Module 6) Python Fundamentals

1.Introduction to Python

Theory:

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

Python is a high-level, interpreted programming language known for its simplicity and readability. Python is an interpreted language, which means the code is executed directly by an interpreter line by line, without needing to be compiled first. This makes the development process faster and easier to debug. It's also dynamically typed, so you don't have to declare the data type of a variable beforehand. The interpreter automatically determines the type at runtime. Python is highly versatile and supports various programming paradigms, including object-oriented, procedural, and functional programming.

History and evolution of Python.

The programming language Python was conceived in the late 1980s, and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands as a successor to ABC capable of exception handling and interfacing with the Amoeba operating system.

Python 2.0 was released on October 16, 2000, with many major new features, such as list comprehensions, cycle-detecting garbage collector, reference counting, memory management and support for Unicode, along with a change to the development process itself, with a shift to a more transparent and community-backed process.

Python 3.0, a major, backwards-incompatible release, was released on December 3, 2008.

As of 9 August 2025, Python 3.13.6 is the latest stable release. This version currently receives full bug-fix and security updates, while Python 3.12—released in October 2023—had active bug-fix support only until April 2025, and since then only security fixes

Advantages of using Python over other programming languages.

Python stands out due to its simplicity and readability, using a clean syntax that's easy for beginners to learn and for experienced developers to maintain. It has a massive collection of libraries and frameworks, which dramatically speeds up development for tasks like web development, data science, and machine learning. Additionally, Python is a versatile, cross-platform language that can be used for a wide variety of applications and runs on various operating systems, making it a highly adaptable tool.

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

Install Python: Download the latest version of Python from the official website and run the installer. Make sure to check the box that says "Add Python to PATH" during installation. This allows you to run Python from any command line window.

Writing and executing your first Python program.

Def funct1():

    Print(“Hello world”)

funct1()

Lab:

Write a Python program that prints "Hello, World!".

Def funct1():

    Print(“Hello world”)

funct1()

Set up Python on your local machine and write a program to display your name.

```
In [1]: def func1():
            print("MALLARAPU YAGNESH NAIDU")

        func1()
```

```
MALLARAPU YAGNESH NAIDU
```

2. Programming Style

Theory:

Understanding Python's PEP 8 guidelines.

PEP 8 is Python's style guide for writing clear, readable, and consistent code. It provides conventions for code layout, naming, and other best practices.

Indentation, comments, and naming conventions in Python.

In Python, indentation is crucial, defining code blocks and structure, unlike other languages that use curly braces. Comments, marked with #, are used for explaining code. Following naming conventions like snake_case for variables and functions (PEP 8) ensures readability and consistency.

Writing readable and maintainable code.

Writing readable and maintainable code is crucial for project longevity and collaboration. This means consistently following PEP 8 guidelines for formatting and style, such as using snake_case for variables and functions. You should use meaningful names for variables, functions, and classes to make the code self-documenting.

Additionally, you should write concise, well-documented code by using docstrings to explain complex functions and inline comments to clarify non-obvious logic. Finally, keeping functions small and focused on a single task makes them easier to test, debug, and reuse. These practices collectively ensure your code is clear, logical, and easy for others—and your future self—to understand.

Lab:

Write a Python program that demonstrates the correct use of indentation, comments, and variables following PEP 8 guidelines.

In [3]:
```
def func2(a,b):
    c=a+b   #indendation              #assigning added sum to value c
    return c


x=2
y=9
print("your addition answer is",func2(x,y))
```

```
your addition answer is 11
```

3. Core Python Concepts

Theory:

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

Integers: Whole numbers without a decimal point.

x=9

Print(x)

Floats: Numbers with a decimal point.

y=5.6

Print(y)

Strings: Sequences of characters enclosed in quotes.

a=”yagnesh”

print(a)

Lists: Ordered, mutable collections of items.b=[1,2,3,”ram”]

Tuples: Ordered, immutable collections of items.C=(1,2,3)

Dictionaries: Unordered collections of key-value pairs. D = {"name": "yagnesh", "roll_no": 20}

Sets: Unordered collections of unique items.E={1,2,43,24}

Python variables and memory allocation.

In Python, a variable is a label or a name that points to an object in memory. When you assign a value (like x = 10), Python creates an object for 10 and the variable x simply references its memory location. Multiple variables can point to the same object, which saves memory.

