

# Basic control flow in Python I

## 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 outside
- Etc.

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

## 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!

## 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.

## 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.

```
In [1]: myNewList = [1, 2, 3, 4, 5]
        for i in myNewList :
            print(i)
```

```
1
2
3
4
5
```

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-explanatory; 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 indentation is the syntax that defines the block of code 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`, then a variable, often `i`, `k`, or `j` (not to be confused with `j` from the complex numbers). This variable is 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. Finally, 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:

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

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

```
In [3]: for i in input_list :  
        root = i**0.5 # remember, the double splat, "**", means "raise to  
        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 indentation is to try the same code without indentation:

```
In [4]: for i in input_list :
        root = i**0.5 # remember, the double splat, "**", means "raise to the
        print('The square root of ', i, ' is ', root)
        print('Now the loop is over.')
```

```
Input In [4]
    root = i**0.5 # remember, the double splat, "**", means "raise t
o the power 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...

```
In [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.')
```

```
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:

```
In [6]: 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 [6]
    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.

```
In [7]: 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]

```
In [181]: myEvens = [2, 4, 6, 8, 10]
for i in myEvens :
    root = i**2
    print('The square of ', i, ' is ', root, '(' ,i, '*', i, ')')

print("The End!")
```

```
The square of  2  is  4 ( 2 * 2 )
The square of  4  is 16 ( 4 * 4 )
The square of  6  is 36 ( 6 * 6 )
The square of  8  is 64 ( 8 * 8 )
The square of 10  is 100 ( 10 * 10 )
The End!
```

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.

## Ranges of data values

Above we created lists by typing values inside square 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:

```
In [12]: 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.

```
In [10]: for i in range(2, 11) :  
         print(i)
```

```
2  
3  
4  
5  
6  
7  
8  
9  
10
```

```
In [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!

```
In [19]: a = range(5)  
a
```

```
Out[19]: range(0, 5)
```

```
In [20]: type(a)
```

```
Out[20]: 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]



```
In [182]: myodds = range(11,30,2)

for i in myodds:
    root = i**3
    print('The cube of ', i, ' is ', root, '(' ,i,"*",i,"*",i,')')

print("The End!")
```

```
The cube of 11 is 1331 ( 11 * 11 * 11 )
The cube of 13 is 2197 ( 13 * 13 * 13 )
The cube of 15 is 3375 ( 15 * 15 * 15 )
The cube of 17 is 4913 ( 17 * 17 * 17 )
The cube of 19 is 6859 ( 19 * 19 * 19 )
The cube of 21 is 9261 ( 21 * 21 * 21 )
The cube of 23 is 12167 ( 23 * 23 * 23 )
The cube of 25 is 15625 ( 25 * 25 * 25 )
The cube of 27 is 19683 ( 27 * 27 * 27 )
The cube of 29 is 24389 ( 29 * 29 * 29 )
The End!
```

## 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 cannot 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:

```
In [69]: aList = range(10,20);  
print(aList)
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

```
In [29]: 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 unknown 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 use a while loop. We would continue asking while meeting people that we have 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:

```
In [30]: 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 restate then, `while` loops are used when the number of times it will be necessary to iterate is unknown, 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]

```
In [177]: number = 0
count = 0
while count < 13 :
    number = number + count
    print('Number up to:', count, 'Sum Value =', number)
    count += 1
```

```
Number up to: 0 Sum Value = 0
Number up to: 1 Sum Value = 1
Number up to: 2 Sum Value = 3
Number up to: 3 Sum Value = 6
Number up to: 4 Sum Value = 10
Number up to: 5 Sum Value = 15
Number up to: 6 Sum Value = 21
Number up to: 7 Sum Value = 28
Number up to: 8 Sum Value = 36
Number up to: 9 Sum Value = 45
Number up to: 10 Sum Value = 55
Number up to: 11 Sum Value = 66
Number up to: 12 Sum Value = 78
```

``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:

```
In [49]: # 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 t
        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]

```
In [180]: x = -1
a = -1
exist = 50
while x < 11:
    x = x + 1
    output = x**2
    a=a+1
    print('The current number is', x, '.', 'And, the multification value
    if output > exist:
        print('The condition was met because', output, 'is exceeded th
        break
```

```
The current number is 0 . And, the multification value of that number
is 0 ( 0 * 0 )
The current number is 1 . And, the multification value of that number
is 1 ( 1 * 1 )
The current number is 2 . And, the multification value of that number
is 4 ( 2 * 2 )
The current number is 3 . And, the multification value of that number
is 9 ( 3 * 3 )
The current number is 4 . And, the multification value of that number
is 16 ( 4 * 4 )
The current number is 5 . And, the multification value of that number
is 25 ( 5 * 5 )
The current number is 6 . And, the multification value of that number
is 36 ( 6 * 6 )
The current number is 7 . And, the multification value of that number
is 49 ( 7 * 7 )
The current number is 8 . And, the multification value of that number
is 64 ( 8 * 8 )
The condition was met because 64 is exceeded the value of 50 . We now
can exist the loop.
```

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

In reality, we could have used the `break` 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.

```
In [52]: 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
```

```
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.]

```
In [104]: # For Loop
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
```

```
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 )
```

```
In [153]: # While Loop
List=100
A = -1
while List<110:
    List=List+1
    OUT = List*2
    A=A+1
    print('The current sum at element ', List, 'is', OUT, '(a is',A,')')
```

```
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 )
```

*I think that for loop is slightly more complex than while loop when you do not have a clear numbers in mind such as the value that less than x. However, I think that for loop is easier than while loop when you have a set of numbers already such as 1, 3, 4, 9, 10, etc.*