Assignment-8.1

Task 1:

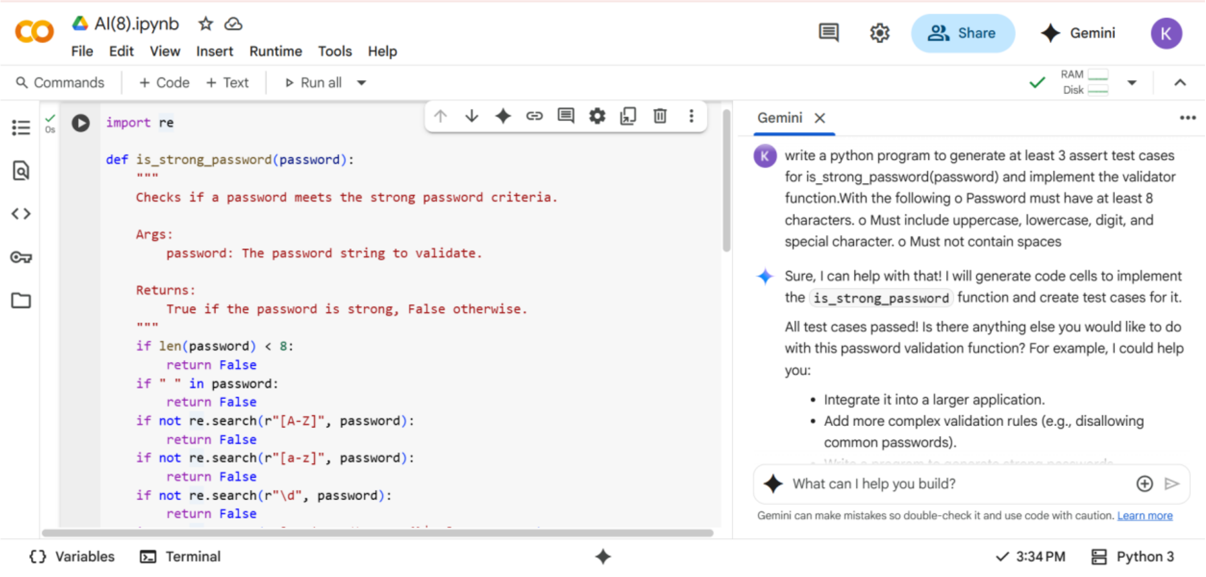
Write a python program to generate at least 3 assert test cases for is_strong_password(password) and implement the validator function.With the following

>Password must have at least 8 characters.

>Must include uppercase, lowercase, digit, and special character.

>Must not contain spaces.

Code and Output:


AI(8).ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share Gemini
RAM Disk
Variables Terminal
3:34 PM Python 3

```python
import re

def is_strong_password(password):
    """
    Checks if a password meets the strong password criteria.

    Args:
        password: The password string to validate.

    Returns:
        True if the password is strong, False otherwise.
    """
    if len(password) < 8:
        return False
    if " " in password:
        return False
    if not re.search(r"[A-Z]", password):
        return False
    if not re.search(r"[a-z]", password):
        return False
    if not re.search(r"\d", password):
        return False
```

Gemini

write a python program to generate at least 3 assert test cases for is_strong_password(password) and implement the validator function.With the following o Password must have at least 8 characters. o Must include uppercase, lowercase, digit, and special character. o Must not contain spaces

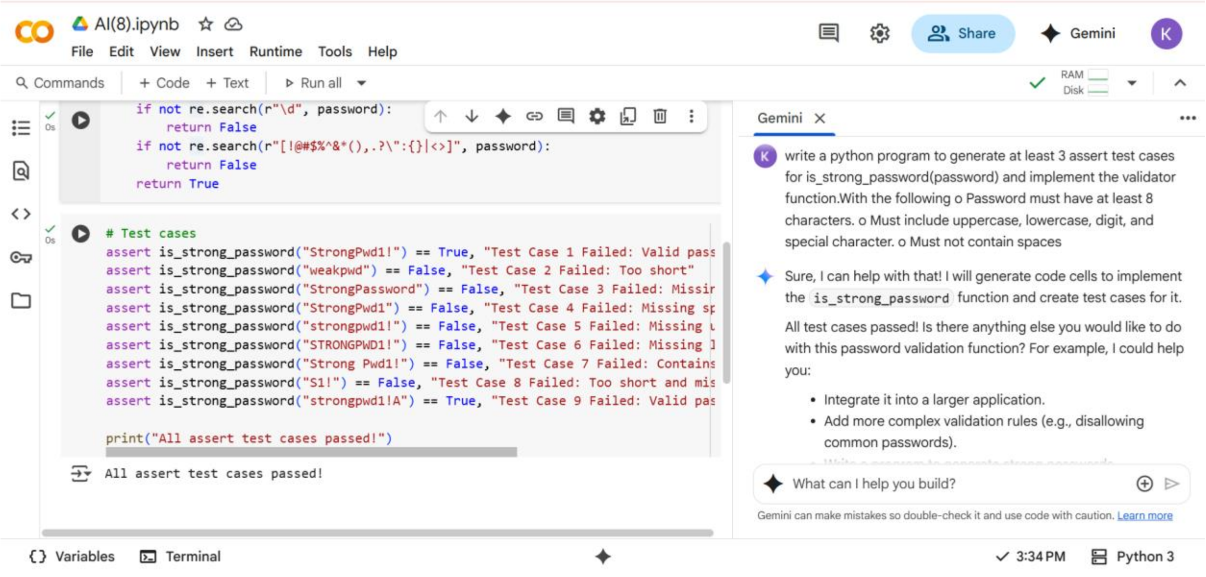
Sure, I can help with that! I will generate code cells to implement the `is_strong_password` function and create test cases for it.

All test cases passed! Is there anything else you would like to do with this password validation function? For example, I could help you:

- Integrate it into a larger application.
- Add more complex validation rules (e.g., disallowing common passwords).

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more


AI(8).ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share Gemini
RAM Disk
Variables Terminal
3:34 PM Python 3

```python
    if not re.search(r"\d", password):
        return False
    if not re.search(r"[!@#$%^&*(),.?\":{}|<>]", password):
        return False
    return True
```

```python
# Test cases
assert is_strong_password("StrongPwd1!") == True, "Test Case 1 Failed: Valid pass
assert is_strong_password("weakpwd") == False, "Test Case 2 Failed: Too short"
assert is_strong_password("StrongPassword") == False, "Test Case 3 Failed: Missir
assert is_strong_password("StrongPwd1") == False, "Test Case 4 Failed: Missing sp
assert is_strong_password("strongpwd1!") == False, "Test Case 5 Failed: Missing u
assert is_strong_password("STRONGPWD1!") == False, "Test Case 6 Failed: Missing l
assert is_strong_password("Strong Pwd1!") == False, "Test Case 7 Failed: Contains
assert is_strong_password("S1!") == False, "Test Case 8 Failed: Too short and mis
assert is_strong_password("strongpwd1!A") == True, "Test Case 9 Failed: Valid pas

print("All assert test cases passed!")
```

```
All assert test cases passed!
```

Gemini

write a python program to generate at least 3 assert test cases for is_strong_password(password) and implement the validator function.With the following o Password must have at least 8 characters. o Must include uppercase, lowercase, digit, and special character. o Must not contain spaces

Sure, I can help with that! I will generate code cells to implement the `is_strong_password` function and create test cases for it.

