EECS 1015 Lab 10

Goal

- Be able to write a script that solves problems using various Python collections

Tasks

1. Learn to break down problems into smaller pieces with collections
2. Learn to debug a script with collections
3. Learn to write a script with collections
4. Learn to write a script that emphasize the a very important concept when working with collections in a function
5. Learn to write a script that emphasize the a very important concept when working with collections in a function
6. Learn to write a script with collections

Total Credit: 100 pts

It is an academic offense to copy code from other students, provide code to other students, let other people (including the teaching staff) debug your code, or debug other students' code.

We will check code similarities at the end of the semester.

Questions 3 – 6 may not be arranged by the difficulty level. You are highly recommended to take a look and decide the order of completing the questions. You will experience something similar in the exams and it is important to learn how to triage your tasks.

Note: For question 3 – 6, you must come up with a function description in your own words, and cannot copy the words from handout.

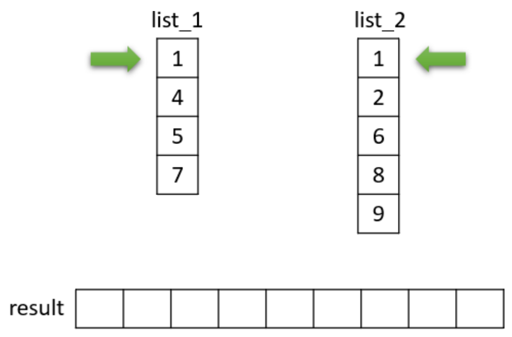
Task 1: Follow the Steps (30 pts)

For this task, we'll familiarize ourselves with working with the list data structure. The objective of this task is to create a function named "merge_lists". This function will accept two distinct lists as input and return a combined list in ascending order. To make it more interesting, we'll also assert that the input lists are in ascending order.

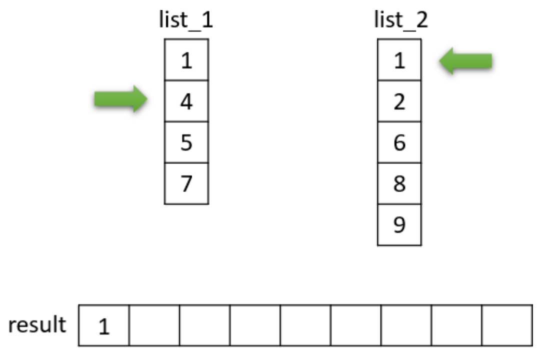
Let’s go over two example scenarios:

Example 1:

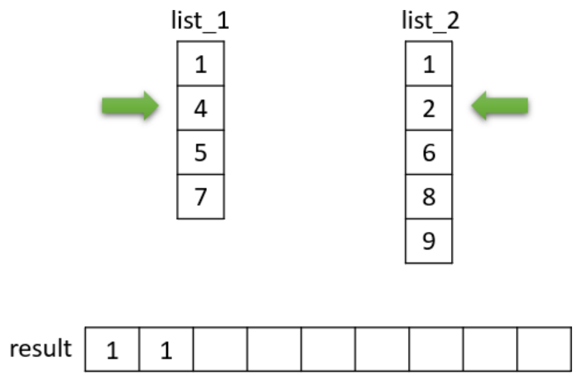
- input: list_1 = [1, 4, 5, 7], list_2 = [1, 2, 6, 8, 9]
- We can see that list_1 and list_2 are in ascending order. We want to merge them so that the resulting list is also in ascending order.
- While there could be a simple way of doing things (e.g., concatenate two lists and then utilize the sort() function on the new list), let’s implement this specific algorithm together which may be more efficient then the sort algorithm (we want faster programs!):
    - It labels which item in each list is of interest (always starts from the first item in each list). Then let’s compare the two items (e.g., 1 and 1), and choose the smaller number.



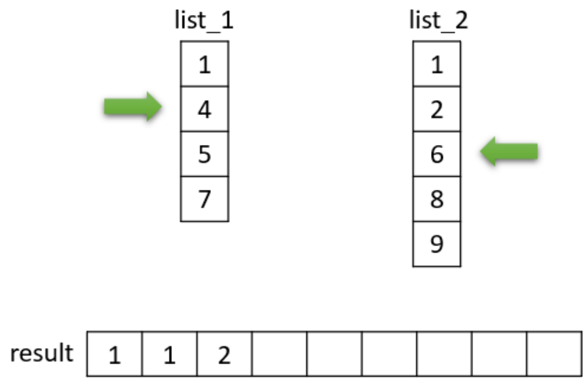
    - Since both numbers are the same, we can choose the number from list_1 and save it in our resulting list, and then move the cursor to the next element in the list, which results in the figure below. Then let’s compare 1 and 4.



