**Module 6) Python Fundamentals**

Introduction to Python

**Theory:**

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

**Python** is a simple, high-level, and interpreted programming language used for web development, data science, AI, and more.

**Features:**

**Simple** syntax, easy to read and write.

**High-level**: No need to manage memory.

**Interpreted**: Executes code line by line.

**Cross-platform**: Runs on Windows, macOS, and Linux.

**Versatile**: Supports OOP, scripting, and more.

**History and evolution of Python.**

**Created by**: Guido van Rossum in the late 1980s.

**First released**: 1991.

**Python 2**: Released in 2000.

**Python 3**: Released in 2008 (current standard).

**Now maintained** by the Python Software Foundation.

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

Easy to learn and use.

Large standard library and community support.

Great for automation, web apps, AI, and data analysis.

Portable and scalable.

Supports multiple programming paradigms.

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

**print("Hello, Python!")**

**Lab:**

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

print("Hello, Python!")

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

#### Steps to Set Up Python:

**Download Python**:  
Go to [https://www.python.org](https://www.python.org" \t "/Users/arjun1104/Documents/x/_new) → Downloads → Choose your OS → Install.

**Verify Installation**:

Open terminal or command prompt: python --version

**2. Programming Style**

**Theory:**

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

**PEP 8** is Python's official style guide. It helps you write **clean, consistent, and readable code**.

Indentation, comments, and naming conventions in Python.

### ****1. Indentation:****

Use **4 spaces per indentation level**

Don’t mix **tabs** and **spaces**.

### ****2.Comments****

Use # for single-line comments.

Use **docstrings** (""" """) for documenting functions.

### ****3. Naming Conventions****

| **Element** | **Style** | **Example** |
| --- | --- | --- |
| Variables | lowercase\_words | total\_amount |
| Functions | lowercase\_words | calculate\_sum() |
| Constants | UPPERCASE | MAX\_LIMIT = 100 |
| Classes | CamelCase | class BankAccount: |
| Private members | \_underscore | \_hidden\_value |

Writing readable and maintainable code.

### ****Writing Readable and Maintainable Code****

Use meaningful names (amount not a).

Write short functions that do one thing.

Add comments and docstrings.

**Lab:**

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

# This program calculates the sum of two numbers

num1 = 10 # First number

num2 = 20 # Second number

sum\_result = num1 + num2 # Calculate sum

print("Sum:", sum\_result) # Print the result

**3. Core Python Concepts**

**Theory:**

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

### ****Understanding Python Data Types:****

**1.Integers (**int**)**  
Whole numbers without decimals.  
Example: 5, -10, 100

**2.Floats (**float**)**  
Numbers with decimals.  
Example: 3.14, -0.001

**3.Strings (**str**)**  
Text enclosed in quotes.  
Example: "Hello", 'Python'

**4.Lists (**list**)**  
Ordered, changeable collections of items (can contain mixed types).  
Example: [1, 2, "apple", 3.5]

**5.Tuples (**tuple**)**  
Ordered, **immutable** (unchangeable) collections.  
Example: (10, 20, 30)

**6.Dictionaries (**dict**)**  
Key-value pairs for storing data.  
Example: {"name": "Arjun", "age": 22}

**7.Sets (**set**)**  
Unordered collections of unique items.  
Example: {1, 2, 3, 3} → {1, 2, 3}

**Python variables and memory allocation.**

Variables store data values.

Python **dynamically allocates memory**, so you don’t declare types explicitly.

You can assign any data type to a variable, and it can change during execution.

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

### ****Python Operators****

| **Operator Type** | **Examples** | **Description** |
| --- | --- | --- |
| **Arithmetic** | +, -, \*, /, %, // (floor div), \*\* (power) | Perform math operations |
| **Comparison** | ==, !=, >, <, >=, <= | Compare values, return True/False |
| **Logical** | and, or, not | Combine boolean expressions |
| **Bitwise** | & (AND), ` | (OR),^(XOR),~(NOT),<<, >>` |

**Lab:**

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

name = "Arjun" # string

age = 22 # integer

height = 5.8 # float

is\_student = True # boolean

print(name)

print(age)

print(height)

print(is\_student)

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

def greet():

print("Hello, Arjun!")

