Nasir Jama (25815104)
Problem Solving and Programming

Video Demonstration (Multiple Sources):

https://jumpshare.com/s/UsZb8dlHvDGqpqI380fg

https://www.youtube.com/watch?v=T9t2UXCKPGc

https://mymedia.northampton.ac.uk/media/AS1+Task/0_l4bh7c1c

Contents

Task 1: Inputs & Loops

Requirements:

The requirements of task 1 involves creating an embedded computer program that converts kilometres into miles. The program must request a manual input process from the user for a numeric input value in kilometres to initialise the conversion, validate that the user has inputted a numeric input value and display the output value in 2 decimal points.

Design:

Abstraction, one of the fundamental concepts in computer science means simplifying a computer system and focusing only on the important details, ignoring the parts that are not essential to solving the problem.

Initial Design (Pseudocode):

STRING distance = INPUT “Enter a distance in kilometres or type stop”

FLOAT conversion_rate = 0.621371

FLOAT miles = distance * conversion_rate

PRINT miles, 2 Decimal Place

Knowing the purpose of the task is to create a program that converts a distance in kilometres into miles, allows me to understand the first approach which is identifying the program purpose.

Prototype:

```
InputText = "Enter a distance in kilometres or type stop: "
UserInput = input(InputText)

Miles = float(UserInput) * 0.621371
print(f"{round(Miles, 2)} miles")
```

This Python prototype is a simple distance converter that asks the user to enter a distance in kilometres, converts it to miles, and displays the result. It first defines a prompt message (InputText) and takes the user's input as text (UserInput).

The input is then converted to a floating point number and multiplied by 0.621371, the conversion factor from kilometres to miles, storing the result in the variable Miles. Finally, the program prints the converted distance rounded to two decimal places using an f-string, displaying the output in the format like “6.21 miles.”

Evaluation:

The strength of this prototype is that it correctly performs the core conversion from kilometres to miles using the accurate conversion factor (1 km = 0.621371 miles) and presents the result rounded to two decimal places, making the output clear and user friendly. It also demonstrates good use of variables and formatted strings, showing the fundamental logic of input, processing, and output.

However, its weaknesses are that it lacks a loop to allow repeated conversions, meaning the program runs only once and then stops. It also does not include input validation, so if the user types a non-numeric value (like “stop” or any text), the program will crash with an error.

To fully meet the task requirements, it would need a while loop for repeated conversions, an option for the user to exit by typing something like “stop,” and input validation to check that the entered value is numeric before performing the conversion.

Implementation & Improvement:

The purpose of this new design program is to enhance functionality by allowing multiple conversions within a single session while ensuring reliable and accurate results. It validates user input to prevent errors, handles the “stop” command to exit the program gracefully, and displays the converted distances clearly formatted to two decimal places. By implementing these improvements, the program becomes more robust, user friendly, and efficient in converting distances from kilometres to miles.

Improved Pseudocode:

```
REPEAT
    INPUT number_1
    INT conversion_rate = 0.621371

    IF TYPE(number_1) IS NOT NUMERIC THEN
        PRINT "Invalid input. Exiting..."
        BREAK
    ELSE
        number_1 = number_1 * conversion_rate
        PRINT "Converted value:", number_1
    END IF
    CONTINUE = ASK USER "Do you want to continue? (YES/NO)"
UNTIL CONTINUE == "NO"
```

This pseudocode starts a loop that keeps running until the user says they don’t want to continue. It first asks the user to input a value (number_1). Then it sets a constant called conversion_rate equal to 0.621371, which is the conversion factor from kilometres to miles. The program checks if the user’s input is numeric — if it isn’t, it prints an error message and stops. If the input is valid, it multiplies that number by the conversion rate to get the converted value, then prints the result. Finally, it asks the user if they want to continue; if the user enters “NO,” the loop ends, otherwise it repeats the process.

During the implementation stage, the pseudocode was translated into a fully functional Python program. The process began by defining a Boolean control variable named Working, which allowed the program to repeatedly execute the main loop using the while statement.