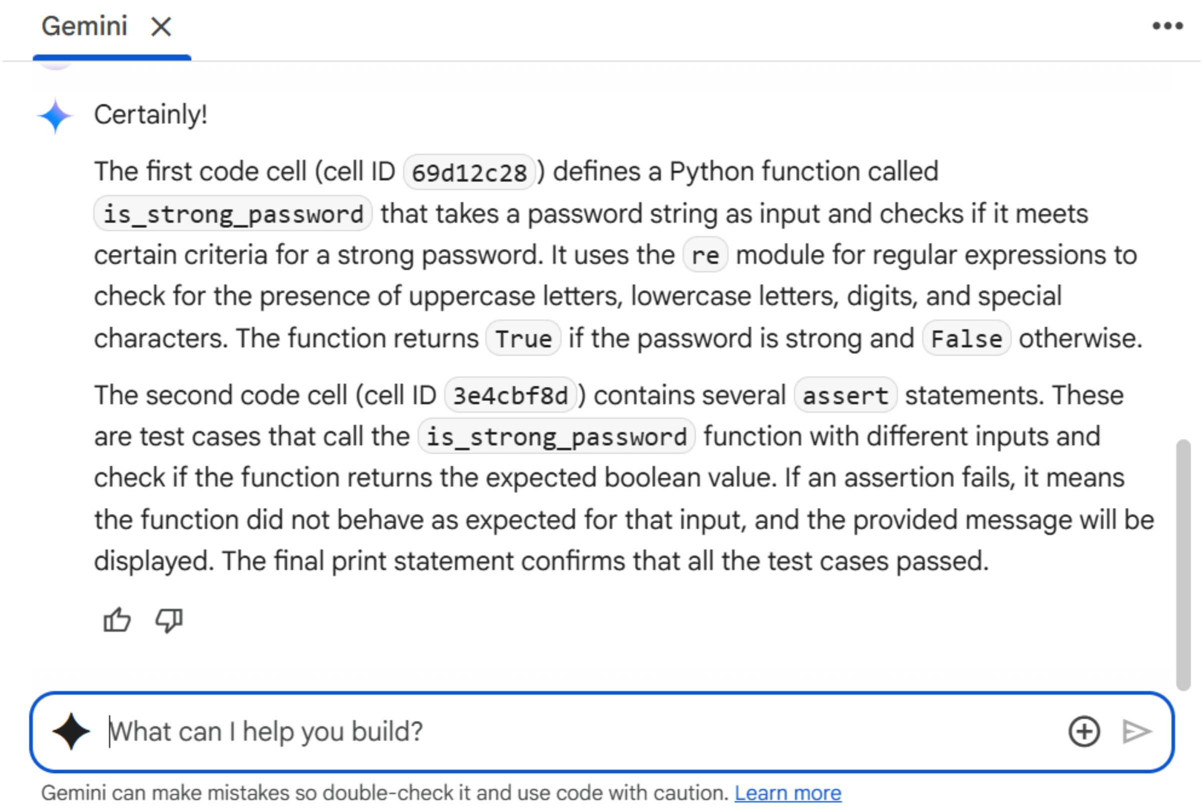
All test cases passed! Is there anything else you would like to do with this password validation function? For example, I could help you:

- Integrate it into a larger application.
- Add more complex validation rules (e.g., disallowing common passwords).

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Explanation:



Certainly!

The first code cell (cell ID `69d12c28`) defines a Python function called `is_strong_password` that takes a password string as input and checks if it meets certain criteria for a strong password. It uses the `re` module for regular expressions to check for the presence of uppercase letters, lowercase letters, digits, and special characters. The function returns `True` if the password is strong and `False` otherwise.

The second code cell (cell ID `3e4cbf8d`) contains several `assert` statements. These are test cases that call the `is_strong_password` function with different inputs and check if the function returns the expected boolean value. If an assertion fails, it means the function did not behave as expected for that input, and the provided message will be displayed. The final print statement confirms that all the test cases passed.

What can I help you build?
Gemini can make mistakes so double-check it and use code with caution. Learn more

Task 2:

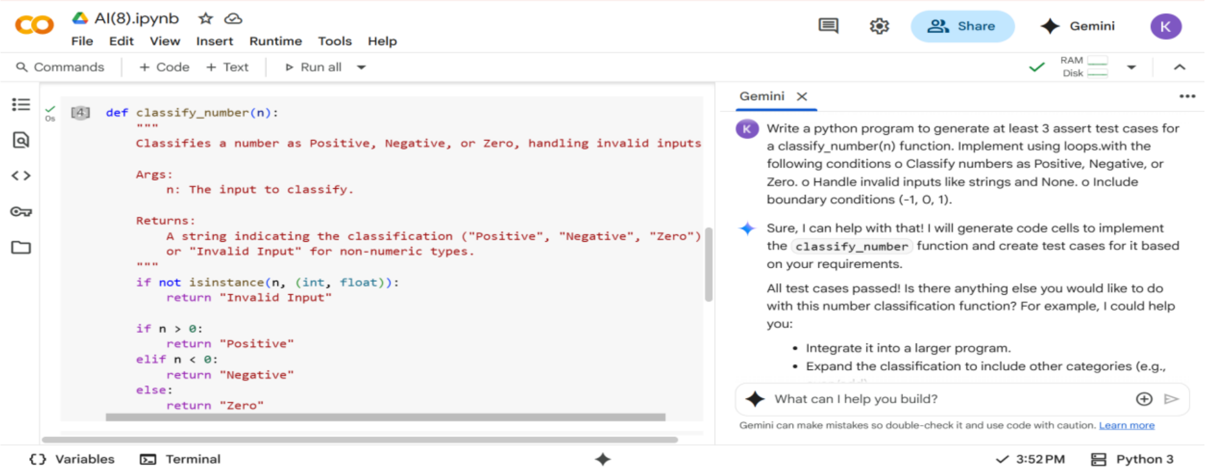
Write a python program to generate at least 3 assert test cases for a classify_number(n) function. Implement using loops.with the following conditions

>Classify numbers as Positive, Negative, or Zero.

>Handle invalid inputs like strings and None.

>Include boundary conditions (-1, 0, 1).

Code and Output:



```
def classify_number(n):
    """
    Classifies a number as Positive, Negative, or Zero, handling invalid inputs

    Args:
        n: The input to classify.

    Returns:
        A string indicating the classification ("Positive", "Negative", "Zero")
        or "Invalid Input" for non-numeric types.
    """
    if not isinstance(n, (int, float)):
        return "Invalid Input"

    if n > 0:
        return "Positive"
    elif n < 0:
        return "Negative"
    else:
        return "Zero"
```

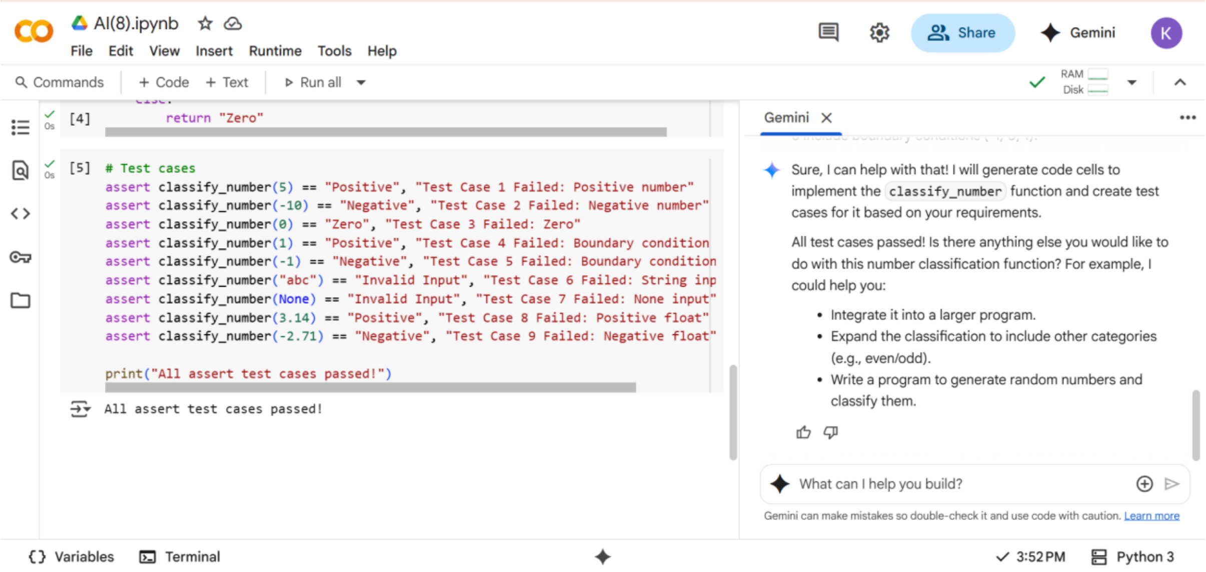
Write a python program to generate at least 3 assert test cases for a classify_number(n) function. Implement using loops.with the following conditions o Classify numbers as Positive, Negative, or Zero. o Handle invalid inputs like strings and None. o Include boundary conditions (-1, 0, 1).