Python operators: arithmetic, comparison, logical, bitwise.

Arithmetic: Used for mathematical operations like addition, subtraction, multiplication, and division.(+,-,*,/,//,%)

Comparison: Used to compare two values, returning a boolean (True or False).

(<,>,=,<=,>=)

Logical: Used to combine conditional statements with and, or, and not.

(&,|,!)

Bitwise: Used to perform operations on binary representations of numbers.

(&&,||).

Lab:

Write a Python program to demonstrate the creation of variables and different data types.

```
In [4]: x=9
        print(type(x))

        <class 'int'>

In [5]: x=9.0
        print(type(x))

        <class 'float'>

In [6]: x="yagnesh"
        print(type(x))

        <class 'str'>

In [7]: x=[1,2,3,"ram"]
        print(type(x))

        <class 'list'>

In [8]: x=(1,2,3)
        print(type(x))

        <class 'tuple'>

In [10]: z={"name":"yagnesh"}
         print(type(z))

         <class 'dict'>

In [11]: a={"a","b","c"}
         print(type(a))

         <class 'set'>
```

Practical Example 1: How does the Python code structure work?

The Python code structure works through a set of rules governing indentation, code blocks, and the execution flow. Unlike many other languages that use curly braces {} to define code blocks, Python uses indentation.

Practical Example 2: How to create variables in Python?

To create a variable, type the variable name, an equal sign, and the value.

Practical Example 3: How to take user input using the input() function.

n=input(“enter number string:”)

print(“the number or string you entered is “,n)

Practical Example 4: How to check the type of a variable dynamically using type().

```
In [13]: n=input("enter value:")
         print(type(n),"is the type of the value you entered",n)

         enter value:yagnesh
         <class 'str'> is the type of the value you entered yagnesh
```

4. Conditional Statements Theory:

Theory:

Introduction to conditional statements: if, else, elif.

Conditional statements (if, elif, else) are used to make decisions in your code. They allow you to execute different blocks of code based on whether a condition is True or False. This structure provides a way for programs to react dynamically to different inputs or situations.

Nested if-else conditions.

Nested if-else conditions are conditional statements placed inside other conditional statements. They allow for more complex decision-making by checking a new condition only after an outer condition is met. This structure creates a hierarchy of checks, providing fine-grained control over code execution.

Lab:

Practical Example 5: Write a Python program to find greater and less than a number using if_else.

In [15]:
```
x=10
y=2
if(x>y):
    print(x,"is greater than ",y)
else:
    print(y,"is greater than",x)
```

```
10 is greater than  2
```

Practical Example 6: Write a Python program to check if a number is prime using if_else.

In [21]:
```
n=int(input("enter a number:"))
def func1(n):
    for i in range(2,n-1):
        if (n%i==0):
            return True
        else:
            return False


if func1(n):
    print(n,"is not a prime number")

else:
    print(n,"is a prime number")
```

```
enter a number:7
7 is a prime number
```

Practical Example 7: Write a Python program to calculate grades based on percentage using if-else ladder.

In [23]:
```
percentage = float(input("Enter the percentage: "))

if percentage >= 90:
    print("Grade: A")
elif percentage >= 80:
    print("Grade: B")
elif percentage >= 70:
    print("Grade: C")
elif percentage >= 60:
    print("Grade: D")
else:
    print("Grade: F")
```

```
Enter the percentage: 98.2
Grade: A
```

Practical Example 8: Write a Python program to check if a person is eligible to donate blood using a nested if.

In [24]:
```
age = int(input("Enter your age: "))
weight = float(input("Enter your weight in kg: "))

if age >= 18:
    if weight >= 50:
        print("You are eligible to donate blood.")
    else:
        print("You are not eligible. Your weight is below the minimum requirement.")
else:
    print("You are not eligible. Your age is below the minimum requirement.")
```

```
Enter your age: 21
Enter your weight in kg: 62
You are eligible to donate blood.
```

5. Looping (For, While)

Theory:

Introduction to for and while loops.

for and while loops are used to execute a block of code multiple times.

A for loop is used when you know exactly how many times you want to loop. It iterates over a sequence (like a list, tuple, or string) and executes a block of code for each item in that sequence. This makes it ideal for tasks where you need to perform an action on every item in a collection.

A while loop, on the other hand, is used when the number of iterations is unknown. It continues to execute a block of code as long as a specified condition is true. The loop stops only when the condition becomes false. This makes it perfect for scenarios like waiting for user input or processing data until a certain state is reached.

