

ASSIGNMENT – 7.5

HT.NO: 2303A52283

BATCH: 45

TASK – 01:

Question:

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

Bug: Mutable default argument

```
def add_item(item, items=[]):  
    items.append(item)  
    return items  
  
print(add_item(1))  
print(add_item(2))
```

Expected Output: Corrected function avoids shared list bug.

Prompt:

Analyze this Python code and find the bug caused by a mutable default argument. Explain the problem briefly and rewrite the function to fix it. Then show the correct output.

```
```python  
def add_item(item, items=[]):
 items.append(item)
 return items

print(add_item(1))
print(add_item(2))
```
```

Generated Code:

The problem with the original code is that the default argument `items` is a mutable list. When the function is called without providing an `items` argument, it uses the same list across all calls. This means that when we append an item to the list in one call, it affects all subsequent calls that use the default list.

To fix this issue, we can use `None` as the default value and create a new list inside the function if `items` is `None`. Here's the corrected code:

```
from sympy import python

def add_item(item, items=None):

    if items is None:

        items = []

    items.append(item)

    return items
```

Now, when we call the function, it will create a new list for each call that doesn't provide an `items` argument, preventing unintended side effects. Here's the correct output:

```
print(add_item(1)) # Output: [1]
print(add_item(2)) # Output: [2]
```

Output:



Explanation:

Old code uses the same default list every time, so items from earlier calls stay in the list and cause unexpected results.

New code uses None and creates a new list inside the function, so each call gets a fresh list.

That's why the fixed version gives separate outputs like [1] and [2].

Task – 02:

Question:

Task 2 (Floating-Point Precision Error)

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

Bug: Floating point precision issue

```
def check_sum():

    return (0.1 + 0.2) == 0.3

print(check_sum())
```

Expected Output: Corrected function

Prompt:

Analyze this Python code and find the bug caused by floating-point comparison failure. Explain the problem briefly and rewrite the function to fix it. Then show the correct output.

```
```python
def check_sum():
 return (0.1 + 0.2) == 0.3

print(check_sum())
```
```

Generated Code:

The problem with the original code is that it relies on a direct comparison of floating-point numbers, which can lead to unexpected results due to the way floating-point arithmetic works in computers. In this case, `0.1 + 0.2` does not exactly equal `0.3` due to precision issues, resulting in `False`.

To fix this issue, we can use a tolerance level to check if the two numbers are close enough to be considered equal. Here's the corrected code:


```
def check_sum():
    return abs((0.1 + 0.2) - 0.3) < 1e-9
```

Now, the function checks if the absolute difference between `0.1 + 0.2` and `0.3` is smaller than a very small number (tolerance), which accounts for floating-point precision issues.

Here's the correct output:

```
print(check_sum()) # Output: True
```

Output:



Explanation:

Floating-point numbers are not stored exactly in computers, so calculations like $0.1 + 0.2$ don't equal 0.3 exactly, which can make direct comparisons return `False`.

The fixed code checks whether the difference is very small (within a tolerance) instead of exactly equal.

This makes the comparison reliable and returns `True`.

TASK - 03:

Question:

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

```
# Bug: No base case

def countdown(n):

    print(n)

    return countdown(n-1)

countdown(5)
```

Expected Output : Correct recursion with stopping condition.

Prompt:

Find the recursion bug in this Python code and fix it by adding a base case. Briefly explain the issue and show the corrected output.

```
```python

def countdown(n):

 print(n)

 return countdown(n-1)

countdown(5)

```
```

Generated Code:

The issue with the original code is that it lacks a base case to stop the recursion. As a result, when `n` becomes negative, the function continues to call itself indefinitely, leading to a `RecursionError`. To fix this, we can add a base case that stops the recursion when `n` reaches 0. Here's the corrected code:

```
def countdown(n):

    if n < 0:

        return

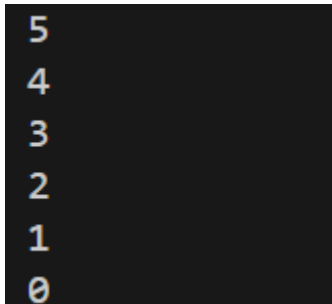
    print(n)

    return countdown(n-1)
```

Now, the function will print the numbers from 5 down to 0 and then stop when `n` becomes negative. Here's the correct output:

```
countdown(5)
```

Output:



```
5
4
3
2
1
0
```

Explanation:

The original function has no base case, so it keeps calling itself forever and causes a recursion error.