Sure, I can help with that! I will generate code cells to implement the `classify_number` function and create test cases for it based on your requirements.

All test cases passed! Is there anything else you would like to do with this number classification function? For example, I could help you:

- Integrate it into a larger program.
- Expand the classification to include other categories (e.g.,

What can I help you build?
Gemini can make mistakes so double-check it and use code with caution. Learn more


AI(8).ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
Gemini
RAM
Disk
Variables
Terminal
3:52 PM
Python 3

```
[4]        return "Zero"
```

```
[5] # Test cases
    assert classify_number(5) == "Positive", "Test Case 1 Failed: Positive number"
    assert classify_number(-10) == "Negative", "Test Case 2 Failed: Negative number"
    assert classify_number(0) == "Zero", "Test Case 3 Failed: Zero"
    assert classify_number(1) == "Positive", "Test Case 4 Failed: Boundary condition
    assert classify_number(-1) == "Negative", "Test Case 5 Failed: Boundary condition
    assert classify_number("abc") == "Invalid Input", "Test Case 6 Failed: String inp
    assert classify_number(None) == "Invalid Input", "Test Case 7 Failed: None input"
    assert classify_number(3.14) == "Positive", "Test Case 8 Failed: Positive float"
    assert classify_number(-2.71) == "Negative", "Test Case 9 Failed: Negative float"

    print("All assert test cases passed!")
```

```
All assert test cases passed!
```

Sure, I can help with that! I will generate code cells to implement the `classify_number` function and create test cases for it based on your requirements.

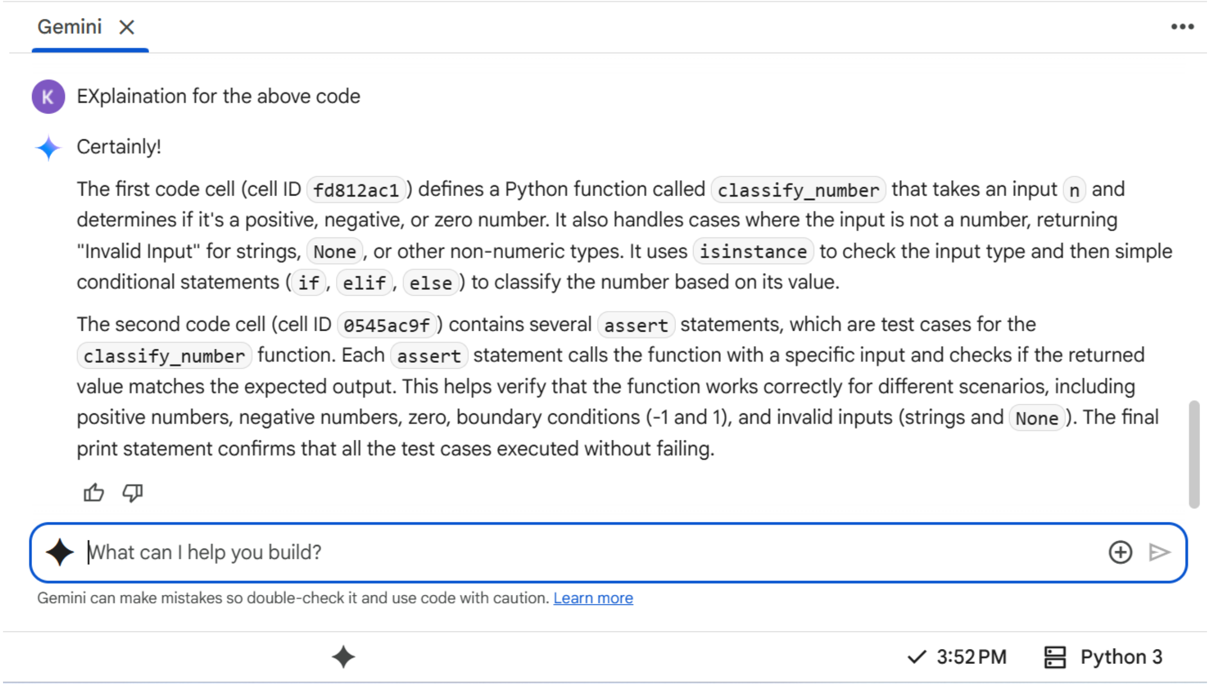
All test cases passed! Is there anything else you would like to do with this number classification function? For example, I could help you:

- Integrate it into a larger program.
- Expand the classification to include other categories (e.g., even/odd).
- Write a program to generate random numbers and classify them.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Explanation:



EXplaination for the above code

Certainly!

The first code cell (cell ID `fd812ac1`) defines a Python function called `classify_number` that takes an input `n` and determines if it's a positive, negative, or zero number. It also handles cases where the input is not a number, returning "Invalid Input" for strings, `None`, or other non-numeric types. It uses `isinstance` to check the input type and then simple conditional statements (`if`, `elif`, `else`) to classify the number based on its value.

The second code cell (cell ID `0545ac9f`) contains several `assert` statements, which are test cases for the `classify_number` function. Each `assert` statement calls the function with a specific input and checks if the returned value matches the expected output. This helps verify that the function works correctly for different scenarios, including positive numbers, negative numbers, zero, boundary conditions (-1 and 1), and invalid inputs (strings and `None`). The final print statement confirms that all the test cases executed without failing.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

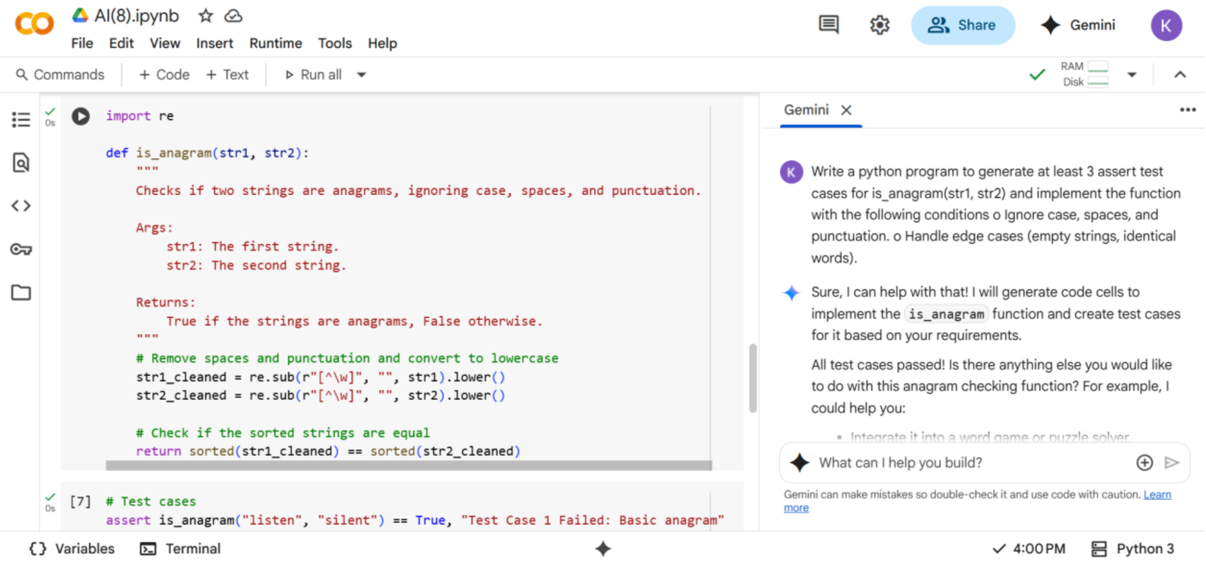
Task 3:

Write a python program to generate at least 3 assert test cases for is_anagram(str1, str2) and implement the function with the following conditions

> Ignore case, spaces, and punctuation.

>Handle edge cases (empty strings, identical words).

