# 02_py_fundementals_loops_conditionals

August 12, 2021

## 1 Basic programing

Lets take a first step into programing.

We will use a 'conditional statement' to make a small program that makes a decision,

And a 'for loop' to iterate through multiple things and do some action

these two little programs will provide an intro to some syntax in python

### 1.1 Credit:

things here are a mix of the really excellent Software Carpentry tutorial on Python: http://swcarpentry.github.io/python-novice-inflammation/ .

I've made some slight adaptations here and there, but the credit goes to those organizations. I hope I am using this correctly under the licences:

https://earth-env-data-science.github.io/LICENSE.html https://swcarpentry.github.io/python-novice-inflammation/LICENSE.html

## 2 `if` statement

#### 2.0.1 how do you write your code to make choices?

and `if` statement is a special construct that checks if some criteria is true. If its true, the code moves on one line to exicute a command. If it's not true the next line is not run

```
if [some statement]:
    print('[some statement] is true')
```

in the example above if the thing following the `if` is true, the python runs the following line, if it is false, then the next line is skipped.

note the structure of the if statement: `if` something is followed by `:`, then the next line is indented with a `tab`. Everything that is indented following `if [some statement]:` is *inside* the if statement

[1]:
```
# try a simple example

if 20 > 10:
    print('so true')
```

```
so true
```

[2]:
```
# change the above so it's not true. what happens?

if 8 > 10:
    print('so true')
```

`if` statements can be more complex, you can build multiple options using the follow-on conditional checks `elif` (i.e. 'else if') and `else`

using these you can check for lots of conditions and have you code branch out in many directions

[3]:
```
# use a variable that we can change to check several conditions
# change the value of x and see how that changes the output

x = 100
if x > 0:
    print('Positive Number')
elif x < 0:
    print('Negative Number')
else:
    print ('Zero!')
```

```
Positive Number
```

## 3 `for` loop

the `for` loop is another very important structure in programming that allows you to iterate over a group of things and do some action many times without repeating lines of code.

There are a number of reasons we don't want to have to repreat code. Let explore this

As an example, let's say have a `variable` called `word` which was assigned the string `"lead"`.

[4]:
```
word = 'lead'
print(word)
```

```
lead
```

We can access a character in a string using something call an index. The index is just a reference to the position of some part of the data in a variable.

For example, we can get the first character of the word `lead`, by using `word[0]`.

**note!** in python we count from 0, so the first letter in `word` is in position `0` and the second letter is in position `1`

[5]:
```
# print the first letter in the variable word
# note! python starts with 0
word[0]
```

[5]:
```
'l'
```

One way to print each character is to use four `print` statements:

[6]:
```
print(word[0])
print(word[1])
print(word[2])
print(word[3])
```

```
l
e
a
d
```

This is a bad approach for three reasons:

- **Not scalable**. Imagine you need to print characters of a string that is hundreds of letters long. It might be easier just to type them in manually.
- **Difficult to maintain**. If we want to decorate each printed character with an asterix or any other character, we would have to change four lines of code. While this might not be a problem for short strings, it would definitely be a problem for longer ones.
- **Fragile**. If we use it with a word that has more characters than what we initially envisioned, it will only display part of the word's characters. A shorter string, on the other hand, will cause an error because it wi

[7]:
```
word = "tin"

print(word[0])
print(word[1])
print(word[2])
print(word[3])
```

```
t
i
n
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-7-ab80842014c4> in <module>
      4 print(word[1])
      5 print(word[2])
----> 6 print(word[3])

IndexError: string index out of range
```

Why did we get an error?

Now let's try a better way:

[8]:
```
word = 'lead'
for char in word:
    print(char)
```

```
l
e
a
d
```

Do you recognize the structure there?

This is shorter — certainly shorter than something that prints every character in a hundred-letter string — and more robust as well:

[9]:
```
word = "oxygen"
for char in word:
    print(char)
```

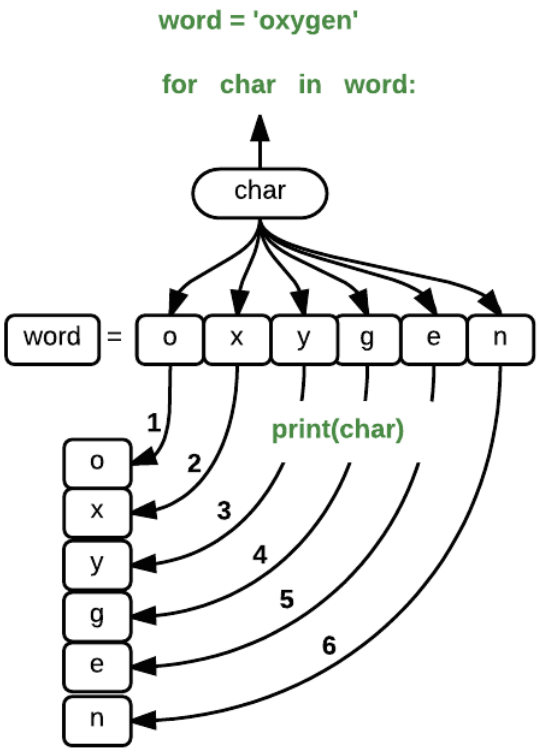
```
o
x
y
g
e
n
```

### 3.0.1 `For` loop structure

The improved version uses a `for` loop to repeat an operation — in this case, printing — once for each thing in a sequence. The general form of a loop is:

```
for element in collection:
    do things using element
```

Using the oxygen example above, the loop might look like this:




where each character (`char`) in the variable `word` is looped through and printed one letter after another. The numbers in the diagram denote which loop cycle the character was printed in (1 being the first loop, and 6 being the final loop).

We can call the loop variable anything we like, but there must be a colon, `:`, at the end of the line starting the loop, and we must **indent** anything we want to run inside the loop. Unlike many other languages, there is no command to signify the end of the loop body (e.g. end ); **what is indented after the for statement belongs to the loop.**

In the example above, the loop variable was given the name `char` as a mnemonic; it is short for 'character'. We can choose any name we want for variables. We might just as easily have chosen the name `banana` for the loop variable, as long as we use the same name when we invoke the variable inside the loop:

[10]:
```
word = "oxygen"
for banana in word:
    print(banana)
```

```
o
x
y
g
e
n
```

Here's another loop that repeatedly updates a variable:

[11]:
```
length = 0
for vowel in 'aeiou':
    length = length + 1
print('There are', length, 'vowels')
```

```
There are 5 vowels
```

It's worth tracing the execution of this little program step by step. Since there are five characters in 'aeiou', the statement on line 3 will be executed five times. The first time around, `length` is zero (the value assigned to it on line 1) and `vowel` is 'a'. The statement adds 1 to the old value of `length`, producing 1, and updates `length` to refer to that new value. The next time around, `vowel` is 'e' and `length` is 1, so `length` is updated to be 2. After three more updates, `length` is 5; since there is nothing left in 'aeiou' for Python to process, the loop finishes and the print statement on line 4 tells us our final answer.

However, after programming a while you'll notice that finding the length of a collection of items, such as a string, is a pretty common task. So much so that python even has a built-in function called `len()` that returns the amount of elements in collection.

[12]:
```
print(len('aeiou'))
```

```
5
```

## 4 Key points

Use `for` variable in sequence to process the elements of a sequence one at a time.

The body of a `for` loop must be indented.

`if` statements can control what happens *if* something is true

The end…

## 5 Exercise 02 / Breakout

Look for the notebook file starting exercise_02