Data Structure

Solution:

To solve this problem, we can iterate over the courses in the `student_data` dictionary, calculate the product of grade and credits for each course, and accumulate the total grade points and total credits. Finally, we divide the total grade points by the total credits to calculate the GPA.

Here's the solution code:

```python
def calculate_gpa(student_data):
    total_grade_points = 0
    total_credits = 0

    for course, course_data in student_data["courses"].items():
        grade = course_data["grade"]
        credits = course_data["credits"]
        total_grade_points += grade * credits
        total_credits += credits

    gpa = total_grade_points / total_credits
    return gpa
```

Explanation:

The `calculate_gpa` function takes the `student_data` dictionary as input. It initializes two variables, `total_grade_points` and `total_credits`, to keep track of the cumulative grade points and credits.

We then iterate over each course in the nested `courses` dictionary using the `.items()` method. For each course, we extract the grade and credits using the corresponding keys (`"grade"` and `"credits"`). We multiply the grade by the credits and add the result to `total_grade_points`. We also increment `total_credits` by the course credits.

After iterating through all the courses, we calculate the GPA by dividing `total_grade_points` by `total_credits`. The result is then rounded to two decimal places using the `round()` function.

Finally, the calculated GPA is returned as the output of the `calculate_gpa` function.

Interpreting Exception

Solution:

(a) The traceback for the given code will display a `TypeError` with a message indicating that the unsupported operand type occurred on the line where the division operation is performed (`average = total_sum / len(numbers)`) in function calculate_average. The cause of the error is that the cause of the exception is the presence of a string value ("25") in the `numbers` list. The division operation expects numeric values, and trying to divide an integer by a string results in a `TypeError`.

Explanation:

The traceback reveals the following information:

1. The exception type is `TypeError`.
2. The exception was raised in the `calculate_average` function, specifically on line Y.
3. The unsupported operation is division (`+`) in the sum function.

From this information, we can conclude that the cause of the exception is the presence of a string value ("25") in the `numbers` list. The plus operation in the sum function expects numeric values, and trying to plus an integer by a string results in a `TypeError`.

(b) To fix the issue, we need to ensure that all elements in the `numbers` list are of numeric type. In this case, the string "25" should be removed or converted to an integer before performing calculations. Here's an updated code snippet with the fix:

```python
def calculate_average(numbers):
    total_sum = sum(numbers)
    average = total_sum / len(numbers)
    return average

numbers = [5, 10, 15, 20, 25]  # Removed the string "25"
result = calculate_average(numbers)
print("The average is:", result)
```

By removing the string value or converting it to an integer, the code will execute without any exceptions, and the correct average will be calculated and printed.

Debugging

Solution:

The bug in the code lies in the initialization of the `max_value` variable to 0. This initialization assumes that all numbers in the list are positive and greater than 0. However, if the list contains negative numbers, the code will fail to identify the correct maximum value.

To fix this issue, we can initialize `max_value` with the first number from the list instead of 0. This way, we start with a valid maximum value, and then we compare and update it as we iterate through the rest of the numbers.

Here's the corrected code:

```python
def find_maximum(numbers):
    if len(numbers) == 0:
        return None

    max_value = numbers[0]
    for number in numbers:
        if number > max_value:
            max_value = number
    return max_value

numbers = [4, 9, 3, 6, 12, 7, 5]
result = find_maximum(numbers)
print("The maximum value is:", result)
```

Testing

Solution:

```python
def test_calculate_factorial(n):

# Test Cases
# Test Case 1: Calculate factorial of 0 (edge case)
# Expected Output: 1
assert calculate_factorial(0) == 1

# Test Case 2: Calculate factorial of 5 (normal case)
# Expected Output: 120 (5! = 5 * 4 * 3 * 2 * 1 = 120)
assert calculate_factorial(5) == 120

# Test Case 3: Calculate factorial of 10 (large input)
# Expected Output: 3628800 (10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 3628800)
assert calculate_factorial(10) == 3628800

# Additional Test Cases
# Test Case 4: Calculate factorial of 1 (minimum input)
# Expected Output: 1
assert calculate_factorial(1) == 1

# Test Case 5: Calculate factorial of -1 (edge case)
# Expected Output: None
assert calculate_factorial(-1) == None
```

Test cases that cover edge cases are essential to verify the behavior of a program at its limits. For this particular question, your solution should contain at least one normal case, one large input, and one edge case.

Graph

Solution:

All individuals (Alice, Bob, Charlie, David, and Emma) in the social network have the same number of friends.

Explanation:

To find the person with the highest number of friends in the social network, we need to determine the node with the highest degree. The degree of a node represents the number of edges connected to that node, which corresponds to the number of friends in this context.

By examining the provided graph, we can see that all nodes have the same degree, indicating that each person has an equal number of friends. Therefore, there is no person with a higher number of friends than others. In this case, all individuals in the social network have the same number of edges.

Class

Solution:

The expected output will be:

```
Bookstore Inventory:
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Price: 10.99
Title: To Kill a Mockingbird
Author: Harper Lee
Price: 7.99
Title: Pride and Prejudice
Author: Jane Austen
Price: 6.99
```

Explanation:

The provided code snippet defines two classes: `Book` and `Bookstore`.

