CSE 1321L: Programming and Problem Solving I Lab

Assignment 5 Fall 2025

Module 4

What students will learn

- Problem Solving.
- Basic Program Structure.
- Input and Output with the user.
- Write code that includes while/for loop and nested loops logic with sequence types.
- Structure program to include methods as well.

Content

Overview:

For this assignment, you're going to practice making logic in your code. It will include loops, selection statements, functions and sequence types. In practical terms, this means you're going to expand on the concepts from previous assignments but also include things like lists and tuples. Again, start early, practice, and ask a lot of questions.

Final note: *Do not cheat*

If your temptation is to look online, don't. Come see us instead and ask questions – we are here to help. Remember, you are going to have to write codes in your future job interviews, so learn it now to secure a high-paying job later.

Assignment5A: Selection Sort from Scratch

Write a method called **myselectionsort(list)** to sort a list of integers in ascending order without using the built-in sort() or sorted() functions. Your function will take unsorted list as parameter and **return** a sorted list.

You will implement your own sorting algorithm with nested loops inside your function and then **compare your result** with the result of Python's default sorting function.

Details:

- Implement **Selection Sort** as described below:
  - **Selection Sort Algorithm (Step-by-Step Example):**
    - Start with the first position (index 0).
    - Find the **smallest value** in the list and **swap it** with the element at index 0.
    - Move to the next index (index 1) and again find the smallest value in the **remaining unsorted part** of the list.
    - Swap it with the element at index 1.
    - Continue this process until all elements are sorted.
    - You will be using nested loops.
  - **Example Walkthrough:**
    - Starting list: [5, 2, 9, 1, 6]
    - **Iteration 1:** smallest element in [5, 2, 9, 1, 6] is 1 → swap with 5 → [1, 2, 9, 5, 6]
    - **Iteration 2:** smallest element in [2, 9, 5, 6] is 2 → already in place → [1, 2, 9, 5, 6]
    - **Iteration 3:** smallest element in [9, 5, 6] is 5 → swap with 9 → [1, 2, 5, 9, 6]
    - **Iteration 4:** smallest element in [9, 6] is 6 → swap with 9 → [1, 2, 5, 6, 9]
    - **Iteration 5:** last element already in place.
- Final sorted list: [1, 2, 5, 6, 9]

Requirements:

- Prompt the user to enter a comma-separated list of numbers.
- Convert the input into a list of integers before passing it to the function.
  - You can use split function.
- Call myselectionsort(list) function and save the returned list.
- Print your sorted list.
- Create another list using Python's built-in sorted() function and print it.
- Compare the two lists:
  - If both are identical → print "Both lists are identical!"
  - Otherwise → print "The lists are not the same."

Example runs are shown below. The user input is shown in **red**.

Sample Output #1:

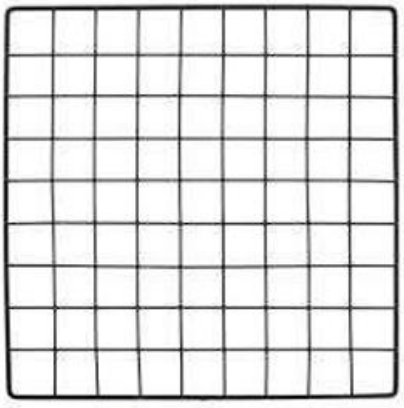
```
Enter numbers separated by commas: 5,2,1,9,6
Original list: [5, 2, 1, 9, 6]
Sorted list (using my selection sort): [1, 2, 5, 6, 9]
Sorted list (using default function): [1, 2, 5, 6, 9]
Both lists are identical!
```

Sample Output #2:

```
Enter numbers separated by commas: 8,7,6,5,4,3,2,1
Original list: [8, 7, 6, 5, 4, 3, 2, 1]
Sorted list (using selection sort): [1, 2, 3, 4, 5, 6, 7, 8]
Sorted list (using default function): [1, 2, 3, 4, 5, 6, 7, 8]
Both lists are identical!
```

Assignment5B: Dungeon Treasure Map

In this assignment you will create a treasure hunt board game. Players will play on a board which resembles a grid like this:



The exact dimensions of the grid will be determined by the player. Each square of the grid will either have an 'O' representing an open square, a 'T' representing treasure, or an 'X' representing treasure that has been collected.

Steps:

- Prompt the user for the size of the grid (width and height)
- Create a list called board, each cell will represent a row of the grid.
- In the first cell of the board list, you'll place a list.
    - Pick a random number between 0 and 1. If the number is greater than or equal to 0.7 you'll add a Treasure 'T' to the next cell of the list. If the number is less than 0.7 you'll add an open 'O' to the next cell of the list.
    - Keep track of how many treasures you are adding to the board in a separate variable called numberOfUndiscoveredTreasures.
    - Repeat step (a) until you have a list that is the height the user asked for in step (1).
- Repeat step (3) until the board is the width the user asked for in step 1.
- Tell the user how many treasures you have hidden.
- Next, you'll ask the user to guess coordinates, you'll check if they found treasure or not:
    - Ask the user to enter in a row number (0 to the width of the board -1)
    - Ask the user to enter in a column number (0 to the height of the board -1)
    - Check that location to see if it is a "T" (Treasure) or an "O" (Open).
        - If it's a treasure tell the user they got treasure, change that cell of the board to an "X" to indicate that it was already discovered. Lower the number of undiscovered treasures by one.
        - If it's not a treasure, tell the user to try again.
        - Keep asking the user to guess locations until the user has discovered all the Treasures, then print out the whole board, and end the game.

