Iteration: the 'for' Loop

In this exercise you will compute the gross weight and CG position for a larger aircraft which may have many stations, and where the number of stations might vary. You will learn to use the `for` statement to repeat the same section of a script over and over again.

On large cargo or passenger aircraft there may be many weight and balance stations, and the way in which the information is recorded and fed to the computer may be different. For example, as a cargo plane is loaded the Loadmaster might fill in the amount of weight added to various stations on a pre-printed form created just for that purpose. For a passenger plane, the crew might use a similar form to record which seats are filled and which are empty. The data would be entered into a computer from one of these forms later on, perhaps by someone else. In such cases it may be more useful to read in the weight and balance data for all of the stations on the list all at once. To do this it is easiest to use a `for` loop to repeat the input step as many times as needed.

You will therefore modify your program from the previous exercise (or write a new one) so that it first reads in the number of stations, followed by the arm and weight information for exactly that many stations, using a `for` loop. After all of the data have been read, the script will compute and display the position of the center of gravity and the total weight. You should use `if` statements, as in the previous exercise, to print out a warning if the center of gravity or weight are not within acceptable limits.

Iteration: Counting up or down in Python

There are several different ways to repeat a set of instructions in Python – what is called "looping". For this exercise we just need the simplest – counting up to a given number. Or counting down, which is almost the same. Here is a simple example you might try:

```
for i in range(10, 0, -1):
    print(i)
print("Blast off!")
```

Here is how it works: the `range` function creates a list of numbers starting with the first number, incremented by the last number (the "increment"), going to (but not including!) the second number. The `for` command sets the variable `i` to each value in the list, in turn. Then every indented line which follows the `for` statement is executed repeatedly, once for each value of `i`. After all that, the non-indented print statement is executed and the script continues from there.

Reading in two numbers

Your program would be more convenient to use if you could type in two values at once, both the arm and weight for a single station. Here is a simple example of how to read two numbers on one line, where they are separated by a comma:

```
x, y = map(float, input("Enter x, y: ").split(","))
print(x+y)
```

This may look a little complicated, but it's less opaque once you've been through how it works:

1. The `input()` function prints the prompt and then reads in whatever is typed, just as it is.*
2. The `split(",")` function is then applied to the results of the `input()` function, and the argument (the thing in the parentheses) tells it to split the input data at a comma.
3. The `map()` function takes another function as its first argument – in this case `float()` – and applies it sequentially to every element in the list from the second argument.
4. Thus all values on one input line are separated and automatically converted to floating-point numbers. This works with any number of values on one line separated by commas, but then the assignment just puts the first two values into the variables `x` and `y`.

Running Python from the Command Line

You have already seen how easy it is to run a Python script when you are editing it with IDLE (just press the F5 key!), or directly by double-clicking on the file icon. There is one other way to run a python script, and even if you don't use it very often, it's useful to know about it. You can also run a Python script from the "command line."

First, you have to open up a window which lets you give commands directly to the computer. This is not the same as the interactive Python shell window in IDLE. Here's what you do:

- ▷ **Windows:** From the Start Menu, open up the "Windows System" folder (on Windows 10) or "Windows Accessories" folder (on Windows 8) and launch "Command Prompt". (On Windows 10 you can use the Windows PowerShell instead.)
- ▷ **macOS:** Use Launchpad (in the dock) to find a folder called "Utilities" or "Other". In that folder you will find an app called "Terminal". Launch that. If you don't have Launchpad in your dock then you can go to the main `/Applications` folder and find the `Utilities` folder and double click there on the `Terminal` app.

* In Python 2 you would have to use `raw_input()` instead of `input()`.

▷ **Raspberry Pi:** Click on the "Terminal" app icon in the menu bar across the top of the screen.

The commands you type into this window are given directly to the computer itself. When the window first opens your commands will operate by default on files in your "home" directory. (In the command line world, a "directory" is another name for a folder.) You probably want to work with files in your Desktop directory (folder). If so, the first command you should give is "`cd`," which stands for "change directory," like so:

```
>   cd Desktop
```

(The "`>`" character represents the prompt printed by the computer, so you don't type that. It might be some other character, depending on what kind of computer you are using, and it might be a lot more than just a single "`>`".)

