Python Practical File

A Practical Report for the Course:

Multidisciplinary Course in Computer Science (CBEGS)

(Syllabus for the batch 2024-2025)

Submitted by:

Jashanpreet Singh Dingra
MSC FYIP Physics Semester III
Roll number: 28122406337

Submitted to:

Prithvipal Singh
Department of Computer Science

Guru Nanak Dev University

Amritsar, Punjab

Contents

Introduction

This report documents the practical programming work completed as part of the Multidisciplinary Course in Computer Science (CBEGS). The objective of this file is to provide a structured, hands-on demonstration of the core concepts of the Python programming language, as outlined in the course syllabus for the 2024-2025 batch.

The report is divided into four main sections, mirroring the syllabus structure:

- **Section A** begins with the fundamentals, covering Python's basic data types, variables, I/O operations, and the decision-making structures (if/elif/else) that form the basis of logical programming.
- **Section B** builds on this foundation, introducing repetition structures (for and while loops) and the concept of functions, which allow for modular and reusable code.
- **Section C** delves into data structures, exploring the manipulation of lists and tuples, as well as file I/O (reading from and writing to files) and exception handling.
- **Section D** concludes the practical work by covering advanced data structures (dictionaries and sets) and introducing the principles of Object-Oriented Programming (OOP) with classes and objects.

Each practical follows a consistent format:

1. **Topic:** A brief explanation of the concept.
2. **Aim:** A detailed description of the practical's objective.
3. **Code:** The Python source code, implemented in a Jupyter or Colab Notebook.
4. **Output:** The text-based result of executing the code.

This file serves as a comprehensive record of the programming exercises completed, demonstrating a practical understanding of Python from basic syntax to more advanced programming paradigms.

Chapter 1

Section A: Introduction to Python Basics

1.1 Practical 1: Variables, Constants, and Data Types

Topic

This practical covers the absolute basics. We'll learn about **variables**, which are like labeled boxes for storing data. We'll also look at Python's main **data types**—like `int` for whole numbers, `str` for text, and `list` for collections of items.

Aim

The goal is to get comfortable with creating variables (like `x = 10`), understanding the difference between a number (`10`) and text (`"10"`), and seeing how data types like lists and dictionaries store collections of information. We'll learn how to create a variable, assign it a value, and check what "type" of data it's holding.

Code

```
# --- Variables and Assignment ---
# A variable is like a labeled box. Here, the box is 'website'
# and the data we put in it is "github.com".
website = "github.com"
print(website)

# We can change the data in the box (re-assignment)
website = "baidu.com"
print(website)

# Multiple assignment in one line
a, b, c = 6, 9.3, "Hello"
print(a)
print(b)
print(c)

# --- Constants (By Convention) ---
# Constants are variables that shouldn't change.
# We write them in ALL_CAPS to show this.
PI = 3.14
GRAVITY = 9.8
print(PI)
```

```python

# --- Data Types (The 'type' of data) ---

# 1. Numbers (int, float, complex)
a_int = 6            # int (whole number)
a_float = 3.0        # float (decimal number)
a_complex = 1+2j     # complex (math number)
print(a_int, "is of type", type(a_int))
print(a_float, "is of type", type(a_float))
print(a_complex, "is of type", type(a_complex))

# 2. String (str) - for text
s = "Hello world!"
print(s, "is of type", type(s))

# 3. List (list) - ordered, changeable
x = [6, 99, 77, 'Apple']
print(x, "is of type", type(x))
x[1] = 100 # Lists are mutable (changeable)
print("Modified list:", x)

# 4. Tuple (tuple) - ordered, NOT changeable
t = (6, 'program', 1+3j)
print(t, "is of type", type(t))
# t[0] = 10 # This would cause an error!

# 5. Set (set) - unordered, unique items
a_set = {1, 2, 2, 3, 3, 3} # Duplicates are automatically removed
print(a_set, "is of type", type(a_set))

# 6. Dictionary (dict) - key-value pairs
d = {1: 'Apple', 2: 'Cat', 3: 'Food'}
print(d, "is of type", type(d))
print("Value for key 1 is:", d[1])
```

