Python #5
Looping to process many data items

Location: https://people.sc.fsu.edu/~jburkardt/classes/python_2022/python05/python05.pdf

Freely adapted from the Python lessons at https://software-carpentry.org/

Loops

- *A loop begins with the* `for` *statement;*
- *An iteration index can be defined, often* `i`, `j` *or* `k`*;*
- *The loop index values can be specified by a* `range()` *statement;*
- *The loop index values can be given explicitly by a list;*
- *The* `for` *statement is followed by indented statements to be repeated;*
- *A loop allows us to operate on each element of a vector (one-dimensional array);*
- *A pair of nested loops operate on each element of a matrix (two-dimensional array);*
- *A loop can be used to carry out an iteration;*

1 We will need loops to process our medical data

In a previous Python exercise, we wrote code to plot values of interest from a file containing medical data, `inflammation-01.csv`. In fact, the researcher has supplied us with 12 such sets of data, and there may be more on the way. To save work, we'd like to be able to process all the data with a single statement. In order to do that, we need to learn about Python loops, which allow us to repeat a given action on multiple sets of data.

2 A loop replaces a sequence of similar statements

The formula $t(n) = \frac{n(n+1}{2}$ evaluates the n-th triangular number, which counts the total number of dots in a triangular grid whose base has n dots. In other words,

$$t(n) = 1 + 2 + 3 + ... + n$$

Suppose we are asked to find the smallest value of n for which $1000 \leq t(n)$. How could we proceed? Unfortunately, at the moment, we really only know enough Python to simply try a list of statements, like this:

```
  print ( 1 * ( 1 + 1 ) / 2 )
  print ( 2 * ( 2 + 1 ) / 2 )
  print ( 3 * ( 3 + 1 ) / 2 )
  ...
```

Clearly, it would be much better to write the formula once, for a symbolic value n, and then somehow specify that we want to start with $n = 1$, evaluate the formula, then increase n by 1, and repeat the process...“for a while”. Although we probably have to search much higher, we could start cautiously, intending only to evaluate the formula up through $n = 10$. So we would like to express, in Python, the idea:

```
starting with n = 1, and continuing up to n = 10
  print the value of n * ( n + 1 ) / 2
```

This kind of repetitive operation is known as a *loop*, and Python uses the `for()` statement to describe how such a loop should be controlled. For our example, we could write

```
for n in range ( 1, 11 ):
  print ( '  t(', n, ') = ', n * ( n + 1 ) / 2 )
```

Notice several things about these lines of code:

- The `for` statement names a variable `n` to be used in the loop;
- The `range()` statement specifies the first and last values of `n`;
- The `range(a,b)` statement actually stops at `b-1`;
- The `for` statement concludes with `:`, a colon;
- The code to be repeated can use the value `n` as it changes;
- The code to be repeated is indented. Indenting two spaces is a good idea.

You may notice that the computed values are printed as real numbers. That's because, by default, Python carries out all numeric division using real arithmetic. In some other languages, the results of dividing an integer by an integer is always rounded to an integer value. Here, even though the results are guaranteed mathematically to be integers, they are treated as though they are real. If for some reason we want a division operation to be output as an integer (whether or not it requires rounding), we replace the single division slash by two slashes.

```
for n in range ( 1, 11 ):
  print ( '  t(', n, ') = ', n * ( n + 1 ) // 2 )
```

For this example, that is only a matter of printing out nicely. In other cases, we may compute $\frac{14}{3}$ and really need the result to be rounded to 4.

3 Indentation

In our example, the “body” of the loop only involves a single statement, but often we will have several statements that all form part of a repeated calculation. We could even rewrite our calculation as:

```
for n in range ( 1, 11 ):
  t = n * ( n + 1 ) // 2
  print ( '  t(', n, ') = ', t )
```

Notice that both statements must be indented, and both statements must be indented in the same way (here, using two spaces). If there are more calculations to be made once the loop is completed, these go back to being unindented.

As an example of using a loop as part of a more complicated program, let's suppose that not only do we want to compute the first 10 triangular numbers, but we want to determine their final sum, and print that

out. We could do this by starting a variable `s` at 0, and then each time we compute the next value of t, adding that value to `s`. Our first statement, which starts `s` at 0, is not part of the loop; nor is our final statement, which prints `s`. Look carefully at how indentation is used to indicate these facts:

```
s = 0
print ( '  S starts out at ', s )
for n in range ( 1, 11 ):
  t = n * ( n + 1 ) // 2
  s = s + t
  print ( '  t(', n, ') = ', t )
print ( '  Sum of triangular numbers is ', s )
```

4 The range() statement creates the loop index values

In our examples of a `for()` loop, we have used the function `range(a,b)` to generate the list of values used for the loop index. Note that this statement can be used all by itself, and simply returns a standard Python list of the values from a up to, but not including, b. It is assumed that $a < b$. However, if for some reason, we want to count *down*, that is, in a decreasing sequence, then we can specify a third argument, the *increment*, as -1. In fact, if we specify the increment, it can be any value. Our loop index can go up by two's, or down by three's, and so on, although only the increments -1 and +2 are commonly used.

