AI-527 Python Programming Laboratory

Lab File

Submitted To:

Name : Dr. Diksha Kumari
Dr. Ruchika Arora

Designation : Assistant Professor

Department : Centre of AI

Submitted By:

Name: Avinash Kumar Prajapati
Roll No: 25901326

Course: M.Tech

Branch: AI

CONTENTS:

Name:- Avinash Kumar Prajapati

Roll No:- 25901326

Course:- M.Tech

Branch:- Artificial Intelligence

1. Installation and Basic Programs (Variables and Data Types, Arithmetic, Comparison, Assignment, Logical Operators)

a. Write a program to display data of different types using variables and literal constants.

```
print("Displaying different data types using variables and literal constants:")

# Integer type
my_integer = 10
print(f"Variable (Integer): {my_integer}")
print(f"Literal (Integer): {123}")

# Float type
my_float = 20.5
print(f"Variable (Float): {my_float}")
print(f"Literal (Float): {3.14}")

# String type
my_string = "Hello, Python!"
print(f"Variable (String): {my_string}")
print(f"Literal (String): {'Colab'}")

# Boolean type
is_active = True
print(f"Variable (Boolean): {is_active}")
print(f"Literal (Boolean): {False}")

# List type
my_list = [1, 2, 3, "apple"]
print(f"Variable (List): {my_list}")
print(f"Literal (List): {[10, 20]}")

# Tuple type
my_tuple = (4, 5, 6, "banana")
print(f"Variable (Tuple): {my_tuple}")
print(f"Literal (Tuple): {(True, False)}")

# Dictionary type
my_dict = {"name": "Alice", "age": 30}
print(f"Variable (Dictionary): {my_dict}")
print(f"Literal (Dictionary): {{'city': 'New York', 'zip': '10001'}}")

# None type
my_none = None
print(f"Variable (NoneType): {my_none}")
```

```
Displaying different data types using variables and literal constants:
Variable (Integer): 10
Literal (Integer): 123
Variable (Float): 20.5
Literal (Float): 3.14
Variable (String): Hello, Python!
Literal (String): Colab
Variable (Boolean): True
Literal (Boolean): False
Variable (List): [1, 2, 3, 'apple']
Literal (List): [10, 20]
Variable (Tuple): (4, 5, 6, 'banana')
Literal (Tuple): (True, False)
Variable (Dictionary): {'name': 'Alice', 'age': 30}
Literal (Dictionary): {'city': 'New York', 'zip': '10001'}
Variable (NoneType): None
```

This Python code demonstrates how to display various data types using both variables and literal constants. Let's break it down:

1. **Introduction:** The first `print` statement simply displays a descriptive header for the output.
2. **Integer Type:**

- `my_integer = 10` assigns the integer value `10` to the variable `my_integer`.
- `print(f"Variable (Integer): {my_integer}")` prints the value stored in the `my_integer` variable.
- `print(f"Literal (Integer): {123}")` directly prints the integer literal `123`.

3. **Float Type:** Similar to integers, it shows a float value stored in `my_float` (a variable) and a direct float literal `3.14`.
4. **String Type:** It demonstrates a string stored in `my_string` ("Hello, Python!") and a string literal `'Colab'`.
5. **Boolean Type:** This section illustrates a boolean variable `is_active` set to `True` and a boolean literal `False`.
6. **List Type:** It displays a list stored in `my_list` (which can contain mixed data types) and a list literal `[10, 20]`.
7. **Tuple Type:** It shows a tuple stored in `my_tuple` and a tuple literal `(True, False)`.
8. **Dictionary Type:** This part demonstrates a dictionary stored in `my_dict` (key-value pairs) and a dictionary literal `{'city': 'New York', 'zip': '10001'}`.
9. **None Type:** Finally, it shows a variable `my_none` assigned the special `None` value (representing the absence of a value).

In essence, the code iterates through common Python data types, first assigning a value to a variable and printing it, and then directly printing a literal of the same type. The `f-strings` (e.g., `f"Variable (Integer): {my_integer}"`) are used for easy formatting of the output.

b. Write a program to show the application of the += operator on strings.

```
# Initialize a string
my_string = "Hello"
print(f"Initial string: {my_string}")

# Use += to append another string
my_string += ", World!"
print(f"String after +=: {my_string}")

# Another example
word1 = "Python"
word2 = " is awesome"
word1 += word2
print(f"Concatenated string: {word1}")
```

```
Initial string: Hello
String after +=: Hello, World!
Concatenated string: Python is awesome
```

This code demonstrates the application of the `+=` operator for string concatenation in Python. Let's break it down:

1. **Initialize a string:** `my_string = "Hello"` creates a string variable named `my_string` and assigns it the value "Hello". The first `print` statement then displays this initial value.
2. **Append using `+=`:** `my_string += ", World!"` is the key part here. The `+=` operator is used to append the string ", World!" to the existing `my_string`. This is a shorthand for `my_string = my_string + ", World!"`. The second `print` statement shows the new, concatenated string.
3. **Another example:** This section further illustrates the concept with `word1` and `word2`. `word1` is initialized to "Python", and `word2` to " is awesome". Then, `word1 += word2` appends the content of `word2` to `word1`, resulting in "Python is awesome". The final `print` statement displays this concatenated result.

In essence, the `+=` operator provides a convenient way to append one string to another, modifying the original string variable in place.

c. Write a program that performs addition and multiplication on string variables.

```
# String Addition (Concatenation)
string1 = "Hello"
string2 = " World"
result_addition = string1 + string2
print(f"Addition (Concatenation): '{string1}' + '{string2}' = '{result_addition}'")

# String Multiplication (Repetition)
string3 = "Python"
multiplier = 3
result_multiplication = string3 * multiplier
print(f"Multiplication (Repetition): '{string3}' * {multiplier} = '{result_multiplication}'")

# Another example of multiplication
pattern = "=-"
repetition = 5
repeated_pattern = pattern * repetition
print(f"Repetitive pattern: '{pattern}' * {repetition} = '{repeated_pattern}'")
```

```
Addition (Concatenation): 'Hello' + ' World' = 'Hello World'
Multiplication (Repetition): 'Python' * 3 = 'PythonPythonPython'
Repetitive pattern: '=-' * 5 = '=-=-=-=-=-'
```

This code demonstrates how addition (`+`) and multiplication (`*`) operators work with string variables in Python.

1. **String Addition (Concatenation):**
   - `string1 = "Hello"` and `string2 = " World"` define two string variables.
   - `result_addition = string1 + string2` concatenates the two strings. When the `+` operator is used with strings, it joins them together to form a new string. The output shows `'Hello' + ' World' = 'Hello World'`.
2. **String Multiplication (Repetition):**
   - `string3 = "Python"` and `multiplier = 3` define a string and an integer.
   - `result_multiplication = string3 * multiplier` repeats the string `string3` as many times as the `multiplier` indicates. The output for this is `'Python' * 3 = 'PythonPythonPython'`.
   - A second example `pattern = "=-"` and `repetition = 5` further illustrates this, resulting in `'=-' * 5 = '=-=-=-=-=-'`.

In summary, the `+` operator joins strings, and the `*` operator repeats a string a specified number of times.

2. Operators based programs

a. Write a program to calculate area of a triangle using Heron's Formula.

```
# --- Simple program to calculate area of a triangle using Heron's Formula ---

# Define the side lengths (example values)
side_a = 5.0
side_b = 6.0
side_c = 7.0

print(f"Calculating area for a triangle with sides: a={side_a}, b={side_b}, c={side_c}")

# Calculate the semi-perimeter
s = (side_a + side_b + side_c) / 2

# Calculate the area using Heron's formula
area = (s * (s - side_a) * (s - side_b) * (s - side_c)) ** 0.5

print(f"The area of the triangle is: {area:.2f}")
```

```
Calculating area for a triangle with sides: a=5.0, b=6.0, c=7.0
The area of the triangle is: 14.70
```

This Python code calculates the area of a triangle using **Heron's Formula**. Here's a breakdown of what it does:

1. **Defining Side Lengths:**
   - It starts by defining three variables: `side_a`, `side_b`, and `side_c`, which represent the lengths of the three sides of the triangle. These values are hardcoded for simplicity in this version.
2. **Displaying Input:**
   - A `print` statement shows the user which side lengths are being used for the calculation.
3. **Calculating the Semi-perimeter (`s`):**
   - Heron's formula requires the semi-perimeter of the triangle, which is half of its perimeter. This is calculated as `s = (side_a + side_b + side_c) / 2`.
4. **Applying Heron's Formula:**
   - The core of the calculation is `area = (s * (s - side_a) * (s - side_b) * (s - side_c)) ** 0.5`.
   - The expression `** 0.5` is used to calculate the square root, as Heron's formula involves taking the square root of the product.
5. **Displaying the Result:**
   - Finally, the calculated `area` is printed to the console, formatted to two decimal places using an f-string (`:.2f`).

This simple code directly applies Heron's formula to find the area of a triangle given its three side lengths.

b. Write a program to calculate the total amount of money in the piggy bank, given the coins of Rs10,5,2,1.

```
# Define the number of each coin denomination
coins_of_10 = 5  # Example: 5 ten-rupee coins
```

```
coins_of_5 = 8   # Example: 8 five-rupee coins
coins_of_2 = 12  # Example: 12 two-rupee coins
coins_of_1 = 15  # Example: 15 one-rupee coins

# Calculate the total value for each denomination
total_10 = coins_of_10 * 10
total_5 = coins_of_5 * 5
total_2 = coins_of_2 * 2
total_1 = coins_of_1 * 1

# Calculate the grand total
grand_total = total_10 + total_5 + total_2 + total_1

# Display the result
print(f"Number of Rs 10 coins: {coins_of_10}")
print(f"Number of Rs 5 coins: {coins_of_5}")
print(f"Number of Rs 2 coins: {coins_of_2}")
print(f"Number of Rs 1 coins: {coins_of_1}")
print(f"\nTotal amount in the piggy bank: Rs {grand_total}")
```

```
Number of Rs 10 coins: 5
Number of Rs 5 coins: 8
Number of Rs 2 coins: 12
Number of Rs 1 coins: 15

Total amount in the piggy bank: Rs 129
```

This Python code calculates the total amount of money in a piggy bank given a certain number of coins for different denominations. Let's break it down:

1. **Define Coin Counts:**
   - `coins_of_10`, `coins_of_5`, `coins_of_2`, and `coins_of_1` are variables initialized with example integer values representing the quantity of each coin denomination (Rs 10, Rs 5, Rs 2, and Rs 1 respectively).
2. **Calculate Total Value per Denomination:**
   - `total_10 = coins_of_10 * 10`: Calculates the total value contributed by Rs 10 coins.
   - `total_5 = coins_of_5 * 5`: Calculates the total value contributed by Rs 5 coins.
   - `total_2 = coins_of_2 * 2`: Calculates the total value contributed by Rs 2 coins.
   - `total_1 = coins_of_1 * 1`: Calculates the total value contributed by Rs 1 coins.
3. **Calculate Grand Total:**
   - `grand_total = total_10 + total_5 + total_2 + total_1`: All the individual denomination totals are summed up to get the final total amount in the piggy bank.
4. **Display the Result:**
   - Several `print` statements use f-strings to display the number of coins for each denomination and, finally, the `grand_total` in a user-friendly format.

