Carleton University
Department of Systems and Computer Engineering
ECOR 1041 — Computation and Programming — Winter 2021

Lab 6 - Learning About Lists

Objective

- To learn some of the operators supported by Python's `list` type, and some of the built-in functions that operate on lists.
- To develop functions that take lists as arguments and/or create and return lists.
- (Parts 2 and 3) To develop some functions that use loops to process lists

Submitting Lab Work for Grading

You don't have to finish the lab by the end of your lab period. The deadline for submitting lab work to cuLearn for grading is shown in the *Wrap Up* section of this document. Solutions that are emailed to your instructor or a TA will not be graded, even if they are emailed before the deadline.

Please read *Important Considerations When Submitting Files to cuLearn*, on the last page of the course outline.

Prerequisite Reading

Before you start Part 1, read these sections from *Practical Programming*, Chapter 8, *Storing Collections of Data Using Lists*:

- all sections, except for the following: *Slicing Lists* (pp. 137 - 139), *Working with Lists of Lists* (pp. 142 - 144)
- read the box *Where Did My List Go?* at the top of page 143

Before you start Parts 2 and 3, read these sections from *Practical Programming*, Chapter 9: *Repeating Code Using Loops*:

- *Processing Items in a List* (pp. 149 - 151)
- *Processing Characters in Strings* (pp. 151 - 152)
- *Looping Over a Range of Numbers* (pp. 152 - 154)
- *Processing Lists Using Indices*, up to but not including *Processing Parallel Lists Using Indices* (pp. 154 - 156)

Part 1

This part consists of three exercises. **None of the solutions requires a loop.**

Your solutions can use:

- the `[]` operator to get and set list elements (e.g., `a = lst[i]` and `lst[i] = a`).
- the `in` and `not in` operators (e.g., `a in lst`).
- the list concatenation operator (e.g., `lst1 + lst2`).
- the list replication operator (e.g., `lst * n` or `n * lst`).
- Python's built-in `len`, `min` and `max` functions.

Your solutions **cannot** use:

- `for` or `while` loops.
- list slicing (e.g., `slice = lst[i : j]` or (`lst[i : j] = t`).
- the `del` statement (e.g., `del lst[0]`).
- Python's built-in `sum`, `reversed` and `sorted` functions.
- any of the methods provided by type `list`; e.g., `append`, `clear`, `copy`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse` and `sort`.
- list comprehensions (which aren't taught in this course).

Debugging hint: If a function fails one or more tests and inspecting your code doesn't reveal the problem, go to the Python Tutor website (pythontutor.com). Copy/paste your function from Wing 101 into the PyTutor editor. You'll need to write a short script to call the function (suggestion: use examples in your docstring as a starting point). Execute the function, statement-by-statement, and use the memory diagrams produced by PyTutor to help you locate the flaw in your solution.

Exercise 1

Step 1: Create a new editor window and save it. Use lab6ex1.py as the file name.

Step 2: Use the function design recipe from Chapter 3 of *Practical Programming* to develop a function named `middle_way`. The function takes two lists that each contain three integers. The function returns a new list containing their middle elements. For example, when the arguments are `[1, 2, 3]` and `[4, 5, 6]`, the function returns `[2, 5]`.

Your function definition must have type annotations and a complete docstring. Use the Python shell to test your function.

Exercise 2

Step 1: Create a new editor window and save it. Use lab6ex2.py as the file name.

Step 2: Use the function design recipe to develop a function named `make_ends`. The function takes a list of integers. Assume that the list will not be empty. The function returns a new list containing the first and last elements from the original list. For example, when the argument is `[4, 5, 6, 7]`, the function returns `[4, 7]`.

Your function definition must have type annotations and a complete docstring. Use the Python shell to test your function.

Exercise 3

Step 1: Create a new editor window and save it. Use lab6ex3.py as the file name.

Step 2: Use the function design recipe to develop a function named `common_end`. The function takes two lists of integers that are not empty, but which may have different lengths. The function returns `True` if they have the same first element or the same last element or if the first and last elements of both lists are the same. Otherwise, the function returns `False`.

Your function definition must have type annotations and a complete docstring. Use the Python shell to test your function.