If only a single argument `s` is given to the `range` function, it behaves as though the user had typed `range(0,s,1)`, that is, it will count up from 0 to s-1, in steps of 1.

If the second limit or the increment not chosen carefully, the range statement returns an empty list.

We have described `range()` as returning a list. Actually, it generates the values of the list one at a time. So if you want to save the values, or print them, you need to force range() to complete its work and save the values in a list, as follows:

```
  r0 = list ( range ( 10 ) )
  r1 = list ( range ( 0, 10 ) )
  r2 = list ( range ( 10, 10 ) )  # Oops
  r3 = list ( range ( 10, 0 ) )   # Oops
  r4 = list ( range ( 10, 0, -1 ) )
  r5 = list ( range ( 0, 10, 2 ) )
```

Of course, here we are using `list()` simply so that we can see what's going on. If we are using `range()` as part of a `for()` statement, we do not use the `list()` statement!

Why this peculiar behavior of `range()`? If your loop goes from 0 to one million, then `range()` only generates one index at a time. If it instead first created a list of one million loop indices, this would eat up a good chunk of your computer memory.

5 Using an explicit list instead of the range() statement

Now there are times when the `range()` statement isn't flexible enough to provide the index values for a `for()` loop. Obviously, the values generated by `range()` must be a sequence of equally spaced numbers.

But suppose we wanted our loop to consider some sequence of numbers that were not equally spaced? It turns out that, if we can provide a list of these numbers, we can use that in place of the `range()` function:

```
UScoins = [ 1, 5, 10, 25, 50, 100 ]
sum = 0
for coin in UScoins:
  sum = sum + coin
print ( '  The sum of a collection of US coins is ', sum )
```

Also, instead of numeric data, we might want to cycle through a sequence of string values:

```
friends = [ 'Alice', 'Bob', 'Carol', 'David' ]
for friend in friends:
  print ( '  My friend ', friend, ' has a name of length ', len ( friend ) )
```

Now if we are willing to use a list to run our loop, suppose we use a *nested* list? That is, each time we ask for an entry of the list, we are getting a row of a table? Recall how we set up a table of patients in the previous exercise. Each row of the table contained the name, weight, height and vaccination status of a patient. We can run a loop which examines, one at a time, the data for each patient. If we give a name for each item, then we can use these items inside the loop.

```
  patient0 = [ 'Robert Baratheon', 235.4, 73, False ]
  patient1 = [ 'Arya Stark', 134.7, 68, True ]
  patients = [ patient0, patient1 ]

  for name, weight, height, vax in patients:
    print ( name, ' weighs ', weight, 'lbs and is ', height, 'inches tall.' )
```

This means that if we set up a nested list that started

```
  months = [
    ['January', 31],
    ['February', 28],
    ['March', 31],
    ...
    ]
```

then we *almost* have enough information to print a calendar! (See how far you can get with this idea!)

6 Nested loops

Especially in numerical calculations, we often deal with matrices, typically symbolized by a capital letter such as A, comprising m rows and n columns, with the typical element represented by $A_{i,j}$. Generally, as part of a calculation, we will need to "examine" every entry of such an array. The natural way to do this requires us to create a pair of `for()` loops, which are nested. Here is a simple example in which we will compute the average value of the matrix elements:

```
m = 4
n = 3
A = np.random.rand ( m, n )
average = 0.0
for i in range ( 0, m ):
  for j in range ( 0, n ):
    average = average + A[i,j]
average = average / m / n
print ( '  average matrix entry is ', average )
```

Notice how the indentation is used to show that the `j` loop is nested inside the `i` loop.

Now that we have examined the various features of `for` loops, we will next be able to return to our medical data study, and process the entire set of files that the researcher has provided.

7 `while` loops

To define a `for` loop, we need to know in advance the range of values we wish to explore. But there are some cases where it may be inconvenient or impossible to do this, and yet our computation really involves some

kind of repetitive action. Sometimes, the right way to handle such situations is to use a `while` loop. The body of the loop might have the same format as we used in a `for` loop, but the loop begins with a statement of the form

```
while ( condition ):
  ...body of loop...
```

The idea is that before we are allowed to begin the loop, we must check that the given condition is true (something like "You must be THIS tall to get on the rollercoaster""). If the condition is false, we simply skip the entire loop and pick up execution afterwards. But if the condition is true we

- execute the statements in the loop body;
- go back to the top of the loop and check the condition again;
- depending on the current value of condition, we repeat the loop, or skip ahead.

It should be obvious that the execution of the loop must, at least potentially, change the value of the condition from True to False, otherwise we will loop forever. Before this description sounds too vague, let's look at a simple example of a loop for which we would prefer to use a `while` statement.
We return to the sequence of triangular numbers $t_n = \frac{n(n+1)}{2}$. We are interested in finding out the smallest value of n for which $1000 \leq t(n)$. Obviously, we could create a `for` loop with a fixed limit, and keep increasing that limit until we see that we have passed 1000. But a better approach would be something like this:

```
n = 1
while ( t(n) < 1000 )
  increase n
```

To do this in Python, we can write:

```
n = 1
while ( n * ( n + 1 ) // 2 < 1000 ):
  n = n + 1
print ( ' 1000 <= t(', n,') = ', n * ( n + 1 ) // 2 )
```