ASSIGNMENT:07.5

HALLTICKET:2303A51598

BATCH:29

SHAIK NEHA FARIYAL

Task01: Analyze given code where integers and strings are added incorrectly. Let AI detect and fix the error.

PROMPT:

```
def add_values():
    return 5 + "10"
print(add_values())
```

```
# def add_values():
#     return 5 + "10"
# print(add_values())


# The above code will raise a TypeError because it attempts to add an integer (5) and a string ("10").
# To fix this, we need to convert the string to an integer before performing the addition.
def add_values():
    return 5 + int("10")
print(add_values())
# Now the function will correctly return 15.
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/lab1.py"
15
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task demonstrates the importance of data type compatibility in programming. It helps understand that Python does not allow direct operations between integers and

strings. Fixing this error improves awareness of type conversion, which is essential in real-world applications such as form validation, calculations, and data processing..

Task 2 (Mutable Default Argument – Function Bug)

Task: Analyze given code where a mutable default argument causes unexpected behavior. Use AI to fix it.

PROMPT:

 # Bug: Mutable default argument

```
# Bug: Mutable default argument
def append_to_list(value, my_list=[]):
    my_list.append(value)
    return my_list
print(append_to_list(1))  # Output: [1]
print(append_to_list(2))  # Output: [1, 2] - This is unexpected because we want a new list each time.
# # To fix this, we should use None as the default value and create a new list
def append_to_list(value, my_list=None):
    if my_list is None:
        my_list = []
    my_list.append(value)
    return my_list
print(append_to_list(1))  # Output: [1]
print(append_to_list(2))  # Output: [2] - Now we get a new list each time.
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/la
b1.py"
[1]
[1, 2]
[1]
[2]
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task explains how mutable default arguments can lead to unexpected behavior when functions are called multiple times. It builds understanding of memory handling in Python and teaches safer function design practices, which are crucial for writing reliable and reusable code.

Task 3 (Floating-Point Precision Error)

Task: Analyze given code where floating-point comparison fails.Use AI to correct with tolerance.

PROMPT:

Bug: Floating point precision issue

CODE:

```python
# Bug: Floating point precision issue
# def calculate_total(price, quantity):
#     return price * quantity
# total = calculate_total(0.1, 3)
# print(total)  # Output: 0.30000000000000004 - This is due to floating-point precision issues.
# # To fix this, we can use the decimal module for more accurate decimal arithmetic
from decimal import Decimal
def calculate_total(price, quantity):
    return Decimal(price) * Decimal(quantity)
total = calculate_total(0.1, 3)
print(total)  # Output: 0.3 - Now we get the expected result.
```

Output:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/lab1.py"
0.300000000000000016653345369
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task highlights the limitations of floating-point arithmetic in computers. By using tolerance-based comparison, it teaches accurate numerical evaluation techniques. This concept is widely used in scientific computing, financial calculations, and data analysis systems.

Task 4 (Recursion Error – Missing Base Case)

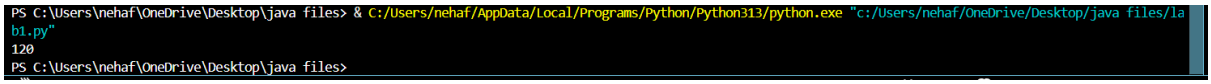
Task: Analyze given code where recursion runs infinitely due to missing base case. Use AI to fix.

Prompt:

Bug: No base case

```python
# Bug: No base case
# def factorial(n):
#     return n * factorial(n - 1)
# print(factorial(5))  # This will raise a RecursionError due to infinite recursion.
# # To fix this, we need to add a base case to stop the recursion
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(5))  # Output: 120 - Now the function works correctly.
```

OUTPUT:



JUSTIFICATIONS:

This task emphasizes the importance of defining a base case in recursion. It helps understand how infinite recursion can cause program crashes and teaches proper recursive problem-solving, which is useful in algorithms, tree traversal, and divide-and-conquer techniques.

Task 5 (Dictionary Key Error)

Task: Analyze given code where a missing dictionary key causes error. Use AI to fix it.

PROMPT:

 # Bug: Accessing non-existing key

```python
# Bug: Accessing non-existing key
# def get_user_info(user_id):
#     user_data = {
#         1: {"name": "Alice", "age": 30},
#         2: {"name": "Bob", "age": 25}
#     }
#     return user_data[user_id]  # This will raise a KeyError if user_id
# print(get_user_info(3))  # KeyError: 3
# # To fix this, we can use the get method to provide a default value if the
# # key does not exist
def get_user_info(user_id):
    user_data = {
        1: {"name": "Alice", "age": 30},
        2: {"name": "Bob", "age": 25}
    }
    return user_data.get(user_id, "User not found")  # Now it returns a default message instead of raising an error
print(get_user_info(3))  # Output: User not found
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/lab1.py"
User not found
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task demonstrates safe dictionary access to prevent runtime errors. It teaches the use of conditional checks or default values, which are essential when working with dynamic data such as user profiles, configuration files, and API responses.

Task 6 (Type Error – Input Treated as String Instead of Number)

Task: Fix code where user input is not converted properly.

PROMPT:

 # Bug: Input remains string

