True | False |

* **Bitwise Operators :** Operate on bits (0s and 1s) of integers.

|  |  |  |  |
| --- | --- | --- | --- |
| **&** | And | 6 & 3 | 2 |
| **|** | Or | 6 | 3 | 7 |
| **^** | XOR | 6 ^ 3 | 5 |
| **~** | Not | ~6 | -7 |
| **<<** | Left shift | 6 << 3 | 48 |
| **>>** | Rightshift | 6 >> 3 | 0 |

**12) Introduction to conditional statements: if, else, elif.**

* **If :** This statement checks a condition and condition is true, then block of code run.

**Ex: if a > b:**

**print(“a is greater than b.”)**

* **else :** When the if condition is false, then execute else block.

**Ex: if a > b:**

**print(“a is greater than b.”)**

**else:**

**print(“b is greater than a.”)**

* **elif :** Used to check multiple condition, only one block (the first True condition) will execute.

**Ex: if a > b and a > c:**

**print(“a is greater than b and c.”)**

**elif b > a and b > c:**

**print(“b is greater than a and c.”)**

**else:**

**print(“c is greater than a and b.”)**

**13) Nested if-else conditions.**

* A nested if-else means placing one if or else statement inside another. This is useful when decisions depend on more than one condition.

**Ex**: **mark = 84**

**if mark >= 40:**

**if mark >= 75:**

**print(“Distinction”)**

**else:**

**print(“Pass”)**

**else:**

**print(“Fail!!”)**

**14) Introduction to for and while loops.**

* **For :** This loop is used to iterate over a sequence (like a list, string, range, etc.).

**Ex : for i in range(1,11):**

**print(i)**

* **While :** This loop runs as long as a condition is True.

**Ex : i = 1**

**while n <= 10:**

**print(i)**

**i += 1**

**15) How loops work in Python.**

* **For :** Thisloop automatically takes each item from a sequence (like a list, string, or range) and executes code for each item.

**In above example, how it works?**

range(1,11) gives [1, -----, 10]

Here 11(end) is not counted in loop.

Then, i takes [1, -----, 10]

And print on output window.

* **While :** This loop runs as long as a condition is True. It keeps checking the condition before every iteration, and stops when the condition becomes False.

**In above example, how it works?**

Start with i = 1

Check : i <= 10 ? Yes --> print 1

Increment value of i ----> i += 1

Check : i <= 10 ? Yes --> print 2

Repeats until i reaches 11

Then stop the loop.

And print on output[1, ----, 10] window.

**16) Using loops with collections (lists, tuples, etc.).**

* **Collections :** Lists, tuples, sets, and dictionaries store multiple items.
* **For loop :** Best for iterating directly over elements.

**Ex : for item in my\_list:**

**print(item)**

* **Index-based**: Use range(len(collection)) or enumerate() if you need the index.

**Ex : for i, val in enumerate(my\_list):**

**print(i, val)**

* **Tuples :** Iterated like lists but immutable.
* **Sets :** Iterated like lists but order is not guaranteed.
* **Dictionaries:**

Keys --> for key in dict:

Keys & values --> for k, v in dict.items():

* **While loop :** Used with index control.

**Ex : i = 0**

**while i < len(my\_list):**

**print(my\_list[i])**

**i += 1**

**17) Understanding how generators work in Python.**

* **Definition:** Generators are special functions that yield values one at a time instead of returning them all at once.
* **Why use them:**

Memory efficient (don’t store all results in memory)

Lazy evaluation (produce items only when needed)

* **How to create:**Use yield instead of return inside a function.

**Ex : def my\_gen():**

**yield 1**

**yield 2**

* **How to use :**

**Ex : for value in my\_gen():**

**print(value)**

* **Normal function :-** Returns value once, ends.
* **Generator function :-** Pauses at yield, resumes from there on next call.

