COSC 130 – HW 04 Instructions

General Instructions

Create a new notebook named **`HW_04_YourLastName.ipynb`** and complete problems 1 – 8 described below.

Any set of instructions you see in this document with an orange bar to the left will indicate a place where you should create a markdown cell. For each new problem, create a markdown cell that indicates the title of that problem as a level 2 header.

Any set of instructions you see with a blue bar to the left will provide instructions for creating a single code cell.

Read the instructions for each problem carefully. Each problem is worth 6 points. An additional 2 points are allocated for formatting and following general instructions.

The assignment should be completed using only base Python commands. Do not use any external packages.

Assignment Header

Create a markdown cell with a level 1 header that reads: "COSC 130 - Homework 04". Add your name below that as a level 3 header

Problem 1: Dot Product

In this problem, you will write a function named that performs the dot product operation on two vectors. In mathematics, a vector is essentially a sequence of numbers, similar to a Python list. Given to vectors of equal size, the **dot product** of those vectors is calculated by multiplying corresponding elements of the two vectors and then summing the products.

As an example, consider vectors represented by the following two lists:

```
a = [3, 4, 2, 1]
b = [5, 2, 3, 7]
```

Then the dot product of these two vectors would be $3 \cdot 5 + 4 \cdot 2 + 2 \cdot 3 + 1 \cdot 7 = 15 + 8 + 6 + 7 = 36$.

Write a function called **`dot_product()`** that takes two parameters named **`x`** and **`y`**. The parameters are intended to represent vectors of the same size. The function should calculate and return the dot product of these two vectors.

The function should not print anything. The function should not create any new lists, and should not alter any lists provided to it as arguments. It should involve only a single loop.

We will now test the dot product function.

In a new code cell, create two lists named **`v1`** and **`v2`** as shown below.

```
v1 = [38,  9, 40, 34, 20, 16, 42, 36, 12,  1, 23, 46, 31, 19, 30, 33, 16, 43, 24, 41]
v2 = [43, 13, 35, 14, 26,  3, 36, 15, 42, 44, 45, 20, 17,  6, 47, 40, 38, 41, 31, 24]
```

Calculate the dot product of these vectors. Print the result in the formal shown below. Match the formal exactly.

```
The dot product of v1 and v2 is xxxx.
```

Problem 2: Amortization

In this problem, we will create a function that counts the number of monthly payments required to pay back a loan

Define a function named **`count_payments()`**. The function should accept three parameters: **`amount`**, **`rate`**, and **`pmt`**. The parameter **`amount`** represents the initial size of a loan that is to be repaid with monthly payments. The parameter **`rate`** is the monthly rate at which the loan collects interest, and **`pmt`** is the size of the monthly payment.

The function should calculate and return the number of monthly payments required to repay the loan. This is calculated by using a loop. Every time the loop executes, the following steps should be performed:

- Calculate the new balance. This will be done by multiplying the current balance by **`1+i`**, and then subtracting the payment amount. Store the result back into **`balance`**, rounded to 2 decimal places.
- Increment a variable used to count the number of payments that have been made.

The loop should continue to run for as long as the balance is greater than zero.

The function should not print anything. It should not create any new lists, and should involve only one loop.

We will now test the function. Create a new code cell for the instructions below.

Assume that a loan is made in the amount of $160,000 and is charged a monthly interest rate of 0.4%. Use a loop along with the function **`count_payments()`** to determine the number of monthly payments that must be made in order to repay the loan, assuming that the payments are in the size of each of the following values:

```
850, 900, 950, 1000, 1050
```

Each time the loop executes, the message shown below should be displayed, with the `xxxx` symbols replaced with the appropriate values. Match the format exactly. Do not include extra spaces.

```
xxxx monthly payments of $xxxx would be required.
```

For full credit, I would like for you to use a formula to determine the new payment amount each time the loop executes, rather than creating a list to store the payment amounts. However, if you use a list to store the payments, you will get partial credit.

Let's test the function with a different set of values. Create a new code cell for the instructions below.

Assume that the borrow mentioned in the previous set of instructions has an opportunity to borrow the $160,000 at a lower monthly interest rate of 0.35%. Repeat the instructions from the previous cell using the new interest rate. Use the same payments amounts as before.

Problem 3: Minimum

In this problem, we will write a function to find the smallest element of a list. We are, in a sense, re-inventing the wheel since the **`min()`** function already performs this exact task. However, the purpose this exercise is to have you think through the logic of how such a function would be implemented from scratch.

Define a function named **`minimum()`**. The function should accept a single parameter named **`x`**, which is expected to be a list of elements of the same type. The function should return the smallest element of **`x`**. The function should work on lists of integers, floats, and strings. In each case, the "smallest" element is defined as the one with the lowest ranking with respect to the **`<`** comparison operator. For strings, this should yield the earliest string when ordered alphabetically.

