**Module – 13**

**Introduction To Python**

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

**Ans. Python is an interpreted, object-oriented, high level programming language with dynamic semantics.**

* **Clean syntax plus high level data types**
* **Uses white space to delimit blocks**
* **Variables do not need declaration**

1. **History and evolution of Python.**

**Ans. Python is a widely used general-purpose, high level programming language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed to emphasize code readability, and its syntax allows programmers to express concepts in fewer lines of code.**

**Evolution: The language was finally released in 1991. When it was released, it used a fewer codes to express the concepts, when we compare it with java, c++ & c. its design philosophy was quite good too.**

1. **Advantages of using Python over other programming languages.**

**Ans. Advantages are as follows:**

* **Simplicity & Readability**
* **Large standard library**
* **Vast eco-system**
* **Cross platform compability**
* **Rapid prototyping**
* **Strong community support**

1. **Understanding Python’s PEP 8 guidelines.**

**Ans. Guidelines are as follows:**

* **Indentation**
* **Maximum line length**
* **Blank lines**
* **Imports**
* **Naming convention**
* **White space**
* **Comments & Docstrings**

1. **Indentation, comments, and naming conventions in Python.**

**Ans. Indentation: python uses indentation to define code blocks(like loops, conditional & functions), not braces like C/C++.**

* **Uses 4 spaces per indentation level (never tab).**
* **Consistent indentation is critical – mismatched spacing will raise an Indentation Error.**

**Comments: Comments help explain your code to others (and your future self).**

* **Single line comments Start with #**
* **Block comments explain a section of code and usually placed above.**
* **Docstring (‘’’Triple quotes’’’)**

**Naming convention: PEP 8 recommends specific style for different identifiers.**

* **Variables & Functions**
* **Constants**
* **Classes**
* **Avoids single-character names except for counters (I,j) in loops.**

1. **Writing readable and maintainable code.**

**Ans.**

* **Use clear and descriptive names**
* **Keep function small and focused**
* **Consistent formatting**
* **Comment thoughtfully**
* **Avoid code duplication**
* **Structure your code logically**
* **Write tests**
* **Refractor regularly**

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

**Ans.**

* **Integer (int): E.g.: x=10**
* **Float (float): E.g.: pi=3.14**
* **String (str): E.g.: name= ”Bhagirath”**
* **List (list): E.g.: fruits=[“apple”, “banana”, “cherry”]**
* **Tuple (tuple): E.g.: coordinates=(10.0, 20.0)**
* **Dictionary (dict): E.g.: student={“name”: “bhagirath”, “age”:27}**
* **Set (set): E.g.: Unique\_numbers={1,2,3}**

1. **Python variables and memory allocation.**

**Ans. Variables are references, not containers. When you write x = 10, you're creating an integer object 10 in memory and binding the name x to it.**

* **You don’t declare types explicitly—Python figures it out at runtime**

**Memory allocation: Python handles memory automatically using a system of:**

* **Private Heap: All Python objects and data structures are stored in a private heap managed by the interpreter.**
* **Reference Counting: Each object keeps track of how many references point to it. When the count hits zero, the memory is released.**
* **Garbage Collection: Python also uses a garbage collector to clean up objects involved in reference cycles (like two objects referencing each other).**

**Behind the scenes**

**- Stack memory stores references and function calls.**

**- Heap memory stores the actual objects.**

**- Python optimizes memory by reusing immutable objects like mall integers and strings.**

1. **Python operators: arithmetic, comparison, logical, bitwise.**

**Ans.**

**Arithmetic Operators:**

* **Addition + : 5+2=7**
* **Subtraction - : 5-3=2**
* **Multiplication \* : 5\*3=15**
* **Division / : 5/2=2.5**
* **Floor division // : 5//2=2**
* **Modulus % : 5%2=1**
* **Exponentiation \*\* : 2\*\*3=8**

**Comparison operator Used to compare values; return True or False:**

* **Equal to ==**
* **Not equal to !=**
* **Greater than >**
* **Less than <**
* **Greater than or equal to >=**
* **Less than or equal to <=**

