Nested Loops

In this exercise you will use "nested" loops to study how variations in altitude and airspeed affect the position of where our bombs land. You will also learn how to define a function in Python, and how to format numerical output.

The FAA only requires that new pilots be able to demonstrate that they can hold an altitude to ± 100 feet and an airspeed to ± 10 knots. Besides this, there may be errors of this magnitude in the aircraft instruments. We would like to see how variations of the initial airspeed and altitude of the projectile change the final position where it lands. One way to do this is to simply repeat the calculations with many different values for the initial position and velocity. This is sometimes referred to as performing a "parametric study" or "parametric analysis." It could also be called a "Variational Error Analysis."

In this assignment you will perform this kind of analysis for a falling object, and report your results in an organized table.

"Nested" loops

Loops are "nested" when one loop is entirely inside another. For example, suppose you wanted to count the total number of people in a school building. You could "loop" over classrooms, visiting each one in turn, and then for each classroom you could "loop" over people, pointing at and counting each person in succession. In this case the loop over people is "nested" inside the loop over classrooms.

The `break` and `continue` statements

Now that you know how to perform iteration using `for` and `while` loops, it may help to know that you can modify the execution of a loop using the `break` and `continue` statements:

`break`
: When the `break` command is invoked the loop ends, regardless of the stopping condition or the iteration count. Since the loop consists of those lines of code that are indented relative to the `for` or `while` statement that began the loop, execution will continue with the next line of code after (outside of) the indented block.

`continue`
: The `continue` statement ends only the current iteration of the loop, but not the loop. Regardless of how ever many lines of code remain in the block, the loop will begin the next iteration of the code from the top of the block.

Formatted Output with the % Operator

As you may have noticed, Python can print a list of numbers and strings separated by comas, but it does not necessarily line them up very well. To make a readable table of data you will want to have each of the values on each line appear in the same position, and with the same width. One way to do this is to first specify a format string to print, which contains placeholders (which begin with "%") to indicate where to insert data values. After this format string you use the same % character, but now as an operator, followed by a *list* of values to insert into the format string. Notice that this means that the "%" character is used here in two different ways (which can be confusing at first).

A simple example will make this clearer. Suppose you have a script which generates an inventory of items, and which includes the price per item and the total value of all of the items in stock. The following line of code would print one line of the report:

```
print("#%2d: %4d units at $%6.2f = $%7.2f"  %  (i, Nunit, P, Nunit*P) )
```

The character string containing the % markers is a template for the line. Each of the format placeholders beginning with % specifies the type of data it expects ('d' for integer "decimal digits", 'f' for "floating" point values) along with some size information. So, for example, "`%6.2f`" means a floating point number 6 characters wide (that includes one place for the decimal point) with 2 digits past the decimal point. After the template string comes the % *operator,* and after that a Python *list* of values to be inserted into the placeholders. In this example the variable `i` is the item number (presumably just a counter starting at 1 and increasing), `Nunit` is the number of units in stock, `P` is the price per unit, and the value of all of them together is the number of items times their individual price. Each of the 4 values in the list is inserted into successive placeholders when the template string is printed. The result looks like:

```
#47:   17 units at $  8.99 = $ 152.83
```

If you have any experience with the C programming language, or those based on C, it may help you to know that the "% codes" used to format output in Python are very similar to those used in C.

Defining Functions

One of the most powerful features of Python is that it is very easy to define functions, (and in Python functions can return not just single values, but lists). To define a function you use the `def` statement, giving the name of the function and a list of the inputs to the function (often called the "arguments"). The `def` statement ends in a colon (don't forget it!), and then the body of the function consists of all the lines which follow which are indented. When the indentation of code ends, that ends the definition of the function.

A simple example should make this clearer. Suppose we want a function which will create and return the Fibonacci sequence up to a given value. Just to remind you, the Fibonacci sequence is the set of numbers "1, 1, 2, 3, 5, 8, 13, ..." and the next number in the sequence is always the sum of the previous two numbers. Here is one way to do this:

```
def FibonacciSequence(maxN) :
    seq = [1, 1]
    while 1 :
        i = len(seq)-1
        next = seq[i] + seq[i-1]
        if next > maxN:  break
        seq.append(next)
    return seq
```

Here is how it works: The name of the function is `FibonacciSequence`, and it takes one argument, which can be referred to in the function by the name `maxN`. The first thing the function does when it is started is create a list called `seq` and puts in it the first two Fibonacci numbers. The `while` loop is an example of an infinite loop, since "1" is always "true." You have to be careful with this to be sure there is a stopping condition inside the loop (in this case there is), but this is a common construct in Python. The variable `i` takes the length of the list in `seq` so far and deducts 1 from it, so that it can be used as an index* into the list `seq`. The variable `next` then gets the next Fibonacci number, which is the sum of the previous two values in the list. The value is tested against `maxN`, and if it is greater than `maxN` the `break` statement terminates the `while` loop. Otherwise, this next number is appended to the list `seq`, and the loop repeats. Once the `break` statement ends the `while` loop, the last line of the function, the `return` statement, makes the final value of the list `seq` the result which is returned by the function.