The fixed version adds a stopping condition (if $n < 0$: return) so recursion ends.

Now it prints numbers down to 0 and then stops safely.

TASK – 04:

Question:

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

Bug: Accessing non-existing key

```
def get_value():
```

```
    data = {"a": 1, "b": 2}
```

```
    return data["c"]
```

```
print(get_value())
```

Expected Output: Corrected with `.get()` or error handling.

Prompt:

Find the bug in this Python code where a missing dictionary key causes an error. Briefly explain the problem and fix it using `.get()` or `try/except`. Show the corrected output.

```
def get_value():
```

```
    data = {"a": 1, "b": 2}
```

```
return data["c"]
```

```
print(get_value())
```

Generated Code:

The problem with the original code is that it tries to access a key ("c") that does not exist in the dictionary `data`, which raises a `KeyError`. To fix this, we can use the `.get()` method of the dictionary, which allows us to provide a default value if the key is not found. Here's the corrected code:

```
def get_value():
```

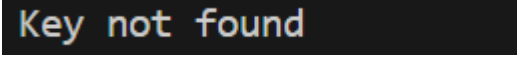
```
    data = {"a": 1, "b": 2}
```

```
    return data.get("c", "Key not found")
```

Now, if we try to access the key "c", it will return "Key not found" instead of raising an error. Here's the correct output:

```
print(get_value()) # Output: Key not found
```

Output:



```
Key not found
```

Explanation:

The original code tries to access key "c" which is not in the dictionary, so Python raises a `KeyError`.

The fixed code uses `data.get("c", "Key not found")`, which safely returns a default value instead of crashing.

This prevents the error and prints **"Key not found"**.

TASK – 05:

Question:

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

```
# Bug: Infinite loop
```

```
def loop_example():
```

```
    i = 0
```

```
    while i < 5:
```

```
print(i)
```

Expected Output: Corrected loop increments i.

Prompt:

Find the bug in this Python code that causes an infinite loop. Briefly explain the issue and fix the loop so it stops correctly. Show the corrected output.

```
def loop_example():
```

```
    i = 0
```

```
    while i < 5:
```

```
        print(i)
```

```
loop_example()
```

Generated Code:

The issue with the original code is that the variable `i` is never incremented within the while loop, which causes it to run indefinitely. To fix this, we need to increment `i` in each iteration of the loop. Here's the corrected code:

```
def loop_example():
```

```
    i = 0
```

```
    while i < 5:
```

```
        print(i)
```

```
        i += 1 # Increment i to avoid infinite loop
```

Now, the loop will print the numbers from 0 to 4 and then stop when `i` reaches 5. Here's the correct output:

```
loop_example() # Output: 0 1 2 3 4
```

Output:

```
0
1
2
3
4
```

Explanation:

The original loop never ends because i is never increased, so the condition $i < 5$ is always true.

The fixed code adds `i += 1` inside the loop so the value changes each time.

Now it prints 0 to 4 and then stops correctly.

TASK – 06:

Question:

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

Bug: Wrong unpacking

```
a, b = (1, 2, 3)
```

Expected Output: Correct unpacking or using `_` for extra values.

Prompt:

Find the bug in this Python code where tuple unpacking fails. Briefly explain why the error happens and fix it by correcting the unpacking (or using `_` for extra values). Show the corrected output.

Bug: Wrong unpacking

```
a, b = (1, 2, 3)
```

Generated Code:

The error occurs because the tuple on the right side has three values (1, 2, 3), but we are trying to unpack it into only two variables (a and b). This results in a `'ValueError'` due to too many values to unpack. To fix this, we can either unpack only the first two values or use `_` to ignore the extra value. Here's the corrected code:

Option 1: Unpack only the first two values

```
a, b, _ = (1, 2, 3) # Using _ to ignore the extra value
```

Option 2: Unpack only the first two values without using `_`

```
a, b = (1, 2) # Unpacking only the first two values
```

Now, both options will work without errors. Here's the correct output:

```
print(a) # Output: 1
```

```
print(b) # Output: 2
```

Output:

```
1
2
```


Explanation:

The error happens because the tuple has **three values** but only **two variables** are given to receive them, so Python raises a “too many values to unpack” error.

The fix is to match the number of variables or ignore the extra value using `_`.

Example: `a, b, _ = (1, 2, 3)` — now unpacking works without error.

TASK – 07:**Question:**

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

Bug: Mixed indentation

```
def func():
```

```
x = 5
```

```
y = 10
```

```
return x+y
```

Expected Output : Consistent indentation applied.