**Logical operators Used to combine conditional statements:**

* **and : true if both are true**
* **or : true if at least one is true**
* **not : inverts the true value**

**Bitwise operator Operate at the binary level:**

* **& AND**
* **| OR**
* **^ XOR**
* **~ NOT**
* **<< Left shift**
* **>> Right shift**

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

**Ans. if Statement executes a block of code only if a condition is true.**

* **else Statement Provides an alternative block if the if condition is false.**
* **elif (short for "else if") Checks another condition if the previous if or elif was false.**

**Key points:**

* **Conditions must evaluate to True or False.**
* **Indentation (usually 4 spaces) defines the block of code.**
* **You can chain multiple elif statements for complex logic.**

1. **Nested if-else conditions.**

**Ans.**

**x = 10**

**if x > 0:**

**print("Positive number")**

**if x % 2 == 0:**

**print("Even number")**

**else:**

**print("Odd number")**

**else:**

**print("Not a positive number")**

**- First, it checks if x is positive.**

**- If true, it goes deeper to check if it's even or odd.**

**- if the first condition is false, it skips the inner block and prints the final else.**

1. **Introduction to for and while loops.**

**Ans. For loop: Used when you want to iterate over a sequence (like a list, string, or range of numbers).**

* **range(3) generates numbers 0, 1, 2.**
* **range(3) generates numbers 0, 1, 2.**

**While loop: Used when you want to repeat a block of code as long as a condition is true.**

**The loop continues until count is no longer less than 3.**

**Key difference: Use for when you know how many times to loop.**

**Use while when the number of repetitions depends on a condition.**

1. **How loops work in Python.**

**Ans. Loops in Python allow you to repeat a block of code multiple times, making your programs more efficient and concise.**

**for Loops – Iterate Over a Sequence**

**while Loops – Repeat While a Condition is True**

**A while loop keeps running as long as a condition is true.**

**Loop control system:**

* **break: Exit the loop early.**
* **Continue: skip the current iteration and move to the next**
* **Else: runs after the loop finishes (unless it was broken)**

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

**Ans. Using loops with collections in Python is a powerful way to process data efficiently. Here's how it works with different types of collections.**

**List & Tuple: You can iterate over each element directly**

**String: Strings are sequences too, so you can loop through each character.**

**Dictionary: You can loop through keys, values, or both**

**Sets: Sets are unordered collections of unique items**

1. **Understanding how generator works in phyton.**

**Ans. Generators in Python are a powerful way to create iterators—but with a twist: they let you produce values *on the fly*, rather than storing them all in memory at once. This makes them perfect for handling large datasets or infinite sequences**

**What is generator?**

**A generator is a special type of function that uses the yield keyword instead of return. When called, it returns a generator object that can be iterated over one value at a time.**

**Each time yield is hit, the function pauses and saves its state. The next time you call next() on the generator, it resumes right where it left off.**

**How it works internally?**

**When the generator function is called, it doesn’t run immediately—it returns a generator object.**

* **Each call to next() resumes execution until the next yield.**
* **When the function ends or hits a return, it raises StopIteration.**

**Why use generator:**

* **Memory-efficient: They don’t store the entire sequence in memory.**
* **Lazy evaluation: Values are generated only when needed.**
* **Clean syntax: Easier to write and read than manually managing iterators.**

1. **Difference between yield and return.**

**Ans. The key difference between yield and return in Python lies in how they send values back and how the function behaves after doing so**

|  |  |  |
| --- | --- | --- |
|  | **Yield** | **Return** |
| **1** | **Pauses the function and remembers its state.** | **Ends the function immediately** |
| **2** | **Returns a value to the caller, but the function can resume from where it left off.** | **Sends a single value (or a tuple of values) back to the caller.** |
| **3** | **Used to create generators, which are memory-efficient and lazy-evaluated** | **The function’s state is not preserved.** |

1. **Understanding iterators and creating custom iterators.**

**Ans. In Python, iterators are objects that allow you to traverse through all the elements of a collection, one at a time. They’re the backbone of loops like for and are built on a simple but powerful protocol.**