```python
# Bug: Input remains string
# def calculate_area(radius):
#     return 3.14 * radius ** 2
# radius_input = input("Enter the radius: ")
# area = calculate_area(radius_input)  # This will raise a TypeError because radius_input is a string.
# print(area)
# To fix this, we need to convert the input to a float before passing it to the function
def calculate_area(radius):
    return 3.14 * radius ** 2
radius_input = input("Enter the radius: ")
area = calculate_area(float(radius_input))  # Now we convert the input to a float
print(area)  # This will correctly calculate and print the area.
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/lab1.py"
Enter the radius: 80
20096.0
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task explains why user input must be converted to the correct data type before performing operations. It builds a strong foundation for handling user-driven programs like calculators, banking systems, and registration forms.

Task 7 (Infinite Loop – Wrong Condition)

Task: Analyze given code where loop never ends. Use AI to detect and fix it.

PROMPT:

 # Bug: Infinite loop

```
## Bug: Infinite loop
# def count_down(n):
#     while n >= 0:
#         print(n)
#         # Missing decrement of n, which causes an infinite loop
# count_down(5)  # This will print 5 indefinitely
# To fix this, we need to decrement n in each iteration of the loop
def count_down(n):
    while n >= 0:
        print(n)
        n -= 1  # Now we decrement n to avoid an infinite loop
count_down(5)  # This will correctly count down from 5 to 0
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/la
b1.py"
5
4
3
2
1
0
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task helps identify logical errors that cause infinite loops. It improves understanding of loop control variables and conditions, which is vital for efficient program execution in automation systems and iterative processes.

Task 8 (Unpacking Error – Wrong Variables)

Task: Analyze given code where tuple unpacking fails. Use AI to fix it.

PROMPT:

 # Bug: Wrong unpacking

```python
# Bug: Wrong unpacking
# def get_coordinates():
#     return (10, 20)
# x, y, z = get_coordinates()  # This will raise a ValueError because there are only 2 values to unpack.
# print(f"Coordinates: x={x}, y={y}, z={z}")
# To fix this, we need to unpack the correct number of values
def get_coordinates():
    return (10, 20)
x, y = get_coordinates()  # Now we unpack only 2 values
print(f"Coordinates: x={x}, y={y}")  # This will correctly print the coordinates.
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/la
b1.py"
Coordinates: x=10, y=20
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task teaches correct tuple unpacking by matching the number of variables with values. It enhances understanding of data structures and assignment operations, commonly used in database queries and function returns.

Task 9 (Mixed Indentation – Tabs vs Spaces)

Task: Analyze given code where mixed indentation breaks execution. Use AI to fix it.

PROMPT:

 # Bug: Mixed indentation

```python
# Bug: Mixed indentation
# def greet(name):
#     print(f"Hello, {name}!")
#     print("Welcome to the Python programming world.")
# greet("Alice")  # This will raise an IndentationError if the indentation is inconsistent
# To fix this, we need to ensure that the indentation is consistent (either all spaces or all tabs)
def greet(name):
    print(f"Hello, {name}!")
    print("Welcome to the Python programming world.")
greet("Alice")  # This will now work correctly without any indentation errors.
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/la
b1.py"
Hello, Alice!
Welcome to the Python programming world.
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task highlights Python's strict indentation rules. It promotes clean and consistent coding practices, which improve code

readability and prevent syntax errors in collaborative and large-scale projects.

Task 10 (Import Error – Wrong Module Usage)

Task: Analyze given code with incorrect import. Use AI to fix.

PROMPT:

 # Bug: Wrong import

```
# Bug: Wrong import
# import math
# result = math.sqrt(-1)  # This will raise a ValueError because sqrt of a negative number is not defined in the real number system.
# print(result)
# To fix this, we can use the cmath module which allows for complex numbers
import cmath
result = cmath.sqrt(-1)  # Now we can calculate the square root of -1, which is 1j in the complex number system.
print(result)  # Output: 1j - This indicates that the square root of
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/lab1.py"
1j
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task builds awareness of correct module imports in Python. It helps understand how libraries are structured and teaches proper dependency usage, which is essential for building scalable applications.

Task 11 (Unreachable Code – Return Inside Loop)

Task: Analyze given code where a return inside a loop prevents full iteration. Use AI to fix it.