3. String Manipulation, Number System and Conversions

a. Write a Python program that asks the user to enter an integer in decimal (base 10) and then prints its equivalent in binary, octal, and hexadecimal using built-in functions.

```
# Ask the user to enter an integer in decimal (base 10)
decimal_num = int(input("Enter an integer in decimal (base 10): "))

print(f"\nDecimal: {decimal_num}")

# Print its equivalent in binary using bin()
print(f"Binary: {bin(decimal_num)}")

# Print its equivalent in octal using oct()
print(f"Octal: {oct(decimal_num)}")

# Print its equivalent in hexadecimal using hex()
print(f"Hexadecimal: {hex(decimal_num)}")
```

```
Enter an integer in decimal (base 10): 69

Decimal: 69
Binary: 0b1000101
Octal: 0o105
Hexadecimal: 0x45
```

This Python program asks the user to enter an integer in decimal (base 10) and then converts it to its equivalent representations in binary, octal, and hexadecimal using Python's built-in functions.

1. `decimal_num = int(input("Enter an integer in decimal (base 10): "))`:
   - This line first displays the prompt "Enter an integer in decimal (base 10): " to the user.
   - The `input()` function reads whatever the user types as a string.
   - The `int()` function then converts this string into an integer, and this integer is stored in the `decimal_num` variable.
2. `print(f"\nDecimal: {decimal_num}")`:
   - This line simply prints the original decimal number that the user entered.
3. `print(f"Binary: {bin(decimal_num)}")`:
   - The `bin()` function is a built-in Python function that takes an integer as an argument and returns its binary representation as a string. The string is prefixed with `0b` to indicate it's a binary number (e.g., `0b101`).
4. `print(f"Octal: {oct(decimal_num)}")`:
   - Similarly, the `oct()` function converts an integer to its octal (base 8) representation, returning a string prefixed with `0o` (e.g., `0o75`).
5. `print(f"Hexadecimal: {hex(decimal_num)}")`:
   - The `hex()` function converts an integer to its hexadecimal (base 16) representation, returning a string prefixed with `0x` (e.g., `0x3F`).

In essence, this program demonstrates how easily Python can handle conversions between different number bases using simple, dedicated functions.

b. Write a Python program that reads a string representing a floating-point number (for example, "25.67") from the user, converts it to a float, adds 10.5 to it, and prints the result.

```
# Read a string representing a floating-point number from the user
float_string = input("Enter a floating-point number (e.g., '25.67'): ")

# Convert the string to a float
original_float = float(float_string)

# Add 10.5 to the float
result = original_float + 10.5

# Print the result
print(f"Original number: {original_float}")
print(f"Added 10.5: {original_float} + 10.5 = {result}")
```

```
Enter a floating-point number (e.g., '25.67'): 69.69
Original number: 69.69
Added 10.5: 69.69 + 10.5 = 80.19
```

This Python program reads a string representing a floating-point number from the user, converts it to a float, adds 10.5 to it, and prints the result. Let's break it down:

1. `float_string = input("Enter a floating-point number (e.g., '25.67'): ")`:
   - This line prompts the user to enter a floating-point number as a string. For example, the user might type "25.67".
   - The `input()` function reads whatever the user types and stores it as a string in the `float_string` variable.
2. `original_float = float(float_string)`:
   - The `float()` function is a built-in Python function that takes a string (in this case, `float_string`) and converts it into a floating-point number. This allows mathematical operations to be performed on it.
   - The resulting float value is stored in the `original_float` variable.
3. `result = original_float + 10.5`:
   - This line performs an arithmetic operation: it adds the floating-point number `10.5` to the `original_float` value.
   - The sum is then stored in the `result` variable.
4. `print(f"Original number: {original_float}")` `print(f"Added 10.5: {original_float} + 10.5 = {result}")`:
   - These lines display the original converted floating-point number and the final result of the addition in a user-friendly format using f-strings.

4. Operator Precedence, Conditional Statements, Loops (For and While)

a. Write a program to solve the expression: x = 10 + 2 * 3 ** 2 - 8 / 4 and show each expression calculation and output.

```
# Original expression
print("Original expression: x = 10 + 2 * 3 ** 2 - 8 / 4")
x = 10 + 2 * 3 ** 2 - 8 / 4
print(f"Final result: x = {x}")
print("\n--- Step-by-step Calculation ---")

# Step 1: Exponentiation (**) highest precedence
step1_val = 3 ** 2
print(f"1. Exponentiation (3 ** 2): {step1_val}")
current_expression = f"10 + 2 * {step1_val} - 8 / 4"
print(f"   Expression now: {current_expression}")

# Step 2: Multiplication (*) and Division (/) from left to right
step2_val_mult = 2 * step1_val
print(f"2. Multiplication (2 * {step1_val}): {step2_val_mult}")
current_expression = f"10 + {step2_val_mult} - 8 / 4"
print(f"   Expression now: {current_expression}")

step3_val_div = 8 / 4
print(f"3. Division (8 / 4): {step3_val_div}")
current_expression = f"10 + {step2_val_mult} - {step3_val_div}"
print(f"   Expression now: {current_expression}")

# Step 4: Addition (+) and Subtraction (-) from left to right
step4_val_add = 10 + step2_val_mult
print(f"4. Addition (10 + {step2_val_mult}): {step4_val_add}")
current_expression = f"{step4_val_add} - {step3_val_div}"
print(f"   Expression now: {current_expression}")

step5_val_sub = step4_val_add - step3_val_div
print(f"5. Subtraction ({step4_val_add} - {step3_val_div}): {step5_val_sub}")
current_expression = f"{step5_val_sub}"
print(f"   Final calculated value: {current_expression}")

# Verify with the actual Python calculation
print(f"\nVerification with Python's direct calculation: {10 + 2 * 3 ** 2 - 8 / 4}")
```

```
Original expression: x = 10 + 2 * 3 ** 2 - 8 / 4
Final result: x = 26.0

--- Step-by-step Calculation ---
1. Exponentiation (3 ** 2): 9
   Expression now: 10 + 2 * 9 - 8 / 4
2. Multiplication (2 * 9): 18
   Expression now: 10 + 18 - 8 / 4
3. Division (8 / 4): 2.0
   Expression now: 10 + 18 - 2.0
4. Addition (10 + 18): 28
   Expression now: 28 - 2.0
5. Subtraction (28 - 2.0): 26.0
   Final calculated value: 26.0

Verification with Python's direct calculation: 26.0
```

This code demonstrates how the `x = 10 + 2 * 3 ** 2 - 8 / 4` expression is evaluated in Python, explicitly showing the order of operations (operator precedence).

1. **Original Expression and Final Result:**
   - The code first prints the original expression. It then calculates the full expression `x = 10 + 2 * 3 ** 2 - 8 / 4` directly and prints the final result (`x`). Python follows standard mathematical order of operations, often remembered by acronyms like PEMDAS/BODMAS (Parentheses/Brackets, Exponents/Orders, Multiplication and Division (from left to right), Addition and Subtraction (from left to right)).
2. **Step-by-step Calculation Breakdown:**
   - The program then proceeds to break down the calculation into individual steps, adhering to Python's operator precedence rules:
     - **Step 1: Exponentiation (`**`)**
       - `3 ** 2` is calculated first, as exponentiation has the highest precedence. The result `9` is stored in `step1_val` and the expression is updated to show this intermediate result.

- **Step 2: Multiplication (`*`) and Division (`/`)**
  - These operators have equal precedence and are evaluated from left to right.
  - First, `2 * step1_val` (which is `2 * 9`) is calculated, resulting in `18`. This is stored in `step2_val_mult`.
  - Next, `8 / 4` is calculated, resulting in `2.0`. This is stored in `step3_val_div`.
  - After these operations, the expression conceptually becomes `10 + 18 - 2.0`.
- **Step 3: Addition (`+`) and Subtraction (`-`)**
  - These operators also have equal precedence and are evaluated from left to right.
  - First, `10 + step2_val_mult` (which is `10 + 18`) is calculated, resulting in `28`. This is stored in `step4_val_add`.
  - Finally, `step4_val_add - step3_val_div` (which is `28 - 2.0`) is calculated, giving the final result of `26.0`. This is stored in `step5_val_sub`.

3. **Verification:**
   - A final `print` statement verifies that the step-by-step calculation matches Python's direct calculation of the expression, confirming the correct application of operator precedence.

b. Write a Python program that creates a dictionary to store three students' names as keys and their marks as values, then:
i. Prints the dictionary
ii. Prints the marks of a student whose name is entered by the user

```
# Create a dictionary to store student names and their marks
student_marks = {
    "Alice": 85,
    "Bob": 92,
    "Charlie": 78
}

# i. Print the entire dictionary
print("1. Full Student Marks Dictionary:")
print(student_marks)

print("\n" + "-" * 30 + "\n") # Separator for clarity

# ii. Print the marks of a student whose name is entered by the user
print("2. Search for a student's marks:")
student_name_search = input("Enter the name of the student to search for: ")

# Check if the student name exists in the dictionary
if student_name_search in student_marks:
    marks = student_marks[student_name_search]
    print(f"Marks for {student_name_search}: {marks}")
else:
    print(f"Student '{student_name_search}' not found in the dictionary.")
```

```
1. Full Student Marks Dictionary:
{'Alice': 85, 'Bob': 92, 'Charlie': 78}

------------------------------

2. Search for a student's marks:
Enter the name of the student to search for: Bob
Marks for Bob: 92
```

This code demonstrates how to use a Python dictionary to store and retrieve student information. It's broken down into two main parts:

1. **Creating and Printing the Dictionary:**
   - `student_marks = {"Alice": 85, "Bob": 92, "Charlie": 78}`: This line initializes a dictionary named `student_marks`. In this dictionary, student names (e.g., "Alice", "Bob") are used as **keys**, and their corresponding marks (e.g., `85`, `92`) are stored as **values**. Dictionaries are excellent for storing data in key-value pairs, allowing for quick lookups based on the key.
   - The `print(student_marks)` statement then outputs the entire dictionary, showing all student names and their marks.
2. **Searching for a Student's Marks:**
   - `student_name_search = input("Enter the name of the student to search for: ")`: This line prompts the user to enter the name of a student they want to find. Whatever the user types is stored in the `student_name_search` variable.
   - `if student_name_search in student_marks:`: This is a conditional statement that checks if the `student_name_search` (the name entered by the user) exists as a key in the `student_marks` dictionary. This is an efficient way to see if a student is recorded.