Code and Output:



```
import re

def is_anagram(str1, str2):
    """
    Checks if two strings are anagrams, ignoring case, spaces, and punctuation.

    Args:
        str1: The first string.
        str2: The second string.

    Returns:
        True if the strings are anagrams, False otherwise.
    """
    # Remove spaces and punctuation and convert to lowercase
    str1_cleaned = re.sub(r"[^\w]", "", str1).lower()
    str2_cleaned = re.sub(r"[^\w]", "", str2).lower()

    # Check if the sorted strings are equal
    return sorted(str1_cleaned) == sorted(str2_cleaned)
```

```
[7] # Test cases
    assert is_anagram("listen", "silent") == True, "Test Case 1 Failed: Basic anagram"
```

Gemini

Write a python program to generate at least 3 assert test cases for is_anagram(str1, str2) and implement the function with the following conditions o Ignore case, spaces, and punctuation. o Handle edge cases (empty strings, identical words).

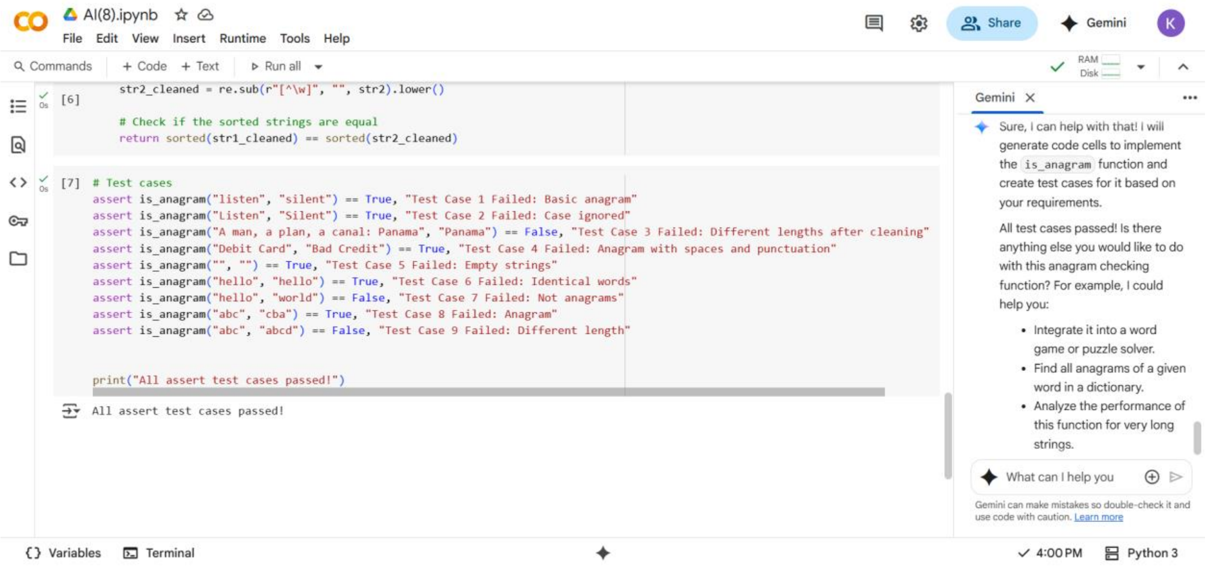
Sure, I can help with that! I will generate code cells to implement the `is_anagram` function and create test cases for it based on your requirements.

All test cases passed! Is there anything else you would like to do with this anagram checking function? For example, I could help you:

- Integrate it into a word game or puzzle solver.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more



```
[6]     str2_cleaned = re.sub(r"[^\w]", "", str2).lower()

        # Check if the sorted strings are equal
        return sorted(str1_cleaned) == sorted(str2_cleaned)
```

```
[7] # Test cases
    assert is_anagram("listen", "silent") == True, "Test Case 1 Failed: Basic anagram"
    assert is_anagram("Listen", "Silent") == True, "Test Case 2 Failed: Case ignored"
    assert is_anagram("A man, a plan, a canal: Panama", "Panama") == False, "Test Case 3 Failed: Different lengths after cleaning"
    assert is_anagram("Debit Card", "Bad Credit") == True, "Test Case 4 Failed: Anagram with spaces and punctuation"
    assert is_anagram("", "") == True, "Test Case 5 Failed: Empty strings"
    assert is_anagram("hello", "hello") == True, "Test Case 6 Failed: Identical words"
    assert is_anagram("hello", "world") == False, "Test Case 7 Failed: Not anagrams"
    assert is_anagram("abc", "cba") == True, "Test Case 8 Failed: Anagram"
    assert is_anagram("abc", "abcd") == False, "Test Case 9 Failed: Different length"


    print("All assert test cases passed!")
```

```
All assert test cases passed!
```

Gemini

Sure, I can help with that! I will generate code cells to implement the `is_anagram` function and create test cases for it based on your requirements.

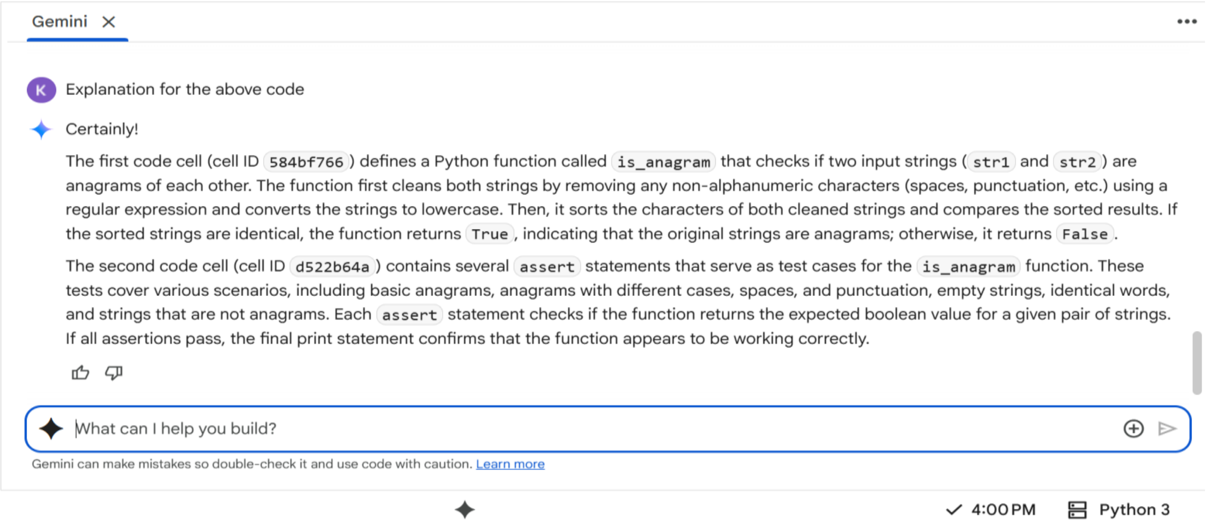
All test cases passed! Is there anything else you would like to do with this anagram checking function? For example, I could help you:

- Integrate it into a word game or puzzle solver.
- Find all anagrams of a given word in a dictionary.
- Analyze the performance of this function for very long strings.

What can I help you

Gemini can make mistakes so double-check it and use code with caution. Learn more

Explanation:



Explanation for the above code

Certainly!

The first code cell (cell ID `584bf766`) defines a Python function called `is_anagram` that checks if two input strings (`str1` and `str2`) are anagrams of each other. The function first cleans both strings by removing any non-alphanumeric characters (spaces, punctuation, etc.) using a regular expression and converts the strings to lowercase. Then, it sorts the characters of both cleaned strings and compares the sorted results. If the sorted strings are identical, the function returns `True`, indicating that the original strings are anagrams; otherwise, it returns `False`.

