tutorial009-ControlFlowOne-Loops

October 7, 2022

1 Basic control flow in Python I

1.0.1 Learning outcomes:

- Understand and use control flow in python
- `for` loops
- `While` Loops

Let's make a thought experiment as a start. Imagine giving yourself, a slightly different different, a robot-like person resembling yourself, instructions to navigate outdoor, away from this room, into the open air and sunshine.

You might give yourself instructions to do so, and say things like the following:

- Stand up from desk
- Walk to the door
- *If* the door is closed *then* open the door
- *If* the length of the corridor to the right is shorter than that to the left *then* take the corridor to the right, *else* take corridor to the left
- Continue walking *for* the duration of the corridor
- *While* walking avoid people
- When the end of the corridor is reached open door to the outisde
- Etc.

Instructions like the ones above are often needed when programming a complex task, just like navigating the college campus.

1.0.2 Control flow

Control flow comprises of the set of commands available in a programming language to control the flow of information processing, just like the instructions above control your navigation.

Control flow comprises the set of operations that make sure the set of code operations are executed by a program. A control flow statement allows the program to make a choice among two or more alternatives.

A set of control flow operations are generally organized into blocks with a beginning and an end. The end and beginning of a control flow block is codified by dedicated words, or syntax.

Control flow is core to all computer programming, not just Python. In this tutorial we are going to get started with control flow and learn about some of the core elements in Python. More specifically we will look at

- for and while loops
- conditional tests and Boolean logic
- control flow
- functions

We will explore these things using fairly simple examples (that will also give us practice with indexing, operators, Python lists, etc). Later, we will see how useful these core elements are when they are combined!

1.0.3 Loops

Things in life can be repetitive.

Often, we need to repeat an entire, long, process over when only small changes are needed. For example, most of us follow the same exact routine every morning (shower/brush teeth/shave/make up/whatever) even though the only thing that has changed is one little number on a calendar. The same is true for computational tasks; a teacher might need to go through the exact same steps to compute a grade for each student, or a data scientist might need to go through the exact same steps to create a plot for several different but identically structured data sets.

Such repetitive tasks are very boring for humans (and bored humans tend to make mistakes!). While computers can't brush our teeth yet (still waiting for those tarter-eating nanobots!), they can help with reapeating calculations over and over using ***loops***.

There are two kinds of loops. There are

- `for` loops, which run a calculation *for* a pre-determined number of times
- `while` loops, which run a calculation *while* some critereon is met

Let's look at these in turn.

1.0.4 Control flow: 'For' loops

The `for` loop will be your workhorse for a lot of tasks.

Let's run this very simple `for` loop, and then we'll look at it and dissect it.

The first line, `myNewList = [1, 2, 3, 4, 5]`, creates a Python ***list*** of numbers. A list in Python is a kind of ***iterable***, which is a Python object that will automatically spit out its values one-at-a-time if it's put in a `for` loop.

The next line, `for i in myNewList:`, sets up the for loop. It says that:

- each value in myNewList (the iterable) will be assigned to the variable `i` in turn
- every ***indented*** line under this line is executed with each value of `i` in turn

The third line self-explanitory; we are just printing the values of `i` to confirm that `i` is, in fact, getting assigned each value of `myNewList` in turn.

(The use of `i` here is by convention only. You can use anything, like `Phredrick` even, as the name of your looping variable. But, just like having numpy nicknamed np, the use of `i` will generally make your code more readable to others, including future you!)

Also, note the indentation in the lines following the `for` loop line is key. As described above that indentention is the syntax that defines the block of conde that is part of the

Python was designed from the ground up to be a very human readable programming language. Appropriate indentation helps make code pretty and readable. As such, Python enforces its use in certain circumstances, like inside a `for` loop. The indentation tells Python "Yep, this line is inside the `for` loop." and the end of indentation tells Python (and you) "Okay, now we're back outside the `for` loop."

Four spaces, that is the number of spaces Python expects in a `for` loop block of code.

In most other programming languages, we are encouraged to indent our code to make it pretty. In Python, indentation is actually a part of the language!