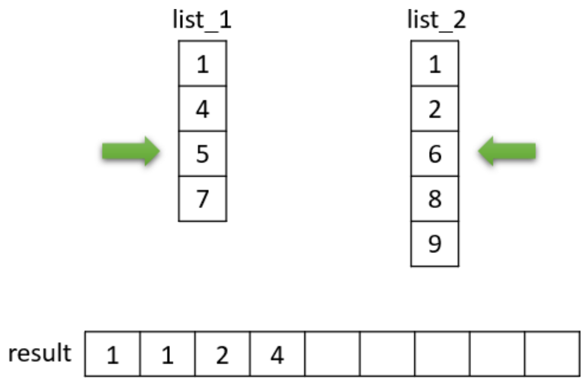
- 1 is smaller so choose 1 and save it in our resulting list, and for list_2, move on to the next item for list_2, which results in the figure below. Now we compare 4 and 2.



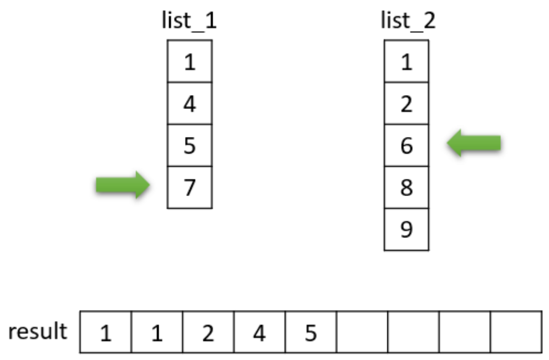
- 2 is smaller than 4, so we choose 2. Put it in our result list and move the cursor in list_2 to the next element, as shown in the figure below. Now we compare 4 and 6.



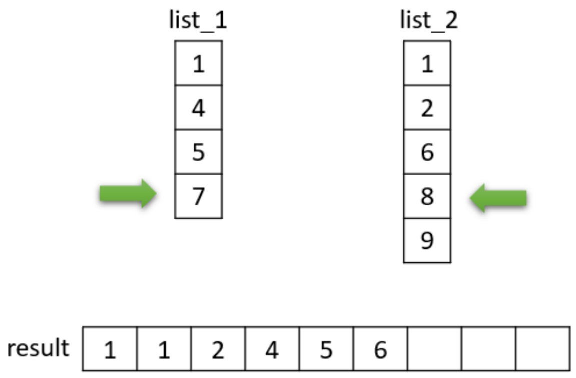
- 4 is smaller than 6, so we choose 4. Put it in our result list and move the cursor in list_1 to the next element, as shown in the figure below. Now we compare 5 and 6.



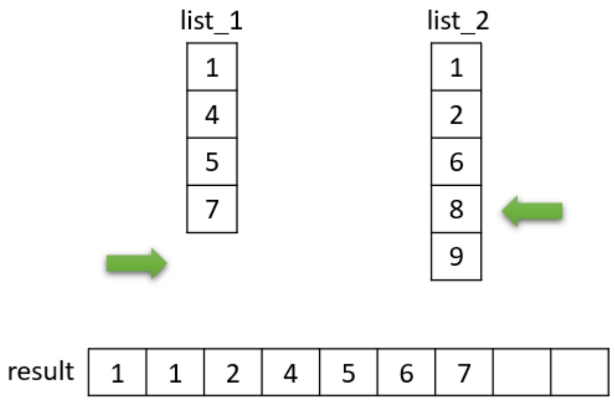
- 5 is smaller than 6, so we choose 5. Put it in our result list and move the cursor in list_1 to the next element, as shown in the figure below. Now we compare 7 and 6.



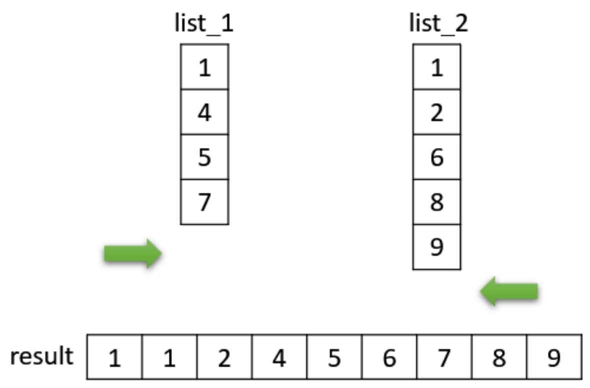
- 6 is smaller than 7, so we choose 6. Put it in our result list and move the cursor in list_2 to the next element, as shown in the figure below. Now we compare 7 and 8.



