Writing Your first Python Code

Estaimted time needed: **25** minutes

Objectives

After compliting this lab you will be able to:

- Write basic code in Python
- Work with various types of data in python
- Convert the data from one type to another
- Use expressions and variables to perform operations

Table of Contents

- Say "Hello" to the world in Python
  - What version of Python are we using?
  - Writing comments in Python
  - Errors in Python
  - Does Python know about your error before it runs your code?
  - Exercise: Your First Program
- Types of objects in Python
  - Integers
  - Floats
  - Converting from one object type to a different object type
  - Boolean data type
  - Exercise: Types
- Expressions and Variables
  - Expressions
  - Exercise: Expressions
  - Variables
  - Exercise: Expression and Variables in Python

Estimated time needed: **25 min**

Say "Hello" to the world in Python

When learning a new programming language, it is customary to start with an "hello world" example. As simple as it is, this one line of code will ensure that we know how to print a string in output and how to execute code within cells in a notebook.

[Tip]: To execute the Python code in the code cell below, click on the cell to select it and press `Shift` + `Enter`.

In [1]:
```
# Try your first Python output

print('Hello, Python!')
```

```
Hello, Python!
```

After executing the cell above, you should see that Python prints `Hello, Python!`. Congratulations on running your first Python code!

[Tip:] `print()` is a function. You passed the string `'Hello, Python!'` as an argument to instruct Python on what to print.

What version of Python are we using?

There are two popular versions of the Python programming language in use today: Python 2 and Python 3. The Python community has decided to move on from Python 2 to Python 3, and many popular libraries have announced that they will no longer support Python 2.

Since Python 3 is the future, in this course we will be using it exclusively. How do we know that our notebook is executed by a Python 3 runtime? We can look

in the top-right hand corner of this notebook and see "Python 3".

We can also ask directly Python and obtain a detailed answer. Try executing the following code:

In [2]:
```
# Check the Python Version

import sys
print(sys.version)
```

```
3.6.11 | packaged by conda-forge | (default, Jul 23 2020, 22:18:32)
[GCC 7.5.0]
```

[Tip:] `sys` is a built-in module that contains many system-specific parameters and functions, including the Python version in use. Before using it, we must explictly `import` it.

Writing comments in Python

In addition to writing code, note that it's always a good idea to add comments to your code. It will help others understand what you were trying to accomplish (the reason why you wrote a given snippet of code). Not only does this help **other people** understand your code, it can also serve as a reminder **to you** when you come back to it weeks or months later.

To write comments in Python, use the number symbol `#` before writing your comment. When you run your code, Python will ignore everything past the `#` on a given line.

In [3]:
```
# Practice on writing comments

print('Hello, Python!') # This line prints a string
# print('Hi')
```

```
Hello, Python!
```

After executing the cell above, you should notice that `This line prints a string` did not appear in the output, because it was a comment (and thus ignored by Python).

The second line was also not executed because `print('Hi')` was preceded by the number sign (`#`) as well! Since this isn't an explanatory comment from the programmer, but an actual line of code, we might say that the programmer *commented out* that second line of code.

Errors in Python

Everyone makes mistakes. For many types of mistakes, Python will tell you that you have made a mistake by giving you an error message. It is important to read error messages carefully to really understand where you made a mistake and how you may go about correcting it.

For example, if you spell `print` as `frint`, Python will display an error message. Give it a try:

In [4]:
```
# Print string as error message

frint("Hello, Python!")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-313a1769a8a5> in <module>
      1 # Print string as error message
      2
----> 3 frint("Hello, Python!")

NameError: name 'frint' is not defined
```

The error message tells you:

1. where the error occurred (more useful in large notebook cells or scripts), and
2. what kind of error it was (NameError)

Here, Python attempted to run the function `frint`, but could not determine what `frint` is since it's not a built-in function and it has not been previously defined by us either.

You'll notice that if we make a different type of mistake, by forgetting to close the string, we'll obtain a different error (i.e., a `SyntaxError`). Try it below:

In [5]:
```
# Try to see build in error message

print("Hello, Python!)
```

```
  File "<ipython-input-5-63a21a726720>", line 3
    print("Hello, Python!)
                          ^
SyntaxError: EOL while scanning string literal
```

Does Python know about your error before it runs your code?

Python is what is called an *interpreted language*. Compiled languages examine your entire program at compile time, and are able to warn you about a whole class of errors prior to execution. In contrast, Python interprets your script line by line as it executes it. Python will stop executing the entire program when it encounters an error (unless the error is expected and handled by the programmer, a more advanced subject that we'll cover later on in this course).

