

CptS 111 Laboratory 4: Lists, Loops, and Some Stats

Goal: This lab uses lists and the `range()` function in various ways. The goal is to make you comfortable with some of the ways lists can be used and to ensure that you understand the behavior of the `range()` function.

You are free to help your classmates and seek help from your classmates. Collaboration is strongly encouraged. However, please make sure the help you provide takes the form of explaining things and you do not merely provide the answers/solutions.

For this lab you should ensure the TA notes your completion of the following “graded” tasks:

Task 2: Using `range()` to Blast Off.

Task 3: Obtaining a `list` of Numbers.

Task 4: Summing a `list` the Hard Way.

Task 5: Finding the Average.

Task 6: Finding the Standard Deviation.

Task 1: `lists`, `for`-Loops, and the `append()` Method.

A `list` can be empty, meaning that it has no elements. By themselves, empty lists are not very useful. However, in some applications we start with an empty list and then build it up. Alternatively, in other applications we start with a non-empty list and remove elements from the list, one at a time, doing something with each element before moving on to the next one. We stop when the list is empty. An empty list consists of square brackets with no elements.

Recall that the template for a `for`-loop is

```
1   for <item> in <iterable>:  
2       <body>
```

where `<iterable>` produces one value for each pass of the loop. On each pass of the loop this value is assigned to `<item>`. As we will see in the coming weeks, in Python various things, including `lists` are iterables. Thus, at the start of the first pass of the loop, the `list`'s first element is assigned to `<item>`. At the start of the second pass, its second element is assigned to `<item>`, and so on. `<item>` serves as the “loop variable.” Any valid identifier can be used for `<item>`. In some loops we are interested in using `<item>` in the body of the loop while in other loops this value is ignored (in such loops `<item>` is typically being used as a counter and there may be no need to use this counter).

The following code starts by demonstrating the behavior of a *non-empty list*. The list `x` defined in line 1 has three elements and is subsequently used in both a `print()` statement and a `for`-loop. Following this, an *empty list* is created and also used in a `print()` statement and a `for`-loop.

```

1 >>> x = [7, 10, 2] # x is a non-empty list.
2 >>> print(x)      # print(x) shows the entire list.
3 [0, 1, 2]
4 >>> # Use a for-loop to print each element of x and some text.
5 >>> for item in x: # Three items in x, thus three lines of output.
6 ...     print(item, "hello")
7 ...
8 7 hello
9 10 hello
10 2 hello
11 >>> y = []       # y is an empty list.
12 >>> print(y)     # print(y) shows brackets but nothing inside.
13 []
14 >>> for item in y: # There are no items in y, so no output is produced.
15 ...     print(item, "hello")
16 ...
17 >>>

```

Recall that lists have a method called `append()` that appends its argument to the end of the given list. The following code starts with an empty list `z`. The user is then solicited for two values that are appended to `z`.

```

1 >>> z = []
2 >>> number = float(input("Enter a number: "))
3 Enter a number: 27.3
4 >>> z.append(number)
5 >>> number = float(input("Enter a number: "))
6 Enter a number: 4.9876
7 >>> z.append(number)
8 >>> print(z)
9 [27.3, 4.9876]

```

A list can be returned by a function. This is illustrated in the following code. Here the code for prompting the user for two values is placed in a function called `get_two()`.

```

1 >>> def get_two():
2 ...     z = []
3 ...     number = float(input("Enter a number: "))
4 ...     z.append(number)
5 ...     number = float(input("Enter a number: "))
6 ...     z.append(number)
7 ...     return z
8 ...
9 >>> xlist = get_two()
10 Enter a number: 47
11 Enter a number: 52
12 >>> print(xlist)
13 [47.0, 52.0]

```

If you have no questions about any of this code, you may move on to the following task. If you do have questions, now is the time to ask!

Task 2: Using `range()` to Blast Off.

`range()` is described in Sec. 6.5 of the textbook. Technically it is an “iterator” but you can think of it as a function that generates a list of integers.¹ In practice `range()` generates the elements of this list one at a time and as needed. But, as will be demonstrated shortly, we can force `range()` to generate all the integers at once by enclosing `range()` in the built-in function `list()`.²

`range()` can be passed either one, two, or three parameters. All parameters must be integers or expressions that evaluate to integers. With the one-parameter form, `range()` will generate a list of integers from zero up to one less than the parameter. So, for example, `list(range(5))` will return the list

```
[0, 1, 2, 3, 4]
```

Note that this list has five elements, i.e., the number of elements in the list corresponds to the argument of `range()`, and yet the maximum value in the list is one less than this argument. When `range()` has a single parameter, this is the `stop` value: *The numbers produced by `range()` never include the stop value!* It is important to remember that `range(n)` produces `n` integers ranging from 0 to `n - 1`.