Now, assuming that you have a Python script in your Desktop folder (directory) called `hello.py`, the command to run it with Python is simply:

```
>   python hello.py
```

Try it, and consult with your instructor if you have any difficulties with this simple example.

Assignment

Your goal for this assignment is to compute the gross weight and CG position for a larger aircraft with any number of stations.

To compete this exercise you must do the following:

1. As with all good programs, your script should start by outputting a brief summary telling the user what it does. (And it should also have proper comments at the beginning of the script.)

2. The first thing your script should ask for and accept is the number of stations for which data follows. After this, read the station weight and arm (in that order, as a pair, separated by a comma) for exactly that many stations, no more and no less.

   Your script should echo the values which it reads, both for confirmation and to make a record of the inputs.

   Negative weights are not allowed. If a negative weight is entered your program should print an error message and treat the weight as zero.

   Keep in mind also that many stations could have zero weight, including the first. This would correspond to blank lines on a pre-made form used by the Loadmaster, or empty seats in a passenger plane. Be careful not to divide by a zero total weight when computing the center of gravity.

Table 1: Weight and balance table for a small aircraft.

Cessna 150 – N6440S	Weight (lbs)	Arm (in)	Moment
Basic Empty Weight (w/ oil)	1107.3	34.15	
Pilot		39.12	
Passenger		39.12	
Fuel (22.5 gallons max)		42.22	
Baggage Area 1 (120 lbs max)		63.77	
Baggage Area 2 (40 lbs max)		85.00	
TOTAL:			

3. Once all the data have been read, your program should produce the following output:

   a. Print the gross weight of the loaded aircraft and the position of the center of gravity, as before.

   b. Also print the number of stations from which these computations were obtained, for verification. You may print this either before or after the other information. Try to organize your output into a nice, easy to read summary report.

   c. If the weight or center of gravity are not within acceptable limits then print an appropriate warning message. Your warning message should help the user understand what might be wrong. In particular, you should print different warning messages depending on whether the CG is too far forward or too far aft.

   d. Your program must correctly handle the case of an aircraft with weight and center of gravity above the diagonal line at the left of the C.G. limits shown in the figure in the last exercise (this was optional then; now it is not).

4. It is useful to test your program with a smaller aircraft which is already familiar. Run your script using the data from Table 1 for the positions of each station, and for the empty weight of the aircraft, and using the following scenarios:

   a. Pilot+Passenger weight: 365 lbs / Fuel: 90 lbs / No baggage

   b. Pilot only weight: 191 lbs / No passenger / Fuel: 135 lbs
   Baggage area 1: 99 lbs / Baggage area 2: 39 lbs

   c. Pilot weight: 191 lbs / Fuel: 135 lbs
   Injured passenger prone in Baggage area 1: 173 lbs
   Baggage area 2: empty

You may find it useful to check your code by performing the same calculations with a hand calculator.

5. Submit to your instructor a copy of your script, and a copy of the output from all the test scenarios listed above (and perhaps others as well).

Optional Improvements:

You do not have to make these improvements to your program, but you can do so if you really want to make it work nicely and to make it easier to read:

- Print the station number each time before you read in the arm and weight for that station.
- Print the total weight (and perhaps also the CG location) after each set of data points has been entered. (But don't compute CG if the total weight is zero!)
- Insure that each line of code in your script, and each line of output it produces, is no longer than 80 characters. As you may notice, the windows IDLE opens for editing and for output are both 80 characters wide, and Thonny shows a line after the 80th column. This dates back to when physical computer terminal screens were 80 characters wide, which in turn dates back to when punch cards were 80 characters wide. It is possible to make the windows wider, but its easier to read the code and output if the reader does not have to change the window size. A great deal of software, not just IDLE, defaults to 80 characters, so you should try to stay within that limit.

Some design advice: get your basic program working first, then add the optional improvements, one by one. It is much easier to build from a working program and fix any small problems which occur as you make changes. In contrast, if you make many changes at once you will then have to try to figure out which of these introduced whatever problems that arise. It's always a good idea to change only one thing at a time and then test that it worked.