- If the student's name *is* found, `marks = student_marks[student_name_search]` retrieves the marks associated with that name from the dictionary, and it's then printed. `student_marks[key]` is the standard way to access a value using its key in a dictionary.
- If the student's name is *not* found (`else` block), a message indicating that the student was not found is printed.

c. Write a program to print the reverse of a number.

```
# Program to print the reverse of a number

# Get input from the user
num = int(input("Enter an integer: "))

# Store the original number for display
original_num = num

reversed_num = 0

# Handle negative numbers
is_negative = False
if num < 0:
    is_negative = True
    num = abs(num) # Work with the absolute value

# Reverse the number using a while loop
while num > 0:
    remainder = num % 10  # Get the last digit
    reversed_num = reversed_num * 10 + remainder # Add the digit to the reversed number
    num = num // 10       # Remove the last digit

# Apply the sign back if it was negative
if is_negative:
    reversed_num = -reversed_num

print(f"The original number is: {original_num}")
print(f"The reversed number is: {reversed_num}")
```

```
Enter an integer: 69
The original number is: 69
The reversed number is: 96
```

This Python code takes an integer input from the user and prints its reverse. Here's a breakdown of its logic:

1. **Get Input:**
   - `num = int(input("Enter an integer: "))` prompts the user to enter an integer. The `input()` function reads the user's input as a string, and `int()` converts it into an integer, which is then stored in the `num` variable.
2. **Store Original Number:**
   - `original_num = num` stores the initial value of `num` so it can be displayed at the end, showing both the original and reversed numbers.
3. **Initialize `reversed_num`:**
   - `reversed_num = 0` initializes a variable that will store the reversed number as it's constructed.
4. **Handle Negative Numbers:**
   - `is_negative = False` a flag to track if the original number was negative.
   - `if num < 0:` checks if the number is negative.
     - If it is, `is_negative` is set to `True`, and `num = abs(num)` converts the number to its absolute (positive) value. This is done because the reversal logic is simpler with positive numbers; the negative sign will be re-applied later.
5. **Reverse the Number (using a `while` loop):**
   - `while num > 0:`: This loop continues as long as `num` is greater than 0. In each iteration, it processes one digit from `num`.
     - `remainder = num % 10`: The modulo operator (`%`) gives the last digit of `num`. For example, `123 % 10` is `3`.
     - `reversed_num = reversed_num * 10 + remainder`: This is the core logic for reversing. It shifts the existing `reversed_num` one decimal place to the left (by multiplying by 10) and then adds the `remainder` (the last digit of the original number).
       - Example: If `reversed_num` is `0` and `remainder` is `3`, `reversed_num` becomes `3`.
       - Next, if `reversed_num` is `3` and `remainder` is `2`, `reversed_num` becomes `3 * 10 + 2 = 32`.

- `num = num // 10`: The floor division operator (`//`) removes the last digit from `num`. For example, `123 // 10` is `12`.

6. **Apply Sign Back:**
   - `if is_negative:`: After the loop finishes, if the original number was negative (as indicated by the `is_negative` flag),
     - `reversed_num = -reversed_num` re-applies the negative sign to the now reversed number.
7. **Display Result:**
   - Finally, `print(f"The original number is: {original_num}")` and `print(f"The reversed number is: {reversed_num}")` display both the input and the calculated reversed number.

d. Write the program to print multiplication table of n, where n is entered by the user.

```
# Program to print the multiplication table of n

# Get input from the user for the number n
num = int(input("Enter a number to see its multiplication table: "))

print(f"\nMultiplication Table for {num}:")

# Use a for loop to generate and print the table from 1 to 10
for i in range(1, 11):
    print(f"{num} x {i} = {num * i}")
```

```
Enter a number to see its multiplication table: 5

Multiplication Table for 5:
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

This Python program generates and prints the multiplication table for a number `n` that the user enters. Let's break down its components:

1. **Get Input (`num = int(input(...))`):**
   - The program first prompts the user to "Enter a number to see its multiplication table: ".
   - The `input()` function reads the user's entry as a string.
   - `int()` converts this string into an integer, which is then stored in the `num` variable. This `num` is the number for which the multiplication table will be generated.
2. **Display Header (`print(f"\nMultiplication Table for {num}:")`):**
   - Before displaying the table, a clear header is printed, indicating which number's multiplication table is about to be shown.
3. **Generate Table (using a `for` loop):**
   - `for i in range(1, 11):` This is the core of the program. A `for` loop is used to iterate through a sequence of numbers.
     - `range(1, 11)` generates numbers starting from `1` up to (but not including) `11`. So, `i` will take values `1, 2, 3, ..., 10`.
   - `print(f"{num} x {i} = {num * i}")`: Inside the loop, for each value of `i`:
     - It calculates the product of `num` (the user's number) and `i` (the current multiplier from 1 to 10).
     - It then prints a formatted string showing the multiplication equation (e.g., "5 x 1 = 5", "5 x 2 = 10"), making the output easy to read.

In summary, the program takes an integer from the user and then uses a loop to systematically print its products with numbers from 1 to 10.

5. Nested Loops, String Multiplication, Keyword Arguments

a. Write a program to print the following pattern:

```
1
```

```
   12
  123
 1234
12345
```

```
# Program to print the given numeric pattern

num_rows = 5

for i in range(1, num_rows + 1):
    # Create the number sequence for the current row (e.g., '1', '12', '123')
    num_sequence = ""
    for j in range(1, i + 1):
        num_sequence += str(j)

    # Right-justify the sequence with spaces to form the pattern
    # The total width should be equal to the width of the last row ('12345')
    print(num_sequence.rjust(num_rows))
```

```
    1
   12
  123
 1234
12345
```

This Python program generates and prints the specified numeric pattern using nested loops and string formatting. Let's break down its components:

1. `num_rows = 5`:
   - This line initializes a variable `num_rows` to `5`, which determines how many rows the pattern will have. You can change this value to print a pattern with more or fewer rows.
2. **Outer `for` loop (`for i in range(1, num_rows + 1):`)**:
   - This loop iterates `num_rows` times, controlling the current row number. The variable `i` will take values from `1` to `num_rows` (inclusive).
   - For each iteration of this loop, one complete line of the pattern is constructed and printed.
3. **Inner `for` loop (`for j in range(1, i + 1):`)**:
   - Inside the outer loop, this inner loop is responsible for creating the sequence of numbers for the *current* row.
   - It iterates from `1` up to the current value of `i`.
   - `num_sequence += str(j)`: In each iteration, it converts the number `j` to a string and appends it to `num_sequence`. This builds strings like "1", "12", "123", and so on.
4. `print(num_sequence.rjust(num_rows))`:
   - After the inner loop completes and `num_sequence` contains the desired numbers for the current row (e.g., "123" for `i=3`), this line prints the sequence.
   - `.rjust(num_rows)` is a string method that right-justifies the string within a field of a specified width. The width used here is `num_rows` (which is 5). This ensures that the numbers are aligned to the right, creating the triangular pattern with leading spaces.
     - For `i=1`, `num_sequence` is "1". `"1".rjust(5)` becomes " 1".
     - For `i=2`, `num_sequence` is "12". `"12".rjust(5)` becomes " 12".
     - ...and so on.

In summary, the outer loop controls the row count, the inner loop builds the numeric string for each row, and `rjust()` ensures the correct alignment to form the desired pattern.

b. Write the program to print multiplication table of n, where n is entered by the user.

```
# Program to print the multiplication table of n

# Get input from the user for the number n
num = int(input("Enter a number to see its multiplication table: "))

print(f"\nMultiplication Table for {num}:")

# Use a for loop to generate and print the table from 1 to 10
```

```
    for i in range(1, 11):
        print(f"{num} x {i} = {num * i}")
```

```
Enter a number to see its multiplication table: 10

Multiplication Table for 10:
10 x 1 = 10
10 x 2 = 20
10 x 3 = 30
10 x 4 = 40
10 x 5 = 50
10 x 6 = 60
10 x 7 = 70
10 x 8 = 80
10 x 9 = 90
10 x 10 = 100
```

This Python program generates and prints the multiplication table for a number `n` that the user enters. Let's break down its components:

1. **Get Input (`num = int(input(...))`):**
    - The program first prompts the user to "Enter a number to see its multiplication table: ".
    - The `input()` function reads the user's entry as a string.
    - `int()` converts this string into an integer, which is then stored in the `num` variable. This `num` is the number for which the multiplication table will be generated.
2. **Display Header (`print(f"\nMultiplication Table for {num}:")`):**
    - Before displaying the table, a clear header is printed, indicating which number's multiplication table is about to be shown.
3. **Generate Table (using a `for` loop):**
    - `for i in range(1, 11):` This is the core of the program. A `for` loop is used to iterate through a sequence of numbers.
        - `range(1, 11)` generates numbers starting from `1` up to (but not including) `11`. So, `i` will take values `1, 2, 3, ..., 10`.
    - `print(f"{num} x {i} = {num * i}")`: Inside the loop, for each value of `i`:
        - It calculates the product of `num` (the user's number) and `i` (the current multiplier from 1 to 10).
        - It then prints a formatted string showing the multiplication equation (e.g., "5 x 1 = 5", "5 x 2 = 10"), making the output easy to read.

In summary, the program takes an integer from the user and then uses a loop to systematically print its products with numbers from 1 to 10.

c. Write a Python function student info that takes three keyword arguments: name, age, and course. Call the function using keyword arguments in any order and print the information in a readable format.

```
def student_info(name, age, course):
    """
    Prints student information using keyword arguments.

    Args:
        name (str): The name of the student.
        age (int): The age of the student.
        course (str): The course the student is enrolled in.
    """
    print("\n--- Student Information ---")
    print(f"Name: {name}")
    print(f"Age: {age}")
    print(f"Course: {course}")
    print("---------------------------")

# Call the function using keyword arguments in different orders
print("Calling with arguments in standard order:")
student_info(name="Alice", age=20, course="Computer Science")

print("Calling with arguments in a different order:")
student_info(course="Physics", name="Bob", age=22)

print("Calling with yet another order:")
student_info(age=21, course="Mathematics", name="Charlie")
```

```
Calling with arguments in standard order:

--- Student Information ---
```

```
Name: Alice
Age: 20
Course: Computer Science
--------------------------
Calling with arguments in a different order:

--- Student Information ---
Name: Bob
Age: 22
Course: Physics
--------------------------
Calling with yet another order:

--- Student Information ---
Name: Charlie
Age: 21
Course: Mathematics
--------------------------
```

This Python function `student_info` demonstrates the use of keyword arguments. Let's break down its components:

1. **Function Definition (`def student_info(name, age, course):`)**:
   - This defines a function named `student_info` that expects three parameters: `name`, `age`, and `course`.
   - These parameters are set up to receive values through keyword arguments when the function is called.
   - The docstring (the triple-quoted string right after `def`) provides a brief description of what the function does, its arguments, and what it might return (though this function just prints, it's good practice).
2. **Printing Information**:
   - Inside the function, several `print` statements use f-strings to display the student's name, age, and course in a clean, readable format.
3. **Calling the Function with Keyword Arguments**:
   - `student_info(name="Alice", age=20, course="Computer Science")`:
     - This is a standard way to call the function, explicitly mapping the values to the parameter names (`name`, `age`, `course`).
   - `student_info(course="Physics", name="Bob", age=22)`:
     - This call demonstrates the key advantage of keyword arguments: **the order doesn't matter**. Even though `course` is listed first here, Python correctly assigns "Physics" to the `course` parameter because the keyword `course=` is used.
   - `student_info(age=21, course="Mathematics", name="Charlie")`:
     - This further emphasizes that keyword arguments allow you to pass values in any order, improving readability and making your function calls more robust if the parameter order changes in the function definition (though that's less common).

In summary, keyword arguments make function calls more explicit and flexible, allowing you to pass arguments by their name rather than relying solely on their positional order.

d. Write a program that prints absolute value, square root and cube of a number.