greet() # Call the function

**Practical Example 2: How to create variables in Python?**

city = "Ahmedabad"

pin\_code = 380001

print("City:", city)

print("PIN Code:", pin\_code)

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

name = input("Enter your name: ")

print("Hello", name)

**4. Conditional Statements**

**Theory:**

**Introduction to conditionalstatements: if, else, elif.**

### 1.if

Executes a block if the condition is **true**.

if condition:

# code runs if condition is true

### 2.if-else

Runs one block if true, another if false.

if condition:

# true blockelse:

# false block

### 3.if-elif-else

Checks multiple conditions in order.

if condition1:

# block 1elif condition2:

# block 2else:

# fallback block

**Nested if-else conditions**.

### ****Nested if-else****

An if inside another if. Used for **multiple levels** of checks.

if condition1:

if condition2:

# nested block

else:

# another blockelse:

# outer else

**Lab:**

**Practical Example 5: Write a Python program to find greater and less than a number using** **if\_else.**

# Take input from the user

number = int(input("Enter a number: "))

# Check if number is greater or less than 10

if number > 10:

print("The number is greater than 10.")

else:

print("The number is less than or equal to 10.")

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

# Take input from the user

num = int(input("Enter a number: "))

# 1 is not a prime number

if num <= 1:

print("Not a prime number")

else:

# Check for factors

is\_prime = True

for i in range(2, num):

if num % i == 0:

is\_prime = False

break

# Final result

if is\_prime:

print("It is a prime number")

else:

print("It is not a prime number")

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

# Take percentage input from user

percentage = float(input("Enter your percentage: "))

# Grade calculation using if-elif-else ladder

if percentage >= 90:

print("Grade: A+")

elif percentage >= 80:

print("Grade: A")

elif percentage >= 70:

print("Grade: B")

elif percentage >= 60:

print("Grade: C")

elif percentage >= 50:

print("Grade: D")

elif percentage >= 35:

print("Grade: E (Pass)")

else:

print("Grade: F (Fail)")

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

# Take user input

age = int(input("Enter your age: "))

weight = float(input("Enter your weight in kg: "))

# Nested if to check eligibility

if age >= 18:

if weight >= 50:

print("You are eligible to donate blood.")

else:

print("You are not eligible because your weight is less than 50 kg.")

else:

print("You are not eligible because your age is less than 18.")

**5. Looping (For, While)**

**Theory:**

**Introduction to for and while loops. and How loops work in Python.**

### What is a Loop?

A **loop** allows you to repeat a block of code multiple times — great for tasks that involve repetition.

## while Loop

while condition:

# code block

The code runs **as long as the condition is True**.

You must manually update variables inside the loop to avoid infinite loops.

## for Loop

for item in collection:

# code block

**Iterates **over a sequence** like a list, tuple, string, or range**.

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

You can loop through **lists**, **tuples**, **strings**, and more using for loops.

### ➤ With a List:

fruits = ["apple", "banana", "cherry"]for fruit in fruits:

print(fruit)

### ➤ With a Tuple:

coordinates = (10, 20, 30)for point in coordinates:

print("Point:", point)

### ➤ With a String:

word = "hello"for char in word:

print(char)

**Lab:**

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

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

for fruit in List1:

print(fruit)

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

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

for fruit in List1:

print(f"The length of '{fruit}' is {len(fruit)}")

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

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

search\_item = 'banana'

found = False

for fruit in List1:

if fruit == search\_item:

print(f"'{search\_item}' found in the list.")

found = True

break

if not found:

print(f"'{search\_item}' not found in the list.")

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

**markdown**

**Copy code**

**\***

**\*\***

**\*\*\***

**\*\*\*\***

\*\*\*\*\*

rows = 5

for i in range(1, rows + 1): # Outer loop for rows

for j in range(i): # Inner loop for printing stars

print("\*", end="")

print() # Move to next line after inner loop

**6. Generators and Iterators**

**Theory:**

**Understanding how generators work in Python.**

**generator** is a special type of function that **yields values one at a time** using the yield keyword instead of returning all values at once.

It **saves memory** and is great for **large data streams** or infinite sequences.

The function doesn't return a full list — it **yields one value at a time** and resumes where it left off.