Let's take a look the anatomy of a `for` loop:

The code block starts with the word `for`, the a variable, often `i`, `k`, or `j` (not to be cofunded with `j` from the complex numbers). This variable is be used as an index (hence the `i`), the holder of the current element's index in a list of elements. After that word `in` is used to indicate that the next variable contains the list of elements to iterate over. Finaly, the sentence is closed with `:`

`for i in list_of_values :`

After, that first line a block of operations follow, operations that are iterated over for each value in the `list_of_values`. This block of operations needs to be indented by 4-spaces, as per Python Syntax.

Let's look at a fully executable example. Let's experiment with this by computing the square root of some numbers. This `for` loop should run as expected. Let's make a Python `list` with `int` values in it:

[1]:
```
input_list = [1, 2, 3, 4, 5]
```

Let's then run the `for` loop, print out the current squre root and print out when the `for` loop has ended:

[2]:
```
for i in input_list :
    root = i**0.5  # remember, the double splat, "**", means "raise to the␣
 ↪power of"
    print('The square root of ', i, ' is ', root)
print('Now the loop is over.')
```

```
The square root of  1  is  1.0
The square root of  2  is  1.4142135623730951
The square root of  3  is  1.7320508075688772
The square root of  4  is  2.0
The square root of  5  is  2.23606797749979
Now the loop is over.
```

A good test for how important is indentention is to try the same code without indentation:

[3]:
```
for i in input_list :
root = i**0.5  # remember, the double splat, "**", means "raise to the power of"
print('The square root of ', i, ' is ', root)
print('Now the loop is over.')
```

```
  Input In [3]
    root = i**0.5  # remember, the double splat, "**", means "raise to the powe␣
↪of"
    ^
IndentationError: expected an indented block
```

Oops! Python barfs because the indentation – an integral part of the code – is wrong.

Even if we try to make our intent clear with blank lines, the indentation determines what is in the loop or not. Below, it looks like we want the second print to be outside the loop…

[4]:
```
aList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in aList :
    root = i**0.5
    print('The square root of ', i, ' is ', root)

    print('Now the loop is over.')
```

```
The square root of  0  is  0.0
Now the loop is over.
The square root of  1  is  1.0
Now the loop is over.
The square root of  2  is  1.4142135623730951
Now the loop is over.
The square root of  3  is  1.7320508075688772
Now the loop is over.
The square root of  4  is  2.0
Now the loop is over.
The square root of  5  is  2.23606797749979
Now the loop is over.
The square root of  6  is  2.449489742783178
Now the loop is over.
The square root of  7  is  2.6457513110645907
Now the loop is over.
The square root of  8  is  2.8284271247461903
Now the loop is over.
The square root of  9  is  3.0
Now the loop is over.
```

… but it's not – the indentation makes it part of the `for` loop. Above, we have just wrongly added `print('Now the loop is over.')` to the `for` loop code block, by simply increasing the indentation.

And because indentation is SO important, we can't indent willy-nilly just because we feel like it:

[5]:
```
aList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in aList :
    root = i**0.5
print('The square root of ', i, ' is ', root)

    print('Now the loop is over.')
```

```
  Input In [5]
    print('Now the loop is over.')
    ^
IndentationError: unexpected indent
```

Okay, let's go back to our working version. It's nice and pretty and it's clear what's in the loop and what's not.

[6]:
```
aList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in aList :
    root = i**0.5  # the double splat, "**", is "raise to the power of"
    print('The square root of ', i, ' is ', root)

print('Now the loop is over.')
```

```
The square root of  0  is  0.0
The square root of  1  is  1.0
The square root of  2  is  1.4142135623730951
The square root of  3  is  1.7320508075688772
The square root of  4  is  2.0
The square root of  5  is  2.23606797749979
The square root of  6  is  2.449489742783178
The square root of  7  is  2.6457513110645907
The square root of  8  is  2.8284271247461903
The square root of  9  is  3.0
Now the loop is over.
```

Complete the following exercise.

- Write a `for` loop to compute the square of the first few even numbers.