```
# Program to print absolute value, square root, and cube of a number

# Get input from the user
num = float(input("Enter a number: "))

# Calculate absolute value
abs_value = abs(num)

# Calculate square root
# The square root is only defined for non-negative numbers in real numbers
if num >= 0:
    square_root = num ** 0.5
else:
    square_root = "Undefined (for real numbers)"

# Calculate cube
cube_value = num ** 3

# Print the results
print(f"\nOriginal Number: {num}")
print(f"Absolute Value: {abs_value}")
print(f"Square Root: {square_root}")
print(f"Cube: {cube_value}")
```

```
Enter a number: 4

Original Number: 4.0
Absolute Value: 4.0
Square Root: 2.0
Cube: 64.0
```

This Python program takes a number from the user and calculates its absolute value, square root, and cube. Let's break it down:

1. **Get Input (`num = float(input("Enter a number: "))`)**:
   - The program prompts the user to "Enter a number: ".
   - The `input()` function reads the user's entry as a string.
   - `float()` converts this string into a floating-point number, which is then stored in the `num` variable. This allows the program to handle both integers and decimal numbers.
2. **Calculate Absolute Value (`abs_value = abs(num)`)**:
   - The built-in `abs()` function returns the absolute value of the number (`num`). The absolute value is the non-negative value of a number, regardless of its sign (e.g., `abs(-5)` is `5`, and `abs(5)` is `5`).
3. **Calculate Square Root (`if num >= 0: ... else: ...`)**:
   - The square root of a number is typically defined for non-negative numbers in the domain of real numbers. So, there's a conditional check:
     - `if num >= 0:`: If the input number is non-negative, `square_root = num ** 0.5` calculates the square root using the exponentiation operator (`**`). Raising a number to the power of `0.5` is equivalent to taking its square root.
     - `else:`: If the input number is negative, the square root is considered "Undefined (for real numbers)", and this string is assigned to `square_root`.
4. **Calculate Cube (`cube_value = num ** 3`)**:
   - `cube_value = num ** 3` calculates the cube of the number by raising it to the power of 3.
5. **Print the Results (`print(f"\nOriginal Number: {num}") ...`)**:
   - Finally, the program uses f-strings to display the original number, its absolute value, square root, and cube in a clear and readable format.

6. Program based on Functions

a. Write a Python program that uses a lambda function to:

i. Take a number as input.

ii. Use a lambda function to square the number.

iii. Print the squared value.

```
# 1. Take a number as input
num_input = float(input("Enter a number: "))

# 2. Use a lambda function to square the number
square_number = lambda x: x ** 2

# 3. Print the squared value
squared_result = square_number(num_input)
print(f"The square of {num_input} is: {squared_result}")
```

```
Enter a number: 5
The square of 5.0 is: 25.0
```

This Python program demonstrates how to use a `lambda` function for a simple operation. Let's break it down:

1. **`num_input = float(input("Enter a number: "))`**:
   - This line prompts the user to "Enter a number: ".
   - The `input()` function reads whatever the user types (which is always a string).
   - `float()` then converts that string into a floating-point number, allowing for both whole numbers and decimals. This number is stored in the `num_input` variable.
2. **`square_number = lambda x: x ** 2`**:
   - This is the core of the `lambda` function demonstration.

- `lambda` is a keyword in Python used to create small, anonymous (unnamed) functions.
- `x` is the argument that the `lambda` function accepts.
- `x ** 2` is the expression that the `lambda` function evaluates and returns. In this case, it squares the input argument `x`.
- The entire `lambda` function is assigned to the variable `square_number`, making `square_number` behave like a regular function.

3. **`squared_result = square_number(num_input)`**:
   - This line calls the `square_number` lambda function, passing `num_input` (the number entered by the user) as its argument.
   - The result of squaring `num_input` is then stored in the `squared_result` variable.
4. **`print(f"The square of {num_input} is: {squared_result}")`**:
   - Finally, this line prints a formatted output (using an f-string) showing both the original number and its calculated square.

b. Write a recursive function count digits(n) that returns the number of digits in a positive integer n.

```
def count_digits(n):
    """
    Recursively counts the number of digits in a positive integer.

    Args:
        n (int): A positive integer.

    Returns:
        int: The number of digits in n.
    """
    # Handle the case of 0 separately if it's considered a positive integer in context (1 digit)
    if n == 0:
        return 1

    # Ensure n is positive before proceeding with recursion
    if n < 0:
        n = abs(n)

    # Base case: if n is a single digit (0-9)
    if n < 10:
        return 1
    # Recursive step: remove the last digit and add 1 to the count
    else:
        return 1 + count_digits(n // 10)

# --- Example Usage ---
print("Program to count digits in a number using a recursive function")

user_input = int(input("Enter a positive integer: "))

if user_input < 0:
    print("Please enter a non-negative integer for this function.")
else:
    num_digits = count_digits(user_input)
    print(f"The number of digits in {user_input} is: {num_digits}")
```

```
Program to count digits in a number using a recursive function
Enter a positive integer: 5
The number of digits in 5 is: 1
```

This Python code defines a **recursive function** `count_digits(n)` to determine the number of digits in a given integer. Here's a detailed breakdown:

`count_digits(n)` Function:

1. **Handling Zero:**
   - `if n == 0: return 1`:
     - The number `0` is considered to have one digit, so this is a specific base case.
2. **Handling Negative Numbers:**
   - `if n < 0: n = abs(n)`:
     - If a negative number is provided, the code converts it to its absolute value. This simplifies the logic, as the number of digits in -123 is the same as in 123.
3. **Base Case for Single-Digit Numbers:**
   - `if n < 10: return 1`:

- This is the primary base case for positive numbers. If `n` is a single-digit number (e.g., 1, 5, 9), it directly returns `1`, as it has one digit.

4. **Recursive Step:**
   - `else: return 1 + count_digits(n // 10)`:
     - This is the recursive part. If `n` has more than one digit:
       - `n // 10` performs integer division, effectively removing the last digit of `n`. For example, `123 // 10` becomes `12`.
       - The function then calls itself with this new, smaller number (`n // 10`).
       - `1 + ...` means that for each digit removed, we add `1` to the total count from the subsequent recursive calls. This continues until `n` becomes a single-digit number (triggering the base case).

Example Usage:

- The program prompts the user to "Enter a positive integer:".
- It checks if the `user_input` is negative and provides a message, as the problem statement specifies a 'positive integer'. The `count_digits` function itself can handle negative numbers by converting them to positive internally.
- Finally, it prints the original number and the total number of digits returned by the `count_digits` function.

c. Write a Python program to:

i. Take n integers into a list.

ii. Print the list in original order.

iii. Print the list in reverse order (without using [::-1] or reverse()).

```
# Python program to handle a list of integers

my_list = []

# i. Take n integers into a list
n = int(input("Enter the number of integers you want to add to the list: "))

if n <= 0:
    print("Please enter a positive number for the count of integers.")
else:
    print("Enter your integers one by one:")
    for i in range(n):
        num = int(input(f"Enter integer {i + 1}: "))
        my_list.append(num)

    # ii. Print the list in original order
    print(f"\nOriginal list: {my_list}")

    # iii. Print the list in reverse order (without using [::-1] or reverse())
    print("List in reverse order:", end=" ")
    for i in range(len(my_list) - 1, -1, -1):
        print(my_list[i], end=" ")
    print() # Newline after printing reversed list
```

```
Enter the number of integers you want to add to the list: 3
Enter your integers one by one:
Enter integer 1: 4
Enter integer 2: 5
Enter integer 3: 7

Original list: [4, 5, 7]
List in reverse order: 7 5 4
```

This Python program takes a user-specified number of integers, stores them in a list, and then prints the list in both its original and reverse order without using Python's built-in `[::-1]` slicing or the `reverse()` method. Let's break it down:

1. **Initialize an Empty List (`my_list = []`)**:
   - An empty list named `my_list` is created to store the integers that the user will input.
2. **Take 'n' Integers into a List (`n = int(input(...))` and `for` loop)**:
   - The program first prompts the user to "Enter the number of integers you want to add to the list:". The input is converted to an integer and stored in `n`.
   - An `if n <= 0:` check ensures that the user enters a positive number for the count of integers.
   - If `n` is valid, a `for` loop `for i in range(n):` iterates `n` times.

- Inside the loop, `num = int(input(f"Enter integer {i + 1}: "))` prompts the user to enter each integer one by one. The input is converted to an integer and then appended to `my_list` using `my_list.append(num)`.

3. **Print the List in Original Order (`print(f"\nOriginal list: {my_list}")`)**:
   - After all integers are collected, this line simply prints the `my_list` directly, displaying its contents in the order they were entered.
4. **Print the List in Reverse Order (without `[::-1]` or `reverse()`)**:
   - `print("List in reverse order:", end=" ")`: This starts printing a descriptive label, and `end=" "` prevents a newline, allowing the reversed numbers to appear on the same line.
   - `for i in range(len(my_list) - 1, -1, -1):`: This is the core logic for manual reversal. A `for` loop is used with `range()`:
     - `len(my_list) - 1`: This is the starting index for the loop. Lists in Python are zero-indexed, so the last element's index is `length - 1`.
     - `-1`: This is the stop value for the range. The loop will go *down to* (but not include) -1, meaning it will process elements from the last index down to index 0.
     - `-1`: This is the step value, indicating that the loop counter `i` should decrease by 1 in each iteration.
   - `print(my_list[i], end=" ")`: Inside this loop, `my_list[i]` accesses the element at the current index `i` (starting from the end and moving towards the beginning), and it's printed, again using `end=" "` to keep numbers on the same line.
   - `print()`: Finally, an empty `print()` statement is used to move to the next line after all reversed numbers have been printed.

7. Modules and packages

a. Write a Python program that:

i. Imports the random module with alias name rnd.

ii. Prints 5 random integers between 1 and 100 using the alias.