Example runs are shown below. The user input is shown in **red.**

Sample Output #1:

```
Enter the width of the grid: 2
Enter the height of the grid: 2
Number of treasures hidden: 1
Enter the row number (0 to 1): 0
Enter the column number (0 to 1): 0
No treasure here, try again!
Enter the row number (0 to 1): 1
Enter the column number (0 to 1): 1
No treasure here, try again!
Enter the row number (0 to 1): 0
Enter the column number (0 to 1): 1
No treasure here, try again!
Enter the row number (0 to 1): 1
Enter the column number (0 to 1): 0
Congratulations! You found a treasure!
O O
X O
Congratulations! You've found all the treasures!
O O
X O
```

Sample Output #2:

```
Enter the width of the grid: 3
Enter the height of the grid: 3
Number of treasures hidden: 3
Enter the row number (0 to 2): 0
Enter the column number (0 to 2): 0
No treasure here, try again!
Enter the row number (0 to 2): 0
Enter the column number (0 to 2): 1
Congratulations! You found a treasure!
O X O
O O O
O O O
Enter the row number (0 to 2): 0
Enter the column number (0 to 2): 2
No treasure here, try again!
Enter the row number (0 to 2): 1
Enter the column number (0 to 2): 1
No treasure here, try again!
Enter the row number (0 to 2): 1
Enter the column number (0 to 2): 0
No treasure here, try again!
Enter the row number (0 to 2): 1
Enter the column number (0 to 2): 2
Congratulations! You found a treasure!
O X O
O O X
O O O
Enter the row number (0 to 2): 2
```

```
Enter the column number (0 to 2): 1
No treasure here, try again!
Enter the row number (0 to 2): 2
Enter the column number (0 to 2): 0
No treasure here, try again!
Enter the row number (0 to 2): 2
Enter the column number (0 to 2): 2
Congratulations! You found a treasure!
O X O
O O X
O O X
Congratulations! You've found all the treasures!
O X O
O O X
O O X
```

Assignment5C: Frequency Dictionary

Write a Python program that repeatedly displays a menu asking the user to choose whether they want to analyze **letters** or **numbers**.
Based on the choice, take comma-separated input, store it in a **tuple**, and pass it to a function find_frequency(data) which **returns a dictionary** containing the frequency of each element.
In the **main function**, find the element(s) with the **highest frequency** and display them along with their count.
The program should continue running until the user chooses to **quit**.

Requirements:

- Display the following menu repeatedly:
  - Main Menu
    1.Enter letters
    2. Enter numbers
    3. Quit
- Based on the choice:
  1 → letters: ask Enter letters separated by commas:
  2 → numbers: ask Enter numbers separated by commas:
  3 → quit: print "Exiting program..." and terminate.
- Convert input into a **tuple**:
  - Letters → keep as strings
  - Numbers → convert each to int
- Call find_frequency(data) with the tuple.
- This function **returns a dictionary** where:
  - key = element, value = frequency
- In the **main function**:
  - Determine the **maximum frequency**.
  - Identify **all element(s)** with that frequency.
  - Display the tuple, frequency dictionary, and the most frequent element(s) with their frequency.

Example runs are shown below. The user input is shown in **red**.

Sample Output #1:

```
Main Menu
1. Enter letters
2. Enter numbers
3. Quit
Enter your choice: 1
Enter letters separated by commas: a,a,r,t,c,a,c
Tuple: ('a', 'a', 'r', 't', 'c', 'a', 'c')
Frequency dictionary: {'a': 3, 'r': 1, 't': 1, 'c': 2}
Most frequent element(s): a
Frequency: 3

Main Menu
1. Enter letters
2. Enter numbers
3. Quit
Enter your choice: 3
Exiting program...
```

Sample Output #2:

```
Main Menu
1. Enter letters
2. Enter numbers
3. Quit
Enter your choice: 2
Enter numbers separated by commas: 1,2,3,3,3,2,2,4,8
Tuple: (1, 2, 3, 3, 3, 2, 2, 4, 8)
Frequency dictionary: {1: 1, 2: 3, 3: 3, 4: 1, 8: 1}
Most frequent element(s): 2, 3
Frequency: 3

Main Menu
1. Enter letters
2. Enter numbers
3. Quit
Enter your choice: 3
Exiting program...
```

Submission Instructions:

- Programs must follow the output format provided. This includes each blank line, colons (:), and other symbols.
- Programs must be working correctly.
- Programs must be written in Python.
- Programs must be submitted with the correct **.py** format.
- Programs must be saved in files with the correct file name:
  - Assignment5A.py
  - Assignment5B.py
  - Assignment5C.py
- Programs (source code files) must be uploaded to Gradescope by the due date.