The three-parameter form of the `range()` function, i.e., the most general form, is `range(start, stop, increment)`. The first parameter, `start`, is the starting value. `start` defaults to zero if not given explicitly.

The third parameter, `increment`, is the step (or increment) between the values that `range()` produces. `increment` defaults to one if not given explicitly. `increment` can be negative, in which case the values in the list are decreasing instead of increasing.

The second parameter, `stop`, is, as mentioned above, the stop value. If two parameters are specified, they are assumed to correspond to `start` and `stop`.

To reiterate, the values produced by `range(start, stop, inc)` are given by

```
[start, start + inc, start + 2 * inc, start + 3 * inc, ... ]
```

This sequence stops just prior to the `stop` value (much as you are supposed to stop in front of a stop sign, even if only barely).

¹For the remainder of the course we will typically refer to `range()` as a function.

²In practice we would *not* use the `list()` function to force this kind of behavior. We are just using it here to facilitate seeing the values `range()` generates.

To demonstrate how `range()` behaves, enter the following at the interactive prompt and make sure you understand the results.

```
list(range(10))
list(range(3, 10))
list(range(3, 10, 3))
list(range(10, 0))
list(range(10, 0, -1))
list(range(10, -1, -1))
```

Now, please answer the following questions (and be prepared to show your answers to the TA at the completion of this task):

- 1. Which of the calls to `range()` shown above produces an empty list?**

- 2. What should be the parameters for `range()` to produce integers that start at 1 and go up to, and include, 10?**

- 3. Assume you have the list `xlist` in your code. Given that `len()` returns the length of its argument, what would you use as the argument to the `range()` function to produce a list of numbers that has as many elements as the length of `xlist`?**

Now our goal is to write a function called `countdown()` that generates specific output. It takes no arguments. When run, `countdown()` uses the `input()` function to prompt the user for a positive integer. It records the input and then uses the `range()` function and a `for`-loop to print a countdown that goes from the user-specified value to zero. At that point the function prints the words “Blast off!”

Here is an example where the function is behaving correctly:

```
1 >>> countdown()
2 Enter a positive integer: 5
3 5
4 4
5 3
6 2
7 1
8 0
9 Blast off!
```

Note! There is one slightly new concept that comes into play in this function having to do with “nesting” (but we have mentioned nesting in connection with the definition of functions). The `countdown()` function will include a `for`-loop. We say that the `for`-loop is nested in the function. You already know we have to indent the body of the function, but we have to further indent the “body” of the code associated with the `for`-loop since the loop is nested in the function. So, the structure of the code will be something like this:

```
1 def countdown():
2     # any code that comes before the for-loop
3     for ...:
4         # code within the body of the for-loop
5     # any code that comes after the for-loop
```

Remember, indentation is a way of identifying blocks of code that are tied together in some context. We indent statements as part of a function definition because we want to identify them as the function body. The same thing applies to the bodies of `for`-loops—to be identifiable, the body of a `for`-loop must be indented more than the loop header itself. Therefore, when a `for`-loop is used in a function definition, the loop header should be indented one level (e.g., using one tab), just like the other statements in the function body, but statements that belongs to the body of the `for`-loop, must be indented further (e.g., using two tabs).

If the `print()` statement that prints “Blast off!” is indented the same amount as the body of the `for`-loop, you will blast-off multiple times. Don’t do this.

Once you have this function behaving properly, demonstrate your solution to the TA. If the TA agrees your code is correct (and if the TA is satisfied with your answers to the questions above), you may move on the next task.

Task 3: Obtaining a `list` of Numbers.

We now want to write a function that combines several of these concepts to allow us to create a list of numbers of arbitrary length (although this length must be specified in advance). Write a function called `get_floats()` that takes no arguments. This function does the following:

1. Creates an empty `list`.
2. Prompts for and obtains the number of data points the user wants to enter. This must be an integer.
3. Uses a `for`-loop to obtain all the user’s data. It is assumed the user will enter `floats`. The prompt used to solicit input may be the same for each entry. Or, you can use a counter so that the prompt changes appropriately for each value (this is demonstrated below). Each value should be appended to the list that was created in step 1.
4. Returns this list.

