Assignment-10

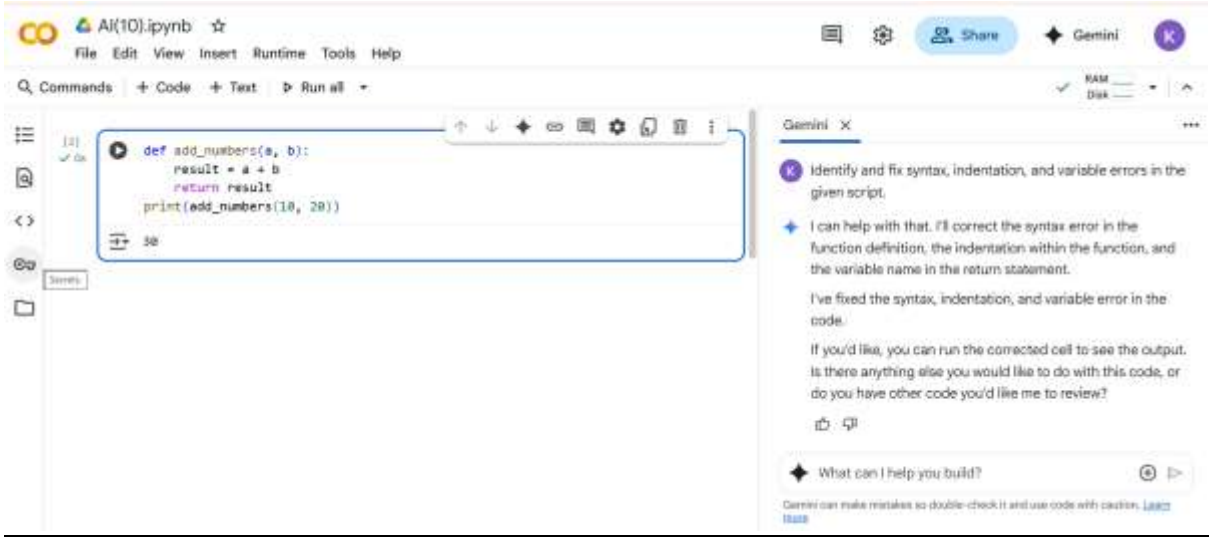
Task 1:

Identify and fix syntax, indentation, and variable errors in the given script.

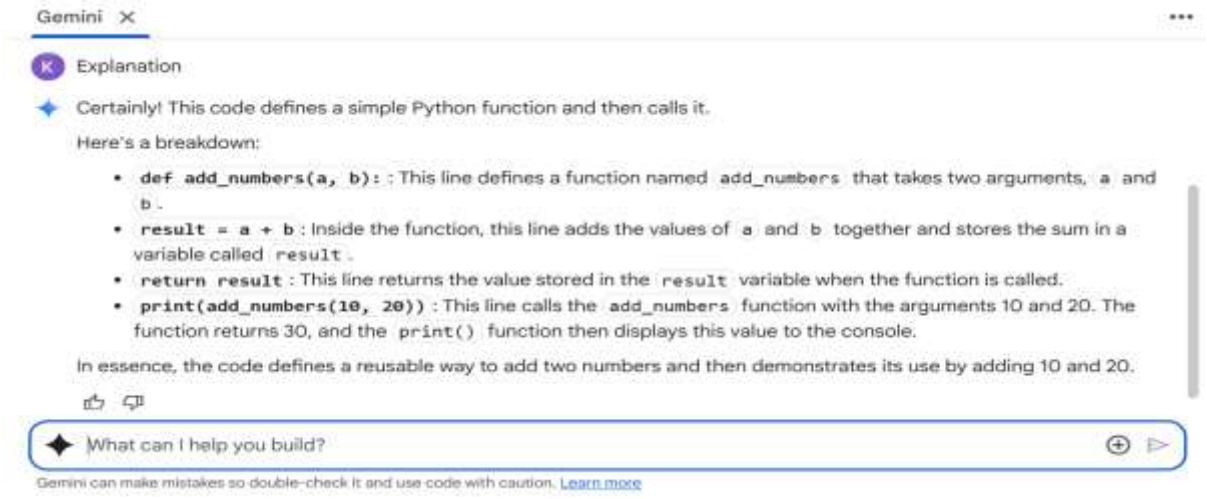
```
# buggy_code_task1.py

def add_numbers(a, b)
    result = a + b
    return reslt
print(add_numbers(10 20))
```

code and output:



Explanation:



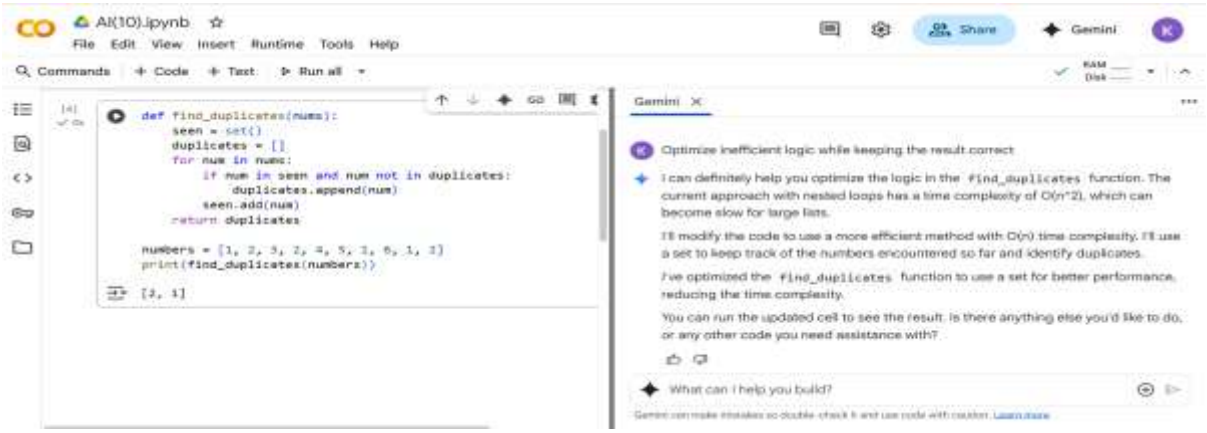
Task 2:

Optimize inefficient logic while keeping the result correct.

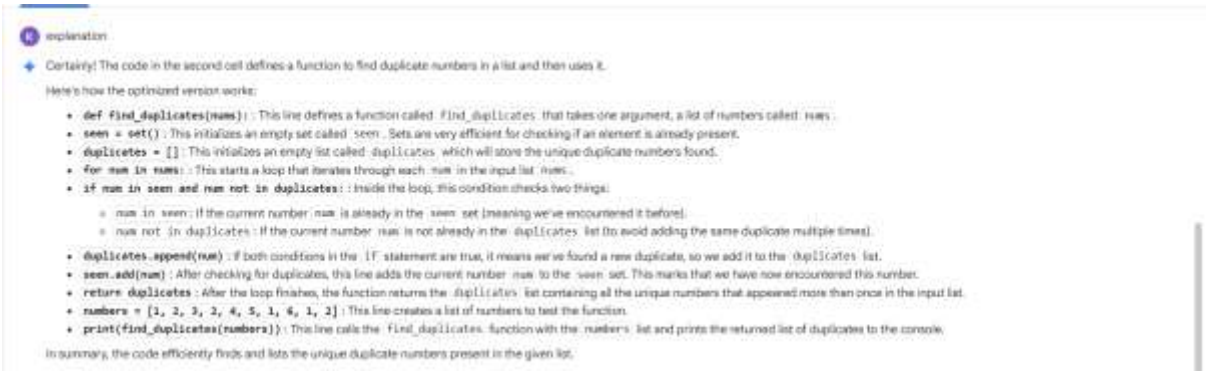
buggy_code_task2.py

```
def find_duplicates(nums):
    duplicates = []
    for i in range(len(nums)):
        for j in range(len(nums)):
            if i != j and nums[i] == nums[j] and nums[i] not in duplicates:
                duplicates.append(nums[i])
    return duplicates
numbers = [1,2,3,2,4,5,1,6,1,2]
print(find_duplicates(numbers))
```

Code and output:



Explanation:



explanation

Certainly! The code in the second cell defines a function to find duplicate numbers in a list and then uses it.

