Basic control flow in Python II

Learning outcomes:

- Understand and use control flow in python
- `if`, `else` statements
- Comparisons and Logical operators

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.

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!

Control flow: Logical Tests and Boolean Operators

Believe it or not, everything that happens on your phone or computer comes down to lots (and I mean **LOTS**) of little decisions based on one or two inputs that can be either "True" or "False", and an output that can also be "True" or "False".

Seriously, everything on any digitial device – from Tik Tok videos to your Python code – comes down to a whole bunch of truths and falsehoods (ones and zeros) that are themselves the result of decisions based on other truths and falsehoods. The "decision makers" are actual physical (but teeny teeny tiny) devices that are combinations of things called *transistors* (https://en.wikipedia.org/wiki/Transistor). Transistors perform conditional tests and logical operations on data.

Two are the primary operations performed by transistors:

- ***Comparison*** operations like `==` (equals) and `>` (greater than) that yield `True` or `False`
- ***Logical*** operations that use ***Boolean logic***, which compares two logical inputs and returns `True` or `False` like `A and B` (`True` only if both A and B are `True`) and `A or B` (`True` if either A or B – or both – are `True`).

Let's play with this. It might seem a bit silly and obvious now, but the power of logical tests will reveal itself soon.

Comparison operators

These are operators that test a single value. Imagine wanting to ask, whether the number of lives my cat has (9 for what I was told yesterday, when I was born (https://en.wikipedia.org/wiki/Cat#Superstitions_and_rituals)) is different than the number of lives of Schrödinger's cat (https://en.wikipedia.org/wiki/Schr%C3%B6dinger%27s_cat) has. Number to number type of questions.

Let's set a variable `x` to 11. (Why only go to ten when you can go to 11?)

```
In [1]: x = 11
        x
```

```
Out[1]: 11
```

Now let's do some logical tests on our variable `x`. Let's see if `x` is less than `42`.

```
In [2]: x < 42
```

```
Out[2]: True
```

Now you test if `x` is greater than `42`.

```
In [3]: x > 42
```

```
Out[3]: False
```

We can also test for equality. Is `x` equal to `42`?

```
In [4]: x == 42
```

```
Out[4]: False
```

```
In [5]: x == 11
```

```
Out[5]: True
```

Finally, we can test for *inequality*. (We test whether it is true that x is *not* equal to a specific number).

```
In [6]: x != 42
```

```
Out[6]: True
```

The exclamation point here means "not", so the experession `x != 42` can be read as "is x not equal to 42?"

And the answer is "That's `True`! The variable `x` is not equal to 42!"

Complete the following exercise.

- Now you test `x` to see if it's not equal to `11`. Is it?

  [Use the cell below to show your code]

```
In [7]: x != 11
```

```
Out[7]: False
```

As you might have noticed, all these operations are *built in.* This means that we did not have to import any specific package to access the operations. Python provides these operations as they are core functionality, the bread and butter of most users, or better said, of most programmers. Like you!

Control flow: <if, else> statements

Fundamental to any mature software or code system is the ability to express conditional statements such as `if`, `then` statements. These are among the most basic building blocks of coding, as they allow controlling the flow and allowing a certain operations to occur. For example, operation `a` might be performed only `if` a specific condition 'C' is met. Say, I can eat my cake only if I have been diligent and went out for a nice and long run today.

A couple of basic numerical examples can get us started.

```
In [8]: x = 3
        a_big_number = 100

        if x > 5 :
            print('Yes, it is a big number!')
        else :
            print('Nope, small!')
```

```
Nope, small!
```

We can even add another test using the elif ("else if") statement. When measuring the temperature in Austin TX, we would say:

In [9]:
```python
current_temp = 70

if current_temp >= 90 :
    print('Too hot!')
elif current_temp <= 50 :
    print('Too cold!')
else :
    print('Just right!')
```

```
Just right!
```

`If, then` statements can be combined with logical operators also and help manage complex decisions in a matter of a few lines:

In [10]:
```python
current_temp = 45
if not( (current_temp >= 90) or (current_temp <= 50) ) :
    print('I will ride my bicycle to school!')
else :
    print('Too cold! I will take the car or walk.')
```

```
Too cold! I will take the car or walk.
```

Complete the following exercise.

- To practice with `if , then` statements, write code that when asked if we should eat cake returns `True` only if we have eaten less then 2 slices of cake today and no cake yesterday and the day before yesterday. The code should otherwise tell us to eat soup.

  [Use the cell below to show your code]

In [15]:
```python
cake_today = 1
cake_yesterday = 0
cake_daybefore = 0

if ( (cake_today < 2) and (cake_yesterday == 0) and (cake_daybefore ==
    print("It's cake day!")
else :
    print("Go eat soup!")
```

```
It's cake day!
```

Again, all these operations are *built in* this means that we did not have to import any specific package. Python provides these basic operations as they are the bread and butter of most users, or better said, of most programmers, like you.

Control flow: Logical operators

So far we have been dealing with testing operations on single numbers. Often times it is important to be able to test multiple operations and compare them, say if `a > 0 and b < 0`. Operations that compare or combine two statements are called **logical**. Logical operations such as `and` and `or` are extremely important and widely used in computer programming, mathematics, neuroscience and in real life.

Python provides binary operations `built in`, so there is not requirement to import a specific library.

Imagine wanting to compare the number of cake slices eaten per day by the average individual in three different countries.

In [16]:
```
# average number of cake slices eaten in
USA = 3  # the United States of America
IT = 2   # Italy
CA = 4   # Canada
```

Imagine wanting to know if BOTH the USA AND Canada eat more cake than Italy. We can first compare if the average citozen eat more cake in Italy or the USA:

In [17]:
```
(USA > IT)
```
Out[17]:
```
True
```

Alright, it looks like more cake is eaten in the USA. What about Canada?