**18) Difference between yield and return.**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **return** | **yield** |
| **Function type** | Normal function | Generator function |
| **Execution** | Ends function immediately | Pauses function and can resume |
| **Values produced** | Returns **one final value** | Can produce multiple values one at a time |
| **Memory usage** | Stores all results in memory | Generates values on demand (memory efficient) |
| **State saving** | Does not save state | Saves current state for next iteration |
| **Usage** | Used when you need all data at once | Used when you want to iterate over results lazily |

**19) Understanding iterators and creating custom iterators.**

* An iterator is an object that can be iterated (looped) over.
* It implements two methods:

\_\_iter\_\_() → Returns the iterator object itself.

\_\_next\_\_() → Returns the next item from the collection.

* Raises Stop Iteration when there are no more items.
* **Built - In :**

Ex : my\_list = [1, 2, 3]

it = iter(my\_list)

print(next(it)) # 1

print(next(it)) # 2

* **Custom :**

Define a class.

Implement \_\_iter\_\_() and \_\_next\_\_().

**20) Defining and calling functions in Python.**

* A function is a block of reusable code.
* Defined using the def keyword.

**Ex : def greet(name):**

**print(“Hello,”, name)**

**greet(“Vivek”)**

**21) Function arguments (positional, keyword, default).**

* **Positional :**

Passed in the same order as defined in the function.

Order matters.

* **Keyword** :

Specify argument name when calling.

Order doesn’t matter.

* **Default :**

Function parameters can have default values.

If no value is given, the default is used.

**22) Scope of variables in Python.**

* Scope of a variable in Python defines where in the program the variable can be accessed.
* Python follows the LEGB Rule:

L = Local (inside a function)

E = Enclosing (in outer function for nested functions)

G = Global (declared at the top level of the program)

B = Built-in (Python's reserved keywords/functions)

**1. Local Scope**

* Variables declared inside a function.
* Accessible only within that function.
* Created when function starts, destroyed when it ends.

**2. Enclosing Scope**

* Variables in the outer (enclosing) function for nested functions.
* Accessible to inner functions if not overridden.

**3. Global Scope**

* Variables declared outside all functions.
* Accessible anywhere in the program unless shadowed by a local variable.

**4. Built-in Scope**

* Name reserved by python (**ex** : print, len, max).
* Always available but can be overridden.

**23) Built-in methods for strings, lists, etc.**

* These methods help in manipulating, accessing, and processing data without writing extra logic.
* **mystr** = “Hello! python is interpred, object oriented and high level programming language.”
* **String :**

**upper() :** Convert to uppercase

**Ex :** print(mystr.upper())

**lower() :** Convert to lowercase

**Ex :** print(mystr.lower())

**count() :** Count occurrences

**Ex :** print(mystr.count(‘o’))

**strip() :** Remove whitespace from start and end

**Ex :** print(mystr.strip())

**replace(a, b) :** Replace Substring

**Ex :** print(mystr.replace(“python”, “java”))

**split() :** Split into substring

**Ex :** print(mystr.split(“ ,”))

**capitalize() :** Capitalize the first letter of the string

**Ex :** print(mystr.capitalize())

**title() :** Capitalize the first letter of each word

**Ex :** print(mystr.title())

**casefold() :** Convert the string to lowercase for case-insensitive comparisons.

**Ex :** print(mystr. casefold())

**startswith() :** Check if the string starts with “Hello” and return boolean value.

**Ex :** print(mystr. startswith(“Hello”))

**endswith() :** Check if the string ends with “!” and return boolean value.

**Ex :** print(mystr.endswith(“!”))

**find() :** Find the first occurrence of “python” and return its index, or -1 if not found.

**Ex :** print(mystr.find(“python”))

**index() :** Check if the string ends with “object” and return boolean value.

**Ex :** print(mystr.index(“object”))

* **String Properties :**

**mystr.isalpha()** = It checks if all characters in the string are alphabetic.

**mystr.isdigit()** = It checks if all characters in the string are digits.

**mystr.isalnum()** = It checks if all characters in the string are alphanumeric (letters and numbers).

**mystr.islower()** = It checks if all characters in the string are lower.

**mystr.isupper()** = It checks if all characters in the string are lower.

**24) Understanding the role of break, continue, and pass in Python loops.**

* **Break :** Stops the loop completely.

