Looping with for() and while() statements

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/python_for/python_for.pdf

---

Loops

- *A* `for()` *statement begins a loop;*
- *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 one-dimensional list or array;*
- *A pair of nested loops operate on each element of a two-dimensional list or array;*
- *A* `while()` *statement repeats a loop as long as a condition is satisfied;*
- *A* `continue()` *statement can finish the current step of a loop;*
- *A* `break()` *statement can exit a loop;*

1 Looping handles a sequence of related tasks

We have talked about several kinds of computations in which we needed to repeat some operation several times.

For instance, we investigated the iteration in which we replace x by $cos(x)$. Starting at 1, we had to issue the same command over and over:

```
x = 1; print ( x )
x = cos ( x ); print ( x )
x = cos ( x ); print ( x )
...
```

Luckily, it only took about 10 such repetitions before it seemed that the values of x were converging.

We have talked about the problem of determining whether an integer n is prime. To check this, we have to do a similar sequence of steps, although now the command changes slightly each time:

```
is_prime = True
if ( n % 2 ):
  is_prime = False
if ( n % 3 ):
```

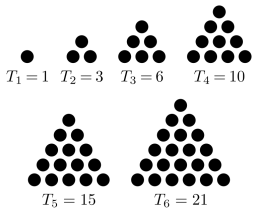
```
  is_prime = False
if ( n % 4 ):
  is_prime = False
...
```

If we are typing these commands interactively, then if we find a divisor early, we know we don't have to run any more commands. But if we were checking $n = 97$, then we would presumably have to write 95 pairs of statements, and execute them all. We really want to be able to carry out a sequence of commands like this, with the option to stop early if something "interesting" happens.

As a new example of this kind of issue, we will consider a problem involving *triangular numbers*.

2 Looping with a `for()` statement

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. For example, there are 10 pins in a standard game of bowling because $T_4 = 10$, and 15 balls in pool because $T_5 = 15$. And you may recall that Gauss, as a elementary school student, was punished by being told to add the numbers from 1 to 100, can came up with the triangular number formula, quicking getting the result $T_{100} = 5050$.



Instead of using the formula for T_n, we could instead simply write it directly as a sum:

$$T_n = 1 + 2 + 3 + ... + n$$

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, 10+1 ):
  print ( n, n * ( n + 1 ) // 2 )  # Why can we use // here?
```

Notice several things about these lines of code:

- This `for()` statement names a variable `n` to be used as the loop index;
- The `range()` statement specifies the first and last values of `n`;
- The `range(a,b)` statement actually stops at `b-1`;
- I wrote `range(1,10+1)` to remind you that the loop runs up to 10 (but not 11);
- 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.

3 Looping with a `while()` statement

As you can see, T_{10} is not as big as 1000, so we have to keep going. We could keep increasing the upper limit until we notice that we have reached our goal. However, our goal wasn't really to run the loop a certain number of times. It was to keep computing a value until that value reached a limit.
Here's a different way to think about our triangular number problem.

```
Let n = 0
Let value = 0
as long as value is less than 1000
  increase n by 1
  increase value by n   (or set value to n*(n+1)//2)
print n and value
```

There are some important differences in this idea. The "as long as" statement tells us to repeat the loop until what we want happens. That seems to guarantee that our loop won't stop too soon, or go on too long, because we can't predict in advance the number of iterations we will need. Secondly, we are suggesting, using indentation, that the print statement only happens once the loop has been completed. So if everything works out, we only see one line of information, rather than the list we saw before.

How do we accomplish this? We use the Python `while()` statement, and specify the condition that we want to see before we will let the loop stop. The statement has the form

```
while ( condition ) :
  one or more indented statements to be repeated
```

Our triangular number problem now becomes:

```
n = 0
value = 0
while ( value < 1000 ) :
  n = n + 1
  value = value + n
print ( n, value )
```

4 How the range() statement generates `for()` loop indexes

Let's go back to our `for()` loop example, and look more closely at how we specified that we wanted n to take on the values 1, 2, 3, ..., 10. We did this using the function `range(a,b)`, in our case `range(1,11)`.

Note that this function can be used all by itself. You might think it returns a list of the numbers. Actually, it's a little more complicated. Strictly speaking, it returns a *generator*, that is, a Python object that will successively spit out the values we want one at a time. If we really want to see those values all at once, we can do so, by asking for a list, literally:

```
  print ( range ( 1, 11 ) )      #  Not what we want!
  values = range ( 1, 11 )
  print ( values )
```

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 is 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 ) )  # What is the last value?
  r5 = list ( range ( 0, 10, 2 ) )   # What is the last value?
```

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 ) )
```

6 Finishing one task early

A `for()` loop usually creates a sequence of tasks for us to carry out. Sometimes, such a task involves several instructions, to be carried out one after another. But we may find at some point that we are already finished with this task, don't need to carry out the later instructions, and can start immediately on the next loop. If we discover we're done early, we can move on using the `continue` statement.

For the *jeopardy* example, suppose the file might contain comment lines, that is, lines of text beginning with a hash mark. Such lines don't contain information, so we don't want to process them. We could deal with this complication as follows:

```
input = open ( filename, 'r' )
for line in input:
  if ( line[0] == '#' ):
    continue
  words = line.split()
  name.append ( words[1] )
  n = n + 1
input.close ( )
```

7 Finishing all tasks early

Sometimes, while executing one cycle of a loop, we may realize that we are completely done, that is, there is no point in completing the current task, and there is no point in doing any further tasks of the loop. This is often a good thing, indicating that you have found something you were looking for, or completed all your work.

If we are asking whether integer n is prime, we presumably have to check all divisors between 2 and $n-1$. However, if we encounter a divisor at any point, then we have our answer early, and can completely stop the iteration. The `break()` statement can be used this way:

```
is_prime = True  # We start with this assumption
for d in range ( 2, n )
  if ( n % d == 0 ):
    is_prime = False
    break
```

This idea will be helpful in speeding up our prime number searches later!

8 Calendar for 2023

With what we know (or almost know!) so far, we could actually try to print a sort of calendar for 2023, really just a list of days in order, like

```
Sun Jan 1
Mon Jan 2
Tue Jan 3
Wed Jan 4
Thu Jan 5
Fri Jan 6
Sat Jan 7
Sun Jan 8
Mon Jan 9
Tue Jan 10
```

To do this, we have to think about how the calendar works. We need some important numbers:

- d = the day of the year, from 0 to 364;
- wd = the day of the week, from 0 to 6;
- m = the month, from 0 to 11;
- md = the day of the month, from 1 to 31 (or less);

Each line of the calendar is created by increasing d by 1. The value of w must change too, but it's just `w = ( d % 7)`. The month m changes, but only if md reaches the last day of the current month.

So we need three lists

- names of the days of the week
- names of the months
- lengths of the months.

Here's a start on your 2023 calendar program:

```
weekday_names = [ 'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat' ]
mon_names = [ 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', '
    Dec' ]
mon_length = [ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ]

d = 0
wd = 0
m = 0
md = 1

while ( d < 365 ):

  print ( weekday_names[wd], mon_names[m], md )
  update md
  update wd
  update d
  if we reached end of this month, what do we do?
```

I realize that we don't really know much about strings and lists and list indexing yet. But if you are adventurous, you might enjoy figuring out how to make this work.

9 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.

10 Check a magic matrix

If the integer n is odd, it's actually pretty easy to create a "magic" matrix, that is, an $n \times n$ matrix for which every row and column has the same sum. I have posted a Python code called *magic_matrix.py* on Canvas, which lets you create a magic matrix of any odd order. For instance,

```
A = magic_matrix ( 3 )
print ( A )
```

will give us

```
8 1 6
3 5 7
4 9 2
```

For a given value of n, the "magic" sum will be $s = \frac{n(n^2+1)}{2}$ which in this case is 15. Using `for()` loops, how can we verify that there is a sum of 15 for

- every row?
- every column?
- every diagonal?

What's the magic sum for $n = 5$? Does our test still work?