[Use the cell below to show your code]

[8]:
```
myEvens = [2, 4, 6, 8, 10]
for i in myEvens :
    root = i**0.5
    print('the square root of', i,'is', root)
```

```
the square root o 2 is 1.4142135623730951
the square root o 4 is 2.0
the square root o 6 is 2.449489742783178
the square root o 8 is 2.8284271247461903
the square root o 10 is 3.1622776601683795
```

Note that when you hit return after typing the `for ... :` line, Python indented the next line automatically for you. How nice! But sometimes you'll want to go back and edit a `for` loop, or add lines to one, etc. So...

Important! When you have to indent code manually, use ***4 spaces*** to indent! Do not use a tab, do not use 3 spaces, do not use 5 spaces, ***use 4 spaces***. This is one thing that Python can be really mean about.

When you become a master coder, you can experiment with this. But don't come running back to me crying when the Python gods smite you and leave you all alone out in the cold having to pick up the pieces of the tattered shambles of your former life.

1.0.5 Ranges of data values

Above we created lists by typing values inside squre brackets. Python has a handy dandy function to create values for `for` loops, called a `range()` that works really well with `for` loops. A `range()` spits out a sequence of numbers perfectly suited to feeding a hungry `for` loop. By default, the range starts at zero and increments by one. Like this:

```
[9]: for i in range(5) :
         print(i)
```

```
0
1
2
3
4
```

But you can change this by providing a start and a stop, or even a start, stop and step.

```
[ ]: for i in range(2, 11) :
         print(i)
```

```
[11]: for i in range(2, 9, 2) :
          print(i)
      print('Who do we appreciate?')
```

```
2
4
6
8
Who do we appreciate?
```

Note that a range does not make a list!

```
[12]: a = range(5)
      a
```

```
[12]: range(0, 5)
```

[13]:
```
type(a)
```

[13]:
```
range
```

Rather, a `range()` can be thought a little machine that spits out numbers for a `for` loop!

Complete the following exercise.

- Write a `for` loop to compute the cube of the first 10 odd numbers starting from 11, use `range()` to create the number. Print each value and a message when the loop is finished.

[Use the cell below to show your code]

[21]:
```
for i in range(11, 31, 2) :
    cubed =  i**1/3
    print(cubed)

print('now the loop is over')
```

```
3.6666666666666665
4.333333333333333
5.0
5.666666666666667
6.333333333333333
7.0
7.666666666666667
8.333333333333334
9.0
9.666666666666666
now the loop is over
```

1.0.6 Control flow: 'while' loops

Above, we have talked about `for` loops. For loops are excellent if we have a good idea of when to stop.

Sometimes, though we do not know the exact number of items we are iterating over, or alternatively, we want to iterate until a certain condition is met (for instead of iterating until the end of a list of numbers we want to iterate while a number is less than another).

In cases when the end condition for a loop is cannto be easily established before entering the loop, the `while` loop is used.

If, for example, we are computing the cumulative sum of the numbers between 10 and 20, we can use a `for` loop:

[22]:
```
aList = range(10,20);
print(aList)
```

```
range(10, 20)
```

[23]:
```python
j = 0
for i in aList :
    s = i + aList[j]
    print('The current sum at element ', i, ' is ', s, '(j is',j,')')
    j = j+1
```

```
The current sum at element  10  is  20 (j is 0 )
The current sum at element  11  is  22 (j is 1 )
The current sum at element  12  is  24 (j is 2 )
The current sum at element  13  is  26 (j is 3 )
The current sum at element  14  is  28 (j is 4 )
The current sum at element  15  is  30 (j is 5 )
The current sum at element  16  is  32 (j is 6 )
The current sum at element  17  is  34 (j is 7 )
The current sum at element  18  is  36 (j is 8 )
The current sum at element  19  is  38 (j is 9 )
```

In many situations the length of a loop might be unkown before entering the loop. Instead, it might be known that the loop will need to be ended, if a specific condition is met.

For example, if we were not in the position to know how many people live in the city of Austin, TX, but we wanted to ask each one of them if they have eaten any cake today, we could could use a while loop. We would continue asking while meeting people that we hve not met before.

In reality, often times it is possible to rewrite a `while` loop as a `for` loop. But there are useful differences between the two loops and for that reason they have both survived.