Inside the loop, the input() function was used to request a distance in kilometres from the user, and the response was stored in a variable called UserInput. To allow users to stop the program at any time, an if condition checked whether the input matched the word “stop” (after using .strip() to remove spaces and .lower() to handle case sensitivity).

The conversion was then performed inside a try block, where the program attempted to convert the input to a floatingpoint number and multiply it by 0.621371 to calculate miles. The except ValueError block was

added to handle any invalid or non-numeric inputs gracefully, displaying an appropriate error message instead of crashing.

Finally, the converted distance was rounded to two decimal places using the round() function and displayed with an f-string for clear formatting. This implementation successfully met the design goals by combining user interaction, looping, input validation, and arithmetic operations in a structured and logical manner.

Final Product

```
'''
Program: Task 1
Author: Abdinasir Jama
Date: October 2025
Description: This program allows the user to convert kilometeres to miles
'''


Working = True
# Working is a variable stored preferrably stored in cpu register, cache memory / ram
# Working is set to the data type of True (which are of two binary states of 0 and 1)
# it is most likely set to one in binary data (guessing by computer conventions)
while Working:
    # I am using this variable working to create a loop using the key-term "while"
    # The while loop continously runs depending on statement its given is true (1)
    InputText = "Enter a distance (km): "
    UserInput = input(InputText)

    # The variable input text is used to be put through into the method "input"
    # This method requests a user input from the user through the terminal tab

    if UserInput.strip().lower() == "stop":
      """
      I used the method "strip" because I learnt that it removes spaces and then
      set what the user has inputed to lower-case and that it matches "stop"

       This dosn't permantly change what the user has inputed instead checks
       if condition matches
      """

      break
      # This set the statment in the while loop to false and permantly breaks the loop
    try:
       # The term "try" attempts to perform a task (conversion) without causing an error

       Miles = float(UserInput) * 0.621371

       # The conversion of miles is set to a float and is multiplied by the conversation
       # rate of kilometres to miles (0.621371)
```

```
        print(f"{round(Miles, 2)} miles")

        # The special f in the start allows for expressions in statement
        # Expressions are used through {} and i run the method of round
        # To round to 2 decimal places

    except ValueError:
        print("Error: Please Enter A Number Or Type Stop to Exit")
        # If the exception of valueerror is presented e.g. if error happends
        # I would present an error message in
```

This program successfully meets the task requirements by allowing the user to repeatedly convert distances from kilometres to miles until they choose to stop. It uses a while loop controlled by a Boolean variable to keep running, and input validation through a try and except ValueError block ensures the program can handle non-numeric input gracefully without crashing. The conversion is correctly calculated using the formula miles = float(UserInput) * 0.621371, and the result is displayed to two decimal places for clarity. Additionally, the use of strip() and lower() allows flexible user input, letting the user type “stop” in any format to exit. The program is well structured and clearly commented, showing a strong understanding of loops, input handling, and data conversion.

Testing:

Image	Description	Inputs	Expected	Outputs	Requirement
Enter a distance (km): 10 6.21 miles	Enter a numeric distance	10	6.21 miles	6.21 miles	Checks Correct kilometres to mile conversion
Enter a distance (km): 3.5 2.17 miles	Enter a decimal value	3.5	2.17 miles	2.17 miles	Ensures decimal values are handled correctly
Enter a distance (km): 0 0.0 miles	Enter zero as a value	0	0 miles	0 miles	Validation Checking / Error
Enter a distance (km): 1000 621.37 miles	Enter a large number	1000	621.37 miles	621.37 miles	Tests accuracy with large inputs

Enter a distance (km): abc Error: Please Enter A Number Or Type Stop to Exit	Enter a non-numeric value	abc	Type Error: Repeat Code Block	Type Error: Repeat Code Block	Tests input validation and try/except block
Enter a distance (km): stop PS C:\Users\Nasir\OneDrive - The University of Northampton\CSY1020\AS1 Tasks>	Enter stop in lowercase	stop	Program exits	Same as expected	Tests loop exit condition
Enter a distance (km): STOP PS C:\Users\Nasir\OneDrive - The University of Northampton\CSY1020\AS1 Tasks>	Enter STOP in upper case		Program exits	Same as expected	Confirms .lower() works for case-insensitivity
Enter a distance (km): Stop PS C:\Users\Nasir\OneDrive - The University of Northampton\CSY1020\AS1 Tasks>	Enter Stop in pascal case		Program exits	Same as expected	Validates .lower() again
Enter a distance (km): -5 -3.11 miles	Enter a negative value		-3.11 miles	-3.1 miles	Ensures negative values work correctly