Output (Text-Based)

```
github.com
baidu.com
6
9.3
Hello
3.14
6 is of type <class 'int'>
3.0 is of type <class 'float'>
(1+2j) is of type <class 'complex'>
Hello world! is of type <class 'str'>
[6, 99, 77, 'Apple'] is of type <class 'list'>
Modified list: [6, 100, 77, 'Apple']
(6, 'program', (1+3j)) is of type <class 'tuple'>
{1, 2, 3} is of type <class 'set'>
{1: 'Apple', 2: 'Cat', 3: 'Food'} is of type <class 'dict'>
Value for key 1 is: Apple
```

—

1.2 Practical 2: Python Operators

Topic

Operators are the symbols we use to perform operations. This includes basic math (like + for addition), comparing things (like > for "greater than"), and logic (like and or or).

Aim

To use Python's built-in operators to perform calculations, compare values, and combine logical conditions.

Code

```python
# --- 1. Arithmetic Operators (Math) ---
x = 16
y = 3
print('x + y =', x + y)        # Addition
print('x - y =', x - y)        # Subtraction
print('x * y =', x * y)        # Multiplication
print('x / y =', x / y)        # Division (always gives a float)
print('x // y =', x // y)    # Floor Division (drops the decimal)
print('x % y =', x % y)        # Modulus (the remainder)
print('x ** y =', x ** y)      # Exponentiation (x to the power of y)

# --- 2. Comparison Operators (True/False) ---
print("\n--- Comparison ---")
print('6 > 3 is', 6 > 3)        # Greater than
print('6 < 3 is', 6 < 3)        # Less than
print('6 == 3 is', 6 == 3)      # Equal to
print('6 != 3 is', 6 != 3)      # Not equal to
print('6 >= 3 is', 6 >= 3)      # Greater than or equal to

# --- 3. Logical Operators (and, or, not) ---
print("\n--- Logic ---")
print('True and False is', True and False) # 'and' checks if BOTH are true
print('True or False is', True or False)   # 'or' checks if AT LEAST ONE is true
print('not True is', not True)             # 'not' flips the value

# Combining operators
print('6 > 3 and 5 < 3 is', 6 > 3 and 5 < 3) # (True and False) -> False
print('6 > 3 or 5 < 3 is', 6 > 3 or 5 < 3)   # (True or False)  -> True
```

Output (Text-Based)

```
x + y = 19
x - y = 13
x * y = 48
x / y = 5.333333333333333
x // y = 5
x % y = 1
x ** y = 4096

--- Comparison ---
6 > 3 is True
```

```
6 < 3 is False
6 == 3 is False
6 != 3 is True
6 >= 3 is True

--- Logic ---
True and False is False
True or False is True
not True is False
6 > 3 and 5 < 3 is False
6 > 3 or 5 < 3 is True

__
```

1.3 Practical 3: Input, Output, and String Formatting

Topic

This practical covers how a program communicates with the user. We use **'input()'** to get information *from* the user (keyboard) and **'print()'** to display information *to* the user (screen).

Aim

To write a program that can ask the user for information, store that information in a variable, and then print it back out in a clean, formatted way.

Code

```python
# --- 1. Output with print() ---
a = 9
print('The value of a is', a) # print() automatically adds a space

# Using 'sep' (separator)
print(1, 2, 3, 4, sep='#') # Use '#' instead of a space

# Using 'end'
print(1, 2, 3, 4, sep='*', end='&')
print("...next print starts right here.")

# --- 2. String Formatting (str.format) ---
print("\n--- Formatting ---")
x = 6
y = 12
# The {} are placeholders
print('The value of x is {} and y is {}'.format(x, y))

# You can use numbers to change the order
print('I love {0} and {1}'.format('Mango', 'Banana'))
print('I love {1} and {0}'.format('Mango', 'Banana'))

# You can use names
print('Hello {name}, {greeting}!'.format(greeting='Good morning', name='Mark'))

# --- 3. Input with input() ---
print("\n--- Input ---")
num_str = input('Enter a number: ')
print("You entered:", num_str)
print("Type of input is:", type(num_str)) # Input is ALWAYS a string!

# --- 4. Type Casting Input ---
# To do math, we must convert the string to a number (int or float)
print("\n--- Type Casting ---")
first_number = int(input("Enter first number: "))
second_number = int(input("Enter second number: "))
sum1 = first_number + second_number
print("Addition of two numbers is:", sum1)
```