Here's how the optimized version works:

- `def find_duplicates(nums):` : This line defines a function called `find_duplicates` that takes one argument, a list of numbers called `nums`.
- `seen = set()` : This initializes an empty set called `seen`. Sets are very efficient for checking if an element is already present.
- `duplicates = []` : This initializes an empty list called `duplicates` which will store the unique duplicate numbers found.
- `for num in nums:` : This starts a loop that iterates through each `num` in the input list `nums`.
- `if num in seen and num not in duplicates:` : Inside the loop, this condition checks two things:
  - `num in seen` : If the current number `num` is already in the `seen` set (meaning we've encountered it before).
  - `num not in duplicates` : If the current number `num` is not already in the `duplicates` list (to avoid adding the same duplicate multiple times).
- `duplicates.append(num)` : If both conditions in the `if` statement are true, it means we've found a new duplicate, so we add it to the `duplicates` list.
- `seen.add(num)` : After checking for duplicates, this line adds the current number `num` to the `seen` set. This marks that we have now encountered this number.
- `return duplicates` : After the loop finishes, the function returns the `duplicates` list containing all the unique numbers that appeared more than once in the input list.
- `numbers = [1, 2, 3, 2, 4, 5, 1, 6, 1, 2]` : This line creates a list of numbers to test the function.
- `print(find_duplicates(numbers))` : This line calls the `find_duplicates` function with the `numbers` list and prints the returned list of duplicates to the console.

In summary, the code efficiently finds and lists the unique duplicate numbers present in the given list.

Task 3:

Refactor messy code into clean, PEP 8–compliant, well-structured code.

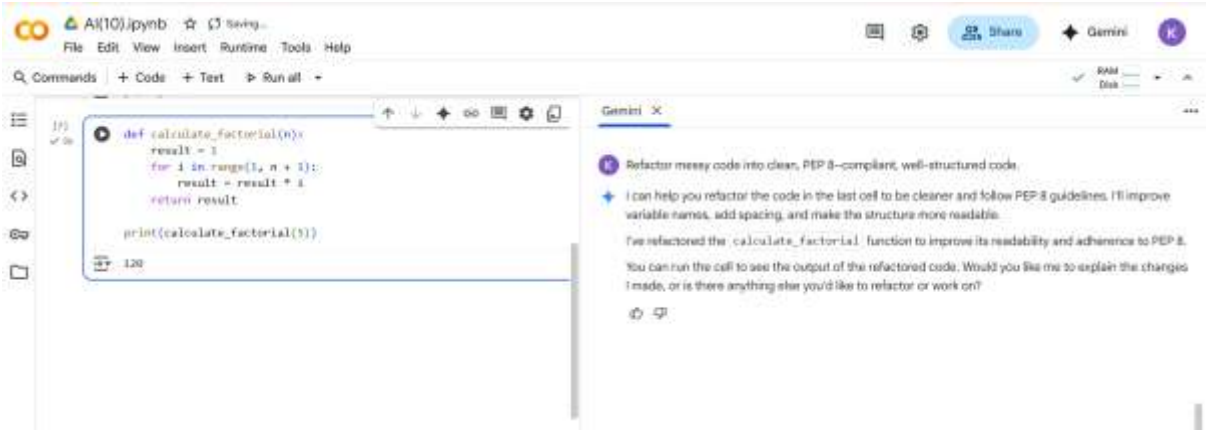
buggy_code_task3.py

```
def c(n):
 x=1
 for i in range(1,n+1):
  x=x*i
 return x
print(c(5))
```