- 7 is smaller than 8, so we choose 7. Put it in our result list and move the cursor in list_1 to the next element, as shown in the figure below.



- Now there are no more elements in list_1. As the items in list_2 is already in acending order, and the item of interest (e.g., 8) is guaranteed to be larger (or equal) to the largest item in the resulting list, we can just add the rest of the items in list_2 in its own order to the resulting list, which ends up with the figure below:



- So the resulting Output is: [1, 1, 2, 4, 5, 6, 7, 8, 9]
- If you still confused about this algorithm, here is a video explanation: https://www.youtube.com/watch?v=XIdigk956u0 . The data structure used in this example is a linked list, so it includes some details that we do not need (e.g., adding the dummy element). The rest of the explanations in the first 3 minutes of the minutes should provide a great visualized explanation.

Example 2:

- input: list_1 = [1, 2, 3, 5], list_2 = [8, 0, 2]
- We can see that list_2 is not in order so the script will end with an assert error

You are highly recommended to attempt the questions yourself before you look at the solution as a way to learn.

Q1.1 Based on the description above, declare the function and write the function recipe including test cases. You can fill the rest of the body of the function with a pass statement for now.

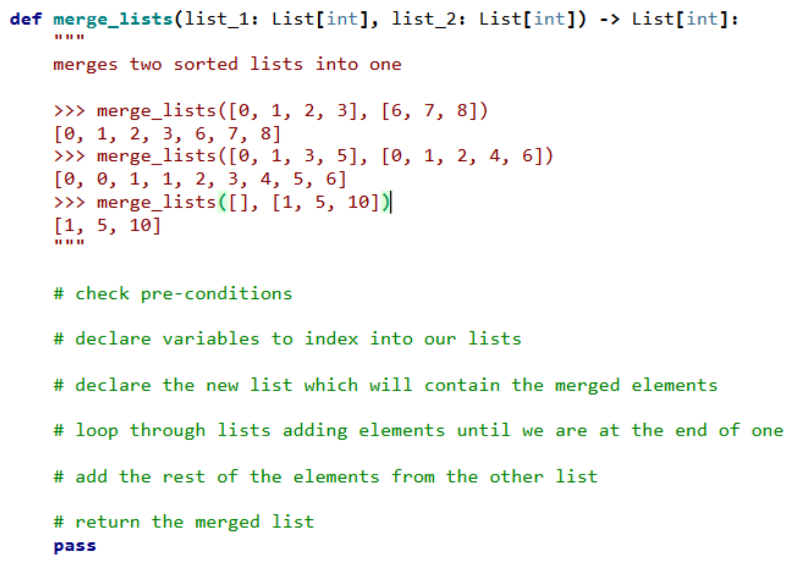
Solution:

```
def merge_lists(list_1: List[int], list_2: List[int]) -> List[int]:
    """
    merges two sorted lists into one

    >>> merge_lists([0, 1, 2, 3], [6, 7, 8])
    [0, 1, 2, 3, 6, 7, 8]
    >>> merge_lists([0, 1, 3, 5], [0, 1, 2, 4, 6])
    [0, 0, 1, 1, 2, 3, 4, 5, 6]
    >>> merge_lists([], [1, 5, 10])
    [1, 5, 10]
    """

    pass
```

(note: remember to do the import: from typing import List)

Q1.2 Add comments to the function, planning out each step you need to take to achieve the function goal.