Output (Text-Based)

```
The value of a is 9
1#2#3#4
```

```
1*2*3*4&...next print starts right here.

--- Formatting ---
The value of x is 6 and y is 12
I love Mango and Banana
I love Banana and Mango
Hello Mark, Good morning!

--- Input ---
Enter a number: 50
You entered: 50
Type of input is: <class 'str'>

--- Type Casting ---
Enter first number: 10
Enter second number: 20
Addition of two numbers is: 30

__
```

1.4 Practical 4: Decision Structures (if-elif-else)

Topic

This is all about making decisions. We'll use **'if'**, **'elif'**, and **'else'**. These statements let our program choose what to do based on a condition (like `age > 18`). It's how a program can respond differently to different situations.

Aim

We want to write a program that doesn't just run from top to bottom. The goal is to use `if` statements to control the "flow" of the program. The program will check a condition (like `num % 2 == 0`) and then run a specific block of code only if that condition is `True`.

Code

```python
# --- 1. Simple if statement ---
# This block only runs if the condition is True.
age = 20
if age >= 18:
    print("You are eligible to vote.")

# --- 2. if-else statement ---
# One of these two blocks will ALWAYS run.
num = int(input("Enter a number: "))
if num % 2 == 0:
    print(num, "is an even number.")
else:
    print(num, "is an odd number.")

# --- 3. if-elif-else statement ---
# 'elif' means "else if". It checks conditions in order.
# Only ONE block will run (the first one that is True).
score = int(input("Enter your test score: "))
if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
    grade = 'D'
else:
    grade = 'F' # The 'else' catches everything else.
print("Your grade is:", grade)
```

Output (Text-Based)

```
You are eligible to vote.
Enter a number: 7
7 is an odd number.
Enter your test score: 85
Your grade is: B
```

Chapter 2

Section B: Repetition Structures and Functions

2.1 Practical 5: Repetition Structures (Loops)

Topic

This practical is about repetition. **Loops** (`while` and `for`) let us run the same block of code over and over again, which saves us from writing it out hundreds of times.

Aim

To learn how to use Python's two kinds of loops:

- **'while' loop**: Keeps running *as long as* its condition is `True`.
- **'for' loop**: Runs *for each item* in a sequence (like a list or a `range`).

Code

```python
# --- 1. while Loop ---
# This loop will run as long as 'i' is less than or equal to 5.
print("--- while Loop ---")
total = 0
i = 1
while i <= 5:
    total = total + i
    i = i + 1   # We must increment 'i' ourselves!
print("The sum of numbers from 1 to 5 is", total)

# --- 2. for Loop (with range) ---
# This is a "count-controlled" loop.
# range(5) gives numbers 0, 1, 2, 3, 4.
print("\n--- for Loop with range(5) ---")
for i in range(5):
    print(i, end=' ')
print() # for new line

# range(1, 6) gives numbers 1, 2, 3, 4, 5.
print("\n--- for Loop with range(1, 6) ---")
```

```python
for i in range(1, 6):
    print(i, end=' ')
print()

# --- 3. for Loop (with a list) ---
# This loop runs once for each item in the 'fruits' list.
print("\n--- for Loop with a list ---")
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)

# --- 4. Nested Loops ---
# A loop inside another loop.
print("\n--- Nested Loops (Pattern) ---")
for i in range(1, 4):      # Outer loop (rows)
    for j in range(i):   # Inner loop (columns)
        print("*", end=" ")
    print() # New line after each row
```