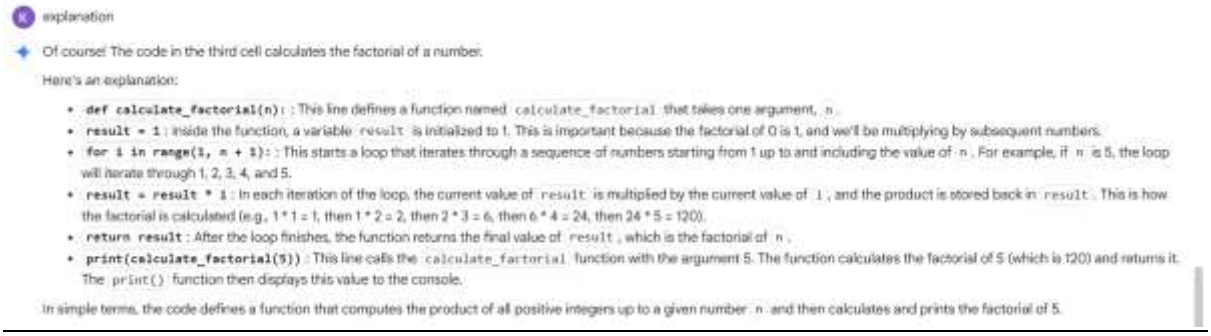
code and output:

```
def c(n):
 x=1
 for i in range(1,n+1):
  x=x*i
 return x
print(c(5))
```

```
120
```



Explanation:



Task 4:

Add security practices and exception handling to the code.

```
# buggy_code_task4.py
import sqlite3
def get_user_data(user_id):
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()
    query = f"SELECT * FROM users WHERE id = {user_id};"  # Potential SQL injection risk
    cursor.execute(query)
    result = cursor.fetchall()
    conn.close()
    return result
user_input = input("Enter user ID: ")
print(get_user_data(user_input))
```

Code and output:

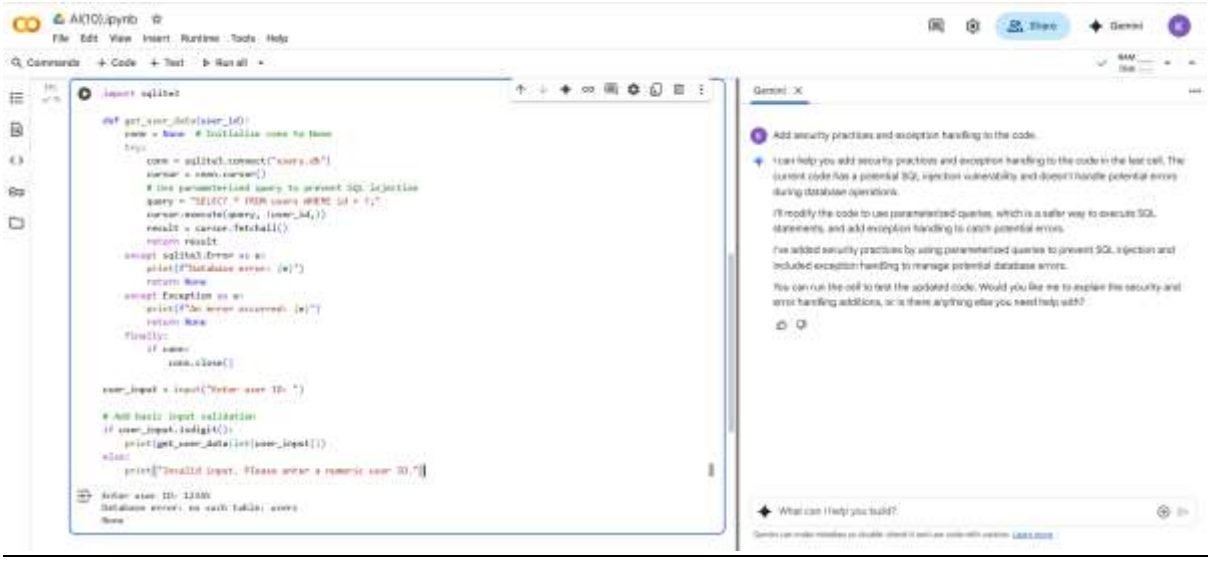
```python
import sqlite3
def get_user_data(user_id):
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()
    query = f"SELECT * FROM users WHERE id = {user_id};"  # Potential SQL injection risk
    cursor.execute(query)
    result = cursor.fetchall()
    conn.close()
    return result
user_input = input("Enter user ID: ")
print(get_user_data(user_input))
```

```
Enter user ID: vinod
---------------------------------------------------------------------------
OperationalError                          Traceback (most recent call last)
/tmp/ipython-input-3726322882.py in <cell line: 0>()
      9     return result
     10 user_input = input("Enter user ID: ")
---> 11 print(get_user_data(user_input))

/tmp/ipython-input-3726322882.py in get_user_data(user_id)
      4     cursor = conn.cursor()
      5     query = f"SELECT * FROM users WHERE id = {user_id};"  # Potential SQL injection risk
----> 6     cursor.execute(query)
      7     result = cursor.fetchall()
      8     conn.close()

OperationalError: no such table: users
```