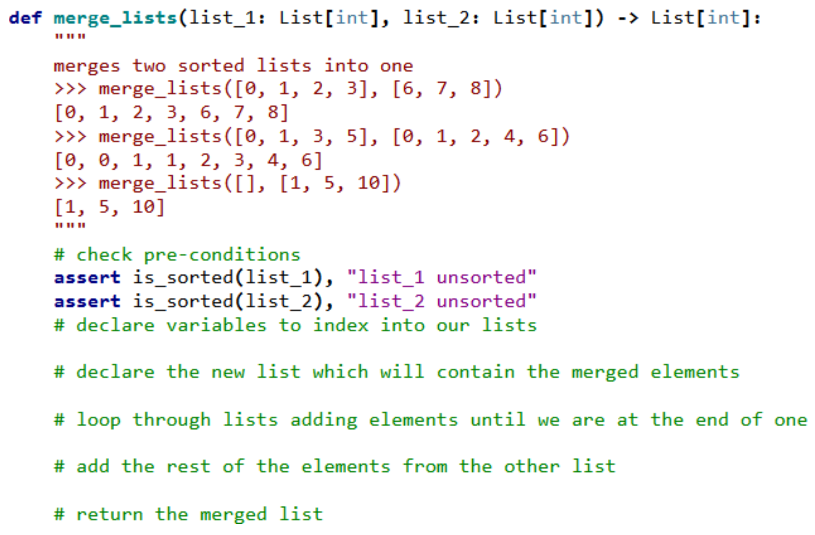
Solution:



```
def merge_lists(list_1: List[int], list_2: List[int]) -> List[int]:
    """
    merges two sorted lists into one

    >>> merge_lists([0, 1, 2, 3], [6, 7, 8])
    [0, 1, 2, 3, 6, 7, 8]
    >>> merge_lists([0, 1, 3, 5], [0, 1, 2, 4, 6])
    [0, 0, 1, 1, 2, 3, 4, 5, 6]
    >>> merge_lists([], [1, 5, 10])
    [1, 5, 10]
    """

    # check pre-conditions

    # declare variables to index into our lists

    # declare the new list which will contain the merged elements

    # loop through lists adding elements until we are at the end of one

    # add the rest of the elements from the other list

    # return the merged list
    pass
```

Q1.3 Check pre-conditions. The pre-condition we need to check is if the lists are in ascending order (i.e., they are sorted) (Note: we skipped the step of checking whether they are int. Can you try it yourself?). To do this we will write another function called is_sorted() which will take as input a list and output whether or not the elements are sorted in ascending order, so that we can modularize the code for the purposes of cleaner structure, code being able to reuse and code readability. We will write the function later, for now we will assume it is done so we can write our assert statements.

Solution:



```
def merge_lists(list_1: List[int], list_2: List[int]) -> List[int]:
    """
    merges two sorted lists into one
    >>> merge_lists([0, 1, 2, 3], [6, 7, 8])
    [0, 1, 2, 3, 6, 7, 8]
    >>> merge_lists([0, 1, 3, 5], [0, 1, 2, 4, 6])
    [0, 0, 1, 1, 2, 3, 4, 5, 6]
    >>> merge_lists([], [1, 5, 10])
    [1, 5, 10]
    """
    # check pre-conditions
    assert is_sorted(list_1), "list_1 unsorted"
    assert is_sorted(list_2), "list_2 unsorted"
    # declare variables to index into our lists

    # declare the new list which will contain the merged elements

    # loop through lists adding elements until we are at the end of one

    # add the rest of the elements from the other list

    # return the merged list
```

Q1.4 Declare the variables to index into our lists and the new list that will contain the merged elements. In order to merge the two input lists we will have to compare their elements to make sure we add the smallest elements first. To get the elements within the list we need to index into them. We will create separate variables for tracking the current index of each list.

Solution:

```
def merge_lists(list_1: List[int], list_2: List[int]) -> List[int]:
    """
    merges two sorted lists into one

    >>> merge_lists([0, 1, 2, 3], [6, 7, 8])
    [0, 1, 2, 3, 6, 7, 8]
    >>> merge_lists([0, 1, 3, 5], [0, 1, 2, 4, 6])
    [0, 0, 1, 1, 2, 3, 4, 5, 6]
    >>> merge_lists([], [1, 5, 10])
    [1, 5, 10]
    """

    # check pre-conditions
    assert is_sorted(list_1), "list_1 unsorted"
    assert is_sorted(list_2), "list_2 unsorted"

    # declare variables to index into our lists
    i = 0 # for iterating through list_1
    j = 0 # for iterating through list_2

    # declare the new list which will contain the merged elements
    merged_list = []
    # loop through lists adding elements until we are at the end of one

    # add the rest of the elements from the other list

    # return the merged list
```

Q1.5 Write the while loop conditional statement. You may leave the body as a pass statement for now.