Output (Text-Based)

```
--- while Loop ---
The sum of numbers from 1 to 5 is 15

--- for Loop with range(5) ---
0 1 2 3 4

--- for Loop with range(1, 6) ---
1 2 3 4 5

--- for Loop with a list ---
apple
banana
cherry

--- Nested Loops (Pattern) ---
* * * * * *
```

—

2.2 Practical 6: User-Defined Functions

Topic

This practical is about making our code reusable. A **function** is a block of code you can name and run whenever you want. This follows the **DRY (Don't Repeat Yourself)** principle. Instead of writing the same five lines of code in three different places, you write them once inside a function and just "call" that function's name.

Aim

The goal is to learn how to *define* a function with `def` and how to *call* it (run it) by using its name. We'll see how to send information *into* a function (using **parameters**) and how to get a result *out* of a function (using the **'return'** keyword).

Code

```python
# --- 1. Defining and Calling a Simple Function ---
# This function takes no input and returns no value. It just does an action.
def greet():
    print("Welcome to Python for Data Science")

print("Calling greet():")
greet() # This is how we call (run) the function

# --- 2. Function with Parameters ---
# 'name' is a parameter. It's a placeholder for the data
# we will send in.
def greet_person(name):
    print("Hello, " + name + ". Good morning!")

print("\nCalling greet_person('Arthur'):")
greet_person('Arthur') # 'Arthur' is the argument (the actual data)
greet_person('Emily')

# --- 3. Value-Returning Function ---
# This function gives a value back using the 'return' keyword.
def add_two_numbers(num_one, num_two):
    total = num_one + num_two
    return total

print("\nCalling add_two_numbers(3, 6):")
sum_result = add_two_numbers(3, 6)
print("The result is:", sum_result)

# --- 4. Function with multiple parameters and return value ---
def generate_full_name(first_name, last_name):
    space = ' '
    full_name = first_name + space + last_name
    return full_name

print("\nCalling generate_full_name('Milaan', 'Parmar'):")
full_name = generate_full_name('Milaan', 'Parmar')
print('Full Name:', full_name)
```

Output (Text-Based)

```
Calling greet():
Welcome to Python for Data Science

Calling greet_person('Arthur'):
Hello, Arthur. Good morning!
Hello, Emily. Good morning!

Calling add_two_numbers(3, 6):
The result is: 9

Calling generate_full_name('Milaan', 'Parmar'):
Full Name: Milaan Parmar
```

Chapter 3

Section C: Files, Exceptions, Lists, and Tuples

3.1 Practical 7: Lists and Tuples Manipulation

Topic

This practical focuses on **Lists** and **Tuples**. Both are used to store ordered collections of items. The main difference is that Lists are *changeable* (you can add/remove items), while Tuples are *not changeable* (immutable).

Aim

To demonstrate how to create, access, and manipulate items in a List. We'll focus on **slicing** (getting a sub-section of the list) and using built-in **methods** (like `.append()` and `.sort()`).

Code

```python
# --- 1. List Slicing ---
# Slicing is like cutting out a piece of the list.
# format is list[start:stop] (stop is not included)
a = [5, 10, 15, 20, 25, 30, 35, 40]
print("Original list:", a)

print("a[2] = ", a[2])           # Get one item (index 2)
print("a[0:3] = ", a[0:3])     # Get items from index 0 up to (but not in) 3
print("a[5:] = ", a[5:])       # Get items from index 5 to the end

# --- 2. List Methods (built-in functions) ---
my_list = [1, 5, 3, 5, 9]
print("\nOriginal my_list:", my_list)

my_list.append(10) # .append() adds an item to the end
print("After append(10):", my_list)

my_list.sort() # .sort() sorts the list in place
print("After sort():", my_list)

item_count = my_list.count(5) # .count() checks how many '5's are in the list
print("Count of 5:", item_count)

my_list.remove(3) # .remove() removes the first '3' it finds
print("After remove(3):", my_list)
```

```python

# --- 3. Two-Dimensional Lists (List of lists) ---
print("\n--- 2D List (Matrix) ---")
matrix = [
    [1, 2, 3], # row 0
    [4, 5, 6], # row 1
    [7, 8, 9]  # row 2
]
print(matrix)
# Access element at row 1, column 1
print("Element at matrix[1][1] is:", matrix[1][1])
```