If you type this into a file and run the script, there will be no output. This only defines the function, but it doesn't use it. You can test the function by adding a line (not indented!) to use the function. Here's how to get the Fibonacci sequence up to 100:

```
print(FibonacciSequence(100))
```

The result, as you can verify yourself, is:

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

One of the most useful things about functions is that you can then use them over and over again. It is now very easy to change the input to the function to find the Fibonacci sequence up to 200, or 400, or 1000.

Not all functions have a return value. Some simply do something and then end. In that case the return value, if tested, will be "`None`". In any case, you can save the results returned by a function in a variable for later use.

As you gain more experience writing Python scripts you will learn how to break your problems up into basic elements of computation or logic, to be able to do the most general work with the fewest functions. As you learn to do this, it can be useful to first write your script without a function, and then break out the useful pieces into separate and more general functions once you have the kinks worked out.

* Remember, in Python numerical indexing into a list starts at zero, not one.

Assignment

Your goal for this exercise is to create a script which computes the trajectory for the same projectile, but over and over again with many different starting conditions. Your script will produce a formatted table showing the time and position of impact. Your script must make use of a function to compute each trajectory.

To complete this exercise you must do the following:

1. **Inputs:** Your program should first read the reference altitude, in feet, and then the reference airspeed, in knots, both on a single line and separated by a comma. Remember to echo the values of all variables when they are read.
2. **Looping :** Your script should compute the drop time and impact position for exactly 9 equally-spaced altitudes between 100 feet above the given altitude down to 100 feet below the given altitude. For each of these 9 altitudes, the program should compute the impact for exactly 5 equally-spaced initial airspeeds, from 10 knots below the given airspeed to 10 knots above the given airspeed. The loop over airspeeds should therefore run "faster" than the loop over altitudes; that is, for each altitude you should repeat the calculation for all possible airspeeds, then move to the next altitude. The loop over altitude will start with the highest value and go down, while the loop over airspeed will start with the lowest value and go up.

   If the altitude read by your program is less than 100 feet then it should make the lowest altitude out of the 9 be zero, and still consider exactly 9 equally-spaced altitudes from 100 feet above the specified altitude down to zero.

   Take care: the loop over different airspeeds should use knots, but inside the loop you will want to convert each airspeed to feet per second.
3. **Computation:** For each initial altitude and airspeed your script will compute the complete trajectory until the falling object hits the ground. You script must do this using a function, called `dropit()`, which takes as input parameters ("arguments") the altitude in feet, and the initial forward speed in feet per second, in that order. The function must return the results as a list which contains the time, the horizontal position of impact, and the final vertical position, in that order.

   As before, you should use a time step of 0.1 seconds, and you can assume that the time of fall will be at most 30.0 seconds. If a drop takes longer than 30.0 seconds your program should not finish the drop. Since you will print the final altitude, and it will be far above zero if the drop does not finish, you should not do anything special to flag this case.

   Your function should not print anything during the drop (there would be way too much output). Instead, it should just return the final time and position.
4. **Output:** The text output of the script should be a formatted table showing the conditions of each drop and the results, created using the formatting strings described above. Each output line should begin with the initial altitude and

airspeed of the drop (in that order, with airspeed in knots), followed by the time until impact, followed by the impact position (x first, then y).

The columns of the table must all line up, and there should be proper table headings over each column. Each line of output should not be longer than 80 characters.

To test your code with representative input values for a Cessna 150, try an altitude of 200 feet, and an airspeed of 60 knots. To test the low altitude behavior you can try a Piper Cub flying at 50 knots at an altitude of 50 feet.

It is now a requirement that your code and the output it produces both have lines that are at most 80 characters long. This is because most computer terminal applications open windows that wide by default. That, in turn, is because in olden times physical computer terminals had displays that were 80 characters wide. And that, in turn, is because the computer punch cards that preceded terminals had 80 columns.

Debugging hint:

While your final program should not print intermediate values while the projectile is in free fall, you may find it useful while testing your program to use a `print` statement to print the whole trajectory of the object. Just be sure to remove or comment out such "debugging" statements from the final version of your program. They are the equivalent of the scaffolding put up to aid in the construction of a building and then removed when the building is completed.