```
# i. Imports the random module with alias name rnd.
import random as rnd

# ii. Prints 5 random integers between 1 and 100 using the alias.
print("Generating 5 random integers between 1 and 100:")
for _ in range(5):
    random_number = rnd.randint(1, 100)
    print(random_number)
```

```
Generating 5 random integers between 1 and 100:
4
90
51
79
14
```

This Python program demonstrates how to import and use the `random` module with an alias to generate random integers. Let's break it down:

1. **`import random as rnd`**:
   - This line imports Python's built-in `random` module. The `as rnd` part is an **alias**, meaning that instead of typing `random.` every time you want to use a function from this module, you can simply type `rnd.`.
   - This is useful for shortening module names or avoiding name clashes if you have multiple modules with similar function names.
2. **`print("Generating 5 random integers between 1 and 100:")`**:
   - This is a simple `print` statement to provide a descriptive header for the output.
3. **`for _ in range(5):`**:
   - This is a `for` loop that iterates 5 times. The underscore `_` is used as the loop variable when its value itself is not needed within the loop (it's just used to control the number of iterations).
4. **`random_number = rnd.randint(1, 100)`**:
   - Inside the loop, `rnd.randint(1, 100)` is called.
     - `rnd` refers to the `random` module (via its alias).

- `randint(a, b)` is a function from the `random` module that returns a random integer `N` such that `a <= N <= b`. In this case, it generates a random integer between 1 and 100 (inclusive).
- The generated random integer is stored in the `random_number` variable.

5. **`print(random_number)`**:
   - This line prints the `random_number` generated in the current iteration of the loop. Since the loop runs 5 times, 5 different random numbers are printed.

b. Write a Python program that:

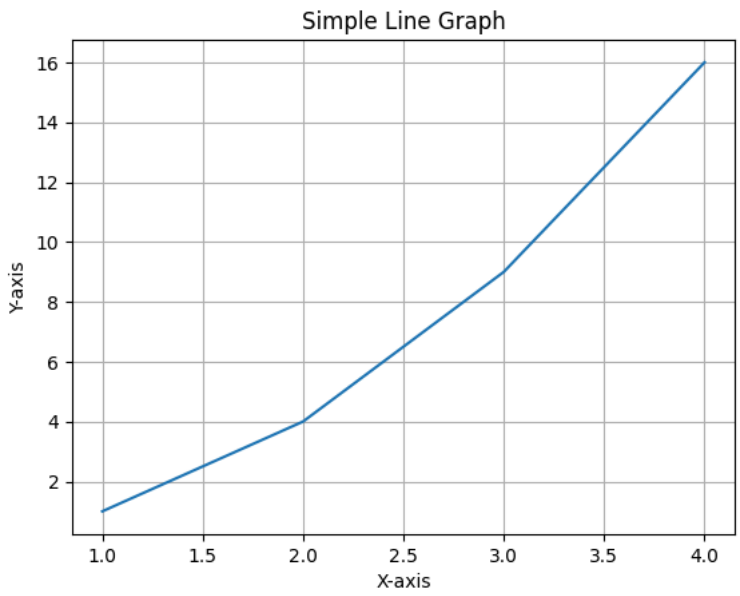
i. Imports pyplot from the matplotlib package as plt.

ii. Plots a simple line graph for x = [1, 2, 3, 4] and y = [1, 4, 9, 16]

```
# i. Imports pyplot from the matplotlib package as plt.
import matplotlib.pyplot as plt

# ii. Plots a simple line graph for x = [1, 2, 3, 4] and y = [1, 4, 9, 16]
x = [1, 2, 3, 4]
y = [1, 4, 9, 16]

plt.plot(x, y)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Simple Line Graph")
plt.grid(True)
plt.show()
```



This Python program demonstrates how to use the `matplotlib` library, specifically its `pyplot` module, to create and display a simple line graph. Let's break down its components:

1. **`import matplotlib.pyplot as plt`**:
   - This line imports the `pyplot` module from the `matplotlib` library. `matplotlib` is a comprehensive library for creating static, animated, and interactive visualizations in Python.
   - `as plt` creates an alias, allowing you to refer to `matplotlib.pyplot` simply as `plt` throughout your code, which is a common convention.
2. **`x = [1, 2, 3, 4]` and `y = [1, 4, 9, 16]`**:
   - These lines define two lists, `x` and `y`, which hold the data points for the graph.
   - `x` represents the values on the horizontal (x-axis), and `y` represents the values on the vertical (y-axis). Notice that `y` values are the squares of `x` values.
3. **`plt.plot(x, y)`**:
   - This is the core command that creates the line graph. It tells `pyplot` to plot `y` against `x`.
4. **`plt.xlabel("X-axis")`**:
   - Sets the label for the x-axis to "X-axis".

5. `plt.ylabel("Y-axis")`:
   - Sets the label for the y-axis to "Y-axis".
6. `plt.title("Simple Line Graph")`:
   - Sets the title of the graph to "Simple Line Graph".
7. `plt.grid(True)`:
   - Adds a grid to the plot, which can help in reading the values more easily.
8. `plt.show()`:
   - This command displays the plot. Without `plt.show()`, the graph might not appear, especially in non-interactive environments.

8. String and its operation

a. Write a program that takes a string as input and prints a new string where all vowels are removed, but the order of the remaining characters is preserved.

```
# Program to remove vowels from a string

# Get input string from the user
input_string = input("Enter a string: ")

# Define a set of vowels for quick lookup (case-insensitive)
vowels = "aeiouAEIOU"

# Create a new string by iterating through the input string
# and appending non-vowel characters
new_string = ""
for char in input_string:
    if char not in vowels:
        new_string += char

# Print the new string without vowels
print(f"Original string: {input_string}")
print(f"String without vowels: {new_string}")
```

```
Enter a string: sri
Original string: sri
String without vowels: sr
```

This Python program takes a string as input and prints a new string where all vowels are removed, but the order of the remaining characters is preserved. Let's break it down:

1. **Get Input String (`input_string = input("Enter a string: ")`)**:
   - This line prompts the user to "Enter a string:".
   - The `input()` function reads whatever the user types and stores it as a string in the `input_string` variable.
2. **Define Vowels (`vowels = "aeiouAEIOU"`)**:
   - A string `vowels` is created containing all lowercase and uppercase English vowels. This is used to quickly check if a character is a vowel.
3. **Initialize New String (`new_string = ""`)**:
   - An empty string `new_string` is created. This string will be built up with only the non-vowel characters from the input string.
4. **Iterate and Filter (`for char in input_string: ...`)**:
   - A `for` loop iterates through each character (`char`) in the `input_string`.
   - `if char not in vowels:`: Inside the loop, this conditional statement checks if the current `char` is *not* present in the `vowels` string.
     - If the character is *not* a vowel, it is appended to the `new_string` using `new_string += char`.
5. **Print Results (`print(f"Original string: {input_string}") ...`)**:
   - Finally, the program prints both the `original_string` and the `new_string` (which contains no vowels), using f-strings for clear output.

b. Given a string, write a program to check whether it is a palindrome or not, ignoring case and all non-alphanumeric characters.

```
# Program to check if a string is a palindrome, ignoring case and non-alphanumeric characters

# Get input string from the user
input_string = input("Enter a string: ")

# 1. Preprocess the string:
#    a. Convert to lowercase
#    b. Remove non-alphanumeric characters
cleaned_string = ""
for char in input_string:
    if 'a' <= char <= 'z' or 'A' <= char <= 'Z' or '0' <= char <= '9':
        cleaned_string += char.lower()

# 2. Check if it's a palindrome
#    Compare the cleaned string with its reverse
is_palindrome = (cleaned_string == cleaned_string[::-1])

# 3. Print the result
print(f"Original string: '{input_string}'")
print(f"Cleaned string (for palindrome check): '{cleaned_string}'")

if is_palindrome:
    print("Result: It is a palindrome!")
else:
    print("Result: It is NOT a palindrome.")
```

```
Enter a string: 343
Original string: '343'
Cleaned string (for palindrome check): '343'
Result: It is a palindrome!
```

This Python program takes a string as input and checks whether it is a palindrome or not, ignoring case and all non-alphanumeric characters. Let's break down its components:

1. **Get Input String (`input_string = input("Enter a string: ")`)**:
   - This line prompts the user to "Enter a string:".
   - The `input()` function reads whatever the user types and stores it as a string in the `input_string` variable.
2. **Preprocess the String (`cleaned_string = "" ...`)**:
   - This is a crucial step for handling the "ignoring case and all non-alphanumeric characters" requirement.
   - An empty string `cleaned_string` is initialized to store only the relevant characters.
   - A `for` loop iterates through each character (`char`) in the `input_string`.
   - `if 'a' <= char <= 'z' or 'A' <= char <= 'Z' or '0' <= char <= '9':`: This condition checks if the current character is an alphabet (lowercase or uppercase) or a digit.
   - If it is an alphanumeric character, `cleaned_string += char.lower()` converts the character to lowercase and appends it to `cleaned_string`. This ensures that case is ignored (e.g., 'A' becomes 'a', so 'Racecar' becomes 'racecar').
   - Characters that are not alphanumeric (like spaces, punctuation, etc.) are simply skipped.
3. **Check if it's a Palindrome (`is_palindrome = (cleaned_string == cleaned_string[::-1])`)**:
   - After preprocessing, `cleaned_string` contains only lowercase alphanumeric characters.
   - `cleaned_string[::-1]` is a Python string slice that creates a reversed version of `cleaned_string`.
   - The `==` operator then compares the `cleaned_string` with its reversed version. If they are identical, the string is a palindrome, and `is_palindrome` is set to `True`; otherwise, it's `False`.
4. **Print the Result (`print(...)`)**:
   - The original and cleaned strings are printed for clarity.
   - An `if-else` statement then checks the value of `is_palindrome` and prints a message indicating whether the input string is a palindrome or not.

c. Write a program that takes a sentence as input and prints the words in reverse order, while keeping the characters inside each word in the same order.

```
# Program to reverse the order of words in a sentence

# Get input sentence from the user
input_sentence = input("Enter a sentence: ")

# Split the sentence into words
# By default, split() splits by any whitespace and handles multiple spaces
words = input_sentence.split()
```

```
# Reverse the order of the words in the list
reversed_words = words[::-1] # Using slicing to reverse the list of words

# Join the reversed words back into a sentence
# Using ' ' as a separator to put a single space between words
reversed_sentence = " ".join(reversed_words)

# Print the result
print(f"Original sentence: '{input_sentence}'")
print(f"Sentence with words reversed: '{reversed_sentence}'")
```

```
Enter a sentence: How are you?
Original sentence: 'How are you?'
Sentence with words reversed: 'you? are How'
```

This Python program takes a sentence as input and prints the words in reverse order, while keeping the characters inside each word in the same order. Let's break it down:

1. **Get Input Sentence (`input_sentence = input("Enter a sentence: ")`)**:
   - This line prompts the user to "Enter a sentence:".
   - The `input()` function reads whatever the user types and stores it as a string in the `input_sentence` variable.
2. **Split the Sentence into Words (`words = input_sentence.split()`)**:
   - The `split()` method, when called without any arguments on a string, splits the string by any whitespace (spaces, tabs, newlines) and handles multiple spaces between words correctly. It returns a list of strings, where each string is a word from the original sentence.
3. **Reverse the Order of the Words (`reversed_words = words[::-1]`)**:
   - This line uses **list slicing** (`[::-1]`) to create a new list `reversed_words` that contains all the elements of the `words` list but in reverse order. This is a very common and Pythonic way to reverse a list.
4. **Join the Reversed Words Back into a Sentence (`reversed_sentence = " ".join(reversed_words)`)**:
   - The `join()` string method is used here. It takes an iterable (like our `reversed_words` list) as an argument and concatenates its elements into a new string.
   - The string on which `join()` is called (`" "` in this case) acts as the separator between the elements. So, a single space will be inserted between each word from the `reversed_words` list.
5. **Print the Result (`print(...)`)**:
   - Finally, the program prints both the `original_sentence` and the `reversed_sentence` using f-strings for clear and informative output.

9. File Handling

a. Write a program that counts the number of tabs, spaces and newline characters in a file.

