

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:



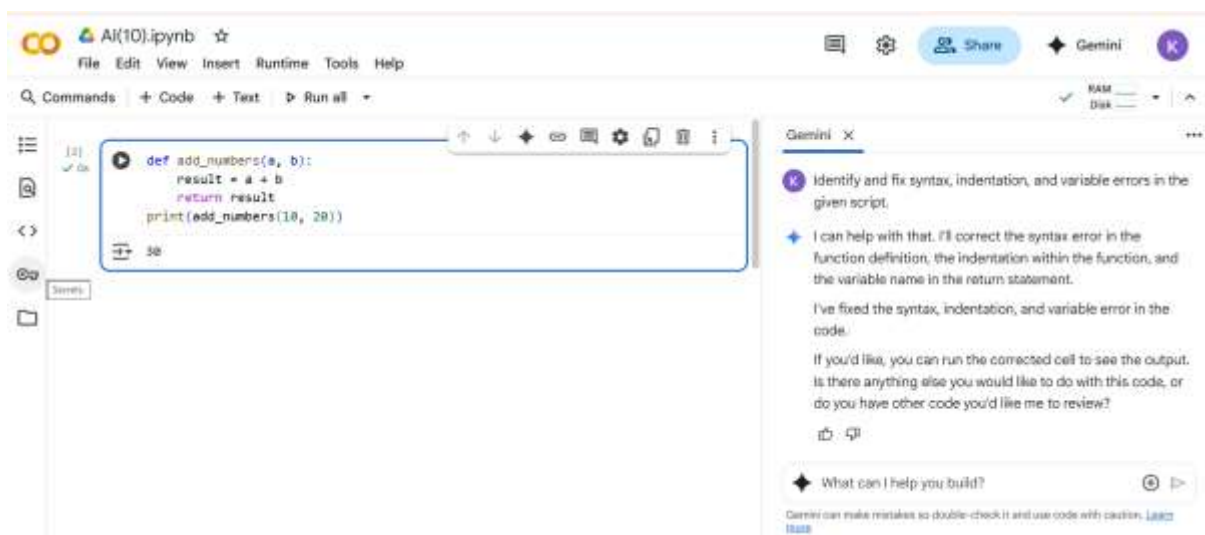
The screenshot shows a code editor with a Python script. The script contains the following code:

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

The code has several errors: a missing colon at the end of the function definition, incorrect indentation for the function body, a misspelled variable name 'reslt', and a missing comma between the arguments in the print statement. The error message displayed is:

```
File "/tmp/ipython-input-2887063201.py", line 1
def add_numbers(a, b)
^
SyntaxError: expected ':'
```

Next steps: [Explain error](#)



The screenshot shows the same code editor with the corrected script:

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

The AI assistant's response is as follows:

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

I can help with that. I'll correct the syntax error in the function definition, the indentation within the function, and the variable name in the return statement.

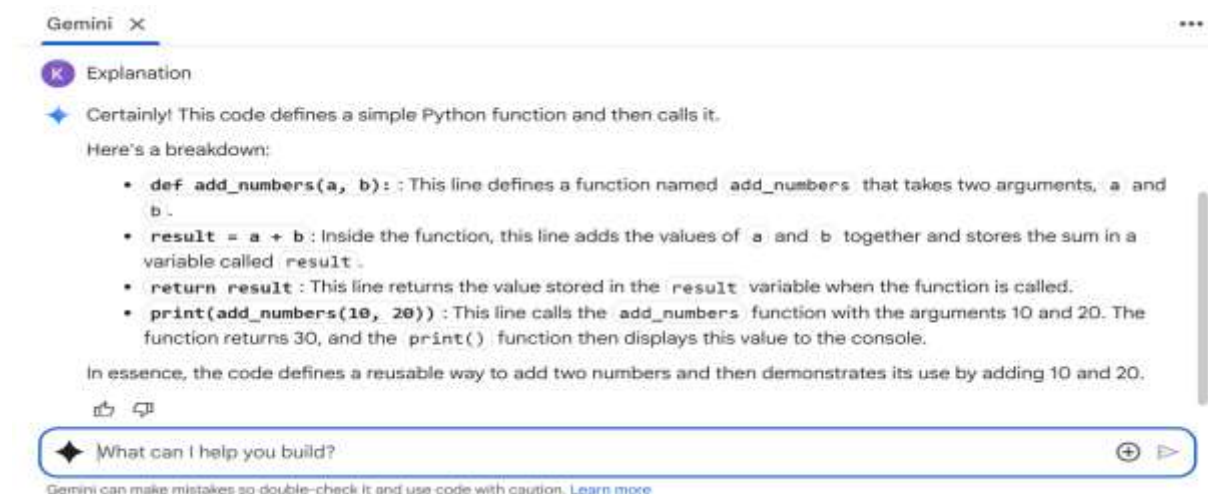
I've fixed the syntax, indentation, and variable error in the code.

If you'd like, you can run the corrected cell to see the output. Is there anything else you would like to do with this code, or do you have other code you'd like me to review?