How loops work in Python.

Loops in Python repeatedly execute a block of code. A for loop iterates over a sequence (like a list), running the code for each item. A while loop continues to execute as long as a specified condition remains True. Both are essential for automating repetitive tasks.

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

Using loops with collections like lists and tuples is a fundamental way to process every item they contain. A for loop is the most common and efficient method for this. It automatically iterates over each element in the collection, from beginning to end, and assigns it to a temporary variable for you to use within the loop's code block. This simplifies tasks like printing every item in a list or performing a calculation on each number. You don't need to manually track indices or check for the end of the collection, which makes the code cleaner and less prone to errors. while loops can also be used, but they require more manual setup with index management.

Lab:

Practical Example 1: Write a Python program to print each fruit in a list using a simple for loop. List1 = ['apple', 'banana', 'mango']

```
In [26]: List1 = ['apple', 'banana', 'mango']
         for i in List1:
             print(i)
```

```
apple
banana
mango
```

- **Practical Example 2: Write a Python program to find the length of each string in List1.**

```
In [29]: List1 = ['apple', 'banana', 'mango']
         for i in List1:
             print(len(i))
```

```
5
6
5
```

- **Practical Example 3: Write a Python program to find a specific string in the list using a simple for loop and if condition.**

```
In [36]: item=input("enter string to find in the list:")


         List1 = ['apple', 'banana', 'mango']


         for i in List1:
             if (item==i):
                 print(item,"found in the list at position",List1.index(i))
                 break
             else:
                 print(item,"entered in not in the list")
```

```
enter string to find in the list:apple
apple found in the list at position 0
```

Practical Example 4: Print this pattern using nested for loop: markdown

Copy code

```
*

**

***

****

*****
```

In [45]:

```
for i in range(1, 7):
    print(i * '*')
```

```
*
**
***
****
*****
******
```

6. Generators and Iterators

Theory:

Understanding how generators work in Python.

Generators are special Python functions that create iterators using the yield keyword. Unlike regular functions that return a single value and exit, generators yield a value, pause their execution, and save their state. When the next value is requested, the function resumes from where it left off. This "lazy evaluation" means they generate values on demand, without storing the entire sequence in memory. This makes them highly memory-efficient for processing large or infinite data streams.

Difference between yield and return.

yield and return are both used to exit a function and pass a value back, but they have a key difference. return terminates a function entirely and sends back a single value. The function's state is lost, and it can't be resumed. In contrast, yield is used in a special type of function called a generator. It pauses the function's execution, returns a value, and saves the function's state. When the generator is iterated over again, the function resumes from where it left off. This makes generators memory-efficient for producing sequences of values one at a time.

Understanding iterators and creating custom iterators.

An iterator is an object that allows you to traverse through a sequence of data, one item at a time.[1] It's defined by two methods: __iter__ and __next__. [2]The __iter__ method returns the iterator object itself, while the __next__ method returns the next item in the sequence.[3] When there are no more items, __next__ raises a StopIteration exception. You can create a custom iterator by defining a class that implements these two methods, giving you full control over how you iterate through your data.

Lab:

Write a generator function that generates the first 10 even numbers.

In [46]:
```
def first_10_even_numbers():
    n = 0
    while n < 20:
        yield n
        n += 2

for number in first_10_even_numbers()
    print(number)
```

```
0
2
4
6
8
10
12
14
16
18
```

Write a Python program that uses a custom iterator to iterate over a list of integers.

In [47]:
```
class MyListIterator:
    def __init__(self, data):
        self.data = data
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.data):
            raise StopIteration

        value = self.data[self.index]
        self.index += 1
        return value

my_list = MyListIterator([10, 20, 30, 40, 50])

for number in my_list:
    print(number)
```

```
10
20
30
40
50
```

7. Functions and Methods

Theory:

Defining and calling functions in Python.

Defining a function in Python means creating a reusable block of code that performs a specific task. You use the def keyword, followed by the function's name, parentheses (), and a colon :. Any parameters the function needs are placed inside the parentheses. The code block for the function must be indented.

Calling a function means executing the code inside it. You do this by simply writing the function's name followed by parentheses () and providing any necessary arguments inside. Functions help organize code, prevent repetition, and make programs easier to read and maintain.

Function arguments (positional, keyword, default).