Next steps: Explain error



Explanation:


explanation

Absolutely! The code in the last cell is designed to fetch user data from an SQLite database, incorporating security and error handling.

Here's a breakdown:

- `import sqlite3` : This line imports the `sqlite3` library, which provides an interface for working with SQLite databases in Python.
- `def get_user_data(user_id):` : This defines a function named `get_user_data` that takes one argument, `user_id` .
- `conn = None` : Initializes the `conn` variable (for the database connection) to `None` .
- `try:` : This block starts a `try` block, which is used to enclose code that might raise an exception.
- `conn = sqlite3.connect("users.db")` : Attempts to establish a connection to the SQLite database file named "users.db".
- `cursor = conn.cursor()` : Creates a cursor object using the connection. Cursors are used to execute SQL commands.
- `query = "SELECT * FROM users WHERE id = ?"` : Defines the SQL query to select all columns from the `users` table where the `id` matches. The `?` is a placeholder for the user ID, which is a key part of preventing SQL injection.
- `cursor.execute(query, (user_id,))` : Executes the SQL query. The `user_id` is passed as a tuple `(user_id,)` as the second argument to `execute` . This is how parameterized queries work; the database driver handles the proper escaping of the value, preventing malicious input from affecting the query structure.
- `result = cursor.fetchall()` : Fetches all rows from the result of the query and stores them in the `result` variable as a list of tuples.
- `return result` : Returns the fetched data.
- `except sqlite3.Error as e:` : This block catches specific exceptions related to SQLite database errors. If a `sqlite3.Error` occurs, it prints an informative error message including the specific error `e` .
- `except Exception as e:` : This block catches any other general exceptions that might occur. It prints a generic error message including the specific error `e` .
- `return None` : In case of any exception, the function returns `None` to indicate that the data could not be retrieved.
- `finally:` : This block contains code that will always execute, regardless of whether an exception occurred or not.
- `if conn:` : Checks if the `conn` variable is not `None` (meaning a connection was successfully established).
- `conn.close()` : If a connection exists, this line closes the database connection, releasing resources.
- `user_input = input("Enter user ID: ")` : Prompts the user to enter a user ID and stores the input in the `user_input` variable.
- `if user_input.isdigit():` : This line adds basic input validation, checking if the user's input consists only of digits.
- `print(get_user_data(int(user_input)))` : If the input is a digit, it converts the input to an integer and calls the `get_user_data` function with this integer ID, then prints the returned data.
- `else:` : If the input is not a digit.
- `print("Invalid input. Please enter a numeric user ID.")` : Prints an error message indicating that the input was invalid.

In summary, this code provides a safer and more robust way to interact with an SQLite database by preventing SQL injection and handling potential errors during the process.

Task 5:

Generate a review report for this messy code.

```
# buggy_code_task5.py

def calc(x,y,z):

 if z=="add":

  return x+y

 elif z=="sub": return x-y

 elif z=="mul":

  return x*y

 elif z=="div":

  return x/y

 else: print("wrong")

  print(calc(10,5,"add"))

print(calc(10,0,"div"))
```

code and output:

```
def calc(x,y,z):
 if z=="add":
  return x+y
 elif z=="sub": return x-y
 elif z=="mul":
  return x*y
 elif z=="div":
  return x/y
 else: print("wrong")

print(calc(10,5,"add"))
print(calc(10,0,"div"))
```