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Explanation:



Gemini X

K Explanation

◆ Certainly! This code defines a simple Python function and then calls it. Here's a breakdown:

- `def add_numbers(a, b):` : This line defines a function named `add_numbers` that takes two arguments, `a` and `b`.
- `result = a + b` : Inside the function, this line adds the values of `a` and `b` together and stores the sum in a variable called `result`.
- `return result` : This line returns the value stored in the `result` variable when the function is called.
- `print(add_numbers(10, 20))` : This line calls the `add_numbers` function with the arguments 10 and 20. The function returns 30, and the `print()` function then displays this value to the console.

In essence, the code defines a reusable way to add two numbers and then demonstrates its use by adding 10 and 20.

👍 🗨

◆ What can I help you build? + ▶

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

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:
```

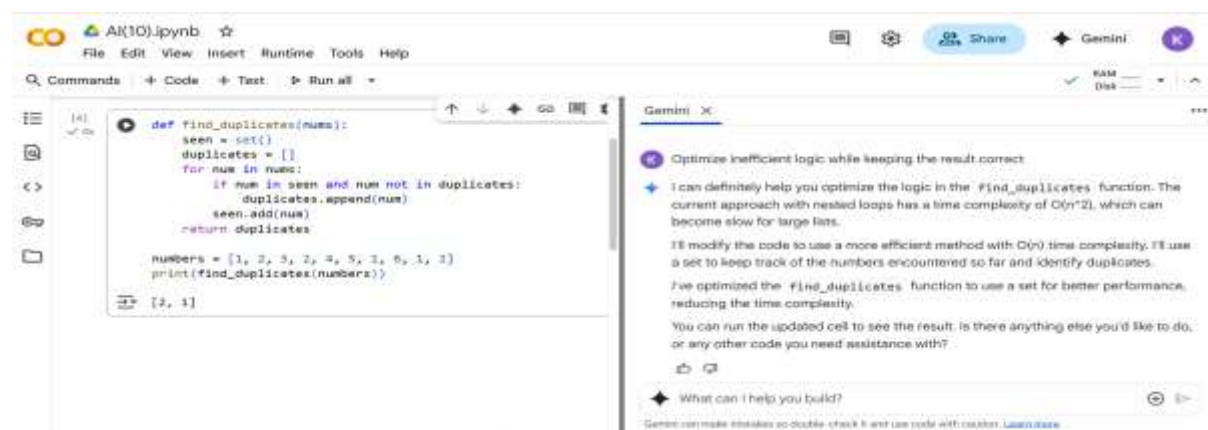
```
                duplicates.append(nums[i])
```

```
    return duplicates
```

```
numbers = [1,2,3,2,4,5,1,6,1,2]
```

```
print(find_duplicates(numbers))
```

Code and output:



AK10.ipynb

File Edit View Insert Runtime Tools Help

Commands Code Text Run all

```
[4]: def find_duplicates(nums):
    seen = set()
    duplicates = []
    for num in nums:
        if num in seen and num not in duplicates:
            duplicates.append(num)
        seen.add(num)
    return duplicates

numbers = [1, 2, 3, 2, 4, 5, 1, 6, 1, 2]
print(find_duplicates(numbers))

[3, 4]
```

Gemini X

◆ Optimize inefficient logic while keeping the result correct


◆ I can definitely help you optimize the logic in the `find_duplicates` function. The current approach with nested loops has a time complexity of $O(n^2)$, which can become slow for large lists. I'll modify the code to use a more efficient method with $O(n)$ time complexity. I'll use a set to keep track of the numbers encountered so far and identify duplicates. I've optimized the `find_duplicates` function to use a set for better performance, reducing the time complexity. You can run the updated cell to see the result. Is there anything else you'd like to do, or any other code you need assistance with?

👍 🗨

◆ What can I help you build? + ▶

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

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
```