Output (Text-Based)

```
Original list: [5, 10, 15, 20, 25, 30, 35, 40]
a[2] =  15
a[0:3] =  [5, 10, 15]
a[5:] =  [30, 35, 40]

Original my_list: [1, 5, 3, 5, 9]
After append(10): [1, 5, 3, 5, 9, 10]
After sort(): [1, 3, 5, 5, 9, 10]
Count of 5: 2
After remove(3): [1, 5, 5, 9, 10]

--- 2D List (Matrix) ---
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Element at matrix[1][1] is: 5
```

—

3.2 Practical 8: File Handling and Exceptions

Topic

This practical covers two related ideas: **File I/O** (Input/Output) for reading and writing data to files, and **Exception Handling** (`try...except`) for managing errors gracefully.

Aim

To learn how to:

1. **Write** content to a new text file.
2. **Read** content from that file.
3. Use a **‘try...except‘** block to catch errors (like `FileNotFoundError`) so our program doesn’t crash if the file doesn’t exist.

Code

```python
# --- 1. Writing to a File ("w" mode) ---
# 'with open' is the safe way to handle files.
# "w" mode means 'write' (it will create or overwrite the file)
file_content = "Hello, this is a test file.\nThis is the second line."
try:
    with open("practical_file.txt", "w") as f:
        f.write(file_content)
    print("File 'practical_file.txt' written successfully.")
except Exception as e:
    print(f"An error occurred while writing: {e}")

# --- 2. Reading from a File ("r" mode) ---
# "r" mode means 'read'
print("\n--- Reading from file ---")
try:
    with open("practical_file.txt", "r") as f:
        content = f.read() # .read() gets the whole content
        print("File content:\n", content)
except FileNotFoundError:
    print("Error: The file 'practical_file.txt' was not found.")

# --- 3. Exception Handling (Catching an error) ---
print("\n--- Testing Exception Handling ---")
try:
    # We try to open a file that doesn't exist
    with open("non_existent_file.txt", "r") as f:
        print(f.read())
except FileNotFoundError:
    # The 'except' block catches the error
    print("Successfully caught expected error: File was not found.")

# --- 4. Processing File line by line (Loop) ---
print("\n--- Reading file line-by-line ---")
try:
    with open("practical_file.txt", "r") as f:
        for line in f:
            print("Line:", line.strip()) # .strip() removes whitespace/newlines
except FileNotFoundError:
    print("Error: The file 'practical_file.txt' was not found.")
```

Output (Text-Based)

```
File 'practical_file.txt' written successfully.

--- Reading from file ---
File content:
 Hello, this is a test file.
This is the second line.

--- Testing Exception Handling ---
Successfully caught expected error: File was not found.

--- Reading file line-by-line ---
Line: Hello, this is a test file.
Line: This is the second line.
```

Chapter 4

Section D: Advanced Data Structures and OOP

4.1 Practical 9: Advanced String Manipulation

Topic

This practical explores more advanced things we can do with **Strings**. Like Lists, we can **slice** them. Strings also have many powerful **methods** for cleaning, searching, and splitting text.

Aim

To demonstrate advanced string operations, including slicing and common string methods like `.strip()`, `.upper()`, `.lower()`, `.replace()`, and `.split()`.