**An iterator is any object that implements two methods:**

* **\_\_iter\_\_() → returns the iterator object itself**
* **\_\_next\_\_() → returns the next value or raises StopIteration when done.**

**Creating a custom iterator: You can build your own iterator by defining a class with \_\_iter\_\_() and \_\_next\_\_() methods.**

**Why use custom iterators?**

* **To model sequences that don’t fit built-in types.**
* **To handle large or infinite data streams efficiently.**
* **To encapsulate complex iteration logic cleanly.**

**You can also use generators (with yield) as a shortcut for creating iterators—they’re simpler and more memory-efficient for many use cases.**

1. **Defining and calling functions in Python.**

**Ans. In Python, functions are reusable blocks of code that perform a specific task. They help make your code modular, readable, and easier to maintain. Here's how you define and call them.**

**Defining:**

**Use the def keyword followed by the function name and parentheses. If the function takes parameters, list them inside the parentheses.**

* **Grit is the function name.**
* **Name is a parameter.**
* **The inside the function is intended.**

**Calling a function: To call a function, just use its name followed by parentheses. If it takes arguments, pass them inside the parentheses.**

1. **Defining and calling functions in Python.**

**Ans. Defining a function: in python to define a function def is used**

**Def greet():**

**Print(“Bhagirath”)**

**Calling a function: To call the function:**

**Greet()**

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

**Ans. Positional Arguments: These are passed to the function in the same order as they’re defined.**

**def describe\_person(name, age):**

**print(f"{name} is {age} years old.")**

**describe\_person("Bhagirath", 21)**

**Keyword Arguments:** **You explicitly name the parameters when calling the function—order doesn’t matter.**

**describe\_person(age=21, name="Bhagirath")**

**Default Arguments**

**You can assign default values so the argument becomes optional unless overridden.**

**def greet(name="User"): print(f"Hello, {name}!") greet() # Uses default greet("Bhagirath") # Overrides default**

1. **Scope of variables in Python**

**Ans. Local Scope: A variable defined *inside* a function has local scope—it only exists within that function.**

**Global Scope:**

**A variable defined *outside* any function is in the global scope—accessible from anywhere.**

**Enclosing (Nonlocal) Scope:**

**This happens in *nested functions*—an inner function can access variables from its enclosing (outer) function.**

**Built-in Scope**

**This is Python's own reserved namespace—like print(), len(), etc.**

1. **Built-in methods for strings, lists, etc.**

**Ans. Built in methods for string and lists are as follows:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| **Lowercase()** | **Converts to lower case** |
| **Upper()** | **Converts to upper case** |
| **Capitalize()** | **Capitalize first letter** |
| **Strip()** | **Removes white space from end** |
| **Replace(a,b)** | **Replace substring a with b** |
| **Split(delim)** | **Splits the string into list** |
| **Find(substr)** | **Finds first index of sub string** |
| **Isanum()** | **Check if all character is alphanumeric** |
| **Isalph()** | **Check if all character is letters** |
| **Startswith(val)** | **Check if string start with val** |

**List Methods:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| **Append(x)** | **Add x to the end of list** |
| **Insert(I,x)** | **Insert x at index I** |
| **Pop(i)** | **Remove and return item at index i** |
| **Remove (x)** | **Removes the first occurrence of x** |
| **Sort()** | **Sorts the list in ascending order** |
| **Reverse()** | **Reverse the list** |
| **Extend(list2)** | **Adds elements of list2 to the end** |
| **Count(x)** | **Counts how many times x occurs** |
| **Index(x)** | **Returns the index of the first x** |

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

**Ans. Break:** **It immediately stops the loop, even if the condition hasn’t finished.**

**for num in range(1, 10):**

**if num == 5:**

**break**

**print(num)**

**Continue — *Skip the current iteration and move to the next one***

***for num in range(1, 6):***

***if num == 3:***

***continue***

***print(num)***

**Pass — *Do nothing (acts as a placeholder)***

**for num in range(1, 6):**

**if num == 3:**

**pass # Placeholder for future logic**

**print(num)**

1. **Understanding how to access and manipulate strings.**

**Ans. Access string:** **Strings in Python are just sequences of characters, so you can access them like lists using *indexing* and *slicing*:**