To show how a `while` loop would work, imagine if I wanted to know if I have eaten more than 14 slices of pizza in the past two weeks (why pizza now? Because we can, pizza is good, too much cake already anyways). Let's assume that I eat 1 slice of pizza per day:

[26]:
```python
day = 1
pizza = 0
while day < 14 :
    pizza = pizza + 1
    print('today is day #,', day, 'so far I ate', pizza, 'pizzas')
    day += 1
```

```
today is day #, 1 so far I ate 1 pizzas
today is day #, 2 so far I ate 2 pizzas
today is day #, 3 so far I ate 3 pizzas
today is day #, 4 so far I ate 4 pizzas
today is day #, 5 so far I ate 5 pizzas
today is day #, 6 so far I ate 6 pizzas
today is day #, 7 so far I ate 7 pizzas
today is day #, 8 so far I ate 8 pizzas
today is day #, 9 so far I ate 9 pizzas
today is day #, 10 so far I ate 10 pizzas
today is day #, 11 so far I ate 11 pizzas
today is day #, 12 so far I ate 12 pizzas
today is day #, 13 so far I ate 13 pizzas
```

The syntax of a `while` loop is similar but different than a `for` loop. A `while` loop requires

A) A condition that will need to be met `while condition`

B) A `:` at the end of the `while` line

C) Operations to be performed while the condition is not met, yet.

That is it.

In our case above, had used a counter and the operation was set to be for the counter to be less than a value. We had something like the following:

```
counter = 1
while counter < max_value :
  operations
  counter += 1
```

To reinstate then, `while` loops are used when the number of times it will be necessary to iterate is unkown, but the condition which determines the end of the loop is known. The `for` loop instead is used when the number of times the loop will have to iterate over is known, independently of whether the condition to exit the iteration is known or not.

Complete the following exercise.

- Write a `while` loop to sum the first 13 numbers starting at 1. Print each value and a message when the loop is finished.

[Use the cell below to show your code]

[39]:
```
counter = 1
cat = 0
while counter < 14 :
    cat = cat + counter
    print(counter, cat)
    counter += 1
print('now the loop is over')
```

```
1 1
2 3
3 6
4 10
5 15
6 21
7 28
8 36
9 45
10 55
11 66
12 78
```

```
13 91
now the loop is over
```

`while` loops also provide a convenient way to specify conditions explicitly. For example, we can use the word `break` to literally break out of a `while` loop, if a condition is met. For example, we can exit (or `break` out of) a `while` loop if a value reaches a certain threshold:

[40]:
```python
# here we break the loop (using the break command)
# if the counter is equal to a certain exit value
counter = 1
exit_val = 6
while counter < 1000:
  print('The counter is',counter,'The exit value is', exit_val)
  if counter == exit_val :
    print('The condition was met',counter,'==', exit_val, ', we exit the loop...
↪')
    break
  counter += 1
```

```
The counter is 1 The exit value is 6
The counter is 2 The exit value is 6
The counter is 3 The exit value is 6
The counter is 4 The exit value is 6
The counter is 5 The exit value is 6
The counter is 6 The exit value is 6
The condition was met 6 == 6 , we exit the loop…
```

The code above, is similar to the one previously used one, yet, the `while` loop is terminated when a condition is met.

The condition is that the counter becomes less than an `exit_val` variable (6). the condition is tested using the `if` syntax (`if` is an important word in python for flow control that we will explore more later).

Complete the following exercise.

- Write a `while` loop to multiply the first 10 numbers starting at 0, but exit the look if the output of the multiplication is more than 50 . Print each value and a message when breaking out of the loop.

[Use the cell below to show your code]

[8]:
```python
counter = 1
exit_val = 50
num = 1
while counter <10 :
    num = num * counter
    print(num)
    if num > exit_val :
        print('the condition was met when', num,'>', exit_val, ', loop is over..
↪.')
```

```
        break
    counter += 1
```

```
1
2
6
24
120
the condition was met when 120 > 50 , loop is over…
```

In some cases, we can set a condition to be met but instead of breaking out of the while loop we can decide to continue. For example, below we use the word `continue` instead of the word `break`, in this