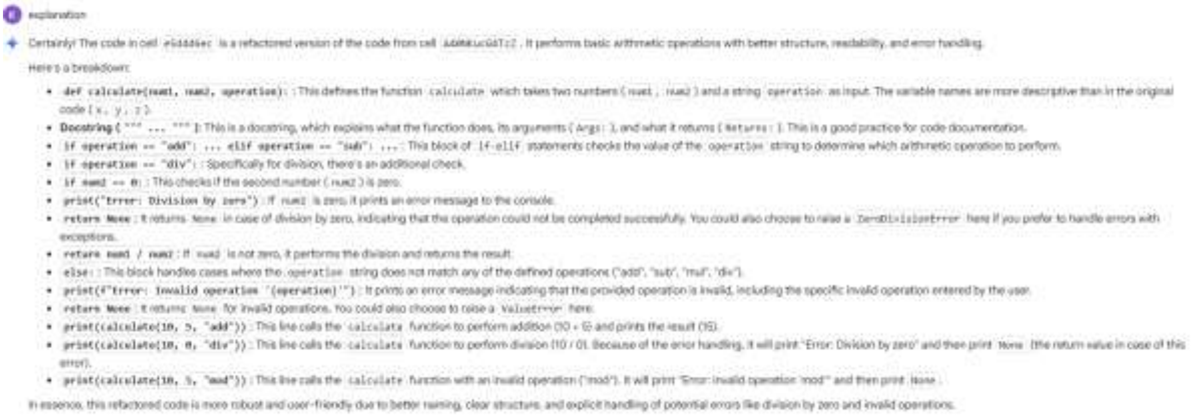
```
15
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipython-input-432503943.py in <cell line: 0>()
     10
     11 print(calc(10,5,"add"))
---> 12 print(calc(10,0,"div"))

/tmp/ipython-input-432503943.py in calc(x, y, z)
      6   return x*y
      7  elif z=="div":
----> 8   return x/y
      9  else: print("wrong")
     10

ZeroDivisionError: division by zero
```

Next steps: Explain error

Explanation:



Certainly! The code in cell `[illegible]` is a refactored version of the code from cell `[illegible]`. It performs basic arithmetic operations with better structure, readability, and error handling.

Here's a breakdown:

- `def calculate(num1, num2, operation):` : This defines the function `calculate` which takes two numbers (`num1`, `num2`) and a string `operation` as input. The variable names are more descriptive than in the original code (`x`, `y`, `z`).
- **Docstring** (`""" ... """`): This is a docstring, which explains what the function does, its arguments (`Args:`), and what it returns (`Returns:`). This is a good practice for code documentation.
- `if operation == "add": ... elif operation == "sub": ...` : This block of `if-elif` statements checks the value of the `operation` string to determine which arithmetic operation to perform.
- `if operation == "div":` : Specifically for division, there's an additional check.
- `if num2 == 0:` : This checks if the second number (`num2`) is zero.
- `print("Error: Division by zero")` : If `num2` is zero, it prints an error message to the console.
- `return None` : It returns `None` in case of division by zero, indicating that the operation could not be completed successfully. You could also choose to raise a `ZeroDivisionError` here if you prefer to handle errors with exceptions.
- `return num1 / num2` : If `num2` is not zero, it performs the division and returns the result.
- `else:` : This block handles cases where the `operation` string does not match any of the defined operations ("add", "sub", "mul", "div").
- `print(f"Error: Invalid operation '{operation}'")` : It prints an error message indicating that the provided operation is invalid, including the specific invalid operation entered by the user.
- `return None` : It returns `None` for invalid operations. You could also choose to raise a `ValueError` here.
- `print(calculate(10, 5, "add"))` : This line calls the `calculate` function to perform addition (10 + 5) and prints the result (15).
- `print(calculate(10, 0, "div"))` : This line calls the `calculate` function to perform division (10 / 0). Because of the error handling, it will print "Error: Division by zero" and then print `None` (the return value in case of this error).
- `print(calculate(10, 5, "mod"))` : This line calls the `calculate` function with an invalid operation ("mod"). It will print "Error: Invalid operation 'mod'" and then print `None`.

In essence, this refactored code is more robust and user-friendly due to better naming, clear structure, and explicit handling of potential errors like division by zero and invalid operations.