**Difference between yield and return.**

| **Feature** | **return** | **yield** |
| --- | --- | --- |
| Use | Ends the function immediately | Pauses the function and resumes |
| Returns | A single value | A generator (iterator) |
| Memory usage | Stores full result in memory | Generates values on the fly |
| Use case | When all data is needed at once | When data is needed one-by-one |

**Understanding iterators and creating custom iterators.**

An **iterator** is any object in Python with:

a \_\_iter\_\_() method that returns the iterator object itself, and

a \_\_next\_\_() method that returns the next value or raises StopIteration.

**Lab:**

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

def even\_numbers():

num = 2

count = 0

while count < 10:

yield num

num += 2

count += 1

# Using the generator

for even in even\_numbers():

print(even)

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

class ListIterator:

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

# Using the custom iterator

numbers = [10, 20, 30, 40, 50]

my\_iterator = ListIterator(numbers)

for num in my\_iterator:

print(num)

**7. Functions and Methods**

**Theory:**

**Defining and calling functions in Python.**

## ****Defining and Calling Functions in Python****

### ➤ Syntax:

### def function\_name(parameters):

# function body

return result

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

## ****Function Arguments in Python****

### There are 3 common types of arguments:

### ➤ a. ****Positional Arguments**** (based on order)

➤ b. **Keyword Arguments** (specify by name)

➤ c. **Default Arguments** (value used if not provided)

**Scope of variables in Python.**

### Scope means where a variable is accessible.

### ➤ a. ****Local Scope**** (inside a function)

➤ b. **Global Scope** (defined outside all functions)

**Built-in methods forstrings, lists, etc.**

### ****String Methods****

upper()

lower()

strip()

lstrip()

rstrip()

replace()

find()

index()

count()

split()

join()

startswith()

endswith()

isalpha()

isdigit()

isalnum()

islower()

isupper()

### ****List Methods****

append()

extend()

insert()

remove()

pop()

clear()

index()

count()

sort()

reverse()

copy()

**Lab:**

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

message = "Hello"

print(message)

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

greeting = "Welcome to Python Programming!"

print(greeting)

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

message = """This is a string

written using triple quotes.

It can span multiple lines."""

print(message)

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

text = "Python"

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

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

text = "Python"

print(text[1:]) # Output: 'ython'

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

'

text = "Python"

print(text[:5]) # Output: 'Pytho'

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

'

text = "Python"

print(text[1:5]) # Output: 'ytho'

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

text = "Python"

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

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

text = "Python"

print(text[1::2]) # Output: 'yhn'

**8. Control Statements (Break, Continue, Pass)**

**Theory:**

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

## ****1.**** break ****Statement****

**Role:** Immediately **stops the loop** and exits it.

Used when you want to **exit a loop early**, typically when a condition is met.

## ****2.**** continue ****Statement****

**Role:** **Skips the current iteration** and moves to the **next iteration** of the loop.

Useful when you want to **skip some steps** in the loop under certain conditions.

## ****3.**** pass ****Statement****

**Role:** Does **nothing**; it’s a **placeholder**.

Used when **syntax requires a statement** but you don’t want to execute any code yet.

Common in empty loops, functions, or conditionals under development.

**Lab:**

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

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

for fruit in List1:

if fruit == 'banana':

continue # Skip 'banana'

print(fruit)

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

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

for fruit in List1:

if fruit == 'banana':

break # Stop loop when 'banana' is found

print(fruit)

**9. String Manipulation**

**Theory:**

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

### ****Accessing Strings:****

Strings are sequences of characters.

Each character can be accessed using **indexing** (starting from 0).

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

#### ****Concatenation**** (+ operator):

oining two or more strings.

#### ****Repetition**** (\* operator):

Repeating a string multiple times.