Part 2

This part consists of four exercises. **Each solution requires exactly one loop.**

Your solutions can use:

- the `[]` operator to get and set list elements (e.g., `a = lst[i]` and `lst[i] = a`).
- the `in` and `not in` operators (e.g., `a in lst`).
- the list concatenation operator (e.g., `lst1 + lst2`).
- the list replication operator (e.g., `lst * n` or `n * lst`).
- Python's built-in `len`, `min` and `max` functions, **unless otherwise noted**.

Your solutions **cannot** use:

- list slicing (e.g., `slice = lst[i : j]` or (`lst[i : j] = t`).
- the `del` statement (e.g., `del lst[0]`).
- Python's built-in `sum`, `reversed` and `sorted` functions.
- any of the methods provided by type `list`; e.g., `append`, `clear`, `copy`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse` and `sort`.
- list comprehensions.

Exercise 4

Step 1: Create a new editor window and save it. Use lab6ex4.py as the file name.

Step 2: Use the function design recipe to develop a function named `count_evens`. The function has one parameter, which is a list of integers. The list may be empty. The function returns the number of even integers in the list.

Your function definition must have type annotations and a complete docstring. Use the Python shell to test your function.

Exercise 5

Step 1: Create a new editor window and save it. Use lab6ex5.py as the file name.

Step 2: Use the function design recipe to develop a function named `big_diff`. The function has one parameter, which is a list of integers. Assume that the list has at least two integers. The function returns the difference between the largest and smallest elements in the list. For example, when the function's argument is `[10, 3, 5, 6]`, the function returns 7.

Although the most Pythonesque solution is:

```
def big_diff(nums):
    return max(nums) - min(nums)
```

don't use this as your solution. Your function is **not** permitted to call Python's `min` and `max` functions. Your function must have **exactly one** loop.

Your function definition must have type annotations and a complete docstring. Use the Python shell to test your function.

Exercise 6

Step 1: Create a new editor window and save it. Use lab6ex6.py as the file name.

Step 2: Use the function design recipe to develop a function named `has22`. The function takes a list of integers, which may be empty. The function returns `True` if the list contains a 2 next to a 2. Otherwise, the function returns `False`. For example, when the function's argument is `[1, 2, 2, 3]`, the function returns `True`. When the function's argument is `[4, 2, 3, 2]`, the function returns `False`.

Your function must have **exactly one** loop.

Your function definition must have type annotations and a complete docstring. Use the Python shell to test your function.

Exercise 7

Step 1: Create a new editor window and save it. Use lab6ex7.py as the file name.

Step 2: The *centered average* of a list of numbers is the arithmetic mean of all the numbers in the list, except for the smallest and largest values. If the list has multiple copies of the smallest or largest values, only one copy is ignored when calculating the mean. For example, given the list `[1, 1, 5, 5, 10, 8, 7]`, one 1 and the 10 are ignored, and the centered average is the mean of the remaining values; that is, (1 + 5 + 5 + 8 + 7) / 5 = 5.2.

Use the function design recipe to develop a function named `centered_average`. The function takes a list of integers. Assume that the list has at least three integers. The function returns the centered average of the list.

Your function can call Python's `min` and `max` functions, but it is **not permitted** to call Python's `sum` function. Your function must have **exactly one** loop.

Your function definition must have type annotations and a complete docstring. Use the Python shell to test your function.

Part 3

This part consists of three exercises. **Each solution requires exactly one loop.**

Your solutions can use:

- the `[]` operator to get and set list elements (e.g., `a = lst[i]` and `lst[i] = a`).
- the `in` and `not in` operators (e.g., `a in lst`).
- the list concatenation operator (e.g., `lst1 + lst2`), unless otherwise noted.
- the list replication operator (e.g., `lst * n` or `n * lst`).
- Python's built-in `len`, `min` and `max` functions.
- any of the methods provided by type `list`; e.g., `append`, `clear`, `copy`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse` and `sort`. (See *Practical Programming*, pages 141-142.)