[9]:
```
# here we continue the loop (using the continue command)
# if the counter is equal to a certain value
counter = 0
val = 6
while counter < 10 :
   counter += 1
   if counter == val :
    print('The condition was met',counter,'==', val, ', but we continue...')
    continue
   print('The counter is',counter,'The value is', val)
```

```
The counter is 1 The value is 6
The counter is 2 The value is 6
The counter is 3 The value is 6
The counter is 4 The value is 6
The counter is 5 The value is 6
The condition was met 6 == 6 , but we continue…
The counter is 7 The value is 6
The counter is 8 The value is 6
The counter is 9 The value is 6
The counter is 10 The value is 6
```

In reality, we could have used the `brake` command also in combination with a `for` loop. So, the situations where a while and for loop diverge are limited. Yet, one or the other are often times preferable, especially for clarity of programming style (e.g., your colleagues reading your code will be happier, if you write simple, easy to read code).

Below one last example with the `while` loop. The loop offers the ability to end with an `else` just like an `if` statement.

[10]:
```
counter = 0
while counter < 10 :
    print("We write INSIDE becuase we are inside the while loop.")
    counter = counter + 1
else :
    print("We write END because we have exited the while loop and are inside␣
↪the else statement")
```

```
We write INSIDE becuase we are inside the while loop.
We write INSIDE becuase we are inside the while loop.
We write INSIDE becuase we are inside the while loop.
We write INSIDE becuase we are inside the while loop.
We write INSIDE becuase we are inside the while loop.
We write INSIDE becuase we are inside the while loop.
We write INSIDE becuase we are inside the while loop.
We write INSIDE becuase we are inside the while loop.
We write INSIDE becuase we are inside the while loop.
We write INSIDE becuase we are inside the while loop.
We write END because we have exited the while loop and are inside the else
statement
```

Complete the following exercise.

- Write the same code twice and compare the complexity of the code

  ```
  LIST = [101, 102, 103, 104, 105, 106, 107, 108, 109, 110]
  A = 0
  for B in LIST :
      OUT = B + LIST[A]
      print('The current sum at element ', B, ' is ', OUT, '(A is',A,')')
      A = A+1
  ```

Complete the code above and rewrite it as a `while` loop.

[Use the cell below to show your code.]

[14]:
```
alist = [101, 102, 103, 104, 105, 106, 107, 108, 109, 110]
A = 0
for B in alist :
    OUT = B + alist[A]
    print('The current sum at element ', B, ' is ', OUT, '(A is',A,')')
    A = A+1
```

```
The current sum at element  101  is  202 (A is 0 )
The current sum at element  102  is  204 (A is 1 )
The current sum at element  103  is  206 (A is 2 )
The current sum at element  104  is  208 (A is 3 )
The current sum at element  105  is  210 (A is 4 )
The current sum at element  106  is  212 (A is 5 )
The current sum at element  107  is  214 (A is 6 )
The current sum at element  108  is  216 (A is 7 )
The current sum at element  109  is  218 (A is 8 )
The current sum at element  110  is  220 (A is 9 )
```

[24]:
```
A = 0
B = 101
out = B*2
while A < 10 :
    print('the current sum at element', B,'is ', out,'(A is',A,')')
    A = A + 1
    B = B + 1
    out = B*2
```

```
the current sum at element 101 is  202 (A is 0 )
the current sum at element 102 is  204 (A is 1 )
the current sum at element 103 is  206 (A is 2 )
the current sum at element 104 is  208 (A is 3 )
the current sum at element 105 is  210 (A is 4 )
the current sum at element 106 is  212 (A is 5 )
the current sum at element 107 is  214 (A is 6 )
the current sum at element 108 is  216 (A is 7 )
the current sum at element 109 is  218 (A is 8 )
the current sum at element 110 is  220 (A is 9 )
```

[Use this cell to describe in your own words which code seems to be more complex and why]

[25]:
```
['The while loop seemed a lot more simple to me, and I did not need to write␣
↪out a whole list of values. It makes more sense than making a generic list␣
↪with no meaning to store elements.']
```

[25]:
```
['The while loop seemed a lot more simple to me, and I did not need to write out
a whole list of values. It makes more sense than making a generic list with no
meaning to store elements.']
```