Reflection:

Overall, Task 1 was successful in meeting the program requirements and demonstrating key programming concepts such as loops, input validation, data conversion, and formatted output.

The final program effectively allows the user to convert any distance in kilometres to miles repeatedly within one session and includes an exit option by typing "stop." Through the use of a while loop and the try/except structure, I learned how to make a program both interactive and robust against invalid inputs.

The use of methods like .strip() and .lower() helped improve usability by accepting flexible input formats, while rounding to two decimal places made the output more professional and readable.

During development, the prototype stage helped me understand the basic conversion process before expanding it with loops and validation. The main challenge I encountered was handling non-numeric input without causing errors, which I resolved by implementing the exception handling structure.

A minor weakness is that the program could be improved by adding clearer feedback after an invalid entry or allowing users to restart without confusion. Overall, this task strengthened my understanding of structured programming, user input handling, and data processing in Python, while also reinforcing the importance of testing and iterative development to meet all given requirements.

Task 2: String Processing

Requirements:

The requirements of Task 2 involve creating a program that processes a user entered sentence to perform several analyses.

My Python program meets these requirements by first prompting the user to enter a sentence and converting it to lowercase using the lower () method, ensuring it is case-insensitive, so words like “Hello” and “hello” are treated the same. It then splits the sentence into individual words, counts the total number of words, and uses a loop to calculate the total length of all words.

The program identifies the longest word by comparing the length of each word during the loop and then calculates the average word length by dividing the total number of characters by the number of words.

Finally, it displays the total word count, the average word length (rounded to two decimal places), and the longest word, fully meeting all the requirements outlined in Task 2.

Design:

The purpose of this task is to demonstrate the use of string handling, iteration, and basic statistical operations in Python.
To achieve this, the program converts the entire sentence to lowercase, splits it into individual words, calculates total and average word lengths, and determines which word is the longest. This ensures consistent processing regardless of capitalisation.

```
STRING distance = INPUT "Enter a distance in kilometres or type stop"
IF distance <> "stop" THEN
     FLOAT conversion_rate = 0.621371
     FLOAT miles = distance * conversion_rate
     PRINT miles, "2 Decimal Place"
END IF

INPUT sentence
LIST words = SPLIT(sentence)
INT word_count = TOTAL(words)
STRING longest_word = ""
INT length = 0
INT average = 0

FOR each word IN words DO
          length = length + TOTAL(word)
          IF TOTAL(word) > TOTAL(longest_word) THEN
               longest_word = word
          END IF
END FOR

IF word_count > 0 THEN
     average = length / word_count
END IF
```

```
PRINT word_count
PRINT average
PRINT longest_word
```

This pseudocode meets the requirements of Task 2 because it prompts the user to enter a sentence, then splits the sentence into individual words to process them. It counts the total number of words using TOTAL(words), calculates the average word length by dividing the total character count by the number of words, and identifies the longest word by comparing the lengths of each word in a loop. The program then displays the total word count, the average word length, and the longest word found. It can easily be made case-insensitive by converting the sentence to lowercase before splitting it, ensuring that words like "Hello" and "hello" are treated the same.