Solution:

```
def merge_lists(list_1: List[int], list_2: List[int]) -> List[int]:
    """
    merges two sorted lists into one

    >>> merge_lists([0, 1, 2, 3], [6, 7, 8])
    [0, 1, 2, 3, 6, 7, 8]
    >>> merge_lists([0, 1, 3, 5], [0, 1, 2, 4, 6])
    [0, 0, 1, 1, 2, 3, 4, 5, 6]
    >>> merge_lists([], [1, 5, 10])
    [1, 5, 10]
    """

    # check pre-conditions
    assert is_sorted(list_1), "list_1 unsorted"
    assert is_sorted(list_2), "list_2 unsorted"

    # declare variables to index into our lists
    i = 0 # for iterating through list_1
    j = 0 # for iterating through list_2

    # declare the new list which will contain the merged elements
    merged_list = []

    # loop through lists adding elements until we are at the end of one
    while i < len(list_1) and j < len(list_2):
        pass

    # add the rest of the elements from the other list

    # return the merged list
```

Q1.6 Complete the while loop. We will compare the current item of each list and add the smaller one to the merged list. if they are equal we will just add the element from the second list, the order here does not matter. When we add an item to the merged list we will iterate the corresponding counter from that item's original list. To add an item to a list we can call the append() method

Solution:

```python
def merge_lists(list_1: List[int], list_2: List[int]) -> List[int]:
    """
    merges two sorted lists into one

    >>> merge_lists([0, 1, 2, 3], [6, 7, 8])
    [0, 1, 2, 3, 6, 7, 8]
    >>> merge_lists([0, 1, 3, 5], [0, 1, 2, 4, 6])
    [0, 0, 1, 1, 2, 3, 4, 5, 6]
    >>> merge_lists([], [1, 5, 10])
    [1, 5, 10]
    """

    # check pre-conditions
    assert is_sorted(list_1), "list_1 unsorted"
    assert is_sorted(list_2), "list_2 unsorted"

    # declare variables to index into our lists
    i = 0 # for iterating through list1
    j = 0 # for iterating through list2

    # declare the new list which will contain the merged elements
    merged_list = []

    # loop through lists adding elements until we are at the end of one
    while i < len(list_1) and j < len(list_2):
        if list_1[i] < list_2[j]:
            merged_list.append(list_1[i])
            i += 1
        else:
            merged_list.append(list_2[j])
            j += 1
    # add the rest of the elements from the other list

    # return the merged list
```

Q1.7 Add the rest of the elements from the list we have not iterated completely through and return the merged list. The previous while loop will exit once one of the lists has been completely explored. Because we know both lists are sorted, that means that all the elements in the other list must be greater than or equal to the elements we have already added to our new list. Therefore, we can add them all to the merged list. We can use the extend() method to add one or more items to a list at once. We will extend the mergedList by the remaining items in the list we have not finished iterating through

Solution:

```python
def merge_lists(list_1: List[int], list_2: List[int]) -> List[int]:
    """
    merges two sorted lists into one

    >>> merge_lists([0, 1, 2, 3], [6, 7, 8])
    [0, 1, 2, 3, 6, 7, 8]
    >>> merge_lists([0, 1, 3, 5], [0, 1, 2, 4, 6])
    [0, 0, 1, 1, 2, 3, 4, 5, 6]
    >>> merge_lists([], [1, 5, 10])
    [1, 5, 10]
    """

    # check pre-conditions
    assert is_sorted(list_1), "list_1 unsorted"
    assert is_sorted(list_2), "list_2 unsorted"

    # declare variables to index into our lists
    i = 0 # for iterating through list1
    j = 0 # for iterating through list2

    # declare the new list which will contain the merged elements
    merged_list = []

    # loop through lists adding elements until we are at the end of one
    while i < len(list_1) and j < len(list_2):
        if list_1[i] < list_2[j]:
            merged_list.append(list_1[i])
            i += 1
        else:
            merged_list.append(list_2[j])
            j += 1

    # add the rest of the elements from the other list
    if i >= len(list_1):
        merged_list.extend(list_2[j:])
    else:
        merged_list.extend(list_1[i:])

    # return the merged list
    return merged_list
```

Q1.8 Declare the is_sorted() function and write the function recipe and test cases. Now that we have finished the merge_lists() function we can write the helper function that we used for checking our preconditions.

Solution:

```python
def is_sorted(nums: List[int]) -> bool:
    """
    determines if a list is sorted

    >>> is_sorted([1, 2, 2, 3, 4])
    True
    >>> is_sorted([9, 5, 10])
    False
    >>> is_sorted([6])
    True
    """

    pass
```

Q1.9 Check pre-conditions and add comments to the is_sorted() function planning out each step you need to take to achieve the function goal

Solution:

```python
def is_sorted(nums: List[int]) -> bool:
    """
    determines if a list is sorted
    >>> is_sorted([1, 2, 2, 3, 4])
    True
    >>> is_sorted([9, 5, 10])
    False
    >>> is_sorted([6])
    True
    """

    # check pre-condition
    assert type(nums) == list, "the input must be a list"

    # create variable to index into the list

    # loop through elements checking if current element is >= the last
```

Q1.10 Create the variable for indexing into the list and write the while loop conditional statement. You may leave the body as a pass statement for now. The reason for setting the variable tracking the list index to 1 will be explained in the next step.