The second code cell (cell ID `d522b64a`) contains several `assert` statements that serve as test cases for the `is_anagram` function. These tests cover various scenarios, including basic anagrams, anagrams with different cases, spaces, and punctuation, empty strings, identical words, and strings that are not anagrams. Each `assert` statement checks if the function returns the expected boolean value for a given pair of strings. If all assertions pass, the final print statement confirms that the function appears to be working correctly.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Task 4:

Write a python program to generate at least 3 assert-based tests for an Inventory class with stock management with the following methods

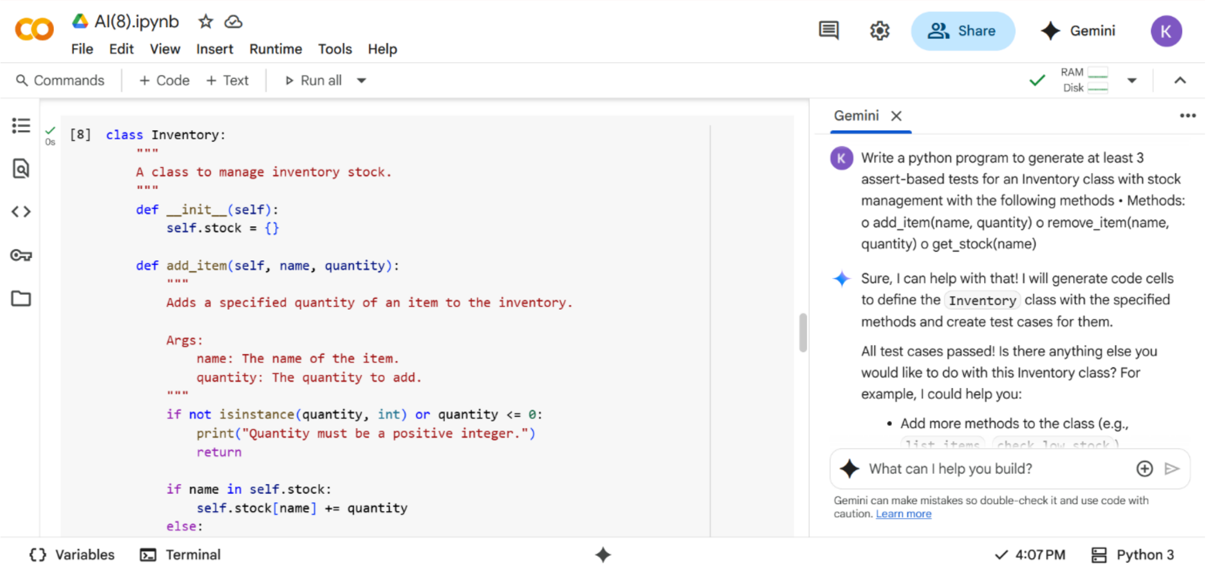
Methods:

>add_item(name, quantity) .

> remove_item(name, quantity) .

>get_stock(name).

Code and Output:



```
[8] class Inventory:
        """
        A class to manage inventory stock.
        """
        def __init__(self):
            self.stock = {}

        def add_item(self, name, quantity):
            """
            Adds a specified quantity of an item to the inventory.

            Args:
                name: The name of the item.
                quantity: The quantity to add.
            """
            if not isinstance(quantity, int) or quantity <= 0:
                print("Quantity must be a positive integer.")
                return

            if name in self.stock:
                self.stock[name] += quantity
            else:
```

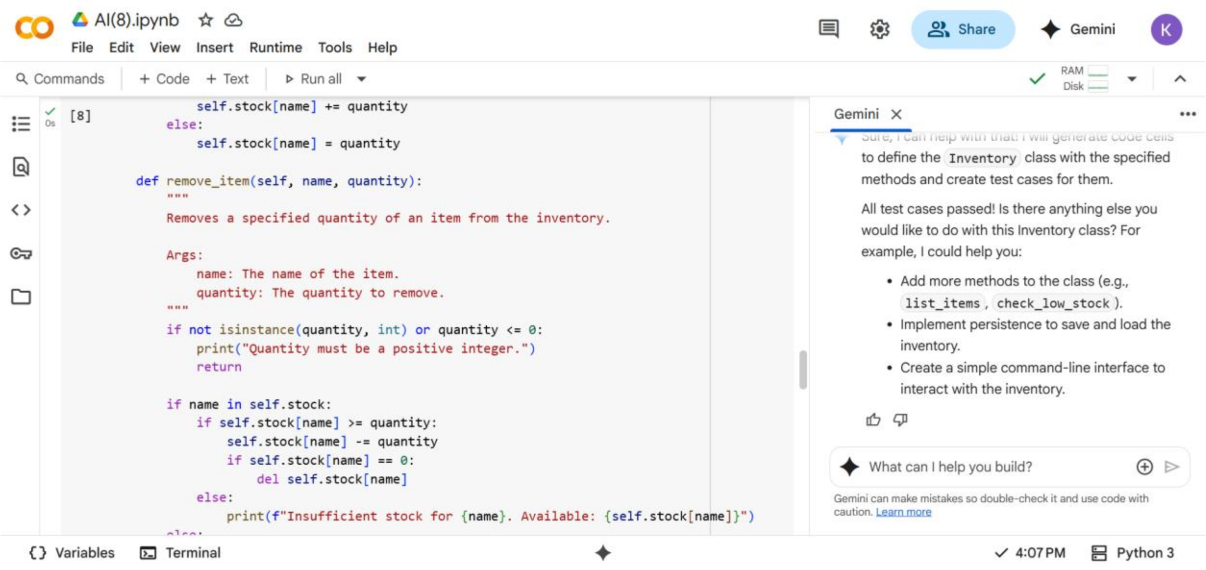
Write a python program to generate at least 3 assert-based tests for an Inventory class with stock management with the following methods • Methods: o add_item(name, quantity) o remove_item(name, quantity) o get_stock(name)

Sure, I can help with that! I will generate code cells to define the `Inventory` class with the specified methods and create test cases for them.

All test cases passed! Is there anything else you would like to do with this Inventory class? For example, I could help you:

- Add more methods to the class (e.g.,

Gemini can make mistakes so double-check it and use code with caution. Learn more



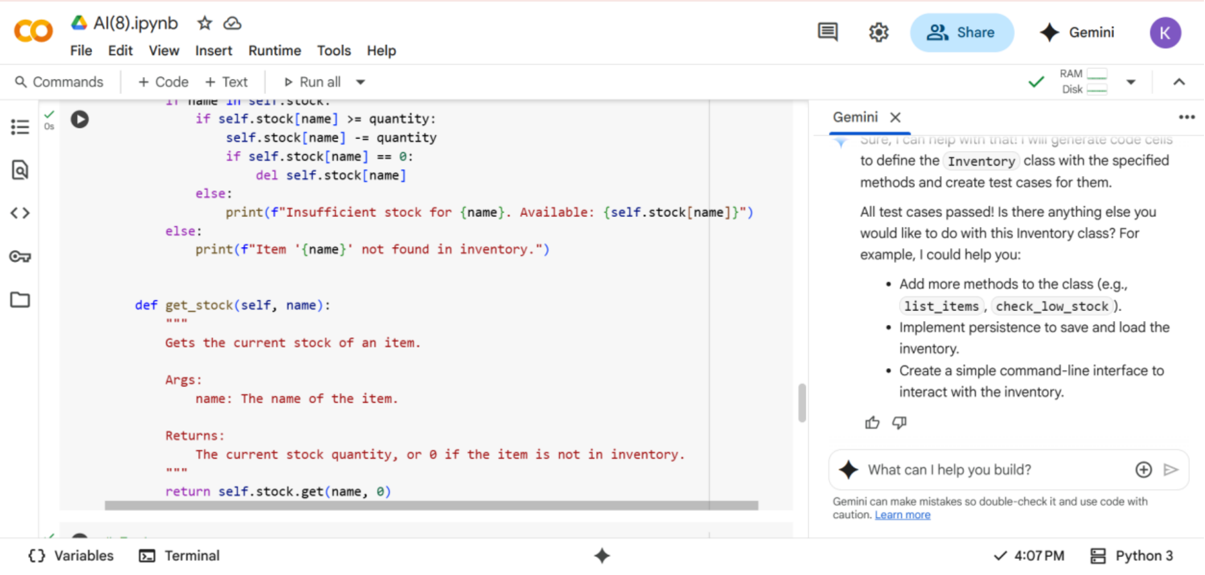
```
[8]             self.stock[name] += quantity
            else:
                self.stock[name] = quantity

        def remove_item(self, name, quantity):
            """
            Removes a specified quantity of an item from the inventory.

            Args:
                name: The name of the item.
                quantity: The quantity to remove.
            """
            if not isinstance(quantity, int) or quantity <= 0:
                print("Quantity must be a positive integer.")
                return

            if name in self.stock:
                if self.stock[name] >= quantity:
                    self.stock[name] -= quantity
                    if self.stock[name] == 0:
                        del self.stock[name]
                else:
                    print(f"Insufficient stock for {name}. Available: {self.stock[name]}")
```

to define the `Inventory` class with the specified methods and create test cases for them.

