**Assignment-1:**

**Code1:**

def reverse\_string(s):

return s[::-1]

def main():

input\_string = "Hello, world!"

reversed\_string = reverse\_string(input\_string)

print(f"Reversed string: {reversed\_string}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

In the previous code there’s a small improvement you can make to the ‘reverse\_string’function.

In this version, the ‘**reverse\_string’** function is simplified using slicing (‘**s[::-1]’**), making the code more concise and Pythonic. This change doesn't affect the functionality but is considered a cleaner and more idiomatic way to reverse a string in Python.

**Code2:**

def get\_age():

age\_str = input("Please enter your age: ")

if age\_str.isnumeric() and int(age\_str) >= 18:

return int(age\_str)

else:

return None

def main():

age = get\_age()

if age >18:

print(f"You are {age} years old and eligible.")

else:

print("Invalid input. You must be at least 18 years old.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

In the above we are having the errors like

1. The ‘input’function returns a string,so when comparing ‘age’ with a numeric value,you need to convert ‘age’ to an integer first.
2. The condition ‘age.isnumeric()’ checks if the string consists of only numeric characters,but it doesn’t guarantee that the value is greater than or equal to 18.

Changes made:

Renamed the variable age to age\_str to make it clear that it is a string.

Checked if int(age\_str) >= 18 to ensure the input is both numeric and greater than or equal to 18.

Used is not None instead of if age to explicitly check for None. This makes the condition more readable.

**Code3:**

def read\_and\_write\_file(filename):

try:

with open(filename, 'r') as file:

content = file.read()

with open(filename, 'w') as file:

file.write(content.upper())

print(f"File '{filename}' processed successfully.")

except Exception as e:

print(f"An error occurred: {str(e)}")

def main():

filename = "sample.txt"

read\_and\_write\_file(filename)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Your code has a potential issue. When you open a file with the mode **'w'** (write), it truncates the file, meaning it erases the existing content. So, if you read the content and then immediately write it back, you will lose the original content of the file.

This modification retains the original content of the file while creating a new file with the uppercase content. If you want to keep both versions (original and uppercase) in the same file, you can use a different approach or store the modified content in a new file.

**Code4:**

def merge\_sort(arr):

if len(arr) <= 1:

return arr

mid = len(arr) // 2

left = arr[:mid]

right = arr[mid:]

merge\_sort(left)

merge\_sort(right)

i = j = k = 0

while i < len(left) and j < len(right):

if left[i] < right[j]:

arr[k] = left[i]

i += 1

else:

arr[k] = right[j]

j += 1

k += 1

while i < len(left):

arr[k] = left[i]

i += 1

k += 1

while j < len(right):

arr[k] = right[j]

j += 1

k += 1

# Example usage

arr = [38, 27, 43, 3, 9, 82, 10]

merge\_sort(arr)

print(f"The sorted array is: {arr}")

The issue in the provided code is related to the recursive calls to ‘**merge\_sort’**. In the current implementation, the sorted subarrays ‘**left’** and ‘**right’** are obtained through recursive calls, but the sorted content is not assigned back to ‘**left’** and ‘**right**.’ As a result, the merging step uses the original, unsorted content of ‘**left’** and ‘**right’**.

Now, the corrected code properly assigns the sorted content back to the ‘**left’** and ‘**right’** subarrays during the recursive calls, ensuring that the merging step operates on sorted subarrays.

**Assignment-2:**

**Code:**

class InventoryManager:

def \_\_init\_\_(self):

self.inventory = {}

def add\_item(self, name, quantity, price):

if name in self.inventory:

print(f"{name} already exists in the inventory.")

else:

self.inventory[name] = {'quantity': quantity, 'price': price}

print(f"{name} added to the inventory.")

def update\_quantity(self, name, new\_quantity):

if name in self.inventory:

self.inventory[name]['quantity'] += new\_quantity

print(f"Quantity of {name} updated to {self.inventory[name]['quantity']}.")

else:

print(f"{name} not found in the inventory.")

def view\_inventory(self):

print("Current Inventory:")

for item, details in self.inventory.items():

print(f"{item}: Quantity - {details['quantity']}, Price - ${details['price']}")

def remove\_item(self, name):

if name in self.inventory:

del self.inventory[name]

print(f"{name} removed from the inventory.")

else:

print(f"{name} not found in the inventory.")

def main():

manager = InventoryManager()

while True:

print("\nMenu:")

print("1. Add new item")

print("2. Update item quantity")

print("3. View current inventory")

print("4. Remove item")

print("5. Exit")

choice = input("Enter your choice (1-5): ")

if choice == '1':

name = input("Enter item name: ")

quantity = int(input("Enter quantity: "))

price = float(input("Enter price: "))

manager.add\_item(name, quantity, price)

elif choice == '2':

name = input("Enter item name: ")

new\_quantity = int(input("Enter new quantity: "))

manager.update\_quantity(name, new\_quantity)

elif choice == '3':

manager.view\_inventory()

elif choice == '4':

name = input("Enter item name to remove: ")

manager.remove\_item(name)

elif choice == '5':

print("Exiting the program.")

break

else:

print("Invalid choice. Please enter a number between 1 and 5.")

if \_\_name\_\_ == "\_\_main\_\_":

main()