

AI Assignment 3:

TASK-1:

- Ask AI to write a function to calculate compound interest, starting with only the function name. Then add a docstring, then input-output example
- Expected Output#1
- Comparison of AI-generated code styles

The screenshot displays a VS Code workspace with a Python file named `fdjhh.py`. The code defines a function `calculate_compound_interest` with a docstring and a main block for user input and calculation. The terminal shows the script's execution with sample inputs and the resulting output. The AI chat panel on the right provides a detailed explanation of the function's logic and the compound interest formula.

```
1 def calculate_compound_interest(principal, rate, time, n):
2     """
3     Calculate the compound interest.
4     principal: initial amount of money
5     rate: annual interest rate (as a decimal, e.g., 0.05 for 5%)
6     time: number of years
7     n: number of times interest applied per year
8     Returns the amount after interest.
9     """
10    amount = principal * (1 + rate / n) ** (n * time)
11    return amount
12
13 if __name__ == "__main__":
14     principal = float(input("Enter the principal amount: "))
15     rate = float(input("Enter the annual interest rate (as a percentage, e.g., 5 for 5%): ") / 100)
16     time = float(input("Enter the number of years: "))
17     n = int(input("Enter the number of times interest applied per year: "))
18     result = calculate_compound_interest(principal, rate, time, n)
19     print("Amount after (time) years: (result)")
20
```

Terminal Output:

```
PS C:\Users\VP\Desktop\wtmp> & C:\Users\HP\AppData\Local\Programs\Python\Python313\python.exe c:\Users\HP\Desktop\wtmp\fdjhh.py
Enter the principal amount: 1000
Enter the annual interest rate (as a percentage, e.g., 5 for 5%): 2
Enter the number of years: 1
Enter the number of times interest applied per year: 1
Amount after 1.0 years: 1020.0
PS C:\Users\VP\Desktop\wtmp>
```

AI Chat Explanation:

User input prompts have been added. Now, when you run `fdjhh.py`, it will ask for the principal, rate, time, and compounding frequency, then print the calculated amount. Let me know if you want further customization!

Workspace

Used 1 reference

The `calculate_compound_interest` function computes the future value of an investment based on the principles of compound interest. It takes four parameters: `principal` (the initial amount of money invested), `rate` (the annual interest rate, expressed as a decimal such as 0.05 for 5%), `time` (the total number of years the money is invested), and `n` (the number of times interest is compounded per year).

Inside the function, the formula used is $\text{principal} * (1 + \text{rate} / n) ** (n * \text{time})$. This is the standard compound interest formula, where the interest is added to the principal at regular intervals, and each time, the new principal includes the interest from previous periods. The expression $(1 + \text{rate} / n)$ calculates the growth factor for each compounding period, and raising it to the power of $(n * \text{time})$ accounts for all compounding periods over the investment's lifetime.

The function returns the total amount after interest has been applied, not just the interest earned. This makes it useful for determining how much an investment will be worth after a certain number of years, given a specific interest rate and compounding frequency. The code is concise and leverages Python's arithmetic operators to directly implement the mathematical formula.

TASK-2:

- Do math stuff, then refine it to: # Write a function to calculate average, median, and mode of a list of numbers.

The image shows a Jupyter Notebook interface with a Python function `calculate_stats` and its output. The function calculates the average, median, and mode of a list of numbers. The output shows the results for three different input lists.

```
from collections import Counter

def calculate_stats(numbers):
    """Calculates the average, median, and mode of a list of numbers.

    Args:
        numbers: A list of numbers.

    Returns:
        A dictionary containing the average, median, and mode.
    """
    if not numbers:
        return {"average": None, "median": None, "mode": None}

    # Calculate average
    average = sum(numbers) / len(numbers)

    # Calculate median
    sorted_numbers = sorted(numbers)
    n = len(sorted_numbers)
    mid_index = n // 2
    if n % 2 == 0:
        median = (sorted_numbers[mid_index - 1] + sorted_numbers[mid_index]) / 2
    else:
        median = sorted_numbers[mid_index]

    # Calculate mode
    counts = Counter(numbers)
    max_count = max(counts.values())
    mode = [num for num, count in counts.items() if count == max_count]
    if len(mode) == len(numbers): # No repeated numbers
        mode = None
    elif len(mode) == 1:
        mode = mode[0] # Return single mode if unique
    return {"average": average, "median": median, "mode": mode}

# Example usage
my_list = [1, 2, 3, 4, 5, 5, 6]
stats = calculate_stats(my_list)
print(stats)

my_list_2 = [1, 1, 2, 3, 3, 4]
stats_2 = calculate_stats(my_list_2)
print(stats_2)

my_list_3 = [1, 2, 3, 4]
stats_3 = calculate_stats(my_list_3)
print(stats_3)
```