Here are some additional comments concerning step 3, i.e., comments about what is done in the body of the `for`-loop. Keep in mind that the `input()` function returns a string. This string must be converted to a `float()`. You use the `append()` method to add values to the list. In this particular loop the loop variable isn't needed in the body of the loop except, perhaps, to be shown in the prompt. So, in this case, we are using the header of the loop to specify the number of times the loop will be executed. (Unlike in some of the previous examples, we are not using the header specifically to access elements of a list via the loop variable. The goal here is to build a list, not access elements of a previously created list.)

The following demonstrates the proper behavior of this function.

```
1 >>> get_floats()
2 Enter the number of elements: 3
3 Enter element 1: 25.1
4 Enter element 2: 19.3
5 Enter element 3: 47.8
6 [25.1, 19.3, 47.8]
7 >>> xlist = get_floats()
8 Enter the number of elements: 4
9 Enter element 1: 2
10 Enter element 2: 4
11 Enter element 3: 8
12 Enter element 4: 10
13 >>> print(xlist)
14 [2.0, 4.0, 8.0, 10.0]
```

Once you have this function behaving properly, demonstrate your solution to the TA. If the TA agrees your code is correct, you may move on the next task.

Task 4: Summing a list the Hard Way.

We want to write a function called `summer()` that sums the values in a list of numbers. Actually, Python already provides a built-in function called `sum()` that will do exactly what we want to implement in this task. This built-in function is demonstrated in the following code:

```
1 >>> x = [9, 3, 21, 15]
2 >>> sum(x)
3 48
4 >>> sum([31.12, -16.32, 24.9, 82.0, 14.0])
5 135.7
```

Despite the fact that this function is already available to us, we will write our own version, called `summer()` that accomplishes the same thing. `summer()` takes a single argument: the list whose elements we want to sum. `summer()` does the following:

1. Initializes a variable to zero. This variable will ultimately hold the sum of all the values in the list.

2. Using a `for`-loop, adds the value of each element of the list to the variable you created in step 1.
3. Returns the sum.

Since `summer()` and the built-in function `sum()` behave identically, the code above also demonstrates the correct behavior of `summer()`.

Once you have this function behaving properly, demonstrate your solution to the TA. If the TA agrees your code is correct, you may move on the next task.

Task 5: Averaging Numbers.

Now we want to write a *program* that can be used to average a set of numbers. (The average is also known as the mean.) This program will use the `get_floats()` function from Task 3 and the `summer()` function from Task 4. Additionally, the program will use a function called `average()` that actually calculates the average and the function `main()` to coordinate everything.

Recall that the average of a list of numbers is simply the sum of the values divided by the number of values. Just to provide a bit of mathematical rigor, for a set of numbers x_1, x_2, \dots, x_N , the average is given by

$$\bar{x} = \frac{1}{N} (x_1 + x_2 + \dots + x_N)$$

where N is the number of terms in the list. The average of a set is often indicated with an overbar, i.e., \bar{x} indicates the average of the set of numbers x_k . Another way to write the average is using summation (or sigma) notation:

$$\bar{x} = \frac{1}{N} \sum_{k=1}^N x_k$$

where the uppercase Greek letter Sigma (Σ) tells us to perform a sum. The expression to the right of Sigma is the term to be summed. The expression below Sigma initializes a summation index (in this case k) and the term above Sigma indicates the final value for the summation index.

Using IDLE, open a new window in which to write the program. You should copy the code for `summer()` and `get_floats()` into this window. The very last statement in this window should be a call to `main()`. Prior to this last statement, you should have defined the `average()` and `main()` functions.

The `main()` function takes no arguments and does not return anything. The body of `main()` can consist of as few as three statements. The body of `main()` does the following:

1. Calls the `get_floats()` function to obtain the list of numbers.
2. Calls the `average()` function to calculate the average of the list of numbers. (`average()` is described in more detail below.)

3. Prints the average.

The last two steps can be combined into a single step. (However, in the task we will want these steps to be separate statements so that we can store the average of the list for later recall.)

The `average()` function takes a single argument, the list of numbers to be averaged. It returns a `float` that is the average of the list. `average()` uses `summer()` to calculate the sum of the elements of the list. (Note that `summer()` does not appear in `main()`.) The body of the `average()` function can be a single line: a `return` statement with the appropriate expression following the keyword `return`.

Here is an example of the proper behavior of this program:

```
1 Enter the number of elements: 3
2 Enter element 1: 2.3
3 Enter element 2: 5.9
4 Enter element 3: -3.872
5 Average = 1.442666666667
```

In the following example all the entries are integers and, in theory, the average is an integer, but because `float` division was used, the result is displayed as a `float` (even though it is a whole number).

```
1 Enter the number of elements: 5
2 Enter element 1: 98
3 Enter element 2: 82
4 Enter element 3: 69
5 Enter element 4: 100
6 Enter element 5: 91
7 Average = 88.0
```

Once you are satisfied your program is behaving properly, show your work to the TA. If the TA agrees your program is correct, move on to the next task.