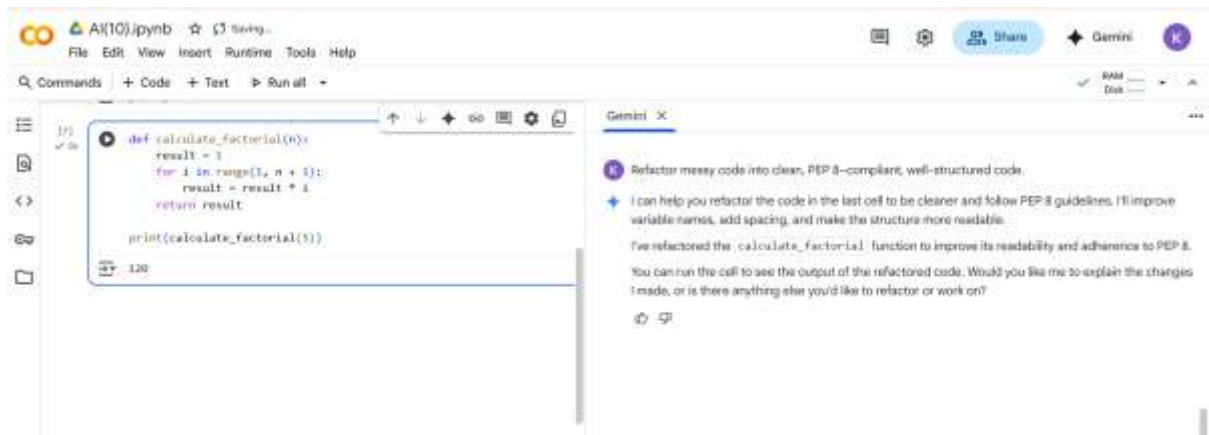
```
    return x
```

```
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:

explanation

Of course! The code in the third cell calculates the factorial of a number. Here's an explanation:

- def calculate_factorial(n):** This line defines a function named `calculate_factorial` that takes one argument, `n`.
- result = 1:** Inside the function, a variable `result` is initialized to 1. This is important because the factorial of 0 is 1, and we'll be multiplying by subsequent numbers.
- for i in range(1, n + 1):** This starts a loop that iterates through a sequence of numbers starting from 1 up to and including the value of `n`. For example, if `n` is 5, the loop will iterate through 1, 2, 3, 4, and 5.
- result = result * i:** In each iteration of the loop, the current value of `result` is multiplied by the current value of `i`, and the product is stored back in `result`. This is how the factorial is calculated (e.g., $1 * 1 = 1$, then $1 * 2 = 2$, then $2 * 3 = 6$, then $6 * 4 = 24$, then $24 * 5 = 120$).
- return result:** After the loop finishes, the function returns the final value of `result`, which is the factorial of `n`.
- print(calculate_factorial(5))** This line calls the `calculate_factorial` function with the argument 5. The function calculates the factorial of 5 (which is 120) and returns it. The `print()` function then displays this value to the console.

In simple terms, the code defines a function that computes the product of all positive integers up to a given number `n`, and then calculates and prints the factorial of 5.

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:

```
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](#)

AK10.py:nb

File Edit View Insert Runtime Tools Help

Commands + Code + Text + Run All +

```
import sqlite3

def get_user_data(user_id):
    """
    Get user data from the database.
    """
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()
    # Use parameterized query to prevent SQL injection
    query = "SELECT * FROM users WHERE id = ?;"
    cursor.execute(query, (user_id,))
    result = cursor.fetchall()
    return result

except sqlite3.Error as e:
    print(f"Database error: {e}")
    return None

except Exception as e:
    print(f"An error occurred: {e}")
    return None

finally:
    if conn:
        conn.close()

user_input = input("Enter user ID: ")

# Add basic input validation
if user_input.isdigit():
    print(get_user_data(int(user_input)))
else:
    print("Invalid input. Please enter a numeric user ID.")
```

Enter user ID: 1234
Database error: no such table: users
None

GenAI X

1 Add security practices and exception handling to the code.

I can help you add security practices and exception handling to the code in the last cell. The current code has a potential SQL injection vulnerability and doesn't handle potential errors during database operations.

I'll modify the code to use parameterized queries, which is a safer way to execute SQL statements, and add exception handling to catch potential errors.

I've added security practices by using parameterized queries to prevent SQL injection and included exception handling to manage potential database errors.

You can run the cell to test the updated code. Would you like me to explain the security and error handling additions, or is there anything else you need help with?

What can I help you build?

Explanation:



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](#)

The screenshot shows a Jupyter Notebook interface. The code cell contains a function `calc(x,y,z)` that performs arithmetic operations based on the string `z`. It handles `add`, `sub`, `mul`, and `div` operations. The output cell shows the execution of `print(calc(10,5,"add"))` and `print(calc(10,0,"div"))`. The first print statement outputs `15`, and the second one raises a `ZeroDivisionError: division by zero`.

The AI-generated explanation in the output cell is as follows:

```
Generate a review report for this messy code.  
I can generate a review report for the code in cell. Just let's call it. It highlights areas that could be improved for readability, maintainability, and robustness.  
I've generated a code review report for cell. Here's a summary of the issues found in the report, such as:  
• Improved variable names for clarity (e.g., x, y, z for num1, num2, operation).  
• Consistent indentation and spacing for better readability following PEP 8.  
• Explicit handling of division by zero to prevent errors.  
• Handling of invalid operations.  
• Addition of a docstring to explain the function's purpose, arguments, and return value.  
You can review the report and the refactored code. Would you like me to explain any specific part of the report or the refactored code in more detail?
```

Explanation:

Explanation

➤ Certainly! The code in cell `#baa55ec` is a refactored version of the code from cell `#d884c687c2`. It performs basic arithmetic operations with better structure, readability, and error handling.

Here's a breakdown:

- `def calculate(x1, x2, operation):`: This defines the function `calculate`, which takes two numbers (`x1`, `x2`) 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 x2 == 0:`: This checks if the second number (`x2`) is zero.
- `print("Error: Division by zero")`: If `x2` 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 x1 / x2`: If `x2` 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("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.