In [18]:
```
(CA > IT)
```
Out[18]:
```
True
```

If we wanted to compare both the USA and Canada at the same time, in python we could conveniently write the operation as follows:

In [19]:
```
(USA > IT) and (CA > IT)
```
Out[19]:
```
True
```

The statment about is *only* true if *both* statements are true. Let's test it.

In [20]:
```
# We will use a temporary value for canada and
# then repeat the logical operation with the new value
CA_temp = 1
(USA > IT) and (CA_temp > IT)
```
Out[20]:
```
False
```

OK what happened there is that whereas the first statement was true (3 > 2 slices of cake) the second was not true (1 is not more then 2 slices of cake) and the whole statement returned `False`. The `and` operator returns `True` only if all composing statements return `True`.

Another logical operation of Key value `OR`. `OR` returns `True` if only one of the two statements is `True`, even if the other is `False`. We can try it:

In [21]:
```
CA_temp = 1
(USA > IT) or (CA_temp > IT)
```
Out[21]:
```
True
```

Complete the following exercise.

- What do you expect would be the result if you were to run `or` between the original statements:
  - `(USA > IT)`
  - `(CA > IT)`

  [The result is `True` for *(USA > IT) or (CA > IT)*]

- What is `CA_temp`?

  [`CA_temp` is used as a placeholder.]

In [22]:
```
(USA > IT) or (CA > IT)
```
Out[22]:
```
True
```

Another helpful operator, often used in similar questions is `not`. The `not` operator is a modifier that changes the value of the output of other operators suchas `>`, `=`, `and`, etc.

For example, if `not` is used, the number of slices of cake eaten by the average citizen in Canada is **not** more than those eaten in the USA:

In [23]:

```python




not(CA > USA)
```

Out[23]:

```
False
```

Whereas this is is obeviously `True`

In [24]:

```python
(CA > USA)
```

Out[24]:

```
True
```

So, I like to think about `not` as a modifier. It can become useful in many cases, especially when the output of two or more statments needs to be modified (flipped) for the code to advance. For example, the following code shows how three boolean statements (all set to `True`) can be modified but a boolean `not` to save my health.

In [25]:

```python
# Save my belly
ihaveeatencake = True
itislatenight = True
ididnotexercisetoday = True

INeedToEatCake = not(ihaveeatencake and itislatenight and ididnotexerc
INeedToEatCake
```

Out[25]:

```
False
```

Complete the following exercises.

- Write code containing a `for` look that tests the following operations:
  - 2 is bigger than 3
  - The square of 2 is smaller than the square of 1
  - 4 times 5 is not equal to 20

  Return all the results during the for loop using `print()`

  [Use the cell below to show your code]

In [29]:

```python
if not ( (2>3) and (2**2 < 1**2) and (4*5 != 20)):
    print("This is wrong. To take your math test again.")
else :
    print("Welcome to the Calculus?")
```

```
This is wrong. To take your math test again.
```

- Make a new Python list. You can put whatever you want in it. Make it at least 5 items long.

```
In [33]: my_list = [1,2,3,4,5]
         print(my_list)
```

```
[1, 2, 3, 4, 5]
```

- Get the first element of your new list.

```
In [37]: print(my_list[0])
```

```
1
```

- Get the last element of your new list in a way that wouldn't depend on list length.

```
In [49]: print(my_list[-1:])
```

```
[5]
```

- Get all but the last two elements in a way that wouldn't depend on list length.

```
In [50]: print(my_list[-2:])
```

```
[4, 5]
```

- Get every other element of your list.

```
In [51]: print(my_list[0:5:2])
```

```
[1, 3, 5]
```

Complete the following exercises.

- Use Markdown to make a table and list all the steps necessary to get the new tutorial every class from github, work on it during class and submit the answers to the exercises in the tutorials. Each rown of the table should have a new step. In the first column the table should list the name of the operation, in the second the command used (the full command-line operation executed), in the third where the work is being done (on the cloud, locally on your computer, in a terminal, etc), in the last column there should be a verbal description of what the operation does.

Type *Markdown* and LaTeX: α^2

Steps	Codes/Command	Description	Location
	`Git Pull Step`		
1	Go to Github Website under your username/FDS-CourseOne	Check if there is anything news	GitHub
2	Use git pull request or sync fork option	Get the updated files	GitHub
3	Open terminal		Terminal
4	`pwd`	Check where you are	Terminal
5	`cd git`	Go you your `git` folder	Terminal
6	`cd FDS-CourseOne`	Go you `FDS-CourseOne` folder	Terminal
7	`git pull`	Update files in your local folder to sync with the GitHub Website	Terminal
	Make a copy of your desired file		
8	`cp -v file_name` ../ `PythonTutorials` /	Copy the desired file to `PythonTutorials` folder	Terminal
	`Open Jupyter Notebook`		
9	`cd ~`	Go back to the original page (optional)	Terminal
10	`jupyter notebook`	Open Jupyter Notebook to work on the assigmentns	Terminal
	Uploeading the completed assignments to GitHub		
11	`cd ~`	Go back to the original page	Terminal
12	`cd git`	Go you your `git` folder that contained the completed assignments	Terminal
13	`cd PythonTutorials`	Go to the folder that contained the completed assignments	Terminal
14	`git status`	Check if there any files in your laptop that different from the files in GitHub	Terminal
15	`git add file_name`	Add updated files to the cloud	Terminal
16	`git commit -am` "any comments"	Commit your files to GitHub	Terminal
17	`git push`	Update your files in GitHub	Terminal
	Complete 😊		