Task 6: Standard Deviation.

If we were only interested in calculating the average of a set of numbers, there is no reason to store the numbers in a list. All we need to obtain the average is the sum of the numbers and this can be calculated “on the fly” without storing old values. The following program accomplishes the same thing as the program you wrote for the previous task:

```
1 def main():
2     """Calculate the average of a set of numbers."""
3     num = int(input("Enter the number of elements: "))
4     sum = 0
5     for i in range(num):
6         sum += int(input("Enter number " + str(i + 1) + ": "))
7     print("Average =", sum / num)
```

8

```
9 main()
```

You should take a minute to ensure you understand exactly how this code works. (You may want to type it in yourself to solidify your understanding.) The statement in line 6 uses the “augmented assignment operator.” This is described in Sec. 2.7.4 of the textbook. The statement `sum += x` is equivalent to `sum = sum + x`.

Although the work you did for the previous tasks was overkill if one only wanted the average, it was definitely not overkill if we want to analyze the values beyond just calculating the average. In this task we will calculate the standard deviation of the numbers.³ The standard deviation is a measure of how much fluctuation there is in a set of numbers after the average has been removed from all the numbers. If all the numbers are equal to the average, then the standard deviation is zero. As part of the calculation of the standard deviation, we must subtract the average from each number. This requires us to calculate the average first and then go back and subtract this from each number, i.e., we have to have a way to recall all the numbers. Because we stored the data in a list, we can easily do this.

The standard deviation is represented by the lowercase Greek letter sigma (σ). For a set of numbers x_1, x_2, \dots, x_N , the standard deviation is given by

$$\sigma = \sqrt{\frac{1}{N} [(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \dots + (x_N - \bar{x})^2]}$$

where \bar{x} is the average value of the set. Recall that in Python we can square a value `z` with the expression `z ** 2` whereas we can calculate the square root with the expression `z ** 0.5`.⁴

For this task you must write a program that will report both the average and the standard deviation of a set of numbers. This program builds on the program you wrote for the previous task, so you should use that program as your starting point (just be sure to save it to a different file). Thus, as before, your program will use the functions `get_floats()`, `summer()`, `average()`, and `main()`. However, you must write one additional function called `std_dev()`.

The function `std_dev()` takes two arguments: a list of numbers and the average of that list. The function returns a float which is the standard deviation of the list. The body of this function does the following:

1. Initializes a variable to zero.
2. Using a `for`-loop, subtracts the average from a number in the list, squares that number, and adds this to the variable initialized in step 1.

³For further discussion of standard deviation, see http://en.wikipedia.org/wiki/Standard_deviation.

⁴The expression given here is the appropriate one when the data used is the entire set of data. At times standard deviation is used with data that represents samples from a larger population. In that case the denominator should be $N - 1$ instead of N in order to obtain an “unbiased” estimate of the standard deviation.

3. The sum of all these values is divided by the number of elements in the list and then the square root is taken. This gives the standard deviation which is returned by the function.

You should modify `main()` so that it calls `std_dev()` to obtain the standard deviation and to print it. The following three runs of the program demonstrate the results you should obtain. Each of these has the same mean but an increasing standard deviation. In the first, all the values are the same and hence the standard deviation is zero.

```
1 Enter the number of elements: 5
2 Enter element 1: 50
3 Enter element 2: 50
4 Enter element 3: 50
5 Enter element 4: 50
6 Enter element 5: 50
7 Average = 50.0
8 Standard deviation = 0.0
```

Here the values are spread out so that they span the range from 30 to 70:

```
1 Enter the number of elements: 5
2 Enter element 1: 30
3 Enter element 2: 40
4 Enter element 3: 50
5 Enter element 4: 60
6 Enter element 5: 70
7 Average = 50.0
8 Standard deviation = 14.1421356237
```

In this final example the values range between 10 and 90:

```
1 Enter the number of elements: 5
2 Enter element 1: 10
3 Enter element 2: 30
4 Enter element 3: 50
5 Enter element 4: 70
6 Enter element 5: 90
7 Average = 50.0
8 Standard deviation = 28.2842712475
```

Once you are satisfied your program is behaving properly, show your work to the TA. If the TA agrees your program is correct, you are done and may leave. Enjoy the weekend!

Homework #4 will be posted in the not-too-distant future (probably by Saturday evening). Email will be sent to the class when it is available. (It will be sent to your WSU address.)