Function arguments are the values you pass to a function. They can be handled in a few ways. Positional arguments are matched to parameters based on their order. For example, in def greet(name, message):, the first argument you pass will always be name.

Keyword arguments are explicitly named in the function call, like greet(message='Hello', name='Alice'). This lets you pass arguments in any order.

Default arguments have a pre-defined value.For instance, in def greet(name, message='Hello'):, message will be 'Hello' if you don't provide a value for it.This makes the argument optional.

Scope of variables in Python.

The scope of a variable determines where it can be accessed in your Python code. There are two main types:

1. Local Scope: A variable defined inside a function is local to that function. It can only be used within that function and is destroyed once the function finishes.
2. Global Scope: A variable defined outside of any function is global. It can be accessed from anywhere in the program, including inside functions.

Built-in methods for strings, lists, etc.

Built-in Methods for Strings

1. str.upper(): Returns a copy of the string with all characters converted to uppercase.
2. str.lower(): Returns a copy of the string with all characters converted to lowercase.
3. str.strip(): Returns a copy of the string with leading and trailing whitespace removed.
4. str.split(separator): Returns a list of strings after splitting the original string at the specified separator.

5. str.replace(old, new): Returns a copy of the string with all occurrences of old replaced by new.

Built-in Methods for Lists

1. list.append(item): Adds an item to the end of the list.
2. list.insert(index, item): Inserts an item at a specified index.
3. list.remove(item): Removes the first occurrence of an item from the list.
4. list.pop(index): Removes and returns the item at a specified index.
5. list.sort(): Sorts the items of the list in place (modifies the original list).

Lab:

Practical Example: 1) Write a Python program to print "Hello" using a string.

```
In [48]: hello="HELLO WORLD"
         print(hello)

         HELLO WORLD
```

Practical Example: 2) Write a Python program to allocate a string to a variable and print it.

```
In [48]: hello="HELLO WORLD"
         print(hello)

         HELLO WORLD
```

Practical Example: 3) Write a Python program to print a string using triple quotes.

```
In [49]: hello='''HELLO YAGNESH'''
         print(hello)

         HELLO YAGNESH
```

Practical Example: 4) Write a Python program to access the first character of a string using index value.

```
n [55]: string="YAGNESH"
        print(string[0])
```

Practical Example: 5) Write a Python program to access the string from the second position onwards using slicing.

```
In [57]: string="YAGNESH"
         print(string[1:])
```

```
AGNESH
```

Practical Example: 6) Write a Python program to access a string up to the fifth character.

```
In [58]: string="YAGNESH"
         print(string[5])
```

```
S
```

Practical Example: 7) Write a Python program to print the substring between index values 1 and 4.

```
In [59]: string="YAGNESH"
         print(string[1:5])
```

```
AGNE
```

Practical Example: 8) Write a Python program to print a string from the last character.

```
In [61]: string="YAGNESH"
         print(string[-1])
```

```
H
```

Practical Example: 9) Write a Python program to print every alternate character from the string starting from index 1.

```
In [62]: my_string = "HelloWorld"
         print(my_string[1::2])
```

```
elWrd
```

8. Control Statements (Break, Continue, Pass)

Theory:

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

break, continue, and pass control the flow of a loop. break immediately exits the current loop entirely, useful for stopping when a condition is met. continue skips the rest of the current iteration and jumps to the next one, perfect for ignoring specific items. pass does nothing; it's a placeholder used to avoid a syntax error where a statement is required, like in an empty loop or if block, when you plan to add code later.

Lab:

Practical Example: 1) Write a Python program to skip 'banana' in a list using the continue statement. List1 = ['apple', 'banana', 'mango']

In [66]:
```
List1 = ['apple', 'banana', 'mango']
for fruit in List1:
    if fruit == 'banana':
        continue
    print(fruit)
```

```
apple
mango
```

Practical Example: 2) Write a Python program to stop the loop once 'banana' is found using the break statement.

In [67]:
```
fruits = ['apple', 'banana', 'mango']
for fruit in fruits:
    if fruit == 'banana':
        break
    print(fruit)
```

```
apple
```

9. String Manipulation

Theory:

Understanding how to access and manipulate strings.