Implementation:

To implement this program in Python, I first ask the user to enter a sentence and convert it to lowercase using the lower() method, which ensures the program is case-insensitive. The sentence is then split into individual words using the split() function, and the total number of words is found using len(Words). I initialise variables to keep track of the total character length and the longest word, then use a for loop to go through each word in the list. Inside the loop, I add up the length of each word to calculate the total and compare each word's length to find the longest one. After the loop, I check if the user entered at least one word before calculating the average word length by dividing the total character count by the number of words. Finally, I display the word count, the average word length (rounded to two decimal places), and the longest word using print() statements.

```
Sentence = str.lower(input("Enter a sentence: "))

# Requests user input with the following text "Enter a sentence: "
# Converts input into lowercase (if applicable) through the string class using
method lower()
# Stores this converted input into variable Sentence


Words = Sentence.split()
# Variable words is a instance of sentence but into split list e.g. ["Hello",
"Word"]
Count = len(Words)
```

```
# Count is the total length of the setence (if there a space it dosnt count
it)
Length = 0
# Length is set to 0 (default value)
Longest_Word = ""
# Longestword is set to "" (default value)

for word in Words:
    # using a foor loop to check through the entire of the list Words
    Length = Length + len(word)
    # We are adding the length of each word into variable Length
    if len(word) > len(Longest_Word):
        # Using a if statement, we check if the length of the current word in
loop
        # is greater than the longest word, this run continously through each
word
        # in the setence and sets it if its the longest word
        Longest_Word = word



Average = 0
# Now I put the variable Average to make global

if Count > 0:
    # I check for if the user has inputted atleast one word before i calculate
the average
    # I calculate the average using the length of each word devided by the
total words
    Average = Length / Count


print("Word Count:", Count )

print("Average Word Count:", round(Average, 2) )

print("Longest Word:", Longest_Word )

# I print the word count, average word count and longest word
# When am printing the average word count, i use the method round() to round
to 2 decimals
```

Testing:

Image	Inputs	Expected	Outputs	Requirement
Enter a sentence: The quick brown fox jumps over the lazy dog Word Count: 9 Average Word Count: 3.89 Longest Word: quick	The quick brown fox jumps over the lazy dog	9 3.89 quick	Same as expected	Normal sentence to test basic functionality
Enter a sentence: HELLO World Word Count: 2 Average Word Count: 5.0 Longest Word: hello	HELLO World	2 5.0 hello	Same as expected	Check case-insensitivity (both words become lowercase)
Enter a sentence: This is a test Word Count: 4 Average Word Count: 2.75 Longest Word: this	This is a test	4 2.75 this	Same as expected	Simple short sentence for correctness
Enter a sentence: I Word Count: 1 Average Word Count: 1.0 Longest Word: i	I	I 1 1 I	Same as expected	Test with a single word input
Enter a sentence: Testing punctuation, like commas and periods. Word Count: 6 Average Word Count: 6.67 Longest Word: punctuation,	Testing punctuation, like commas and periods.		Same as expected	Check how punctuation affects word count and length

Reflection:

My Python program successfully meets all the requirements of the task. It prompts the user to enter a sentence and processes that input accurately. By converting the sentence to lowercase using the lower() method, the program ensures case-insensitivity, so words like “Hello” and “hello” are treated the same. The sentence is split into individual words, allowing the program to count the total number of words correctly. It uses a loop to calculate the total length of all words and to identify the longest word by comparing their lengths. The program then calculates the average word length by dividing the total character count by the number of words and displays the results clearly using formatted print statements. Through testing with different types of inputs, it consistently performs as expected, correctly displaying the word count, average word length, and longest word, showing that it meets all parts of the given requirements.

Task 3: Lists & Dictionaries

Requirements:

The purpose of this program is to manage student marks efficiently using lists and dictionaries. The program must first check whether a file named students.txt exists. If it does, it should read in the student names and marks from that file. If the file does not exist, it should allow the teacher to manually enter the student data. The data will be stored in a dictionary, with student names as keys and their marks as values. After collecting all the data, the program will calculate the average mark and determine the student with the highest mark. Finally, the results, including each student's name and mark, the average mark, and the top student's name and score, will be written to a file called results.txt. This ensures that the program can both process existing data and record new results effectively.

Design:

This pseudocode helps me design the program by clearly outlining each step the program must follow before writing the actual code. It starts by checking whether the students.txt file exists and then either reads student names and marks from that file or allows the teacher to enter them manually if the file is missing. Using structured steps such as IF, FOR, and WHILE loops makes it easy to understand the program's flow and logic. The pseudocode also shows how to store the data in a dictionary, calculate the average and highest marks, and write the results to a new file called results.txt. Writing it this way ensures that the logic is correct, all requirements are covered, and the program will work in any programming language once implemented.