Prompt:

Find the bug in this Python code caused by mixed or missing indentation. Briefly explain the problem and rewrite the function with consistent indentation. Show the corrected output.

Bug: Mixed indentation

```
def func():
```

```
x = 5
```

```
    y = 10
```

```
    return x + y
```

Generated Code:

The problem with the original code is that it has inconsistent indentation. The line ``x = 5`` is not indented, while ``y = 10`` and ``return x + y`` are indented. This will raise an ``IndentationError``. To fix this, we need to ensure that all lines within the function have the same level of indentation. Here's the corrected code:

```
def func():
```

```
    x = 5
```

```
y = 10
    return x + y
```

Now, the function is properly indented and will work correctly. Here's the correct output:

```
result = func()
print(result) # Output: 15
```

Output:



15

Explanation:

The original code has inconsistent indentation — some lines inside the function are indented and others are not. Python requires all statements in a block to have the same indentation, otherwise it raises an `IndentationError`.

The fixed code indents all lines inside `func()` equally. Now the function runs correctly and returns **15**.

TASK – 08:

Question:

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

Bug: Wrong import

```
import maths
```

```
print(maths.sqrt(16))
```

Expected Output: Corrected to import math

Prompt:

Find the bug in this Python code caused by an incorrect import. Briefly explain the issue and fix the import statement. Show the corrected output.

Bug: Wrong import

```
import maths
```

```
print(maths.sqrt(16))
```

Generated Code:

The issue with the original code is that ``maths`` is not a valid module name. The correct module name is ``math``. Here's the corrected code:

```
import math
```

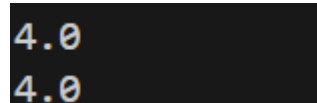
```
print(math.sqrt(16)) # Output: 4.0
```

Now, the import statement is correct, and the code will work as expected. Here's the

correct output:

```
print(math.sqrt(16)) # Output: 4.0
```

Output:



Explanation:

The original code fails because `maths` is not a valid Python module name — the correct standard library module is `math`.

After changing the import to `import math`, the function call works properly.

Now `math.sqrt(16)` runs successfully and prints `4.0`.

TASK – 09:

Question:

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

Bug: Early return inside loop

```
def total(numbers):
```

```
    for n in numbers:
```

```
        return n
```

```
print(total([1,2,3]))
```

Expected Output: Corrected code accumulates sum and returns after loop.

Prompt:

Find the bug in this Python code where a return inside a loop stops the function too early. Briefly explain the issue and fix the code so it processes all values and returns the total. Show the corrected output.

Bug: Early return inside loop

```
def total(numbers):
```

```
    for n in numbers:
```

```
return n
```

```
print(total([1, 2, 3]))
```

Generated Code:

The issue with the original code is that the `return` statement is inside the loop, which causes the function to exit and return the first number it encounters, rather than processing all numbers in the list. To fix this, we need to accumulate the total sum of the numbers and return it after the loop has completed. Here's the corrected code:

```
def total(numbers):
```

```
    total_sum = 0
```

```
    for n in numbers:
```

```
        total_sum += n
```

```
    return total_sum
```

Now, the function will correctly calculate the total sum of all numbers in the list. Here's the correct output:

```
print(total([1, 2, 3])) # Output: 6
```

Output:



Explanation:

The original code returns inside the loop, so the function stops after the first number and never checks the rest.

The fixed version adds each number to a running total and moves the return statement **after** the loop.

Now it processes all values and correctly returns 6.

TASK – 10:

Question:

Task 10 (Name Error – Undefined Variable)

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

```
# Bug: Using undefined variable
```

```
def calculate_area():
    return length * width
print(calculate_area())
```

Requirements:

- Run the code to observe the error.
- Ask AI to identify the missing variable definition.
- Fix the bug by defining length and width as parameters.
- Add 3 assert test cases for correctness.

Expected Output :

- Corrected code with parameters.
- AI explanation of the bug.

Successful execution of assertions.