You can access and manipulate strings in Python using several powerful methods.Individual characters are accessed by their index within square brackets, with the first character at index 0. Slicing lets you extract a substring using [start:stop:step], which is great for getting a part of the string. Strings are immutable, meaning you can't change them directly; instead, methods like .upper() or .replace() return a new, modified string.These features allow for efficient data retrieval and transformation without affecting the original string.

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

In Python, you can easily combine strings using concatenation with the + operator. You can repeat a string multiple times with repetition using the * operator. Strings also come with built-in methods for common tasks. For example, .upper() converts a string to all uppercase, and .lower() converts it to all lowercase. These methods are essential for basic string manipulation and data cleaning.

String slicing.

String slicing extracts a portion of a string using [start:stop:step] syntax, creating a new substring from the specified range and interval.

Lab:

Write a Python program to demonstrate string slicing.

```
In [71]: word="DEMONSTRATION"
         print(word[0:8:2])

         DMNT
```

Write a Python program that manipulates and prints strings using various string methods.

```
In [76]: word="DEMONSTRATION"
         word1="skill"
         print(word[0:8:2])
         print(word.lower())
         print(word1.upper())
         print(word+word1)

         DMNT
         demonstration
         SKILL
         DEMONSTRATIONskill
```

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

Theory:

How functional programming works in Python.

Functional programming in Python treats computation as the evaluation of mathematical functions, avoiding changes to program state and mutable data. It emphasizes using pure functions, which always produce the same output for the same input and have no side effects. This approach relies on functions as first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables. This leads to code that's often more concise, easier to reason about, and better suited for concurrent and parallel execution.

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

Python's map(), filter(), and reduce() functions are tools for functional programming.

map(function, iterable) applies a function to every item in an iterable and returns a new iterable with the results. For example, you can use map to square every number in a list.

filter(function, iterable) constructs an iterator from elements of an iterable for which the function returns True. It's great for selecting items that meet a specific condition.

reduce(function, iterable) (from the functools module) applies a function cumulatively to the items of an iterable, from left to right, to reduce the iterable to a single value.

Introduction to closures and decorators.

A closure is a function that remembers the values of variables from its enclosing scope, even after that scope has finished executing.This allows a nested function to access and modify a variable from its parent function. A decorator is a function that takes another function as an argument, adds new functionality to it, and returns the modified function, all without changing the original's source code. Decorators are essentially closures in action, providing a clean syntax for wrapping functions with reusable behaviors.

Lab:

Write a Python program to apply the map() function to square a list of numbers.

In [77]:
```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)
```

```
[1, 4, 9, 16, 25]
```

Write a Python program that uses reduce() to find the product of a list of numbers.

```
In [78]: from functools import reduce

         numbers = [1, 2, 3, 4, 5]
         product = reduce(lambda x, y: x * y, numbers)
         print(product)
```

```
120
```

Write a Python program that filters out even numbers using the filter() function.

```
In [79]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
         odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))
         print(odd_numbers)
```

```
[1, 3, 5, 7, 9]
```

Assessment:

Create a mini-project where students combine conditional statements, loops, and functions to create a basic Python application, such as a simple calculator or a grade management system.

```
def add(x, y):
    return x + y


def subtract(x, y):
    return x - y


def multiply(x, y):
    return x * y


def divide(x, y):
    if y == 0:
        return "Error! Division by zero."
    return x / y


while True:
```

```
print("\nSelect operation:")
print("1. Add")
print("2. Subtract")
print("3. Multiply")
print("4. Divide")
print("5. Exit")

choice = input("Enter choice(1/2/3/4/5): ")

if choice == '5':
    print("Exiting calculator.")
    break

if choice in ('1', '2', '3', '4'):
    try:
        num1 = float(input("Enter first number: "))
        num2 = float(input("Enter second number: "))
    except ValueError:
        print("Invalid input. Please enter a number.")
        continue

    if choice == '1':
        print(num1, "+", num2, "=", add(num1, num2))
    elif choice == '2':
        print(num1, "-", num2, "=", subtract(num1, num2))
    elif choice == '3':
        print(num1, "*", num2, "=", multiply(num1, num2))
    elif choice == '4':
        print(num1, "/", num2, "=", divide(num1, num2))
```

```
else:
    print("Invalid Input. Please enter a valid choice.")
```

```
Select operation:
1. Add
2. Subtract
3. Multiply
4. Divide
5. Exit
Enter choice(1/2/3/4/5): 1
Enter first number: 2
Enter second number: 4
2.0 + 4.0 = 6.0
```