Output:

```
{'average': 3.742857142857143, 'median': 4, 'mode': 5}
{'average': 2.3333333333333335, 'median': 2.5, 'mode': [1, 3]}
{'average': 2.5, 'median': 2.5, 'mode': None}
```

The Gemini chat interface on the right explains the code:

Please explain this code:

from collections import Counter

This code defines a Python function called `calculate_stats` that takes a list of numbers as input and returns a dictionary containing the average, median, and mode of those numbers.

Here's a breakdown of the code:

- Import Counter:** It imports the `Counter` class from the `collections` module to easily count the occurrences of each number in the list.
- Function Definition:**
 - It defines the `calculate_stats` function that accepts one argument, `numbers`.
 - It includes a docstring explaining what the function does, its arguments, and what it returns.
- Handle Empty List:**
 - It checks if the input `numbers` list is empty. If it is, it returns a dictionary with `None` for average, median, and mode.
- Calculate Average:**
 - It calculates the average by summing all the numbers in the list and dividing by the total count of numbers.
- Calculate Median:**
 - It sorts the input list to find the median.
 - It determines the middle index based on whether the list has an even or odd number of elements.
 - If the list has an even number of elements, the median is the average of the two middle elements.
 - If the list has an odd number of elements, the median is the middle element.
- Calculate Mode:**
 - It uses `Counter` to count the occurrences of each number.
 - It finds the maximum count among all numbers.
 - It identifies the numbers that have the maximum count (these are the modes).
 - It handles two special cases for the mode:
 - If all numbers appear the same number of times (meaning no repeated numbers), the mode is considered `None`.
 - If there is only one mode, it returns that single value instead of a list.
- Return Results:** The function returns a dictionary containing the calculated `average`, `median`, and `mode`.
- Example Usage:** The code includes example calls to the `calculate_stats` function with different lists and prints the results.

What can I help you build?

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

TASK-3:

- Provide multiple examples of input-output to the AI for `convert_to_binary(num)` function. Observe how AI uses few-shot prompting to generalize.

The screenshot shows a VS Code editor with a file named `FTF.py` open. The code defines a function `to_binary` that takes an integer `n` and returns its binary representation as a string, removing the `0b` prefix. The function uses `bin(n)` and slicing `[2:]` to achieve this. Below the function definition, there are example inputs and their corresponding binary outputs.

```
1 def to_binary(n):
2     return bin(n)[2:]
3
4 # Example usage:
5 examples = [5, 10, 255, 0, 1024]
6 for num in examples:
7     print(f"{num} in binary is {to_binary(num)}")
```

The terminal output shows the results of running the code:

```
PS C:\Users\VP\Desktop\utmp> & c:\Users\HP\AppData\Local\Programs\Python\Python313\python.exe c:\Users\HP\Desktop\utmp\FTF.py
5 in binary is 101
10 in binary is 1010
255 in binary is 11111111
0 in binary is 0
1024 in binary is 10000000000
PS C:\Users\VP\Desktop\utmp>
```

The chat window on the right provides an explanation of the function. It notes a potential "gotcha" for beginners: understanding why `[2:]` is used. The `bin()` function always includes the `0b` prefix to indicate a binary literal, but for display or further processing, you often want just the digits. The code handles this by slicing off the first two characters.

Workspace

Used 1 reference

This code defines a Python function named `to_binary` that takes a single argument `n`. The purpose of the function is to convert the integer value `n` into its binary string representation.

Inside the function, the built-in `bin()` function is used. When you pass an integer to `bin()`, it returns a string that represents the number in binary format, prefixed with `0b`. For example, `bin(5)` returns the string `'0b101'`. However, in many cases, you may only want the binary digits themselves, without the `0b` prefix.