Solution:

```python
def is_sorted(nums: List[int]) -> bool:
    """
    determines if a list is sorted

    >>> is_sorted([1, 2, 2, 3, 4])
    True
    >>> is_sorted([9, 5, 10])
    False
    >>> is_sorted([6])
    True
    """

    # check pre-condition
    assert type(nums) == list, "the input must be a list"

    # create variable to index into the list
    i = 1

    # loop through elements checking if current element is >= the last
    while i < len(nums):
        pass
```

Q1.11 Finish the while loop and return whether or not the list was sorted. For a list to be sorted in ascending order, each subsequent element in the list must be greater than or equal to the previous item in the list. So in this while loop we will check if the item at the current index is greater than or equal to the element one index before. If this condition is not met at each index then we know the list is not sorted in ascending order. The reason we set the index to start at 1 is because the first item in the list is technically sorted already as there is nothing before it so there is nothing to check in that case.

Solution:

```
def is_sorted(nums: List[int]) -> bool:
    """
    determines if a list is sorted

    >>> is_sorted([1,2,2,3,4])
    True
    >>> is_sorted([9,5,10])
    False
    >>> is_sorted([6])
    True
    """

    # check pre-condition
    assert type(nums) == list, "the input must be a list"

    # create variable to index into the list
    i = 1

    # loop through elements checking if current element is >= the last
    while i < len(nums):
        if nums[i] < nums[i - 1]:
            return False
        i += 1
    return True
```

You have completed this task. Congratulations!

Submission

- Go to eClass -> our course -> Week 10 -> Practice -> Lab -> Lab 10 Submission
- Copy your code to Task 1
- You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission.

Rubrics:

- You will get full marks for this question if you submit the solution (with no typo) to the required location on Prairielearn before the deadline.

Task 2: Debugging (30 pts)

For this task, you will debug a function called “contains_duplicates()” wrote by Sam. This function receives a list of numbers as input and will output True if the list contains duplicate elements and False if all elements are distinct.

For example, if you had the following list:

[1, 2, 3, 4, 1]

contains_duplicates would return True as the element 1 is in the list more than once

For another example, suppose you had the following list:

[5, 6, 7, 8]

contains_duplicates would return False as each element appears in the list once.

Note: there are simpler ways of achieving the goal than what Sam wrote. Can you figure it out after debug the code? This is an optional exercise.

Submission

- Copy the Python code to Lab 10 -> Task 2 on Prairielearn.
- You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission
- Note: In order for the autograder to work properly:
  - You must NOT change the name of the function
  - You must NOT change the order of the arguments

Rubric

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
  - you will receive 5 points if your script correctly provides a solution for general test case e.g., [1, 2, 3, 4, 1], ans: True
  - you will receive 5 points if your script correctly provides a solution for condition – no duplicate, e.g., [5, 6, 7, 8], ans: False
  - you will receive 5 points if your script correctly provides a solution for condition – long list, e.g., [2, 0, 1, 4, 5, 9, 10, 0], ans: True
  - you will receive 5 points if your script correctly provides a solution for Edge case – singleton, e.g., [0], ans: False
  - You will 5 points for each hidden case (2 hidden cases in total)

Task 3: Implementation (10 pts)

For this task, you will write a function called “majority_element()” which will receive a list of ints and return the majority element in that list. The definition of a majority element is any element that appears most frequently. If there are multiple element appears equally frequently, then returns all of them as a list and the elements must be in ascending order in this list.

Let’s give an example. Suppose you have the following list:

[2, 2, 2, 3, 1, 2, 2, 3, 3]

looking at this list, we can figure out which elements appear, and how many times:

element	appearances
1	1
2	5
3	3

we can see now that element 2 appears 5 times, which is most frequently. Therefore the answer is [2].