```
def count_special_characters(filepath):
    """
    Counts the number of tabs, spaces, and newline characters in a given file.

    Args:
        filepath (str): The path to the file.

    Returns:
        tuple: A tuple containing (tab_count, space_count, newline_count).
    """
    tab_count = 0
    space_count = 0
    newline_count = 0

    with open(filepath, 'r', encoding='utf-8') as file:
        content = file.read()
        for char in content:
            if char == '\t':
                tab_count += 1
            elif char == ' ':
                space_count += 1
            elif char == '\n':
                newline_count += 1
    return tab_count, space_count, newline_count

# --- Example Usage ---
```

```
# Create a dummy file for demonstration purposes
# In a real scenario, you would use an existing file.
file_content = "Hello\tWorld\nThis is a test with some spaces\nand\ttabs."
with open("sample_file.txt", "w", encoding="utf-8") as f:
    f.write(file_content)
print("Created 'sample_file.txt' for demonstration.")

# Specify the path to your file
# Replace 'sample_file.txt' with your actual file path if needed
file_to_analyze = "sample_file.txt"

counts = count_special_characters(file_to_analyze)

if counts:
    tabs, spaces, newlines = counts
    print(f"\nAnalysis for '{file_to_analyze}':")
    print(f"Tabs: {tabs}")
    print(f"Spaces: {spaces}")
    print(f"Newlines: {newlines}")
```

```
Created 'sample_file.txt' for demonstration.

Analysis for 'sample_file.txt':
Tabs: 2
Spaces: 6
Newlines: 2
```

This Python program counts the number of tabs, spaces, and newline characters in a file. Let's break down its components:

`count_special_characters(filepath)` Function:

1. **Function Definition:**
   - `def count_special_characters(filepath):` defines a function that takes one argument: `filepath` (a string representing the path to the file).
2. **Initialization:**
   - `tab_count = 0`, `space_count = 0`, `newline_count = 0`: These variables are initialized to zero to store the counts of each special character.
3. **File Handling (`try...except` and `with open(...)`):**
   - The code uses a `try...except` block to gracefully handle potential errors during file operations.
   - `with open(filepath, 'r', encoding='utf-8') as file:`:
     - This opens the file specified by `filepath` in read mode (`'r'`).
     - `encoding='utf-8'` is specified to handle a wide range of characters correctly.
     - The `with` statement ensures that the file is automatically closed even if errors occur.
   - `content = file.read()`: Reads the entire content of the file into a single string variable `content`.
4. **Character Counting Loop:**
   - `for char in content:`: This loop iterates through each character in the `content` string.
   - `if char == '\t': tab_count += 1`: If the character is a tab (`\t`), its counter is incremented.
   - `elif char == ' ': space_count += 1`: If the character is a space ( ), its counter is incremented.
   - `elif char == '\n': newline_count += 1`: If the character is a newline (`\n`), its counter is incremented.
   - `return tab_count, space_count, newline_count`: After iterating through all characters, the function returns the three counts as a tuple.
5. **Error Handling:**
   - `except FileNotFoundError: ...`: Catches the error if the specified file does not exist.
   - `except Exception as e: ...`: Catches any other general exceptions that might occur during file processing.

Example Usage:

1. **Dummy File Creation:**
   - `file_content = "Hello\tWorld\nThis is a test with some spaces\nand\ttabs."`: A string containing tabs, spaces, and newlines is defined.
   - `with open("sample_file.txt", "w", encoding="utf-8") as f: f.write(file_content)`: This section creates a temporary file named `sample_file.txt` and writes the `file_content` into it. This is done so the example can run without needing an existing file.
2. **Function Call and Result Display:**

- `file_to_analyze = "sample_file.txt"`: The path to the dummy file is set.
- `counts = count_special_characters(file_to_analyze)`: The `count_special_characters` function is called with the file path, and its returned tuple of counts is stored in `counts`.
- `if counts: ...`: Checks if the function returned valid counts (not `None` in case of an error).
- The `print` statements then unpack the `counts` tuple and display the individual counts for tabs, spaces, and newlines in a readable format.

b. Write a program that copies first 10 bytes of a binary file into another.

This Python program demonstrates how to copy a specific number of bytes from one binary file to another. Let's break down its components:

`copy_first_n_bytes(source_filepath, destination_filepath, num_bytes)` Function:

1. **Function Definition:**
   - `def copy_first_n_bytes(source_filepath, destination_filepath, num_bytes):` defines a function that takes three arguments:
     - `source_filepath` (string): The path to the original binary file.
     - `destination_filepath` (string): The path where the copied bytes will be written.
     - `num_bytes` (integer): The number of bytes to copy from the beginning of the source file.
2. **Reading from Source File:**
   - `with open(source_filepath, 'rb') as source_file:`:
     - This opens the file specified by `source_filepath` in **read binary mode** (`'rb'`). The `'b'` is crucial for handling non-text data.
     - The `with` statement ensures the file is properly closed after its block is exited.
   - `data = source_file.read(num_bytes)`:
     - The `read(num_bytes)` method reads exactly `num_bytes` from the file. If the file contains fewer bytes than `num_bytes`, it reads all available bytes until the end of the file.
     - The read content is stored as a `bytes` object in the `data` variable.
3. **Writing to Destination File:**
   - `with open(destination_filepath, 'wb') as destination_file:`:
     - This opens the file specified by `destination_filepath` in **write binary mode** (`'wb'`). If the file doesn't exist, it's created; if it does, its contents are truncated (emptied) before writing.
   - `destination_file.write(data)`:
     - The `write()` method writes the `bytes` object (`data`) into the destination file.
4. **Confirmation and Note:**
   - `print(f"Successfully copied the first {len(data)} bytes...")`: Confirms how many bytes were copied.
   - `if len(data) < num_bytes: ...`: Informs the user if the source file was smaller than the requested number of bytes to copy.

Example Usage:

1. **Setting File Names and Bytes to Copy:**
   - `source_file_name`, `destination_file_name`, and `bytes_to_copy` are defined for clarity.
2. **Creating a Dummy Source File:**
   - `dummy_content = b"This is a sample binary file content..."`: A `bytes` literal (`b"..."`) is created for demonstration.
   - `with open(source_file_name, 'wb') as f: f.write(dummy_content)`: This creates a file named `source_binary.bin` and writes the `dummy_content` to it. This ensures the program has a file to work with.
3. **Calling the Copy Function:**
   - `copy_first_n_bytes(...)`: The main function is called to perform the copy operation.
4. **Verifying Destination File Content (Optional):**
   - `with open(destination_file_name, 'rb') as f: copied_content = f.read()`: The destination file is opened in read binary mode, and its content is read.

- `print(f"Content of '{destination_file_name}': {copied_content}")`: The content of the destination file is printed to verify that only the first 10 bytes were copied correctly.

10. Data Structure: Coding exercises

a) Write a Python program to:

i. Take n integers from the user and store them in a list.

ii. Create a new list that contains the same elements but without duplicates.

iii. Print both the original list and the new list.

```
# i. Take n integers from the user and store them in a list.
original_list = []

while True:
    try:
        n = int(input("Enter the number of integers you want to add to the list: "))
        if n <= 0:
            print("Please enter a positive number for the count of integers.")
        else:
            break
    except ValueError:
        print("Invalid input. Please enter an integer.")

print("Enter your integers one by one:")
for i in range(n):
    while True:
        try:
            num = int(input(f"Enter integer {i + 1}: "))
            original_list.append(num)
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

# ii. Create a new list that contains the same elements but without duplicates.
# Using a set to efficiently remove duplicates, then convert back to a list.
unique_list = list(set(original_list))

# iii. Print both the original list and the new list.
print(f"\nOriginal list: {original_list}")
print(f"List without duplicates: {unique_list}")
```

```
Enter the number of integers you want to add to the list: 5
Enter your integers one by one:
Enter integer 1: 4
Enter integer 2: 14
Enter integer 3: 24
Enter integer 4: 34
Enter integer 5: 44

Original list: [4, 14, 24, 34, 44]
List without duplicates: [34, 4, 44, 14, 24]
```

This Python program does the following:

1. **Initializes an Empty List (`original_list = []`)**:
   - An empty list is created to store the integers that the user will input.
2. **Takes `n` Integers as Input**:
   - It first prompts the user to enter the `n` (number of integers) they want to add to the list. It includes `try-except` blocks to handle potential `ValueError` if the user enters non-integer input and `while` loops to ensure valid positive integer input for `n` and each subsequent number.
   - A `for` loop then iterates `n` times, prompting the user to enter each integer one by one. Each valid integer is converted to an `int` and appended to the `original_list`.
3. **Creates a New List Without Duplicates (`unique_list = list(set(original_list))`)**:
   - This is the core for removing duplicates:
     - `set(original_list)`: Converts the `original_list` into a `set`. A key property of sets is that they only store unique elements; any duplicate values are automatically discarded during this conversion.
     - `list(...)`: Converts the `set` back into a `list`.

- It's important to note that converting to a set and back to a list does **not** preserve the original order of elements. If order preservation were critical, a different approach (like iterating and checking for existence in a new list) would be needed.

4. **Prints Both Lists**:
   - Finally, the program prints both the `original_list` (with potentially duplicate elements) and the `unique_list` (with all duplicates removed) using f-strings for clear output.

b. Write a program that reads two sequences of integers (allow duplicates) and prints:

i. the set of elements common to both sequences,

ii. the set of elements that appear in exactly one of the two sequences (symmetric difference). Elements in each output set should be printed in sorted order without duplicates.