Pseudocode:

```
STRING input_file = "students.txt"
STRING output_file = "results.txt"
DICTIONARY students = {}

IF FILE_EXISTS(input_file) THEN
     OPEN input_file FOR READ
     FOR each line IN input_file DO
          LIST parts = SPLIT(line, ",")
```

```
          IF TOTAL(parts) = 2 THEN
               STRING name = parts[0]
               FLOAT mark = parts[1]
               students[name] = mark
          END IF
     END FOR
     CLOSE input_file
ELSE
     PRINT "File not found. Enter student data manually."
     WHILE TRUE DO
          STRING name = INPUT "Enter student name (or 'done' to finish)"
          IF LOWER(name) = "done" THEN
               BREAK
          END IF
          FLOAT mark = INPUT "Enter mark for " + name
          students[name] = mark
     END WHILE
END IF

IF TOTAL(students) > 0 THEN
     FLOAT average = SUM(students.values) / TOTAL(students)
     STRING top_student = MAX_KEY(students)
     FLOAT top_mark = students[top_student]
ELSE
     FLOAT average = 0
     STRING top_student = "N/A"
     FLOAT top_mark = 0
END IF

OPEN output_file FOR WRITE
WRITE "Student Results:" TO output_file
FOR each name, mark IN students DO
     WRITE name + ": " + mark TO output_file
END FOR
WRITE "Average mark: " + average TO output_file
WRITE "Top student: " + top_student + " (" + top_mark + ")" TO output_file
CLOSE output_file

PRINT "Results written to results.txt"
```

Implementation:

My actual code follows the same logical structure as the pseudocode but implements it using Python syntax and built-in functions. I started by importing the os module, which allows me to check if the file students.txt exists, just like the pseudocode's FILE_EXISTS step.

If the file is found, the program reads each line, splits it by a comma to separate the student's name and mark, and stores this data in a dictionary where the name is the key and the mark is the value. If the file doesn't exist, the program switches to manual input, asking the teacher to enter names and marks until "done" is typed.

Once the data is collected, the code calculates the average using sum() and len() and finds the top student using Python's max() function, which aligns directly with the pseudocode's calculation section.

```
import os

# import is used to access different scripts, classes and etc (libaries)
# os stands for Operating System so allows me to interact with the operating
system
# i can use os to interfact with files, environment (environ) and but much
more.

# In this caes am going to use os to handle, create or edit files.

# File names
input_file = "students.txt"
output_file = "results.txt"

# I created 2 variables to control what the script should look out or etc.

# dictionary to hold student data
students = {}

# Step 1: Read from file if it exists, otherwise enter manually
if os.path.exists(input_file):
    # This check if the os (operating systm) and the current path folder,
checks for input file
    with open(input_file, "r") as f:
        # now that we know file exists, we open it as read only mode and store
this as f
        for line in f:
            # loops over each line in the text file
            parts = line.strip().split(",")
            # strip removes spaces in that line e.g. "Alice,95"
            # split creates / returns a list seprated by the "," and stored as
parts
            if len(parts) == 2:
                # checks if the data seprated in the list is exactly 2
                # e.g. [mark, 25]
                name = parts[0].strip()
                # python starts list at 0
                try:
                    mark = float(parts[1].strip())
```

```
                    students[name] = mark
                    # now we make sure the mark dosnt have spaces and that its
a decimal
                    # now we store the student name and their marks in the
dictionary
                except ValueError:
                    print(f"Invalid mark for {name}, skipped.")
else:
    # if the file is not found we would manually create it (manual data entry)
    print("File not found. Enter student data manually.")
    while True:
        name = input("Enter student name (or 'done' to finish): ").strip()
        # first we ask for the student name and remove the spaces
        if name.lower() == "done":
            # we check if the student typed "done" to end the program
            break
        try:
            # now we attempt to get the student marks
            mark = float(input(f"Enter mark for {name}: "))
            students[name] = mark
            # now we stored the student name and mark in the dictionary
            # using the student name as its unique identifier
        except ValueError:
            print("Invalid mark. Try again.")
            # remember try just attempts if there an error it goes here


# Step 2: Calculate average and top mark
if students:
    # if students found we do our own calculations to find stuff
    average = sum(students.values()) / len(students)
    # .values just only returns numbers in dictionary [80, 60, 30]
    # sum just adds the values up e.g. 80+60+30
    # now average we calculate by deviding it by the number of students
    top_student = max(students, key=students.get)
    # max just gets the top value, telling max to get it from marks not names
    top_mark = students[top_student]
    # reverse enginnering i guess, this just returns a number since
students[a] = 1

else:
    # if no data in students we create default values
    average = 0
    top_student = "N/A"
    top_mark = 0

# Step 3: Write results to file
with open(output_file, "w") as f:
```

```
    # now we create an output file or rewrite an outputfile as store it as f

    # now we repeatedly just use the f.write function and use the f"" and {}
    f.write("Student Results:\n")
    for name, mark in students.items():
        f.write(f"{name}: {mark}\n")
    f.write(f"\nAverage mark: {average:.2f}\n")
    f.write(f"Top student: {top_student} ({top_mark})\n")


print(f"Results written to {output_file}")

# final output message
```