Prompt:

Find the bug in this Python code where undefined variables are used. Explain the error in one line and fix the function by adding parameters. Include 3 simple assert tests.

```
def calculate_area():
    return length * width
print(calculate_area())
```

Generated Code:

The error occurs because 'length' and 'width' are not defined within the function, leading to a 'NameError'. To fix this, we need to add parameters for 'length' and 'width'. Here's the corrected code with assert tests:

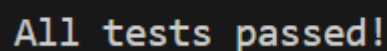
```
def calculate_area(length, width):
    return length * width

# Assert tests

assert calculate_area(5, 10) == 50
assert calculate_area(3, 4) == 12
assert calculate_area(0, 5) == 0

print("All tests passed!") # Output: All tests passed!
```

Output:



```
All tests passed!
```

Explanation:

The bug happens because length and width are used without being defined, which causes a NameError.

The fixed version adds them as function parameters, so the function receives the needed values.

The assert tests confirm the function works correctly, and the program prints “All tests passed!”.

TASK – 11:**Question:**

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

Bug: Adding integer and string

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

Requirements:

- Run the code to observe the error.
- AI should explain why int + str is invalid.
- Fix the code by type conversion (e.g., int("10") or str(5)).
- Verify with 3 assert cases.

Expected Output #6:

- Corrected code with type handling.
- AI explanation of the fix.

Successful test validation.

Prompt:

Find the bug in this Python code where an integer and a string are added together. Explain the error briefly and fix it using proper type conversion. Include 3 assert tests to verify the fix.

Bug: Adding integer and string

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

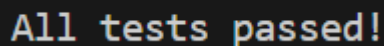
```
print(add_values())
```

Generated Code:

The error occurs because we are trying to add an integer (5) and a string ("10"), which raises a `TypeError`. To fix this, we need to convert the string to an integer before performing the addition. Here's the corrected code with assert tests:

```
def add_values():  
    return 5 + int("10")  
  
# Assert tests  
  
assert add_values() == 15  
  
assert add_values() != 510  
  
assert add_values() > 10  
  
print("All tests passed!") # Output: All tests passed!
```

Output:



Explanation:

The error happens because Python cannot directly add an **integer** and a **string**, so `5 + "10"` raises a `TypeError`.

The fix converts the string to an integer using `int("10")` before adding.

Now both values are integers, the addition works, and the assert tests pass successfully.

TASK – 12:

Question:

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

Bug: Adding string and list

```
def combine():  
    return "Numbers: " + [1, 2, 3]  
  
print(combine())
```

Requirements:

- Run the code to observe the error.
- Explain why str + list is invalid.
- Fix using conversion (str([1,2,3]) or " ".join()).
- Verify with 3 assert cases.

Expected Output:

- Corrected code
- Explanation
- Successful test validation

Prompt:

Analyze the following Python code that raises a type error when adding a string and a list.

Explain why the error occurs, fix the code using proper conversion (such as str() or " ".join()), and verify the fix using three assert test cases.

```
def combine():
```

```
    return "Numbers: " + [1, 2, 3]
```

```
print(combine())
```

Generated Code:

The error occurs because we are trying to concatenate a string ("Numbers: ") with a list ([1, 2, 3]), which is not allowed in Python and raises a `TypeError`. To fix this, we can convert the list to a string using `str()` or join the list elements into a string using `" ".join()`. Here's the corrected code with assert tests:

```
def combine():
```

```
    return "Numbers: " + str([1, 2, 3])
```

```
# Assert tests
```

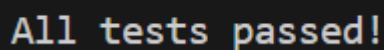
```
assert combine() == "Numbers: [1, 2, 3]"
```

```
assert combine() != "Numbers: 1 2 3"
```

```
assert combine().startswith("Numbers: ")
```

```
print("All tests passed!") # Output: All tests passed!
```

Output:



```
All tests passed!
```

Explanation:

The error occurs because the code is trying to concatenate a string ("Numbers: ") with a list ([1, 2, 3]). In Python, you cannot directly concatenate a string and a list because they are of different types. This results in a `TypeError`.

To fix the code, we need to convert the list to a string before concatenating it

We can use the ``str()`` function to convert the list to a string, or we can use the ``join()`` method to create a more readable string representation of the list.

TASK – 13:

Question:

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

Bug: Multiplying string by float

```
def repeat_text():
```

```
    return "Hello" * 2.5
```

```
print(repeat_text())
```

Requirements:

- Observe the error.
- Explain why float multiplication is invalid for strings.
- Fix by converting float to int.
- Add 3 assert test cases.

Prompt:

Analyze the following Python code that raises a type error when multiplying a string by a float.