```
def get_integer_sequence(prompt):
    """
    Reads a sequence of integers from the user, allowing duplicates.
    Returns a list of integers.
    """
    input_str = input(prompt)
    # Convert string of numbers to a list of integers
    # filter(None, ...) handles multiple spaces by removing empty strings
    numbers = [int(x) for x in input_str.split() if x.strip()]
    return numbers

# Get the first sequence of integers
sequence1_list = get_integer_sequence("Enter the first sequence of integers (space-separated): ")

# Get the second sequence of integers
sequence2_list = get_integer_sequence("Enter the second sequence of integers (space-separated): ")

# Convert lists to sets to easily handle duplicates and perform set operations
set1 = set(sequence1_list)
set2 = set(sequence2_list)

print(f"\nOriginal Sequence 1: {sequence1_list}")
print(f"Original Sequence 2: {sequence2_list}")
print(f"Set 1 (unique elements from sequence 1): {set1}")
print(f"Set 2 (unique elements from sequence 2): {set2}")

# i. Find the set of elements common to both sequences (intersection)
common_elements = sorted(list(set1.intersection(set2)))
print(f"\n1. Common elements in both sequences: {common_elements}")

# ii. Find the set of elements that appear in exactly one of the two sequences (symmetric difference)
symmetric_difference_elements = sorted(list(set1.symmetric_difference(set2)))
print(f"2. Elements in exactly one sequence: {symmetric_difference_elements}")
```

```
Enter the first sequence of integers (space-separated): 1 2 3 4 5
Enter the second sequence of integers (space-separated): 9 8 7 6 5

Original Sequence 1: [1, 2, 3, 4, 5]
Original Sequence 2: [9, 8, 7, 6, 5]
Set 1 (unique elements from sequence 1): {1, 2, 3, 4, 5}
Set 2 (unique elements from sequence 2): {5, 6, 7, 8, 9}

1. Common elements in both sequences: [5]
2. Elements in exactly one sequence: [1, 2, 3, 4, 6, 7, 8, 9]
```

This program effectively demonstrates how to handle sequences of integers and leverage Python's `set` data structure for efficient operations like finding intersections and symmetric differences. Here's a breakdown:

1. **`get_integer_sequence(prompt)` function**:
   - This helper function is designed to robustly take space-separated integer input from the user.
   - It uses a `while True` loop and `try-except ValueError` to ensure that only valid integers are accepted. If the user enters non-integer text, it prompts them again.
   - `input_str.split()`: Splits the input string into a list of strings based on whitespace.
   - `[int(x) for x in input_str.split() if x.strip()]`: This is a list comprehension that iterates through the split parts, `strip()`s any leading/trailing whitespace (to handle extra spaces), and converts each valid part to an integer. This creates the initial list of integers.
2. **Getting Sequences from User**:
   - The program calls `get_integer_sequence` twice to get `sequence1_list` and `sequence2_list`.
3. **Converting to Sets**:

- `set1 = set(sequence1_list)` and `set2 = set(sequence2_list)`: The lists are converted into `set` objects. Sets automatically discard duplicate elements and are optimized for set operations like intersection and difference.

4. **Printing Intermediate Information**:
   - The original lists and the sets (unique elements from each sequence) are printed for clarity.
5. **Finding Common Elements (Intersection)**:
   - `set1.intersection(set2)`: This method returns a new set containing only the elements that are present in *both* `set1` and `set2`.
   - `list(...)` and `sorted(...)`: The resulting set of common elements is converted back into a list and then sorted to meet the requirement of printing in sorted order without duplicates.
6. **Finding Symmetric Difference**:
   - `set1.symmetric_difference(set2)`: This method returns a new set containing all elements that are in *either* `set1` *or* `set2` but *not in both*.
   - `list(...)` and `sorted(...)`: Similar to the intersection, this result is converted to a list and sorted before printing.

11. Programming Exercises with classes and objects, Inheritance

a. Write a program with class Employee that keeps a track of the number of employees in an organization and also stores their name, designation and salary details.

```
class Employee:
    """
    A class to represent an Employee in an organization.
    It keeps track of the total number of employees and individual employee details.
    """
    # Class variable to keep track of the number of employees
    employee_count = 0

    def __init__(self, name, designation, salary):
        self.name = name
        self.designation = designation
        self.salary = salary
        Employee.employee_count += 1 # Increment the count each time a new employee is created
        print(f"Employee {self.name} added. Total employees: {Employee.employee_count}")

    def display_employee_details(self):
        """Prints the details of the individual employee."""
        print(f"\n--- Employee Details ---")
        print(f"Name: {self.name}")
        print(f"Designation: {self.designation}")
        print(f"Salary: ${self.salary:,.2f}")
        print(f"------------------------")

    @classmethod
    def display_total_employees(cls):
        """Prints the total number of employees in the organization."""
        print(f"\nTotal Number of Employees: {cls.employee_count}")

# --- Example Usage ---

# Create some employee objects
emp1 = Employee("Alice Smith", "Software Engineer", 75000)
emp2 = Employee("Bob Johnson", "Project Manager", 90000)
emp3 = Employee("Charlie Brown", "UX Designer", 65000)

# Display individual employee details
emp1.display_employee_details()
emp2.display_employee_details()

# Display the total number of employees using the class method
Employee.display_total_employees()

# Create another employee
emp4 = Employee("Diana Prince", "Data Scientist", 88000)

# Display the updated total number of employees
Employee.display_total_employees()
```

```
Employee Alice Smith added. Total employees: 1
Employee Bob Johnson added. Total employees: 2
Employee Charlie Brown added. Total employees: 3
```

```
--- Employee Details ---
Name: Alice Smith
Designation: Software Engineer
Salary: $75,000.00
------------------------

--- Employee Details ---
Name: Bob Johnson
Designation: Project Manager
Salary: $90,000.00
------------------------

Total Number of Employees: 3
Employee Diana Prince added. Total employees: 4

Total Number of Employees: 4
```

This Python program defines a `Employee` class to manage employee data and track the total number of employees. Here's a breakdown:

`Employee` Class:

1. **`employee_count = 0` (Class Variable):**
   - This is a class-level variable, meaning it belongs to the class itself, not to any specific employee object. All instances of `Employee` share this single `employee_count`.
   - It's initialized to `0` and is used to keep a running tally of how many `Employee` objects have been created.
2. **`__init__(self, name, designation, salary)` (Constructor):**
   - This is a special method that gets called automatically whenever a new `Employee` object is created (e.g., `emp1 = Employee(...)`).
   - `self.name`, `self.designation`, `self.salary`: These are instance variables. Each employee object will have its own `name`, `designation`, and `salary`.
   - `Employee.employee_count += 1`: Crucially, within the constructor, the `employee_count` class variable is incremented. This ensures that every time a new employee is instantiated, the total count goes up.
3. **`display_employee_details(self)` (Instance Method):**
   - This method is called on an *individual employee object* (e.g., `emp1.display_employee_details()`).
   - It prints the `name`, `designation`, and `salary` specific to that particular employee instance.
4. **`@classmethod` `display_total_employees(cls)` (Class Method):**
   - The `@classmethod` decorator indicates that `display_total_employees` is a class method. This means it operates on the class itself, not an instance.
   - `cls` is conventionally used as the first parameter for class methods (similar to `self` for instance methods) and refers to the class (`Employee` in this case).
   - `cls.employee_count`: This method accesses and prints the `employee_count` class variable, providing the total number of employees.

Example Usage:

- The program first creates three `Employee` objects (`emp1`, `emp2`, `emp3`). Each creation automatically increments `employee_count`.
- It then demonstrates calling `display_employee_details()` on individual employees.
- `Employee.display_total_employees()` is called to show the current total number of employees.
- Another employee (`emp4`) is created, and `Employee.display_total_employees()` is called again to show that the count has been updated.

b. Write a program that has classes such as Student, Course, and Department.
• Enroll a student in a course of a particular department.
• Student Class (name, rollno, course, year)
• Course Class (name, year)
• Department Class (name)

```
class Department:
    def __init__(self, name):
        self.name = name

    def display_info(self):
        return f"Department: {self.name}"
```

```
class Course:
    def __init__(self, name, year):
        self.name = name
        self.year = year

    def display_info(self):
        return f"Course: {self.name} ({self.year})"

class Student:
    def __init__(self, name, rollno):
        self.name = name
        self.rollno = rollno
        self.enrolled_course = None  # Will store a Course object
        self.enrolled_department = None # Will store a Department object

    def enroll_in_course(self, course_obj, department_obj):
        if isinstance(course_obj, Course) and isinstance(department_obj, Department):
            self.enrolled_course = course_obj
            self.enrolled_department = department_obj
            print(f"{self.name} (Roll No: {self.rollno}) enrolled in {course_obj.name} of {department_obj.name}.")
        else:
            print("Error: Invalid course or department object provided for enrollment.")

    def display_info(self):
        print(f"\n--- Student Information ---")
        print(f"Name: {self.name}")
        print(f"Roll No: {self.rollno}")
        if self.enrolled_course and self.enrolled_department:
            print(f"Enrolled In: {self.enrolled_course.display_info()}")
            print(f"Under: {self.enrolled_department.display_info()}")
        else:
            print("Enrollment Status: Not currently enrolled in any course.")
        print(f"---------------------------")

# --- Example Usage ---

# 1. Create a Department
computer_science = Department("Computer Science")
print(f"Created {computer_science.display_info()}")

# 2. Create a Course
python_programming = Course("Python Programming", "2023-2024")
print(f"Created {python_programming.display_info()}")

# 3. Create a Student
alice = Student("Alice Johnson", "CS001")
print(f"Created Student: {alice.name}, Roll No: {alice.rollno}")

# 4. Enroll the student in the course of the particular department
alice.enroll_in_course(python_programming, computer_science)

# 5. Display the student's complete information
alice.display_info()

print("\n--- Another Student Example ---")
bob = Student("Bob Williams", "CS002")
bob.display_info() # Not yet enrolled

math_department = Department("Mathematics")
calc_course = Course("Calculus I", "2023-2024")

bob.enroll_in_course(calc_course, math_department)
bob.display_info()
```

```
Created Department: Computer Science
Created Course: Python Programming (2023-2024)
Created Student: Alice Johnson, Roll No: CS001
Alice Johnson (Roll No: CS001) enrolled in Python Programming of Computer Science.

--- Student Information ---
Name: Alice Johnson
Roll No: CS001
Enrolled In: Course: Python Programming (2023-2024)
Under: Department: Computer Science
---------------------------

--- Another Student Example ---

--- Student Information ---
Name: Bob Williams
Roll No: CS002
Enrollment Status: Not currently enrolled in any course.
---------------------------
```

```
Bob Williams (Roll No: CS002) enrolled in Calculus I of Mathematics.