As another example, suppose you have the following list:

[3, 3, 1, 1]

In this example, both number 1 and number 3 appear two times. Therefore, it should return [1, 3]

Requirement

- You cannot use any other libraries or packages to get the answer for you

Submission

- Copy the Python code to Lab 10 -> Task 3 on Prairielearn.
- You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission.
- Note: In order for the autograder to work properly:
  - You must NOT change the name of the function
  - You must NOT change the order of the arguments

Rubric

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
  - You will receive 0.25 pt for function signature

- You will receive 0.25 pt for function description
- You will receive 0.5 pt for doctest (At least 4, and they cannot be the cases below)
- You will receive 1 pt if your script correctly provides the solution to: general case, e.g., [2, 2, 2, 3, 1, 2, 2, 3, 3], ans: [2]
- You will receive 1 pt if your script correctly provides the solution to: condition – multiple majority numbers, e.g., [3, 3, 1, 1], ans: [1, 3]
- You will receive 1 pt if your script correctly provides the solution to: condition – all elements are the same, e.g., [5, 5, 5, 5, 5], ans: [5]
- You will receive 1 pt if your script correctly provides the solution to: condition – long list, e.g., [1, 3, 2, 1, 3, 3, 3, 1, 2, 3, 2, 1, 2, 2, 1], ans: [1, 2, 3]
- You will receive 1 pt if your script correctly provides the solution to: edge case – empty, e.g., [], ans: []
- You will receive 1 pt if your script correctly provides the solution to: element contract violation, e.g., ['1', '2'], ans: AssertionError
- You will receive 1 pt if your script correctly provides the solution to: parameter contract violation, e.g., (3, 3, 4, 5), ans: AssertionError
- You will get 1 points for each hidden case (2 hidden cases in total)

Task 4: Implementation (20 pts)

Imagine you are tasked with developing a simple student grade management system for school.

Given a dictionary called subject_grade to represent a student's current grade, where the key is the subject names (strings) and the value is the grades for the subject of this student (integers).

subject_grade = {"Math": 90, "English": 75, "Introduction to Python": 95, "Biology ":88}

Your task is write a function to update the information. There are two ways of doing so:

- Task 4.1: Update the original dictionary in the function (do you need to return it in this case?)
- Task 4.2: Create a new dictionary and return the new dictionary

Please write the two functions based on the instructions below. Please note that these two functions are not challenging in terms of implementation, but involves important conceptual understanding that we have talked about in class.

Task 4.1

You task is to write the update_subject_grade() function, and here is the function contract:

update_subject_grade(subject_grade, subject, new_grade) -> None

This function updates the new grade by changing the previous grade in the subject_grade to new_grade of the subject, and returns the nothing.

Below are three examples of how update_subject_grade should work.

- Example 1
  - update_subject_grade({"Math": 90, "English": 75, "Introduction to Python": 95, "Biology": 88}, "Math", 70)
  - It should update the math grade in the dictionary itself, but returns nothing.
- Example 2
  - update_subject_grade({"Math": 90, "English": 75, "Introduction to Python": 95, "Biology": 88}, "Math", 70)
  - It will result in AssertionError: grade must be integer
- Example 3
  - update_subject_grade({"Math": 90, "English": 75, "Introduction to Python": 95, "Economy", 70)
  - It will result in AssertionError: subject must be already exist in subject_grade

As shown in the example, update_subject_grade returns nothing but modify the original dictionary.

When using the function, the new_grade must be an integer and the subject must already existing in the subject_grade.

Submission

- Copy the Python code to Lab 10 -> Task 4.1 on Prairielearn.
- You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission
- Note: In order for the autograder to work properly:
    - You must NOT change the name of the function
    - You must NOT change the order of the arguments

Rubrics:

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
    - Doctest is limited and may not be suitable for this particular task. Therefore, we will not require you to write doctests.
    - You will receive 0.4 points if you provide function description.
    - You will receive 0.6 points if you provide the correct signature.
    - (Case 1) You will receive 2 pts if your script correctly provides a solution for
        - update_subject_grade({"Math": 90, "English": 75, "Introduction to Python": 95, "Biology": 88}, "Math", 70)
        - Ans: The function itself returns nothing, the dict passed in as parameter is updated to {"Math": 70, "English": 75, "Introduction to Python": 95, "Biology": 88}
    - (Case 2) You will receive 2 pts if your script correctly provides a solution for
        - update_subject_grade({"Math": 90, "English": 75, "Introduction to Python": 95, "Biology": 88}, "English", 55)
        - Ans: The function itself returns nothing, the dict passed in as parameter is updated to {"Math": 90, "English": 55, "Introduction to Python": 95, "Biology": 88}
    - (Case 3) You will receive 2 pts if your script correctly provides a solution for
        - update_subject_grade({"Math": 90, "English": 75, "Introduction to Python": 95, "Biology": 88}, "English", 60.5)
        - Ans: AssertionError
    - (case 4) You will receive 2 pts if your script correctly provides a solution for
        - update_subject_grade({"Math": 90, "English": 75, "Introduction to Python": 95, "Biology": 88}, "Spanish", 90)
        - Ans: AssertionError
    - You will receive 1 points for providing a solution for one other test cases (hidden test case)

Task 4.2

You task is to write the return_subject_grade() function, and here is the function contract:

return_subject_grade(subject_grade, subject, new_grade) -> dict

This function creates a copy of the original dictionary subject_grade, update the grade for specified subject with the new grade in the copied dictionary, and return the updated dictionary without altering the original dictionary. The function is otherwise similar to the function in task 4.1.