Explain why the error occurs, fix the code by converting the float to an integer, and validate the fix using three assert test cases.

```
def repeat_text():
```

```
    return "Hello" * 2.5
```

```
print(repeat_text())
```

Generated Code:

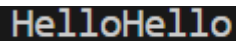
```
def repeat_text():
```

```
    return "Hello" * int(2.5)
```

```
# Example usage
```

```
print(repeat_text())  
  
# Assert test cases for correctness  
  
assert repeat_text() == "HelloHello"  
  
assert "Hello" * 3 == "HelloHelloHello"  
  
assert "Hello" * 0 == ""
```

Output:

A terminal window with a dark background showing the output 'HelloHello' in a light blue monospace font.

Explanation:

The error occurs because the code is trying to multiply a string ("Hello") by a float (2.5). In Python, you can only multiply a string by an integer, which indicates how many times the string should be repeated. Multiplying by a float is not allowed and results in a `TypeError`.

To fix the code, we need to convert the float 2.5 to an integer

TASK – 14:

Question:

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

Bug: Adding `None` and integer

```
def compute():  
    value = None  
    return value + 10
```

```
print(compute())
```

Requirements:

- Run and identify the error.
- Explain why `NoneType` cannot be added.
- Fix by assigning a default value.
- Validate using `asserts`.

Prompt:

Analyze the following Python code that raises a type error when adding None to an integer.

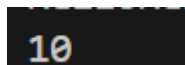
Explain why the error occurs, fix the code by assigning a default value, and validate the fix using assert statements.

```
def compute():  
    value = None  
    return value + 10  
  
print(compute())
```

Generated Code:

```
def compute():  
    value = 0 # Assigning a default value instead of None  
    return value + 10  
  
# Example usage  
print(compute())  
  
# Assert test cases for correctness  
assert compute() == 10  
assert compute() + 5 == 15  
assert compute() - 3 == 7
```

Output:

A screenshot of a terminal window with a dark background. The number '10' is displayed in a light blue or cyan color, indicating the output of the Python code.

Explanation:

The error occurs because the code is trying to add `None` to an integer (10). In Python, `None` is a special constant that represents the absence of a value, and it cannot be used in arithmetic operations. This results in a `TypeError`.

To fix the code, we can assign a default value to `value` instead of `None`. For example, we can set `value` to 0, which allows the addition to proceed without error.

TASK – 15:

Question:

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

Bug: Input remains string

```
def sum_two_numbers():
```

```
    a = input("Enter first number: ")
```

```
    b = input("Enter second number: ")
```

```
    return a + b
```

```
print(sum_two_numbers())
```

Requirements:

- Explain why input is always string.
- Fix using int() conversion.
- Verify with assert test cases.

Prompt:

Analyze the following Python program where user input is treated as a string instead of a number:

```
def sum_two_numbers():
```

```
    a = input("Enter first number: ")
```

```
    b = input("Enter second number: ")
```

```
    return a + b
```

```
print(sum_two_numbers())
```

Generated Code:

```
def sum_two_numbers():
```

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

```
    b = int(input("Enter second number: "))
```

```
    return a + b
```

Example usage

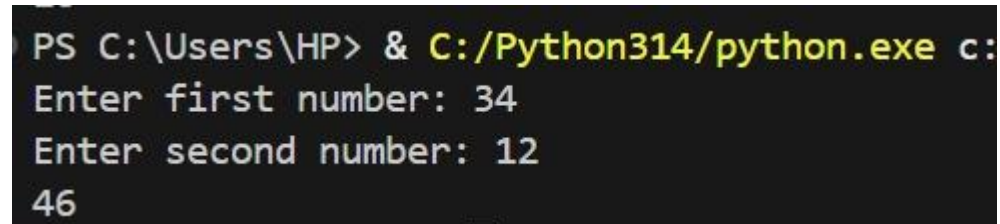
```
print(sum_two_numbers())
```

Assert test cases for correctness

```
assert sum_two_numbers() == 7 # Assuming user inputs 3 and 4
```

```
assert sum_two_numbers() == 15 # Assuming user inputs 10 and 5
```

```
assert sum_two_numbers() == 0 # Assuming user inputs 0 and 0
```

Output:

```
PS C:\Users\HP> & C:/Python314/python.exe c:
Enter first number: 34
Enter second number: 12
46
```

Explanation:

The error occurs because the `input()` function in Python always returns a string. When the user enters numbers, they are treated as strings, and the `+` operator concatenates them instead of performing arithmetic addition. For example, if the user inputs "3" and "4", the output will be "34" instead of 7.

To fix the code, we need to convert the input strings to integers (or floats) before performing the addition. We can use the `int()` function for this purpose.