The `Book` class represents a book in the bookstore inventory. It has an `__init__` method that initializes the `title`, `author`, and `price` attributes of a book object. It also has a `display_info` method that prints the title, author, and price of a book.

The `Bookstore` class represents a bookstore and manages its inventory. It has an `__init__` method that initializes the `inventory` attribute as an empty list. The `add_book` method adds a book to the bookstore's inventory by appending it to the `inventory` list. The `display_inventory` method prints the information of all the books in the inventory by calling the `display_info` method of each book.

In the code snippet, a `Bookstore` object named `bookstore` is created. Three `Book` objects (`book1`, `book2`, and `book3`) are created with different titles, authors, and prices. Each book is added to the bookstore's inventory using the `add_book` method.

When the `display_inventory` method is called on the `bookstore` object, it will print the information of all the books in the inventory.

The output will display the title, author, and price of each book in the bookstore's inventory. Each book's information will be printed on separate lines, and the entire inventory will be preceded by the heading "Bookstore Inventory:".

Function

Solution:

In the context of CSE160, functions are fundamental building blocks of Python programming. They play a crucial role in the development of modular, reusable, and organized code. Here are several reasons why functions are essential in Python programming within the context of CSE160:

1. **Code Reusability and Modularity:** Functions allow you to encapsulate a block of code that performs a specific task or operation. By organizing code into functions, you can reuse that functionality multiple times throughout your program or in different programs altogether. This reusability promotes modularity, making code more manageable, maintainable, and easier to understand.

2. **Abstraction and Encapsulation:** Functions provide a level of abstraction by hiding the internal implementation details of a particular task behind a function name and its parameters. This abstraction allows you to focus on using the function without worrying about how it works internally. Encapsulating code within functions helps manage complexity and promotes a more structured and logical approach to programming.

3. **Code Readability and Maintainability:** Functions improve the readability and maintainability of code by breaking down complex tasks into smaller, self-contained units. Each function can have a specific purpose, making the code easier to understand, review, and modify. Well-designed functions with descriptive names and clear interfaces enhance code comprehension, even for other programmers who may work on the codebase.

4. **Code Organization and Structure:** Functions provide a way to organize code logically and hierarchically. By separating code into functions, you can group related operations together, promoting a modular and structured approach. This structure improves code organization, makes it easier to locate and modify specific functionality, and enhances code clarity and maintainability.

5. **Code Testing and Debugging:** Functions facilitate the testing and debugging process. By dividing your code into smaller functions, you can test each function individually to ensure it produces the desired output. Functions also make it easier to isolate.

Sharing (by reference)

Solution:

The final output of the program will be:

```
{"list": [5, 6, 7, 4], "nested_dict": {"key": "new value"}}
```

Explanation:

In the provided code snippet, we have a function called `modify_data` that takes a dictionary (`data`) as a parameter. Inside the function, three operations are performed on the `data` dictionary.

First, the `append` method is used to add the integer `4` to the list stored in `data["list"]`. This modifies the list in place and adds `4` as a new element at the end of the list.

Next, the value associated with the key `"key"` in `data["nested_dict"]` is changed to `"new value"`. This modifies the nested dictionary within `data`.

Finally, a new dictionary object is assigned to `data`, overwriting the original dictionary. This reassignment creates a new dictionary object and does not modify the original dictionary passed as an argument.

In the main code, a nested data structure is created with a list `[5, 6, 7]` and a nested dictionary `{"key": "old value"}`. The `modify_data` function is called with this data structure as an argument.

Now, let's analyze the pass-by-reference and pass-by-value behavior in this code. In Python, mutable objects like lists and dictionaries are passed by reference. This means that modifications made to the object within the function can affect the original object. However, when a new object is assigned to the parameter within the function, it creates a new local object that does not modify the original object.

In this case, the list within the dictionary (`data["list"]`) is modified in place by appending `4`, and the nested dictionary (`data["nested_dict"]`) is modified by changing the value associated with the key `"key"`. These modifications affect the original data structure outside the function since it is passed by reference.

However, when the function reassigns the parameter `data` to a new dictionary, it creates a new object that is local to the function scope. This reassignment does not modify the original data structure passed as an argument.

The `data_structure` dictionary, which was modified by appending `4` to the list and changing the value associated with the key `"key"`, will reflect these modifications. The reassignment of `data` within the function does not affect the `data_structure` dictionary outside the function.

Performance

Solution:

Both implementations solve the `two_sum` problem, which involves finding a pair of numbers that sum up to a given target value. However, the second_implementation offers a more efficient solution, especially for larger input lists, as it performs the task in a single pass through the list instead of nested iterations.

Explanation:

In the first_implementation, we use nested loops to iterate over each pair of numbers in the `nums` list. We check if the sum of the two numbers equals the target value. If a match is found, we return the indices of the two numbers. This implementation has a runtime complexity of O(n^2), as the number of iterations grows quadratically with the size of the input list.

In the second_implementation, we utilize a dictionary (`num_map`) to store the complement of each number encountered while iterating through the `nums` list. For each number, we calculate its complement with respect to the target value. If the complement is present in the dictionary, it means we have found a pair that sums up to the target value, and we return the corresponding indices. Otherwise, we store the number and its index in the dictionary. This implementation has a runtime complexity of O(n), as we iterate through the list only once.