To achieve this, the code uses slicing: `[2:]`. This removes the first two characters from the string returned by `bin()`, effectively stripping away the `0b` prefix and leaving just the binary digits. So, for `n = 5`, `bin(n)[2:]` would return `'101'`.

Overall, this function provides a concise way to get the binary representation of an integer as a plain string, which can be useful for display, storage, or further processing in your code.

Agent: GPT-4.1

TASK-4:

- Create an user interface for an hotel to generate bill based on customer requirements

The screenshot displays a Jupyter Notebook environment with a Python script for a hotel bill generator. The script uses the `ipywidgets` library to create an interactive UI. The output shows a form with fields for Room Number, Customer Name, Duration of Stay, Service Selection, and Quantity, along with an 'Add Service' button. The current state of the form shows '021' for Room Number, 'karthik' for Customer Name, '02' for Duration of Stay, 'Restaurant' for Service Selection, and '02' for Quantity. The output area displays 'Added 2 of Restaurant'.

```
import ipywidgets as widgets
from IPython.display import display

# Create input fields for customer details
room_number_input = widgets.Text(description="Room Number:")
customer_name_input = widgets.Text(description="Customer Name:")
stay_duration_input = widgets.IntText(description="Duration of Stay (nights):")

# Create a simple service selection
service_options = [('Restaurant', 50), ('Laundry', 20), ('Spa', 100)]
service_dropdown = widgets.Dropdown(options=service_options, description="Select Service:")
service_quantity = widgets.IntText(description="Quantity:")

# Button to add service
add_service_button = widgets.Button(description="Add Service")

# Output area for the bill
bill_output = widgets.Output()

# Display the input fields and buttons
display(room_number_input, customer_name_input, stay_duration_input, service_dropdown, service_quantity, add_service_button, bill_output)

# Dictionary to store added services
added_services = {}

def add_service():
    service_name = service_dropdown.label
    price_per_unit = service_dropdown.value
    quantity = service_quantity.value

    if service_name in added_services:
        added_services[service_name]['quantity'] += quantity
    else:
        added_services[service_name] = {'price': price_per_unit, 'quantity': quantity}

    with bill_output:
        bill_output.clear_output()
        print(f"Added {quantity} of {service_name}")

add_service_button.on_click(add_service)
```

Room Num... 021
Customer... karthik
Duration of... 02
Select Serv... Restaurant
Quantity... 02
Add Service
Added 2 of Restaurant

Genius X
Please let me know how you would like to proceed given this limitation.

1 Please explain this code:

`import ipywidgets as widgets`

This code cell uses the `ipywidgets` library to create a simple interactive interface in the notebook. Here's a breakdown of what it does:

- Imports:** It imports the necessary libraries: `ipywidgets` for creating interactive elements and `display` from `IPython.display` to show these elements in the notebook.
- Input Fields:** It creates several input widgets:
 - `room_number_input`: A text box for entering the room number.
 - `customer_name_input`: A text box for entering the customer's name.
 - `stay_duration_input`: A numerical input box for entering the duration of stay in nights.
- Service Selection:**
 - `service_options`: A list of tuples defining service options with their names and prices.
 - `service_dropdown`: A dropdown menu populated with the `service_options`.
 - `service_quantity`: A numerical input box for entering the quantity of the selected service.
- Add Service Button:**
 - `add_service_button`: A button labeled "Add Service".
- Output Area:**
 - `bill_output`: An output widget where messages or the bill details will be displayed.
- Display Widgets:** It uses the `display()` function to render all the created widgets in the notebook output.
- added_services Dictionary:** An empty dictionary is initialized to store the services added by the user.
- add_service Function:** This function is designed to be called when the `add_service_button` is clicked.
 - It retrieves the selected service's name (`label`), price (`value`), and the entered quantity.
 - It checks if the service already exists in the `added_services` dictionary.
 - If it exists, it updates the quantity.
 - If it doesn't exist, it adds the service to the dictionary with its price and quantity.
 - It clears the `bill_output` area and prints a confirmation message indicating the service and quantity added.
- Button Event Handling:**
 - `add_service_button.on_click(add_service)`: This line connects the `add_service` function to the `add_service_button`. When the button is clicked, the `add_service` function will be executed.