--- Student Information ---
Name: Bob Williams
Roll No: CS002
Enrolled In: Course: Calculus I (2023-2024)
Under: Department: Mathematics
--------------------------
```

This Python program defines three classes: `Department`, `Course`, and `Student`, to model a basic educational enrollment system. It demonstrates how these classes can interact to enroll a student in a course that belongs to a specific department. Let's break down each class and their usage:

1. `Department` Class:

- **Purpose:** Represents an academic department.
- **`__init__(self, name)` (Constructor):**
  - Initializes a `Department` object with a `name` attribute (e.g., "Computer Science").
- **`display_info(self)`:**
  - Returns a formatted string containing the department's name.

2. `Course` Class:

- **Purpose:** Represents an academic course.
- **`__init__(self, name, year)` (Constructor):**
  - Initializes a `Course` object with a `name` (e.g., "Python Programming") and a `year` (e.g., "2023-2024").
- **`display_info(self)`:**
  - Returns a formatted string containing the course's name and year.

3. `Student` Class:

- **Purpose:** Represents a student in the system.
- **`__init__(self, name, rollno)` (Constructor):**
  - Initializes a `Student` object with a `name` and `rollno`.
  - It also initializes `enrolled_course` and `enrolled_department` to `None`, as a student is not enrolled in a course or department initially.
- **`enroll_in_course(self, course_obj, department_obj)`:**
  - This method allows a student to enroll in a course. It takes a `Course` object and a `Department` object as arguments.
  - It includes a type check (`isinstance`) to ensure that valid `Course` and `Department` objects are passed.
  - If valid, it assigns the `course_obj` to `self.enrolled_course` and `department_obj` to `self.enrolled_department`.
  - It then prints a confirmation message.
- **`display_info(self)`:**
  - Prints a comprehensive summary of the student's information.
  - If the student is enrolled (`self.enrolled_course` and `self.enrolled_department` are not `None`), it also displays the details of the enrolled course and department by calling their respective `display_info` methods.

Example Usage:

The example section demonstrates the interaction of these classes:

1. **Department and Course Creation:**
   - `computer_science = Department("Computer Science")` creates a department.
   - `python_programming = Course("Python Programming", "2023-2024")` creates a course.
2. **Student Creation and Enrollment (Alice):**
   - `alice = Student("Alice Johnson", "CS001")` creates a student.
   - `alice.enroll_in_course(python_programming, computer_science)` enrolls Alice in the Python course within the Computer Science department.
   - `alice.display_info()` then prints Alice's complete information, including her enrollment details.
3. **Another Student Example (Bob):**
   - `bob = Student("Bob Williams", "CS002")` creates another student.
   - `bob.display_info()` is called before enrollment to show his initial status (not enrolled).

- New `math_department` and `calc_course` objects are created.
- `bob.enroll_in_course(calc_course, math_department)` enrolls Bob.
- His `display_info()` is called again to show his updated enrollment status.

This structure allows for a clear and organized way to manage related data using object-oriented programming principles.

12. Operator Overloading, Error Handling

a. Write a program that overloads the + operator on a class Student that has attributes name and marks.

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks

    def __str__(self):
        return f"Student(Name: {self.name}, Marks: {self.marks})"

    def __add__(self, other):
        if isinstance(other, Student):
            # Combine names and sum marks
            combined_name = f"{self.name} & {other.name}"
            total_marks = self.marks + other.marks
            return Student(combined_name, total_marks)
        else:
            # Handle cases where `other` is not a Student object
            raise TypeError("Unsupported operand type for +: 'Student' and " + type(other).__name__)

# --- Example Usage ---
student1 = Student("Alice", 85)
student2 = Student("Bob", 90)
student3 = Student("Charlie", 75)

print(f"Student 1: {student1}")
print(f"Student 2: {student2}")
print(f"Student 3: {student3}")

# Overloading the '+' operator
combined_students = student1 + student2
print(f"\nCombined Students (Alice + Bob): {combined_students}")

# Another combination
all_three_students = combined_students + student3
print(f"All three Students (Alice+Bob + Charlie): {all_three_students}")

# Demonstrating type error
try:
    invalid_addition = student1 + 10
except TypeError as e:
    print(f"\nError: {e}")
```

```
Student 1: Student(Name: Alice, Marks: 85)
Student 2: Student(Name: Bob, Marks: 90)
Student 3: Student(Name: Charlie, Marks: 75)

Combined Students (Alice + Bob): Student(Name: Alice & Bob, Marks: 175)
All three Students (Alice+Bob + Charlie): Student(Name: Alice & Bob & Charlie, Marks: 250)

Error: Unsupported operand type for +: 'Student' and int
```

This Python program defines a `Student` class and demonstrates how to **overload the `+` operator** for instances of this class. Operator overloading allows you to define custom behavior for operators (like `+`, `-`, `*`, etc.) when they are used with objects of your own classes.

`Student` Class:

1. **`__init__(self, name, marks)` (Constructor):**
   - This is the constructor method, called when a new `Student` object is created (e.g., `Student("Alice", 85)`).
   - It initializes two instance attributes: `self.name` (a string for the student's name) and `self.marks` (an integer representing their marks).
2. **`__str__(self)` (String Representation Method):**

- This special method defines how an `Student` object should be represented as a string when `print()` is called on it, or when `str()` is used.
- It returns a user-friendly string like `"Student(Name: Alice, Marks: 85)"`.

3. `__add__(self, other)` **(Operator Overloading for `+`):**
   - This is the core of the operator overloading. Python calls this method whenever the `+` operator is used between a `Student` object (`self`) and another object (`other`).
   - `if isinstance(other, Student):`: It first checks if the `other` operand is also an instance of the `Student` class. This is important to ensure that the `+` operation only makes sense between two `Student` objects.
     - If `other` is a `Student`, it performs the custom addition logic:
       - `combined_name = f"{self.name} & {other.name}"`: It concatenates the names of the two students with an " & " separator.
       - `total_marks = self.marks + other.marks`: It sums their marks.
       - `return Student(combined_name, total_marks)`: It returns a *new* `Student` object with the combined name and total marks.
   - `else:`: If `other` is *not* a `Student` object (e.g., an integer or a string), it raises a `TypeError`.
     - `raise TypeError(...)`: This ensures that invalid `+` operations (like `student1 + 10`) are caught and a descriptive error message is provided, making the class more robust.

Example Usage:

1. **Creating Students:** Three `Student` objects (`student1`, `student2`, `student3`) are created and their initial `__str__` representations are printed.
2. **Overloading the `+` Operator:**
   - `combined_students = student1 + student2`: This line triggers the `__add__` method. It creates a new `Student` object named `combined_students` with the name "Alice & Bob" and marks 175.
   - `all_three_students = combined_students + student3`: This further demonstrates chaining the `+` operator. The `combined_students` object is added to `student3`, resulting in a new student with "Alice & Bob & Charlie" and total marks of 250.
3. **Demonstrating Type Error Handling:**
   - `try...except TypeError`: This block attempts an invalid operation: `student1 + 10`.
   - Because the `__add__` method explicitly checks the type of `other`, it raises a `TypeError` when an `int` is encountered. The `except` block catches this error and prints the custom error message, showing how the class prevents nonsensical operations.

b. Write a program that validates name and age as entered by the user to determine whether the person can cast vote or not.

```
def get_valid_name():
    while True:
        name = input("Enter your name: ").strip()
        if name:
            return name
        else:
            print("Name cannot be empty. Please enter a valid name.")

def get_valid_age():
    while True:
        try:
            age = int(input("Enter your age: "))
            if 0 < age < 150: # Assuming a realistic age range
                return age
            else:
                print("Please enter a realistic age between 1 and 149.")
        except ValueError:
            print("Invalid input. Please enter a number for age.")

# Get validated input from the user
person_name = get_valid_name()
person_age = get_valid_age()

# Determine voting eligibility
voting_age_threshold = 18

print(f"\nHello, {person_name}!")
print(f"You are {person_age} years old.")

if person_age >= voting_age_threshold:
```

```
    print("Congratulations! You are eligible to cast your vote.")
else:
    years_to_wait = voting_age_threshold - person_age
    print(f"Sorry, you are not yet eligible to vote. You need to wait {years_to_wait} more year(s).")
```

```
Enter your name: Lucky
Enter your age: 23

Hello, Lucky!
You are 23 years old.
Congratulations! You are eligible to cast your vote.
```

This Python program takes a name and an age as input from the user, validates these inputs, and then determines if the person is eligible to vote based on a predefined age threshold. Let's break down its components:

1. `get_valid_name()` **Function:**
   - **Purpose:** This function is responsible for prompting the user to enter their name and ensuring that the name is not empty.
   - `while True:`: It uses an infinite loop to repeatedly ask for input until a valid name is provided.
   - `name = input("Enter your name: ").strip()`: Prompts the user and stores their input. `.strip()` removes any leading or trailing whitespace (like spaces or newlines).
   - `if name:`: Checks if the `name` string is not empty after stripping whitespace. An empty string evaluates to `False` in a boolean context.
   - `return name`: If the name is not empty, it's considered valid and returned, breaking the loop.
   - `else: print(...)`: If the name is empty, an error message is printed, and the loop continues.
2. `get_valid_age()` **Function:**
   - **Purpose:** This function prompts the user for their age and validates that it's a realistic number within a certain range (1 to 149).
   - `while True:`: An infinite loop ensures valid age input.
   - `try...except ValueError:`: This block is used for error handling. The `input()` function returns a string, and `int()` tries to convert it to an integer. If the user enters something that cannot be converted (e.g., text), a `ValueError` occurs.
     - `age = int(input("Enter your age: "))`: Prompts for age and attempts to convert it to an integer.
     - `if 0 < age < 150:`: Checks if the entered age is a positive number and within a plausible range (e.g., 1 to 149 years). This is a simple form of logical validation.
     - `return age`: If the age is valid, it's returned, and the loop breaks.
     - `else: print(...)`: If the age is outside the realistic range, an error message is printed.
     - `except ValueError: print(...)`: If `int()` conversion fails, an error message for invalid input is printed.
3. **Main Program Logic:**
   - `person_name = get_valid_name()`: Calls the function to get a validated name from the user.
   - `person_age = get_valid_age()`: Calls the function to get a validated age from the user.
   - `voting_age_threshold = 18`: Sets the minimum age required to vote.
   - **Displaying Information:** Prints a greeting with the user's name and confirms their age.
   - **Voting Eligibility Check:**
     - `if person_age >= voting_age_threshold:`: Checks if the person's age meets or exceeds the voting threshold.
     - If true, it prints a congratulatory message indicating eligibility.
     - `else:`: If the person is not old enough, it calculates how many years they need to wait (`years_to_wait = voting_age_threshold - person_age`) and prints a message indicating their ineligibility and the remaining waiting period.

This program effectively combines user input, validation, and conditional logic to provide a clear answer regarding voting eligibility.

c. Write a program that asks the user to enter an integer. If the user enters something that is not an integer (like "abc" or 3.5), the program should catch the error, print "Invalid input, please enter an integer.", and ask again until a valid integer is entered.

```
# Program to ask for integer input until a valid one is entered

while True:
    try:
        user_input = input("Enter an integer: ")
        number = int(user_input)
        print(f"You entered a valid integer: {number}")
        break # Exit the loop if input is a valid integer
    except ValueError:
        print("Invalid input, please enter an integer.")
```

```
Enter an integer: 69
You entered a valid integer: 69
```

This Python program continuously prompts the user to enter an integer until a valid integer input is received. It effectively uses error handling to manage cases where the user inputs non-integer values.

Here's a breakdown of its components:

1. `while True:` **(Infinite Loop)**:
   - This sets up an infinite loop, meaning the code inside it will keep executing repeatedly until a `break` statement is encountered.
   - This ensures that the program will continue to ask for input until a valid integer is provided.
2. `try-except ValueError:` **(Error Handling)**:
   - The `try` block contains the code that might potentially raise an error.
     - `user_input = input("Enter an integer: ")`: This line prompts the user to enter a value, which is always read as a string.
     - `number = int(user_input)`: This is the critical line. It attempts to convert the `user_input` string into an integer. If the string cannot be successfully converted (e.g., if the user types "abc" or "3.5"), a `ValueError` will be raised.
     - `print(f"You entered a valid integer: {number}")`: If the conversion to an integer is successful (no `ValueError` occurred), this line prints the valid integer.
     - `break`: This statement immediately exits the `while True` loop, as a valid integer has been successfully obtained.
   - The `except ValueError:` block is executed *only if* a `ValueError` occurs within the `try` block.
     - `print("Invalid input, please enter an integer.")`: If a `ValueError` is caught (meaning the user's input was not a valid integer), this informative error message is printed.
     - After printing the error message, the loop continues (`while True:`), prompting the user for input again.