All test cases passed! Is there anything else you would like to do with this Inventory class? For example, I could help you:

- Add more methods to the class (e.g., `list_items`, `check_low_stock`).
- Implement persistence to save and load the inventory.
- Create a simple command-line interface to interact with the inventory.

Gemini can make mistakes so double-check it and use code with caution. Learn more



```
        if name in self.stock:
            if self.stock[name] >= quantity:
                self.stock[name] -= quantity
                if self.stock[name] == 0:
                    del self.stock[name]
            else:
                print(f"Insufficient stock for {name}. Available: {self.stock[name]}")
        else:
            print(f"Item '{name}' not found in inventory.")


    def get_stock(self, name):
        """
        Gets the current stock of an item.

        Args:
            name: The name of the item.

        Returns:
            The current stock quantity, or 0 if the item is not in inventory.
        """
        return self.stock.get(name, 0)
```

Gemini

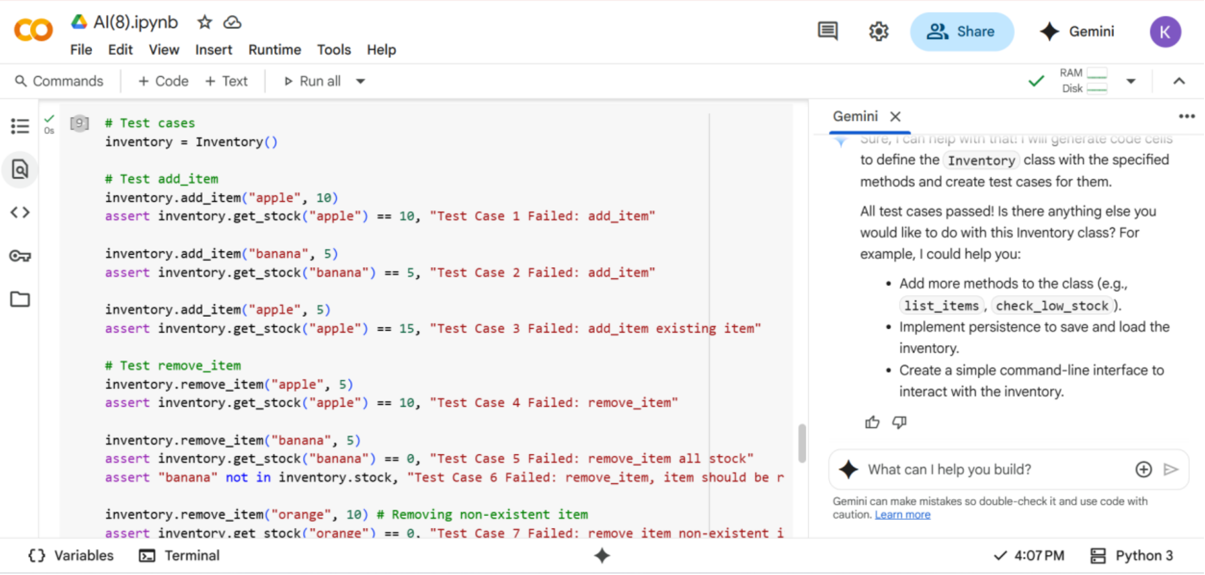
Sure, I can help with that! I will generate code cells to define the `Inventory` class with the specified methods and create test cases for them.

All test cases passed! Is there anything else you would like to do with this Inventory class? For example, I could help you:

- Add more methods to the class (e.g., `list_items`, `check_low_stock`).
- Implement persistence to save and load the inventory.
- Create a simple command-line interface to interact with the inventory.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more



```
# Test cases
inventory = Inventory()

# Test add_item
inventory.add_item("apple", 10)
assert inventory.get_stock("apple") == 10, "Test Case 1 Failed: add_item"

inventory.add_item("banana", 5)
assert inventory.get_stock("banana") == 5, "Test Case 2 Failed: add_item"

inventory.add_item("apple", 5)
assert inventory.get_stock("apple") == 15, "Test Case 3 Failed: add_item existing item"

# Test remove_item
inventory.remove_item("apple", 5)
assert inventory.get_stock("apple") == 10, "Test Case 4 Failed: remove_item"

inventory.remove_item("banana", 5)
assert inventory.get_stock("banana") == 0, "Test Case 5 Failed: remove_item all stock"
assert "banana" not in inventory.stock, "Test Case 6 Failed: remove_item, item should be r

inventory.remove_item("orange", 10) # Removing non-existent item
assert inventory.get_stock("orange") == 0, "Test Case 7 Failed: remove_item non-existent i
```

Gemini

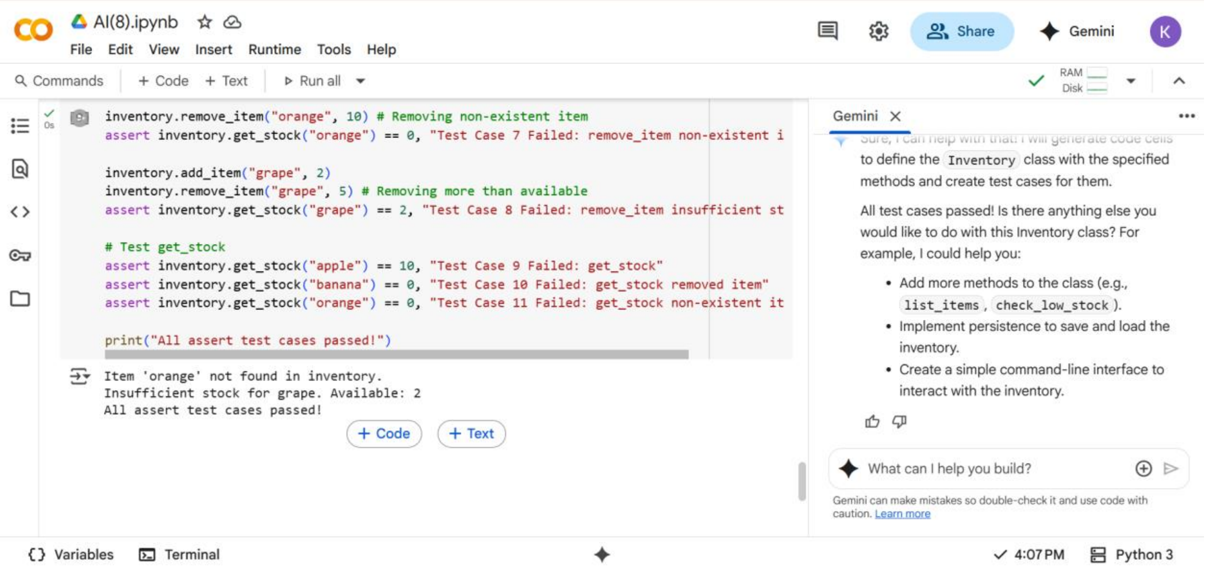
Sure, I can help with that! I will generate code cells to define the `Inventory` class with the specified methods and create test cases for them.

All test cases passed! Is there anything else you would like to do with this Inventory class? For example, I could help you:

- Add more methods to the class (e.g., `list_items`, `check_low_stock`).
- Implement persistence to save and load the inventory.
- Create a simple command-line interface to interact with the inventory.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more



```
inventory.remove_item("orange", 10) # Removing non-existent item
assert inventory.get_stock("orange") == 0, "Test Case 7 Failed: remove_item non-existent i

inventory.add_item("grape", 2)
inventory.remove_item("grape", 5) # Removing more than available
assert inventory.get_stock("grape") == 2, "Test Case 8 Failed: remove_item insufficient st

# Test get_stock
assert inventory.get_stock("apple") == 10, "Test Case 9 Failed: get_stock"
assert inventory.get_stock("banana") == 0, "Test Case 10 Failed: get_stock removed item"
assert inventory.get_stock("orange") == 0, "Test Case 11 Failed: get_stock non-existent it

print("All assert test cases passed!")
```

```
Item 'orange' not found in inventory.
Insufficient stock for grape. Available: 2
All assert test cases passed!
```

Gemini

Sure, I can help with that! I will generate code cells to define the `Inventory` class with the specified methods and create test cases for them.