Finally, it writes all results—including each student's mark, the average, and the top student—to results.txt. The pseudocode acted as a step-by-step plan, helping me translate each part logically into working Python code and ensuring that the final program matched the intended design.

Testing:

<table>
<tr><td colspan="2">TEST ID: 1</td></tr>
<tr><td>Description:

students.txt exists with:
Alice,90
Bob,75
Charlie,60</td><td>Requirement:
File reading, dictionary storage, average & top mark, file output</td></tr>
<tr><td colspan="2">Expected Result:
Reads file, calculates average and top student, writes to output file</td></tr>
<tr><td colspan="2">Test Output:

```
Results written to results.txt
PS C:\Users\Nasir\OneDrive - The University of Northampton\CSY1020\AS1 Tasks>
```

results.txt:

```
Student Results:
Alice: 90.0
Bob: 75.0
Charlie: 60.0

Average mark: 75.00
Top student: Alice (90.0)
```

</td></tr>
<tr><td colspan="2">TEST ID: 2</td></tr>
</table>

Description: students.txt not found	Requirement: Manual data entry, dictionary storage, file creation

Expected Result:
Prompts for manual input, stores data in dictionary, writes to file

Test Output:

```
File not found. Enter student data manually.
Enter student name (or 'done' to finish): Alice
Enter mark for Alice: 90
Enter student name (or 'done' to finish): Bob
Enter mark for Bob: 75
Enter student name (or 'done' to finish): Charlie
Enter mark for Charlie: 60
Enter student name (or 'done' to finish): done
Results written to results.txt
```

results.txt:

```
Student Results:
Alice: 90.0
Bob: 75.0
Charlie: 60.0

Average mark: 75.00
Top student: Alice (90.0)
```

TEST ID: 3

Description: Invalid mark in file students.txt has: Alice,85 Bob, abc Charlie,95	Requirement: Error handling, partial data processing

Expected Result:
Skips invalid marks, continues processing

Test Output:

```
Invalid mark for Bob, skipped.
Results written to results.txt
```

```
PS C:\Users\Nasir\OneDrive - The University of Northampton\CSY1020\AS1 Tasks>
```

results.txt:

```
Student Results:
Alice: 85.0
Charlie: 95.0

Average mark: 90.00
Top student: Charlie (95.0)
```

TEST ID: 4

Description:	Requirement:	Requi
Empty input file students.txt exists but empty	Handles no data scenario gracefully	Error

Expected Result:
Uses default values (no data found)

Test Output:

```
Results written to results.txt
PS C:\Users\Nasir\OneDrive - The University of Northampton\CSY1020\AS1 Tasks>
```

results.txt:

```
Student Results:

Average mark: 0.00
Top student: N/A (0)
```

TEST ID: 5

Description:	Requi
Duplicate student names students.txt contains: Alice,90 Alice,95	Hand

Expected Result:
Later value overwrites earlier (dictionary behavior)

Test Output:

```
Results written to results.txt
PS C:\Users\Nasir\OneDrive - The University of Northampton\CSY1020\AS1 Tasks>
```

results.txt:

```
Student Results:
Alice: 95.0

Average mark: 95.00
Top student: Alice (95.0)
```

Reflection

The program successfully met all the stated requirements. It correctly checks for the existence of the students.txt file, reads data when available, and allows manual input when the file is missing.