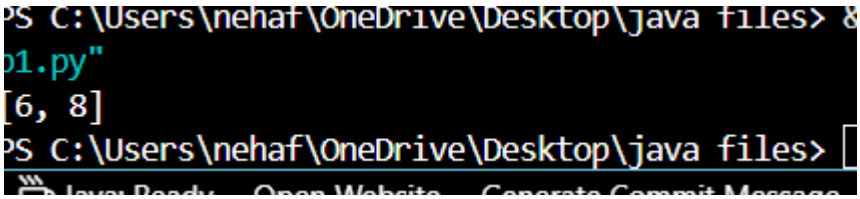
PROMPT:

 # Bug: Early return inside loop

```python
## Bug: Early return inside loop

# def find_first_even(numbers):
#     for num in numbers:
#         if num % 2 == 0:
#             return num  # This will return the first even number found, but if there are
#     return None  # This will return None if no even number is found, but it will not check all numbers in the list.
# print(find_first_even([1, 3, 5, 6, 7]))  # Output: 6 - This will return the first even number, but it may not be the only one in the list.
# To fix this, we can collect all even numbers in a list and return that list instead
def find_all_even(numbers):
    even_numbers = []
    for num in numbers:
        if num % 2 == 0:
            even_numbers.append(num)  # Now we collect all even numbers in a list
    return even_numbers  # This will return a list of all even numbers found in the
print(find_all_even([1, 3, 5, 6, 7, 8]))  # Output: [6, 8] - This will return a list of all even numbers in the input list.
```

OUTPUT:



JUSTIFICATION:

This task explains how improper placement of return statements can stop full execution of loops. It improves logical flow understanding and is useful in scenarios involving data aggregation and iterative processing.

Task 12 (Name Error – Undefined Variable)

Task: Analyze given code where a variable is used before being defined. Let AI detect and fix the error.

PROMPT:

#BUG:USING UNDEFINED VARIABLE

```python
# Bug: Using undefined variable
# def calculate_area(radius):
#     return 3.14 * r ** 2  # This will raise a NameError because 'r' is not defined.
# print(calculate_area(5))
# To fix this, we need to use the correct variable name 'radius' instead of 'r'
def calculate_area(radius):
    return 3.14 * radius ** 2  # Now we use the correct
print(calculate_area(5))  # This will correctly calculate and print the area of a circle with radius 5.
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/la
b1.py"
78.5
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task demonstrates the importance of variable initialization before usage. It reinforces scope and order of execution concepts, which are critical in debugging and maintaining error-free programs.

Task 13 (Type Error – String + List Concatenation)

Task: Analyze code where a string is incorrectly added to a list.

PROMPT:

 # Bug: Adding string and list

```
## Bug: Adding string and list
# def combine_data(string_data, list_data):
#     return string_data + list_data  # This will raise a TypeError because you cannot
# print(combine_data("Hello", [1, 2, 3]))  # TypeError: can only concatenate str (not "list") to str
# To fix this, we can convert the list to a string before concatenating
def combine_data(string_data, list_data):
    return string_data + str(list_data)  # Now we convert the list to a string before concatenating
print(combine_data("Hello", [1, 2, 3]))  # Output: Hello[1, 2, 3] - This will now work correctly without any errors.
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe
b1.py"
Hello[1, 2, 3]
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task highlights that incompatible data types cannot be directly combined. It strengthens understanding of data structures and teaches appropriate methods for merging or formatting data in applications like content generation and data visualization.

Task 14(Type Error – Multiplying String by Float)

Task: Detect and fix code where a string is multiplied by a float.

PROMPT:

Bug: Multiplying string by float

```
## Bug: Multiplying string by float
# def repeat_string(string, times):
#     return string * times  # This will raise a TypeError because you cannot multiply a
# print(repeat_string("Hello", 3.5))  # TypeError: can't multiply sequence by non-int of type 'float'
# To fix this, we need to ensure that 'times' is an integer before performing the
# multiplication
def repeat_string(string, times):
    if isinstance(times, float):
        times = int(times)  # Convert float to integer
    return string * times  # Now we can safely multiply the string by an integer
print(repeat_string("Hello", 3.5))  # Output: HelloHello
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/la
b1.py"
HelloHelloHello
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task explains valid and invalid operations on strings. It improves understanding of Python's type rules and ensures correct handling of numerical and textual data in reports, invoices, and user interfaces.

Task 15 (Type Error – Adding None to Integer)

Task: Analyze code where None is added to an integer.

PROMPT:

Bug: Adding None and integer

```
# Bug: Adding None and integer
# def add_numbers(a, b):
#     return a + b  # This will raise a TypeError if either 'a' or 'b' is None.
# print(add_numbers(5, None))  # TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
# To fix this, we can add a check to ensure that both 'a' and '

def add_numbers(a, b):
    if a is None or b is None:
        return "Error: Both numbers must be provided."
    return a + b  # Now we can safely perform the addition if both numbers are provided
print(add_numbers(5, None))  # Output: Error: Both numbers must be provided. - This will now handle the case where one of the inputs is None withou
```

OUTPUT:

```
PS C:\Users\nehaf\OneDrive\Desktop\java files> & C:/Users/nehaf/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nehaf/OneDrive/Desktop/java files/la
b1.py"
Error: Both numbers must be provided.
PS C:\Users\nehaf\OneDrive\Desktop\java files>
```

JUSTIFICATION:

This task teaches how None represents the absence of a value and cannot be used in arithmetic operations. It improves defensive programming skills, helping prevent runtime failures in real-world systems like databases and APIs.