**text = "Python"**

**print(text[0]) # Output: 'P' (first character)**

**print(text[-1]) # Output: 'n' (last character)**

**print(text[1:4]) # Output: 'yth' (characters from index 1 to 3)**

**Manipulating String:**

|  |  |  |
| --- | --- | --- |
| **Method** | **What it does** | **Example** |
| **Lower()** | **Converts lower case** | **“HELLO”.lower – hello** |
| **Upper()** | **Converts upper case** | **“hello”.upper-HELLO** |
| **Strip()** | **Removes whitespace** | **“name ”.strip - name** |
| **Replace(a,b)** | **Replace character** | **“blue sky”.replace- blue, red** |
| **Split()** | **Split string into list** | **“a,b,c”.split(“,”) – [a,b,c]** |
| **Join()** | **Joins list into string** | **“”.join([‘i’,’love’,’code’])** |

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

**Ans.**

**Concatenation:**

**First\_name = “Bhagirath”**

**Last\_name = “Gajera”**

**Full\_name = First\_name + “ ” + Last\_name**

**Print(Full\_name)**

**Repetition:**

**Line = “-=”\*10**

**Print (line)**

|  |  |  |
| --- | --- | --- |
| **Method** | **Use case** | **Example** |
| **Upper()** | **All uppercase letters** | **“hello”.upper()-‘HELLO’** |
| **Lower()** | **All lowercase letters** | **“HELLO”.lower()-‘hello’** |
| **Capitalize()** | **First letters uppercase** | **“hello”.capitalize()- ‘Hello’** |
| **Title()** | **Every first letter is capitalized** | **“hello world”.capitalized – ‘Hello World’** |
| **Strip()** | **Removes White space from start & end** | **“ name ”.strip() – ‘name’** |
| **Replace(‘a’,’b’)** | **Replace substrings** | **“abc”.replace(“a”,”x”)-‘xbc’** |
| **Split()** | **Splits a string into list** | **“a,b,c”.split(“,”)- [‘a’,’b’,’c’]** |

1. **String slicing.**

**Ans. String[start:stop:step]**

**Start: index to begin (inclusive)**

**Stop: index to end (exclusive)**

**Step: how many characters to skip (optional)**

**Text=”Bhagirath”**

|  |  |  |
| --- | --- | --- |
| **Slice** | **Meaning** | **Output** |
| **Text[0:3]** | **First 3 characters (0 to 2)** | **‘Bha’** |
| **Text[4:]** | **From index 4 to end** | **‘irath’** |
| **Text[:5]** | **From start to index 4** | **‘Bhagi’** |
| **Text[::2]** | **Every second character** | **‘Bagrt’** |
| **Text[::-1]** | **Full reverse of the string** | **‘htarigahB’** |
| **Text[-3:]** | **Last 3 character** | **‘ath’** |

1. **How functional programming works in Python.**

**Ans. Here’s functional programming in Python**

* **Pure Functions: No side effects, same input always gives same output.**
* **Immutability: Avoid changing variables—return new values instead.**
* **First-Class Functions: Functions can be passed, returned, and stored like data.**
* **Lambda: Small, anonymous functions.**
* **map(): Transforms each item in a list.**
* **filter(): Filters items based on a condition.**
* **reduce() (from functools): Combines items into one value.**

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

**Ans. map() - Transforms each item using a function.  
map(lambda x: x\*2, [1, 2, 3]) - [2, 4, 6]**

* **filter() - Keeps items that match a condition.**
  + **filter(lambda x: x % 2 == 0, [1, 2, 3]) - [2]**
* **reduce() - Combines all items into one value.**
  + **reduce(lambda x, y: x + y, [1, 2, 3]) – 6**

1. **Introduction to closures and decorators.**

**Ans. Closures: Functions that *remember* variables from their enclosing scope, even after the outer function has finished.**

**Decorators: Functions that *wrap* another function to add extra behavior—used with @decorator\_name syntax.**