In summary, this code sets up a basic interactive form using `ipywidgets` to collect customer and service information for a hotel billing system, and it includes a function to add selected services to a dictionary when a button is clicked.

What can I help you build?

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

Variables Terminal 4:20 PM Python 3

TASK-5:

● Analyzing Prompt Specificity: Improving Temperature Conversion Function with Clear Instructions

The image shows a Jupyter Notebook on the left and a Gemini AI chat interface on the right. The Jupyter Notebook contains a Python function `convert_temperature` that converts temperatures between Celsius, Fahrenheit, and Kelvin. The function takes three arguments: `temp` (the temperature value), `from_unit` (the unit of the input temperature), and `to_unit` (the desired unit for the output temperature). The function uses a series of `if` and `elif` statements to check the units and apply the appropriate conversion formula. Example usage is provided at the bottom of the code block.

```
[ ] Start coding or generate with AI.

def convert_temperature(temp, from_unit, to_unit):
    """Converts temperature between Celsius, Fahrenheit, and Kelvin.

    Args:
        temp: The temperature value.
        from_unit: The unit of the input temperature ('C', 'F', or 'K').
        to_unit: The desired unit for the output temperature ('C', 'F', or 'K').

    Returns:
        The converted temperature value, or None if the units are invalid.
    """
    if from_unit == 'C':
        if to_unit == 'F':
            return (temp * 9/5) + 32
        elif to_unit == 'K':
            return temp + 273.15
        elif to_unit == 'C':
            return temp
    elif from_unit == 'F':
        if to_unit == 'C':
            return (temp - 32) * 5/9
        elif to_unit == 'K':
            return (temp - 32) * 5/9 + 273.15
        elif to_unit == 'F':
            return temp
    elif from_unit == 'K':
        if to_unit == 'C':
            return temp - 273.15
        elif to_unit == 'F':
            return (temp - 273.15) * 9/5 + 32
        elif to_unit == 'K':
            return temp
    return None

# Example usage:
celsius_temp = 25
fahrenheit_temp = convert_temperature(celsius_temp, 'C', 'F')
print(f"{celsius_temp}°C is {fahrenheit_temp}°F")

fahrenheit_temp = 77
celsius_temp = convert_temperature(fahrenheit_temp, 'F', 'C')
print(f"{fahrenheit_temp}°F is {celsius_temp}°C")

kelvin_temp = 300
celsius_temp = convert_temperature(kelvin_temp, 'K', 'C')
print(f"{kelvin_temp}K is {celsius_temp}°C")

25°C is 77.0°F
77°F is 25.0°C
300K is 26.850000000000002°C
```

The Gemini AI chat interface on the right shows a prompt: "Please explain this code:" followed by the function definition. The AI responds with a detailed explanation of the function's arguments, its logic, and the conversion formulas used. It also provides a breakdown of the formulas and an example of the function's output.

Please explain this code:

```
<> def convert_temperature(temp, from_unit, to_unit):
```

Sure! I can explain the selected code cell.

The code defines a Python function called `convert_temperature` that takes three arguments:

- `temp`: The temperature value you want to convert.
- `from_unit`: The unit of the input temperature ('C' for Celsius, 'F' for Fahrenheit, or 'K' for Kelvin).
- `to_unit`: The desired unit for the output temperature ('C', 'F', or 'K').

The function uses a series of `if` and `elif` statements to check the `from_unit` and `to_unit` and apply the appropriate conversion formula.

Here's a breakdown of the conversion formulas used:

- Celsius to Fahrenheit: $(temp * 9/5) + 32$
- Celsius to Kelvin: $temp + 273.15$
- Fahrenheit to Celsius: $(temp - 32) * 5/9$
- Fahrenheit to Kelvin: $(temp - 32) * 5/9 + 273.15$
- Kelvin to Celsius: $temp - 273.15$
- Kelvin to Fahrenheit: $(temp - 273.15) * 9/5 + 32$

If the `from_unit` and `to_unit` are the same, the function returns the original temperature.

If the provided `from_unit` or `to_unit` are not 'C', 'F', or 'K', the function returns `None`.

The code then provides example usage of the function, demonstrating conversions from Celsius to Fahrenheit, Fahrenheit to Celsius, and Kelvin to Celsius, and prints the results.

What can I help you build?

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