Student names and marks are stored efficiently in a dictionary, and the program accurately calculates both the average mark and the top student. It handles invalid data, empty files, and extra spaces gracefully, ensuring stability across all test scenarios.

The results are written correctly to results.txt, and the output matches the expected format. Overall, the program fulfils its purpose of managing and recording student marks effectively.

Task 4: Functions & Problem Decomposition

Requirements:

The purpose of this program is to allow a user to manage a shopping cart efficiently using functions and data structures such as lists or dictionaries. The program will enable the user to add multiple items along with their prices and store this information appropriately. It will include at least three functions, such as add_item, calculate_total, and display_cart, to organise

e the code and handle specific tasks. When the user proceeds to checkout, the program will display all items in the cart along with their prices and show the total cost, formatted to two decimal places. The final program must be saved as **task4.py**.

Design:

```
DICTIONARY cart = {}

FUNCTION add_item(cart)
     STRING item = INPUT "Enter the item name: "
     FLOAT price = INPUT "Enter the item price: $"
     cart[item] = price
     PRINT "Added '" + item + "' to the cart for $" + FORMAT(price, 2)
END FUNCTION

FUNCTION display_cart(cart)
     IF TOTAL(cart) = 0 THEN
          PRINT "Your cart is empty."
          RETURN
     END IF
     PRINT "--- Shopping Cart ---"
     FOR each item, price IN cart DO
          PRINT item + ": $" + FORMAT(price, 2)
     END FOR
     PRINT "----------------------"
END FUNCTION

FUNCTION calculate_total(cart)
     FLOAT total = SUM(cart.values)
     RETURN total
END FUNCTION

FUNCTION main()
     WHILE TRUE DO
          PRINT "1. Add Item"
          PRINT "2. View Cart"
          PRINT "3. Checkout"
          STRING choice = INPUT "Choose an option (1-3): "

          IF choice = "1" THEN
               CALL add_item(cart)
          ELSE IF choice = "2" THEN
               CALL display_cart(cart)
          ELSE IF choice = "3" THEN
               CALL display_cart(cart)
               FLOAT total = CALL calculate_total(cart)
               PRINT "Total cost: $" + FORMAT(total, 2)
               PRINT "Thank you for shopping!"
```

```
                BREAK
            ELSE
                PRINT "Invalid choice. Please enter 1, 2, or 3."
            END IF
        END WHILE
END FUNCTION

CALL main()
```

Writing the pseudocode was useful because it provided a clear, step-by-step plan for how the shopping cart program would work before actually writing the Python code.

It helped outline the main logic of the program, including how data would be stored, how user input would be handled, and how the functions would interact with each other.

By defining the structure in pseudocode first, it became easier to identify what functions were needed—such as add_item, display_cart, and calculate_total—and how they would contribute to the main loop of the program. This approach also made the implementation process more efficient and reduced errors, since the pseudocode acted as a blueprint that could be directly translated into working Python code.

Overall, it ensured the program was well-organised, readable, and logically sound before any actual coding took place.