Code

```python
# --- 1. String Slicing ---
# Works just like list slicing!
s = 'Hello world!'
print("Original string:", s)

print("s[4] = ", s[4])          # Get one character
print("s[6:11] = ", s[6:11])    # Get a sub-string

# --- 2. String Methods ---
# Note: Strings are IMMUTABLE.
# Methods don't change the original string, they return a NEW string.
text = "  Python is Fun!  "
print("\nOriginal text: '" + text + "'")

# Manipulating
print("strip(): '" + text.strip() + "'")      # Removes whitespace from ends
print("upper(): "D + text.upper())        # Converts to uppercase
print("lower(): " + text.lower())        # Converts to lowercase
print("replace('Fun', 'Great'): " + text.replace("Fun", "Great"))

# Searching / Testing
print("startswith('  Python'):", text.startswith('  Python'))
print("find('is'):", text.find('is')) # Returns the index where 'is' starts
```

```python

# Splitting
# .split() turns a string into a list of strings
words = text.strip().split(' ')
print("split(' '):", words)
```

Output (Text-Based)

```
Original string: Hello world!
s[4] =  o
s[6:11] =  world

Original text: '  Python is Fun!  '
strip(): 'Python is Fun!'
upper():   PYTHON IS FUN!
lower():   python is fun!
replace('Fun', 'Great'):   Python is Great!
startswith('  Python'): True
find('is'): 9
split(' '): ['Python', 'is', 'Fun!']
```

—

4.2 Practical 10: Dictionaries and Sets

Topic

This practical covers two powerful data structures:

- **Sets**: Like lists, but they are *unordered* and *cannot* contain duplicates.
- **Dictionaries**: Store data as **key-value pairs**. Instead of using an index number, you look up a "key" (like a word) to get its "value" (like a definition).

Aim

To understand and use `dict` and `set` data structures, demonstrating set uniqueness, set operations, and dictionary key-value access.

Code

```
# --- 1. Sets ---
# Note how the duplicates (2, 3, 5) are removed
a = {1, 2, 2, 3, 3, 3, 4, 5, 5}
print("Set 'a':", a)
print("Type of 'a':", type(a))

# Set Operations
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
print("\nSet1:", set1)
print("Set2:", set2)
print("Union (set1 | set2):", set1 | set2)           # All items from both
print("Intersection (set1 & set2):", set1 & set2)   # Only items in both
print("Difference (set1 - set2):", set1 - set2)    # Items in set1, but not set2


# --- 2. Dictionaries ---
# {key: value, key: value}
d = {1: 'value', 'key': 2}
print("\nDictionary 'd':", d)
print("Type of 'd':", type(d))

# Accessing values by key
print("d[1] = ", d[1])
print("d['key'] = ", d['key'])

# Adding a new key-value pair
d[3] = 'New Item'
print("After adding d[3]:", d)

# Iterating over a dictionary
print("\nIterating over dictionary:")
for key, value in d.items():
    print(f"Key: {key}, Value: {value}")
\end{lslinting}

\subsection*{Output (Text-Based)}
\begin{verbatim}
Set 'a': {1, 2, 3, 4, 5}
Type of 'a': <class 'set'>
```

```

Set1: {1, 2, 3, 4}
Set2: {3, 4, 5, 6}
Union (set1 | set2): {1, 2, 3, 4, 5, 6}
Intersection (set1 & set2): {3, 4}
Difference (set1 - set2): {1, 2}

Dictionary 'd': {1: 'value', 'key': 2}
Type of 'd': <class 'dict'>
d[1] =  value
d['key'] =  2
After adding d[3]: {1: 'value', 'key': 2, 3: 'New Item'}

Iterating over dictionary:
Key: 1, Value: value
Key: key, Value: 2
Key: 3, Value: New Item
\end{verbatim}

---
\newpage
% --- Practical 11: Object-Oriented Programming (OOP) ---
\section{Practical 11: Classes and Object-Oriented Programming}

\subsection*{Topic}
This is a big one. \textbf{Object-Oriented Programming (OOP)} is a way of
    structuring our code to model the real world. We'll create a \textbf{Class},
    which is like a blueprint (e.g., the blueprint for a `Student`).

\subsection*{Aim}
Our goal is to build a \texttt{Student} "blueprint" (the \textbf{Class}). This
    blueprint will define what every student \emph{has} (data like \texttt{name}
    and \texttt{id}, called \textbf{attributes}) and what every student can \emph
    {do} (functions like \texttt{printStudentData()}, called \textbf{methods}).

Then, we'll use this blueprint to create actual, individual \texttt{Student} \
    textbf{objects} (also called \textbf{instances}).

\subsection*{Code}
\begin{lstlisting}
# --- Defining a Class ---
# This is the "blueprint" for a Student.
class Student:
    # Class attribute (shared by all students)
    species = "students"
    student_count = 0

    # The Constructor (__init__) runs when a NEW object is created
    # 'self' refers to the object being created (e.g., s1 or s2)
    def __init__(self, name, id):
        # Instance attributes (data unique to this object)
        self.name = name
        self.id = id
        self.skills = [] # A new empty list for each new student
        Student.student_count += 1 # Update the shared count

    # Instance method (a function that objects can do)
    def printStudentData(self):
        print("Name:", self.name, ", Id:", self.id)