Your solutions **cannot** use:

- list slicing (e.g., `slice = lst[i : j]` or (`lst[i : j] = t`).
- the `del` statement (e.g., `del lst[0]`).
- Python's built-in `sum`, `reversed` and `sorted` functions.
- list comprehensions.

Exercise 8

Step 1: Create a new editor window and save it. Use lab6ex8.py as the file name.

Step 2: Use the function design recipe to develop a function named `bank_statement`. The function takes a list of floating point numbers, which will always have at least one number. Positive numbers represent deposits into a bank account and negative numbers represent withdrawals from the account.

The function returns a new list containing three numbers: the first will be the sum of the deposits, the second (a negative number) will be the sum of the withdrawals, and the third will be the current account balance. These numbers must be rounded to two digits of precision after the decimal point (read Chapter 3, pages 33-34).

Your function must have **exactly one** loop. Your function can call Python's list methods, but this isn't necessary.

Your function definition must have type annotations and a complete docstring. Use the Python shell to test your function.

Note: when the list returned by the function is displayed, numbers such as 15.0 or -17.3 will be displayed with one digit after the decimal point instead of two, for example, 15.00 or 17.30 (the way amounts of money are normally represented). This is ok.

Exercise 9

Step 1: Create a new editor window and save it. Use lab6ex9.py as the file name.

Step 2: Use the function design recipe to develop a function named `divisors`. The function takes a positive integer *n* and returns a list containing all the positive divisors of *n*. For example, if `divisors` is called with argument 6, the list will contain 1, 2, 3 and 6. If `divisors` is called with argument 9, the list will contain 1, 3 and 9.

Your function must have **exactly one** loop. Your function can call Python's list methods, but this isn't necessary.

Your function definition must have type annotations and a complete docstring. Use the Python shell to test your function.

Exercise 10

Step 1: Create a new editor window and save it. Use lab6ex10.py as the file name.

Step 2: Use the function design recipe to develop a function named `reverse` that takes a list,

which may be empty. The function returns a new list that contains all the elements from the original list in reverse order. For example, when the function's argument is `[4, 2, 3, 2]`, the function returns the new list `[2, 3, 2, 4]`. When the function's argument is an empty list, it returns a new empty list.

Your function must have **exactly one** loop. Your function **is not** permitted to call Python's built-in `reversed` function or the `reverse` method provided by type `list`.

Your function definition must have type annotations and a complete docstring. Use the Python shell to test your function.

Extra Practice

Use PyTutor to visualize the execution of your solutions. Remember, PyTutor doesn't have a shell. After copying your function definitions into the PyTutor editor, type assignment statements that call the functions and assign the returned values to variables.

Instructions for submitting your lab work are on the next page.

Wrap Up

Please read *Important Considerations When Submitting Files to cuLearn*, on the last page of the course outline. The submission deadlines for this lab are:

Lab Section	Lab Date/Time	Submission Deadline (Ottawa Time)
3A	Monday, 8:35 - 11:25	Thursday, Feb. 25, 2021, 23:55
2A	Tuesday, 14:35 - 17:25	Thursday, Feb. 25, 2021, 23:55
1A	Friday, 12:35 - 14:25	Saturday, Feb. 27, 2021, 23:55
4A	Friday, 18:05 - 20:55	Saturday, Feb. 27, 2021, 23:55

1. Login to cuLearn. Go to the cuLearn page **for your lab section** (not the main course page).
2. Click the link Submit Lab 6 for grading.
3. Click Add submission.
4. Submit your solutions by dragging the ten files containing your functions (lab6ex1.py through lab6ex10.py) to the File submissions box.
5. Note that you have to confirm that your solutions are your own work before you can submit them. Click Save changes. The Submissions status page will be displayed. The line labelled *Submission status* will confirm that your submission is now *Submitted for grading*. The line labelled *File submissions* will list the names of the files you've submitted.
6. You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the deadline. Only the most recent submission is saved by cuLearn. To submit revised solutions, follow steps 7 through 9.
7. In cuLearn, click the link Submit Lab 6 for grading.
8. Click Remove submission to delete your previous submission. Click Continue to confirm that you want to do this.
9. Click Edit submission. Repeat steps 4 and 5 to submit your revised solutions.

History: Feb. 19, 2021 (initial release)