The function should not print anything. It should not create any new lists, and should involve only one loop.

Furthermore, this function should not make use of ANY built-in Python functions other than `range()` and `len()`. No credit will be awarded for solutions that use the `min()` function.

We will now test the **`minimum()`** function. Create a new code cell to perform the steps below.

Create three lists as shown below:

```
list1 = [9.8, 7.4, 5.6, 4.8, 4.8, 5.3, 4.1, 9.6, 5.4]
list2 = [3.4, 7.6, 8.7, 7.5, 9.8, 7.5, 6.7, 8.7, 8.4]
list3 = ['St. Louis', 'Kansas City', 'Chicago', 'Little Rock', 'Omaha']
```

Use the **`minimum()`** function to calculate the minimum of each of these lists, printing the results.

Problem 4: Argmin

In this problem, we will write a function to find the **index** of the smallest element of a list. The logic in writing this function is very similar to that which was used in Problem 3, with a few modifications.

Define a function named **`argmin()`**. The function should accept a single parameter named **x**, which is expected to be a list of elements of the same type. The function should return the **index** of smallest element of **x**. If there are multiple elements all equal to the smallest element, then the function should return the index of the first occurrence of an element with this value.

The function should not print anything. It should not create any new lists, and should involve only one loop.

Furthermore, this function should not make use of ANY built-in Python functions other than `range()` and `len()`. No credit will be awarded for solutions that use the `min()` function or the `index()` method.

A potential solution would be to use the **`minimum()`** function from Problem 3 to determine the smallest element, and then loop over **x** to find the index where this value is located. For the sake of efficiency, **this approach should be avoided**. Since the **`minimum()`** function includes a loop, this solution would actually involve looping over the list twice. Although use of the **`minimum()`** function should be avoided here, partial credit will be awarded for this approach if you are unable to find a more efficient solution.

We will now test the **`argmin()`** function. Create a new code cell to perform the steps below.

Call **`argmin()`** on the each list created in Problem 3, printing the results.

Problem 5: Find Smallest Elements

In this problem, we will write a function to find the smallest elements of a list.

Define a function named **`find_smallest()`** that accepts two parameters: **x** and **n**. The parameter **x** is expected to be a list of values of the same time, and **n** is expected to be an either integer, or the value **None**, and should have a default value of **None**.

- If **n** is set to **None**, then the function should return the smallest element of **x** (not as part of a list).
- If **n** is set to a positive integer, the function should return a list consisting of the smallest **n** elements of list **x**. If **n** is greater than the length of the list **x**, then the entire list **x** should be returned.

Note that **n=None** and **n=1**, should produce similar, but not identical results. Both arguments will select only a single value from the list **x**, but if **n=1** then the function should return a list containing the value, whereas if **n=None** then the function should simply return the value.

This problem would be challenging to do without using built-in functions, and so you are allowed to do so in this problem. **As a hint**, I recommend using sorting functions and slicing. However, the list that was provided as an argument to the function **should not** get sorted or altered as a result of the function being called.

We will now test the **`find_smallest()`** function. Create a new code cell to perform the steps below.

Create a list named **`my_list`** containing the values 39, 74, 28, 64, 17, 28, 54, 53 (in that order). Print the results of each of the following function calls.

```
find_smallest(my_list)
find_smallest(my_list, 1)
find_smallest(my_list, 2)
find_smallest(my_list, 5)
find_smallest(my_list, 12)
```

Let's confirm that the original list has not been altered.

Create a new code cell to print the list **`my_list`**.

Problem 6: Find Unique Elements

In this problem, we will create a function named **`unique()`** that will identify the unique elements contained within a list. The function will return a list that contains the same values as the argument list, but with duplicate values removed, and sorted in ascending order. For example, if **`mylist = [4, 6, 3, 3, 9, 6, 4]`**, then **`unique(mylist)`** should return **`[3, 4, 6, 9]`**.

Define a function named **`unique()`** with a single parameter **`x`**, which is expected to be a list. The function should return a new list, containing a single occurrence of every value that appears in **`x`**. The list returned should be sorted in ascending order.

You are allowed to use the **`range()`** and **`len()`** functions (if needed) and the **`append()`** and **`sort()`** list methods, but no other predefined functions or methods should be used. I would recommend using one or both of the following comparison operators: **`in`** and **`not in`**.

We will now test the **`unique()`** function. Create a new code cell to perform the steps below.

Create the lists shown below:

```
int_list = [23, 16, 23, 12, 14, 23, 12, 19, 19]
str_list = ['cat', 'dog', 'dog', 'cat', 'bat', 'frog', 'dog', 'frog']
```

Call the **`unique()`** function on each of these lists, printing the results.