```

```python
    # Another instance method
    def add_skill(self, skill):
        self.skills.append(skill)
        print(f"Added skill '{skill}' for {self.name}")

# --- Creating and Using Instances (Objects) ---
print("Total Students:", Student.student_count)

# Create first instance
print("\n--- Creating s1 ---")
s1 = Student("Arthur", 101) # This calls __init__
s1.printStudentData()       # This calls the object's method
s1.add_skill("HTML")

# Create second instance
print("\n--- Creating s2 ---")
s2 = Student("Emily", 102)
s2.printStudentData()
s2.add_skill("Marketing")

# --- Final Status ---
print("\n--- Final Status ---")
print("s1 skills:", s1.skills) # Note: s1 and s2 have different skill lists
print("s2 skills:", s2.skills)
print("Total Students:", Student.student_count) # The shared count is 2
```

Output (Text-Based)

```
Total Students: 0

--- Creating s1 ---
Name: Arthur , Id: 101
Added skill 'HTML' for Arthur

--- Creating s2 ---
Name: Emily , Id: 102
Added skill 'Marketing' for Emily

--- Final Status ---
s1 skills: ['HTML']
s2 skills: ['Marketing']
Total Students: 2
```

Conclusion

This practical report has successfully documented the fundamental and advanced concepts of the Python programming language as required by the CBEGS syllabus.

The journey began with **Section A**, which established a strong foundation in basic syntax, variable declaration, data types, and logical operators. **Section B** transitioned this knowledge into practical application by introducing repetition through `for` and `while` loops, and code modularity through the definition and use of functions.

Section C explored data in more depth, covering the management of data collections using `lists` and `tuples`, and the critical skill of data persistence through file I/O. The introduction of `try...except` blocks also demonstrated a professional approach to error handling. Finally, **Section D** introduced higher-level programming concepts, including the powerful `dictionary` and `set` data structures, and culminated in the principles of Object-Oriented Programming (OOP), where we bundled data and behavior into logical `Classes` and `Objects`.

Through these practicals, a clear progression has been made from writing simple, linear scripts to designing more complex, organized, and robust programs. The completion of this file demonstrates a working proficiency in Python's core features and a solid foundation for tackling more advanced programming challenges.

References

The code and theoretical concepts in this report were based on the lecture materials and PDF documents provided for the course. The complete source code is also available on GitHub.

- `https://github.com/jsdingra11/python_md_2025/`
- `009_Python_Data_Types.pdf`
- `012_Python_Operators.pdf`
- `011_Python_Input_Output_Import.pdf`
- `001_Python_Functions.pdf`
- `001_Python_OOPs_Concepts.pdf`
- `007_Python_Variables_&_Constants.pdf`
- `002_Python_Classes_and_Objects - Jupyter Notebook.pdf`