Try to run the code in the cell below and see what happens:

In [6]:
```
# Print string and error to see the running order

print("This will be printed")
frint("This will cause an error")
print("This will NOT be printed")
```

```
This will be printed

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-af59af1b345d> in <module>
      2
      3 print("This will be printed")
----> 4 frint("This will cause an error")
      5 print("This will NOT be printed")

NameError: name 'frint' is not defined
```

Exercise: Your First Program

Generations of programmers have started their coding careers by simply printing "Hello, world!". You will be following in their footsteps.

In the code cell below, use the `print()` function to print out the phrase: `Hello, world!`

In [7]:
```
print("Hello, world!")
```

```
Hello, world!
```

Double-click **here** for the solution.

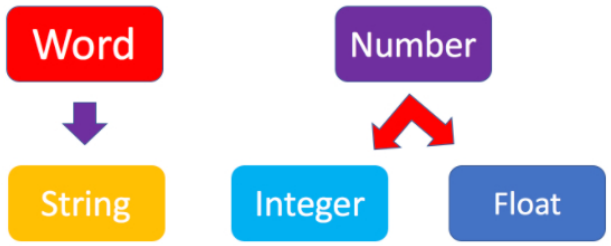
Now, let's enhance your code with a comment. In the code cell below, print out the phrase: `Hello, world!` and comment it with the phrase `Print the traditional hello world` all in one line of code.

In [8]:
```
#Print the traditional hello world
```

Double-click **here** for the solution.

Types of objects in Python

Python is an object-oriented language. There are many different types of objects in Python. Let's start with the most common object types: *strings*, *integers* and *floats*. Anytime you write words (text) in Python, you're using *character strings* (strings for short). The most common numbers, on the other hand, are *integers* (e.g. -1, 0, 100) and *floats*, which represent real numbers (e.g. 3.14, -42.0).



The following code cells contain some examples.

In [9]:
```
# Integer

11
```

Out[9]:
```
11
```

In [10]:
```
# Float

2.14
```

Out[10]:
```
2.14
```

In [11]:
```
# String

"Hello, Python 101!"
```

Out[11]:
```
'Hello, Python 101!'
```

You can get Python to tell you the type of an expression by using the built-in `type()` function. You'll notice that Python refers to integers as `int`, floats as `float`, and character strings as `str`.

In [12]:
```
# Type of 12
```

```
type(12)
```

Out[12]: int

In [13]:
```
# Type of 2.14

type(2.14)
```

Out[13]: float

In [14]:
```
# Type of "Hello, Python 101!"

type("Hello, Python 101!")
```

Out[14]: str

In the code cell below, use the `type()` function to check the object type of `12.0`.

In [15]:
```
type(12.0)
```

Out[15]: float

Double-click **here** for the solution.

Integers

Here are some examples of integers. Integers can be negative or positive numbers:

We can verify this is the case by using, you guessed it, the `type()` function:

In [16]:
```
# Print the type of -1

type(-1)
```

Out[16]: int

In [17]:
```
# Print the type of 4

type(4)
```

Out[17]: int

In [18]:
```
# Print the type of 0

type(0)
```

Out[18]: int

Floats

Floats represent real numbers; they are a superset of integer numbers but also include "numbers with decimals". There are some limitations when it comes to machines representing real numbers, but floating point numbers are a good representation in most cases. You can learn more about the specifics of floats for your runtime environment, by checking the value of `sys.float_info`. This will also tell you what's the largest and smallest number that can be represented with them.

Once again, can test some examples with the `type()` function:

In [19]:
```
# Print the type of 1.0

type(1.0) # Notice that 1 is an int, and 1.0 is a float
```

Out[19]: float

In [20]:
```
# Print the type of 0.5

type(0.5)
```

Out[20]: float

In [21]:
```
# Print the type of 0.56

type(0.56)
```

Out[21]: float

In [22]:
```
# System settings about float type

sys.float_info
```

Out[22]:
```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

Converting from one object type to a different object type

You can change the type of the object in Python; this is called typecasting. For example, you can convert an *integer* into a *float* (e.g. 2 to 2.0).

Let's try it:

In [23]:
```
# Verify that this is an integer

type(2)
```

Out[23]:
```
int
```

Converting integers to floats

Let's cast integer 2 to float:

In [24]:
```
# Convert 2 to a float

float(2)
```

Out[24]:
```
2.0
```

In [25]:
```
# Convert integer 2 to a float and check its type

type(float(2))
```

Out[25]:
```
float
```

When we convert an integer into a float, we don't really change the value (i.e., the significand) of the number. However, if we cast a float into an integer, we could potentially lose some information. For example, if we cast the float 1.1 to integer we will get 1 and lose the decimal information (i.e., 0.1):

In [26]:
```
# Casting 1.1 to integer will result in loss of information

int(1.1)
```

Out[26]:
```
1
```

Converting from strings to integers or floats

Sometimes, we can have a string that contains a number within it. If this is the case, we can cast that string that represents a number into an integer using `int()`:

In [27]:
```
# Convert a string into an integer

int('1')
```

Out[27]:
```
1
```

But if you try to do so with a string that is not a perfect match for a number, you'll get an error. Try the following:

In [28]:
```
# Convert a string into an integer with error

int('1 or 2 people')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-28-b78145d165c7> in <module>
      1 # Convert a string into an integer with error
      2
----> 3 int('1 or 2 people')