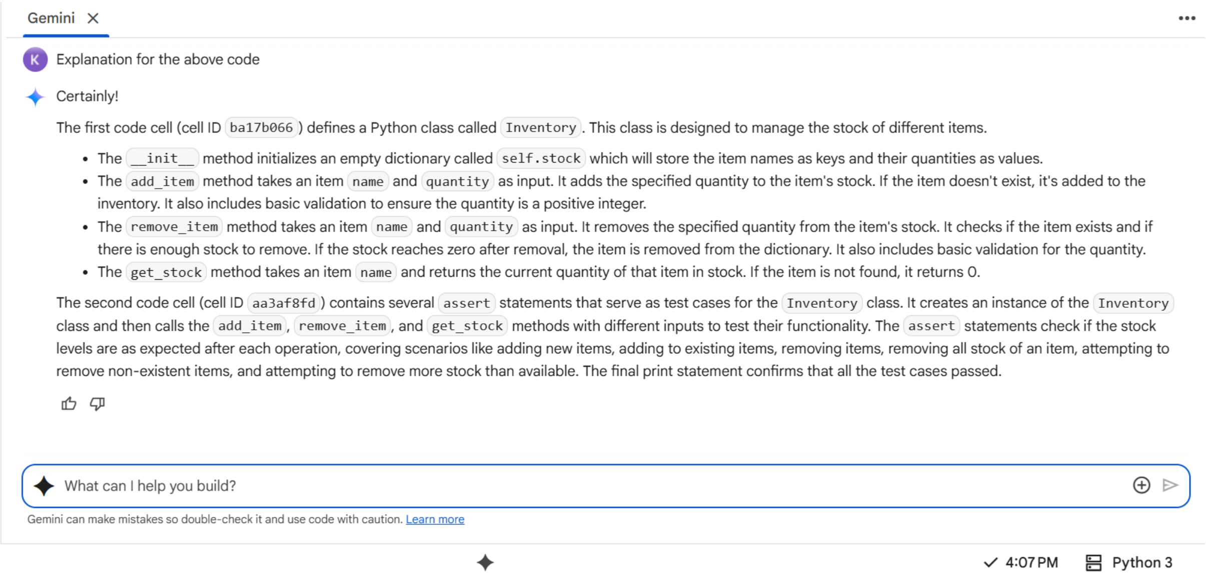
All test cases passed! Is there anything else you would like to do with this Inventory class? For example, I could help you:

- Add more methods to the class (e.g., `list_items`, `check_low_stock`).
- Implement persistence to save and load the inventory.
- Create a simple command-line interface to interact with the inventory.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Explanation:



Explanation for the above code

Certainly!

The first code cell (cell ID `ba17b066`) defines a Python class called `Inventory`. This class is designed to manage the stock of different items.

- The `__init__` method initializes an empty dictionary called `self.stock` which will store the item names as keys and their quantities as values.
- The `add_item` method takes an item `name` and `quantity` as input. It adds the specified quantity to the item's stock. If the item doesn't exist, it's added to the inventory. It also includes basic validation to ensure the quantity is a positive integer.
- The `remove_item` method takes an item `name` and `quantity` as input. It removes the specified quantity from the item's stock. It checks if the item exists and if there is enough stock to remove. If the stock reaches zero after removal, the item is removed from the dictionary. It also includes basic validation for the quantity.
- The `get_stock` method takes an item `name` and returns the current quantity of that item in stock. If the item is not found, it returns 0.

The second code cell (cell ID `aa3af8fd`) contains several `assert` statements that serve as test cases for the `Inventory` class. It creates an instance of the `Inventory` class and then calls the `add_item`, `remove_item`, and `get_stock` methods with different inputs to test their functionality. The `assert` statements check if the stock levels are as expected after each operation, covering scenarios like adding new items, adding to existing items, removing items, removing all stock of an item, attempting to remove non-existent items, and attempting to remove more stock than available. The final print statement confirms that all the test cases passed.

Gemini can make mistakes so double-check it and use code with caution. Learn more

Task 5:

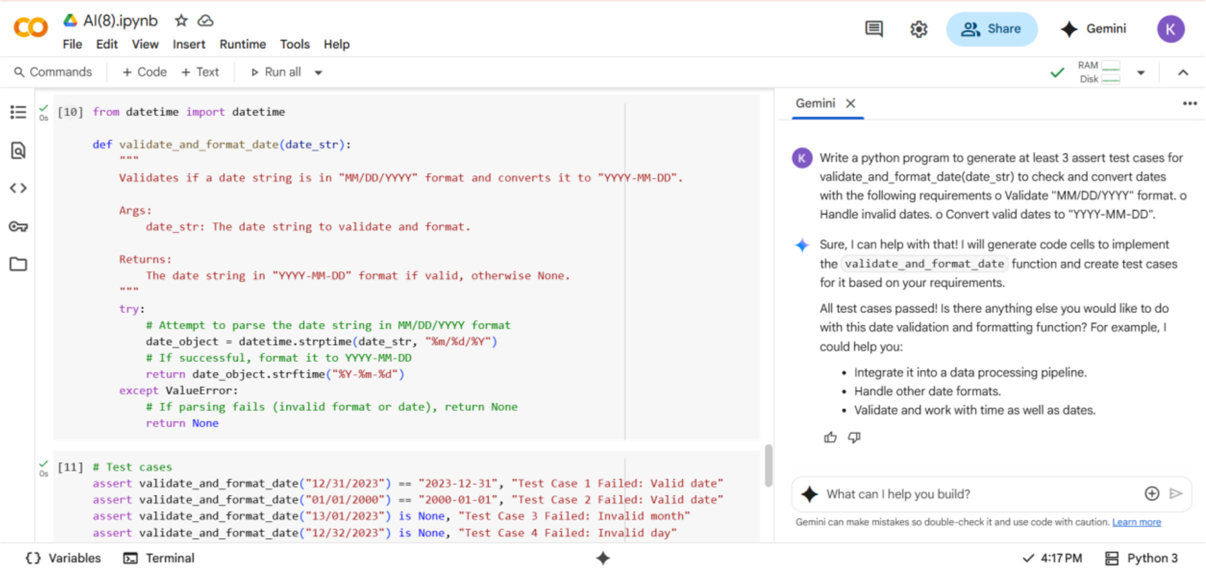
Write a python program to generate at least 3 assert test cases for validate_and_format_date(date_str) to check and convert dates with the following requirements

> Validate "MM/DD/YYYY" format.

>Handle invalid dates.

> Convert valid dates to "YYYY-MM-DD".

Code and Output:



```
[10] from datetime import datetime

     def validate_and_format_date(date_str):
         """
         Validates if a date string is in "MM/DD/YYYY" format and converts it to "YYYY-MM-DD".

         Args:
             date_str: The date string to validate and format.

         Returns:
             The date string in "YYYY-MM-DD" format if valid, otherwise None.
         """
         try:
             # Attempt to parse the date string in MM/DD/YYYY format
             date_object = datetime.strptime(date_str, "%m/%d/%Y")
             # If successful, format it to YYYY-MM-DD
             return date_object.strftime("%Y-%m-%d")
         except ValueError:
             # If parsing fails (invalid format or date), return None
             return None
```

```
[11] # Test cases
     assert validate_and_format_date("12/31/2023") == "2023-12-31", "Test Case 1 Failed: Valid date"
     assert validate_and_format_date("01/01/2000") == "2000-01-01", "Test Case 2 Failed: Valid date"
     assert validate_and_format_date("13/01/2023") is None, "Test Case 3 Failed: Invalid month"
     assert validate_and_format_date("12/32/2023") is None, "Test Case 4 Failed: Invalid day"
```