Implementation:

```
def add_item(cart):
    # Function to add an item to the cart
    item = input("Enter the item name: ")
    price = float(input("Enter the item price: $"))
    cart[item] = price
    print(f"Added '{item}' to the cart for ${price:.2f}\n")


def display_cart(cart):
    # Function to display all items in the cart
    if not cart:
        print("Your cart is empty.\n")
        return
    print("\n--- Shopping Cart ---")
    for item, price in cart.items():
        print(f"{item}: ${price:.2f}")
    print("-----------------------\n")
```

```
def calculate_total(cart):
    # Function to calculate total cost
    total = sum(cart.values())
    return total

def main():
    # Main program loop
    cart = {}
    while True:
        print("1. Add Item")
        print("2. View Cart")
        print("3. Checkout")
        choice = input("Choose an option (1-3): ")

        if choice == "1":
            add_item(cart)
        elif choice == "2":
            display_cart(cart)
        elif choice == "3":
            display_cart(cart)
            total = calculate_total(cart)
            print(f"Total cost: ${total:.2f}")
            print("Thank you for shopping!")
            break
        else:
            print("Invalid choice. Please enter 1, 2, or 3.\n")

# Run the program
main()
```

Testing:

Image	Description	Inputs	Expected	Outputs	Requirement
TEST ID: 1					

1. Add Item 2. View Cart 3. Checkout Choose an option (1-3):	Start program and view menu options		Program displays options: 1. Add Item, 2. View Cart, 3. Checkout	Menu displayed correctly	Program starts and displays main menu
TEST ID: 2					
1. Add Item 2. View Cart 3. Checkout Choose an option (1-3): 1 Enter the item name: Apple Enter the item price: $1.5 Added 'Apple' to the cart for $1.50	Add a single item to the cart	1 Apple 1.50	Message: *Added 'Apple' to the cart for $1.50*	Message displayed correctly	add_item() function works correctly
TEST ID: 3					
1. Add Item 2. View Cart 3. Checkout Choose an option (1-3): 1 Enter the item name: Apple Enter the item price: $1.5 Added 'Apple' to the cart for $1.50 1. Add Item 2. View Cart 3. Checkout Choose an option (1-3): 1 Enter the item name: Bread Enter the item price: $2.0 Added 'Bread' to the cart for $2.00	Add multiple items to the cart	1 Apple 1.50 1 Bread 2.00	Both items added with confirmation messages	Both added successfully	Multiple items can be added
TEST ID: 4					
1. Add Item 2. View Cart 3. Checkout Choose an option (1-3): 1 Enter the item name: Apple	View cart contents	1 Apple 1.50 1 Bread 2.00	Displays: “Apple: $1.50” and “Bread: $2.00”	Items displayed correctly	display_cart() works correctly

Enter the item price: $1.50 Added 'Apple' to the cart for $1.50 1. Add Item 2. View Cart 3. Checkout Choose an option (1-3): 1 Enter the item name: Bread Enter the item price: $2.00 Added 'Bread' to the cart for $2.00 1. Add Item 2. View Cart 3. Checkout Choose an option (1-3): 2 --- Shopping Cart --- Apple: $1.50 Bread: $2.00 ----------		2			
TEST ID: 5					
1. Add Item 2. View Cart 3. Checkout Choose an option (1-3): 1 Enter the item name: Apple Enter the item price: $1.50 Added 'Apple' to the cart for $1.50 1. Add Item 2. View Cart 3. Checkout Choose an option (1-3): 1 Enter the item name: Bread Enter the item price: $2.00 Added 'Bread' to the cart for $2.00 1. Add Item 2. View Cart 3. Checkout Choose an option (1-3): 3	Checkout process	1 Apple 1.50 1 Bread 2.00 3	Displays all items, total cost ($3.50), and exit message	Output as expected	Checkout displays cart and total correctly

--- Shopping Cart --- Apple: $1.50 Bread: $2.00 ---------- Total cost: $3.50 Thank you for shopping!					

Reflection:

The program successfully met all the objectives outlined for managing a shopping cart. It allows users to add multiple items with their prices, stores the data efficiently in a dictionary, and uses separate functions to handle different tasks—ensuring the code is organised and easy to maintain.

The use of functions like add_item, display_cart, and calculate_total made the program modular and clear. Additionally, the output formatting to two decimal places improved readability and gave a professional finish to the checkout display.

The testing process confirmed that the program handled user input correctly, including invalid options, and performed calculations accurately.

Overall, implementing these requirements enhanced my understanding of program structure, function design, and user interaction in Python.