Problem 7: Frequency Distribution

In this problem, we will create a function named **`freq_dist()`** that creates a frequency distribution based on the elements of a list. The function will accept a list of values as its input and will return two lists. One returned list will contain the unique values found in the argument list, while the second list will contains counts of the number of times each of the unique values occurred. For example: If **`mylist = [10, 10, 20, 10, 20, 30, 20, 20, 30, 40]`**, then **`freq_dist(mylist)`** should return the following tuple: **`([10, 20, 30, 40], [3, 4, 2, 1])`**.

Define a function named **`freq_dist()`** with a single parameter **`x`**, which is expected to be a list. This function should return two lists. You can name these lists whatever you like, but for the sake of discussion, let's call them **`values`** and **`counts`**.

- The list **`values`** should be a sorted list of unique values appearing in **`x`**. You can (and should) use the function you wrote in Problem 6 to create this list.
- The list **`counts`** should be a list of integers. Each integer should be a count of the number of times a particular value in values appears in x.

An outline of this function is provided below:

1. Find the list of unique values, referred to above as **`values`**.
2. Create an empty list for **`counts`**.
3. Loop over **`values`**. Each time the loop executes, you should count the number of times the current element of **`values`** appears in the list **`x`**, appending the result to the **`counts`** list. I recommend using the **counts()** list method for this (although you could also perform this using a second loop).
4. Return the two lists **values** and **counts**.

We will now test the **`freq_dist()`** function. Create a new code cell to perform the steps below.

Create the list shown below:

```
grades = ['A', 'D', 'A', 'C', 'B', 'F', 'A', 'D', 'C',
          'B', 'F', 'A', 'C', 'B', 'A', 'B', 'B', 'C',
          'B', 'F', 'D', 'D', 'A', 'C', 'B', 'B', 'D']
```

Call the **`freq_dist()`** function on this list. Display the resulting lists with messages formatted as shown below. Match the formatting exactly. The lists displayed should be left-aligned.

```
Unique Values: xxxx
Frequencies:   xxxx
```

Problem 8: Weighed Means

In this problem, we will create a function named **`mean()`** that can be used to calculate either standard means or weighted means. The function will have two parameters, **`x`** and **`w`**. The parameter **`x`** is expected to be a numerical list, and **`w`** is expected to be either a numerical list or to have the value **None**. If **`w=None`**, then the function should simply return the mean (or average) of the elements in **`x`**. If **`w`** is a list, then the function should return the weighted mean of the elements in **`x`**, using the elements of **`w`** as weights.

If the elements of **`x`** are denoted by $x_0, x_1, \ldots, x_n$ and the elements of **`w`** are denoted by $w_0, w_1, \ldots, w_n$, then the weighted mean is calculated as follows:

$$\frac{w_0 \cdot x_0 + w_1 \cdot x_1 + \cdots + w_n \cdot x_n}{w_0 + w_1 + \cdots + w_n}$$

As an example (**that you are not being asked to include in your notebook**), if:

```
list1 = [200, 400, 100]
list2 = [3, 2, 5]
```

Then **`weighted_mean(list1, list2)`** should return: $\frac{3(200)+2(400)+5(100)}{3+2+5} = \frac{1900}{10} = 190$

Define a function named **`mean()`** with two parameters named **`x`** and **`w`**. . The parameter **`x`** is expected to be a numerical list, and **`w`** is expected to be either a numerical list or to have the value **None**. The default value of **`w`** should be **None**. The function should behave as follows:

- If **`w=None`**, then the function should return the standard mean (average) of the values in **`x`**. You may use the functions **`sum()`** and **`len()`** to calculate this.
- If **`w`** is a list, then the function should return the weighted mean of the values in **`x`**, using the values from **`w`** as weights.

In any case, this function should return only a single value.

The function should not print anything. It should not create any new lists, and should alter either of the lists provided to it as arguments.

We will now test the **`mean()`** function. Create a new code cell to perform the steps below.

Create the list shown below:

```
values = [4, 7, 3, 5, 2, 6, 8, 2, 4, 8]
weights = [2, 1, 3, 1, 2, 3, 1, 4, 2, 1]
```

Use two calls to the function **`mean()`** to calculate the standard mean of **`values`** and then the weighted mean of **`values`**, using the list **`weights`** to supply the weights. Print the results with messages as shown below.

```
Standard Mean: xxxx
Weighted Mean: xxxx
```

The call to **`mean()`** that is used to calculate the standard mean is expected to involve only one argument.

Submission Instructions

When you are done, click **Kernel > Restart and Run All**. If any cell produces an error, then manually run every cell after that one, in order. Save your notebook, and then export the notebook as an HTML file. Upload the HTML file to Canvas and upload the IPYNB file to CoCalc, placing it in the **Homework/HW 04** folder.