**Ex : for i in range(10):**

**if i == 4:**

**break**

**print(i)**

* **Continue :** Skips the current iteration and goes to the next.

**Ex : for i in range(10):**

**if i == 5:**

**continue**

**print(i)**

* **Pass :** Does Nothing (Placeholder)

**Ex :** **for i in range(10):**

**pass**

**25) Understanding how to access and manipulate strings.**

* A string in Python is a sequence of characters enclosed in either:

Single quotes ‘ ’

Double quotes “ ”

Triple quotes ‘’’ ‘’’ or “”” “”” (for multi-line strings)

* Strings are indexed sequences.
* **Indexing :**

Starts from 0 for the first character.

Negative indexing (-1) starts from the last character.

* **Slicing :**

Slicing is used to extract a part of the string.

start :- index to begin (default 0)

end :- index to stop (excluded)

step :- interval between characters (default 1)

* Strings are immutable, once created, their content cannot be changed.
* String Operation : concatenation, repetition, string methods……..
* Strings are widely used for data storage, manipulation, and display.

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

* **Concatenation :** Join the two string.

**Ex: a = “Hello”**

**b = “World”**

**print(a + “ ” + b)**

* **Repetition :** Using the **\*** operator with a string and an integer. It returns a new string repeated **n** times.

**Ex : print(“Hi” \* 3)**

* **upper() :** Converts to uppercase.
* **lower() :** Converts to lowercase.
* **strip() :** Removes spaces from both ends.
* **replace() :** Replace substring.
* **join() :** Joins list into string.
* **split() :** Splits into substring.
* **find() :** Finds index of substring.
* **count() :** Counts occurrences.

**27) String slicing.**

* String slicing in Python extracts a part of a string using the syntax.

* string[start : end : step]
* **mystr** = “Hello! python is interpred, object oriented and high level programming language.”

print(mystr[9]) print(mystr[2:12:2])

print(mystr[4:12]) print(mystr[:5])

**28) How functional programming works in Python.**

* **Definition** **:** A style where functions are first-class citizens (can be stored, passed, and returned).
* **Focus** **:** What to do, not how to do it.
* **Key Ideas :**

**Pure functions** **:** No side effects, same input -- same output.

**Immutability** **:** Avoid changing data directly.

**Higher-order functions** **:** Functions that take/return other functions.

* **Common Tools :**

**map(func, iterable)** **:** Apply function to all items.

**filter(func, iterable)** **:** Keep items that match a condition.

**reduce(func, iterable)** **:** Combine items into one value.

**lambda** **:** Anonymous small function.

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

* **map() :**

**Purpose :** Apply a function to each element of an iterable.

**Syntax :** map(function, iterable)

**Returns :** map object (convert to list() if needed).

* **filter() :**

**Purpose :** Keep elements that match a condition.

**Syntax :** filter(function, iterable)

**Returns :** filter object (convert to list() if needed).

* **reduce() :**

**Purpose :** Reduce sequence to a single value by repeatedly applying a function.

**Syntax :** reduce(function, iterable)

**Returns :** Single value.

**30) Introduction to closures and decorators.**

* **Closures :**
* **Definition :** A closure is a function inside another function that remembers variables from the outer function even after the outer function has finished executing.
* **Key Points :**

Inner function can access non-local variables.

Useful for data hiding and creating function factories.

* **Decorators :**
* **Definition :** A decorator is a function that takes another function as input, adds extrafunctionality, and returns a new function.
* **Often used for :**

Logging

Authentication

Measuring execution time

* **Syntax :**

Uses @decorator\_name before a function.