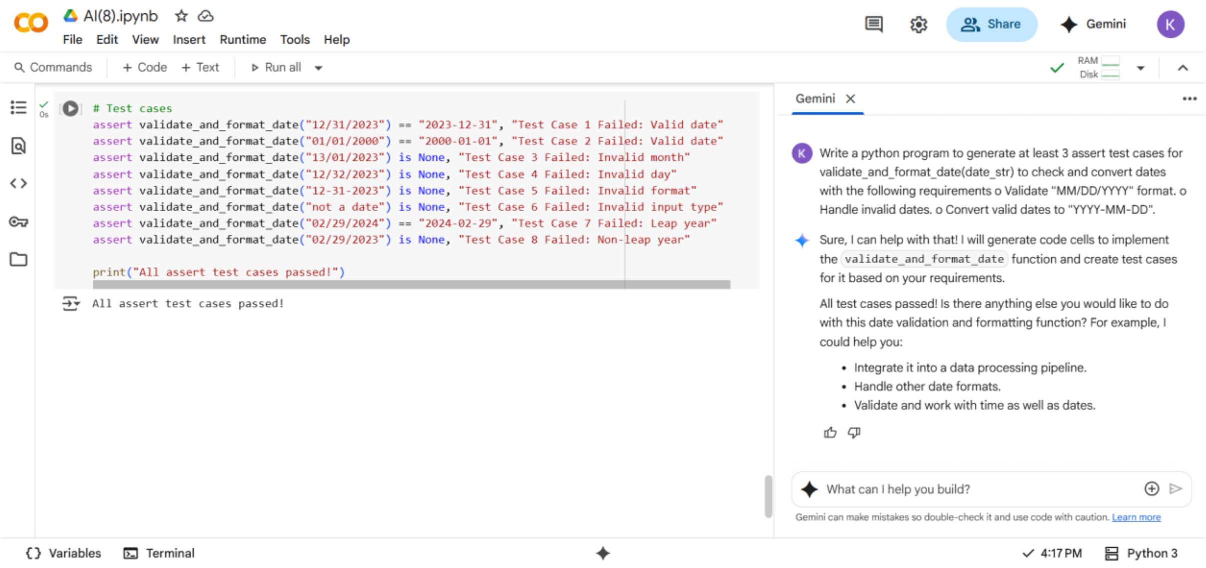
Write a python program to generate at least 3 assert test cases for validate_and_format_date(date_str) to check and convert dates with the following requirements o Validate "MM/DD/YYYY" format. o Handle invalid dates. o Convert valid dates to "YYYY-MM-DD".

Sure, I can help with that! I will generate code cells to implement the `validate_and_format_date` function and create test cases for it based on your requirements.

All test cases passed! Is there anything else you would like to do with this date validation and formatting function? For example, I could help you:

- Integrate it into a data processing pipeline.
- Handle other date formats.
- Validate and work with time as well as dates.

Gemini can make mistakes so double-check it and use code with caution. Learn more



```
# Test cases
assert validate_and_format_date("12/31/2023") == "2023-12-31", "Test Case 1 Failed: Valid date"
assert validate_and_format_date("01/01/2000") == "2000-01-01", "Test Case 2 Failed: Valid date"
assert validate_and_format_date("13/01/2023") is None, "Test Case 3 Failed: Invalid month"
assert validate_and_format_date("12/32/2023") is None, "Test Case 4 Failed: Invalid day"
assert validate_and_format_date("12-31-2023") is None, "Test Case 5 Failed: Invalid format"
assert validate_and_format_date("not a date") is None, "Test Case 6 Failed: Invalid input type"
assert validate_and_format_date("02/29/2024") == "2024-02-29", "Test Case 7 Failed: Leap year"
assert validate_and_format_date("02/29/2023") is None, "Test Case 8 Failed: Non-leap year"

print("All assert test cases passed!")
```

```
All assert test cases passed!
```

Write a python program to generate at least 3 assert test cases for validate_and_format_date(date_str) to check and convert dates with the following requirements o Validate "MM/DD/YYYY" format. o Handle invalid dates. o Convert valid dates to "YYYY-MM-DD".

Sure, I can help with that! I will generate code cells to implement the `validate_and_format_date` function and create test cases for it based on your requirements.

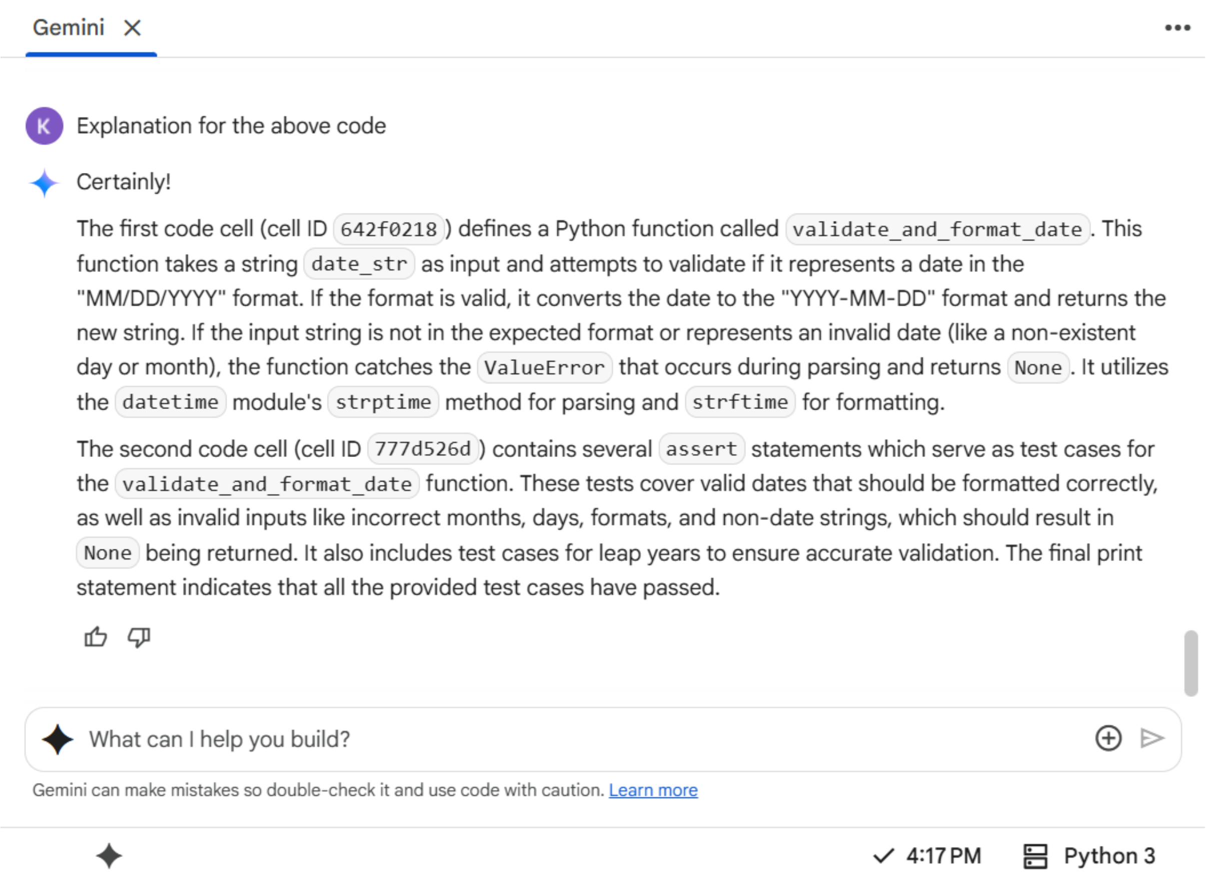
All test cases passed! Is there anything else you would like to do with this date validation and formatting function? For example, I could help you:

- Integrate it into a data processing pipeline.
- Handle other date formats.
- Validate and work with time as well as dates.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Explanation:



Explanation for the above code

Certainly!

The first code cell (cell ID `642f0218`) defines a Python function called `validate_and_format_date`. This function takes a string `date_str` as input and attempts to validate if it represents a date in the "MM/DD/YYYY" format. If the format is valid, it converts the date to the "YYYY-MM-DD" format and returns the new string. If the input string is not in the expected format or represents an invalid date (like a non-existent day or month), the function catches the `ValueError` that occurs during parsing and returns `None`. It utilizes the `datetime` module's `strptime` method for parsing and `strftime` for formatting.

The second code cell (cell ID `777d526d`) contains several `assert` statements which serve as test cases for the `validate_and_format_date` function. These tests cover valid dates that should be formatted correctly, as well as invalid inputs like incorrect months, days, formats, and non-date strings, which should result in `None` being returned. It also includes test cases for leap years to ensure accurate validation. The final print statement indicates that all the provided test cases have passed.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more