### ****Common String Methods****

| **Method** | **Description** | **Example** | **Output** |
| --- | --- | --- | --- |
| upper() | Converts string to uppercase | "hello".upper() | 'HELLO' |
| lower() | Converts string to lowercase | "HELLO".lower() | 'hello' |
| strip() | Removes leading and trailing spaces | " hello ".strip() | 'hello' |
| replace() | Replaces substring | "hello".replace("l", "p") | 'heppo' |
| split() | Splits string into list | "a,b,c".split(",") | ['a', 'b', 'c'] |
| join() | Joins list elements into string | ",".join(['a', 'b', 'c']) | 'a,b,c' |
| find() | Finds substring index or -1 | "hello".find("e") | 1 |
| startswith() | Checks if string starts with substring | "hello".startswith("he") | True |
| endswith() | Checks if string ends with substring | "hello".endswith("lo") | True |

String slicing.

Extract parts of a string using [start:stop:step].

**Lab:**

**Write a Python program to demonstrate string slicing.**

text = "Programming"

print("Original string:", text)

print("Slice [0:6]:", text[0:6]) # From index 0 to 5

print("Slice [3:]:", text[3:]) # From index 3 to end

print("Slice [:5]:", text[:5]) # From start to index 4

print("Slice [::2]:", text[::2]) # Every 2nd character

print("Slice [::-1]:", text[::-1]) # Reversed string

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

message = " Hello, Python World! "

print("Original message:", repr(message))

# Convert to uppercase

print("Uppercase:", message.upper())

# Convert to lowercase

print("Lowercase:", message.lower())

# Remove leading and trailing spaces

print("Stripped:", message.strip())

# Replace 'Python' with 'Java'

print("Replace:", message.replace("Python", "Java"))

# Find the position of 'Python'

print("Find 'Python':", message.find("Python"))

# Split message into words

words = message.strip().split()

print("Split words:", words)

# Join words with '-'

joined = "-".join(words)

print("Joined with '-':", joined)

# Check if message starts with ' He'

print("Starts with ' He':", message.startswith(" He"))

# Check if message ends with 'World! '

print("Ends with 'World! ':", message.endswith("World! "))

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

**Theory:**

**How functional programming works in Python.**

Functional programming in Python is a **programming paradigm** where you build programs by **applying and composing functions**. It emphasizes **immutability**, **pure functions** (functions with no side effects), and **higher-order functions** (functions that take or return other functions).

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

### map(), filter(), and reduce() for Data Processing

#### map(function, iterable)

Applies a function to every item in an iterable.

#### filter(function, iterable)

Filters elements for which the function returns True.

#### reduce(function, iterable)

Applies a function cumulatively to reduce the iterable to a single value.

reduce() is in the functools module.

**Introduction to closures and decorators :**

### Closures

A **closure** is a function that remembers the variables from the outer enclosing scope even if the outer function is done executing.

### Decorators

A **decorator** is a special function that **modifies the behavior of another function** without changing its code.

**Lab:**

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

# Square each number in a list using map()

numbers = [1, 2, 3, 4, 5]

squared\_numbers = list(map(lambda x: x \*\* 2, numbers))

print("Original numbers:", numbers)

print("Squared numbers:", squared\_numbers)

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

from functools import reduce

# Multiply all numbers in a list using reduce

numbers = [1, 2, 3, 4, 5]

product = reduce(lambda x, y: x \* y, numbers)

print("Numbers:", numbers)

print("Product of numbers:", product)

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

# Filter out even numbers from the list using filter()

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

odd\_numbers = list(filter(lambda x: x % 2 != 0, numbers))

print("Original numbers:", numbers)

print("Odd numbers (even numbers removed):", odd\_numbers)

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 calculate\_grade(avg):

if avg >= 90:

return 'A'

elif avg >= 75:

return 'B'

elif avg >= 60:

return 'C'

elif avg >= 40:

return 'D'

else:

return 'F'

def main():

students = {}

while True:

name = input("Enter student name (or type 'done' to finish): ")

if name.lower() == 'done':

break

marks = []

for i in range(3):

mark = float(input(f"Enter mark {i+1} for {name}: "))

marks.append(mark)

avg = sum(marks) / len(marks)

grade = calculate\_grade(avg)

students[name] = {'Marks': marks, 'Average': avg, 'Grade': grade}

print("\n--- Student Grade Report ---")

for student, info in students.items():

print(f"\nName: {student}")

print("Marks:", info['Marks'])

print("Average:", info['Average'])

print("Grade:", info['Grade'])

if \_\_name\_\_ == "\_\_main\_\_":

main()