After completing the lab, think about real-world scenarios where you need to modify the parameters that pass in, and where you should not alter it but create a copy of it.

Submission

- Copy the Python code to Lab 10 -> Task 4.2 on Prairielearn.
- You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission
- Note: In order for the autograder to work properly:
  - You must NOT change the name of the function
  - You must NOT change the order of the arguments

Rubrics:

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
  - You will receive 0.4 points if you provide two correct distinct doctests.
  - You will receive 0.2 points if you provide function description.
  - You will receive 0.4 points if you provide correct signature.
  - (Case 1) You will receive 2 points if your script correctly provides a solution for
    - return_subject_grade({"Math": 90, "English": 75, "Introduction to Python": 95, "Biology": 88}, "Math", 100)
    - Ans: {"Math": 70, "English": 75, "Introduction to Python": 95, "Biology": 88} and the dict passed in as parameter is not modified
  - (Case 2) You will receive 2 points if your script correctly provides a solution for
    - return_subject_grade({"Math": 90, "English": 75, "Introduction to Python": 95, "Biology": 88}, "English", 55)
    - Ans: {"Math": 90, "English": 55, "Introduction to Python": 95, "Biology": 88} and the dict passed in as parameter is not modified

- (Case 3) You will receive 2 points if your script correctly provides a solution for
    - return_subject_grade({"Math": 90, "English": 75, "Introduction to Python": 95, "Biology": 88, "Introduction to Machine Learning": 97, "Database": 100}, "Introduction to Machine Learning", 56)
    - Ans: AssertionError

- (Case 4) You will receive 2 points if your script correctly provides a solution for
    - return_subject_grade({"Math": 90, "English": 75, "Introduction to Python": 95, "Biology": 88, "Introduction to Machine Learning": 97, "Database": 100} and the dict passed in as parameter is not modified
    - Ans: {"Math": 90, "English": 75, "Introduction to Python": 95, "Biology": 88, "Introduction to Machine Learning": 56, "Database": 100}
- You will receive 1 point for providing a solution for one other test cases

Task 5: Implementation (10 pts)

In this task, you will implement a function and no starter code is provided. The function should be called invert_dict. This function accepts a dictionary of students grade with the student's names as the key, and the grades as the value, and converts it to another dictionary with he grade being the key and the name being the value. For example:

```
>>> invert_dict({"Pikachu": 99, "Snorlex": 30})

{99: ['Pikachu'], 30: ['Snorlex']}
```

Note that in the inverted dictionary, the values are lists. Think about why it should be a list. Please also think about why we need to invert to this dictionary.

Submission

Copy the Python code to Lab 10 -> Task 5 on Prairielearn.

You can resubmit your code as many times as you need, but you need to wait for 5 minutes before submission (You need to spend some time debugging!).

Rubrics:

- You will not receive any mark if you do not submit it to the designated location on Prairielearn before the deadline
- You will not receive any mark if your code violates the above requirements
- Otherwise
    - You will receive 0.5 pts if you provide a description of the function.
    - You will receive 0.5 pts if you provide the argument annotations correctly
    - You will receive 1 pt if you include at least 2 test cases (and they cannot be the same as test cases provided) and your code pass those test cases
    - You will receive 2 pts if your function passes general test (e.g., invert_dict({"Pikachu": 99, "Snorlex": 30}), expected output: {99: ['Pikachu'], 30: ['Snorlex']})
    - You will receive 2 pts if your function passes the corner test - zero data (e.g., invert_dict({}), expected output: {})
    - You will receive 2 pts if your function passes the condition test – same grades (e.g., invert_item({"Pikachu": 99, "Snorlex": 30, "Bulbasaur": 30}), expected output: {99: ['Pikachu'], 30: ['Snorlex', 'Bulbasaur']})
    - You will receive 2 pts if your function passes the contract violation (e.g., invert_item([30, 20]), expected output: AssertionError)