ValueError: invalid literal for int() with base 10: '1 or 2 people'
```

You can also convert strings containing floating point numbers into *float* objects:

In [29]:
```
# Convert the string "1.2" into a float

float('1.2')
```

Out[29]:
```
1.2
```

[Tip:] Note that strings can be represented with single quotes (`'1.2'`) or double quotes (`"1.2"`), but you can't mix both (e.g., `"1.2'`).

Converting numbers to strings

If we can convert strings to numbers, it is only natural to assume that we can convert numbers to strings, right?

In [30]:
```
# Convert an integer to a string

str(1)
```

Out[30]:
```
'1'
```

And there is no reason why we shouldn't be able to make floats into strings as well:

In [31]:
```
# Convert a float to a string

str(1.2)
```

Out[31]:
```
'1.2'
```

Boolean data type

Boolean is another important type in Python. An object of type *Boolean* can take on one of two values: `True` or `False`:

In [32]:
```
# Value true

True
```

Out[32]: True

Notice that the value `True` has an uppercase "T". The same is true for `False` (i.e. you must use the uppercase "F").

In [33]:
```
# Value false

False
```

Out[33]: False

When you ask Python to display the type of a boolean object it will show `bool` which stands for *boolean*:

In [34]:
```
# Type of True

type(True)
```

Out[34]: bool

In [35]:
```
# Type of False

type(False)
```

Out[35]: bool

We can cast boolean objects to other data types. If we cast a boolean with a value of `True` to an integer or float we will get a one. If we cast a boolean with a value of `False` to an integer or float we will get a zero. Similarly, if we cast a 1 to a Boolean, you get a `True`. And if we cast a 0 to a Boolean we will get a `False`. Let's give it a try:

In [36]:
```
# Convert True to int

int(True)
```

Out[36]: 1

In [37]:
```
# Convert 1 to boolean

bool(1)
```

Out[37]: True

In [38]:
```
# Convert 0 to boolean

bool(0)
```

Out[38]: False

In [39]:
```
# Convert True to float

float(True)
```

Out[39]: 1.0

Exercise: Types

What is the data type of the result of: `6 / 2`?

In [40]:
```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
```

Double-click **here** for the solution.

What is the type of the result of: `6 // 2`? (Note the double slash `//`.)

In [41]:
```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
```

Double-click **here** for the solution.

Expression and Variables

Expressions

Expressions in Python can include operations among compatible types (e.g., integers and floats). For example, basic arithmetic operations like adding multiple numbers:

In [42]:
```
# Addition operation expression

43 + 60 + 16 + 41
```

Out[42]: 160

We can perform subtraction operations using the minus operator. In this case the result is a negative number:

In [43]:
```
# Subtraction operation expression

50 - 60
```

Out[43]: -10

We can do multiplication using an asterisk:

In [44]:
```
# Multiplication operation expression

5 * 5
```

Out[44]: 25

We can also perform division with the forward slash:

In [45]:
```
# Division operation expression

25 / 5
```

Out[45]: 5.0

In [46]:
```
# Division operation expression

25 / 6
```

Out[46]: 4.166666666666667

As seen in the quiz above, we can use the double slash for integer division, where the result is rounded to the nearest integer:

In [47]:
```
# Integer division operation expression

25 // 5
```

Out[47]: 5

In [48]:
```
# Integer division operation expression

25 // 6
```

Out[48]: 4

Exercise: Expression

Let's write an expression that calculates how many hours there are in 160 minutes:

In [49]:
```
160/60
```

Out[49]: 2.6666666666666665

Double-click **here** for the solution.

Python follows well accepted mathematical conventions when evaluating mathematical expressions. In the following example, Python adds 30 to the result of the multiplication (i.e., 120).

In [50]:
```
# Mathematical expression

30 + 2 * 60
```

Out[50]: 150

And just like mathematics, expressions enclosed in parentheses have priority. So the following multiplies 32 by 60.

In [51]:
```
# Mathematical expression

(30 + 2) * 60
```

Out[51]: 1920

Variables

Just like with most programming languages, we can store values in *variables*, so we can use them later on. For example:

In [52]:
```
# Store value into variable

x = 43 + 60 + 16 + 41
```

To see the value of `x` in a Notebook, we can simply place it on the last line of a cell:

In [53]:
```
# Print out the value in variable

x
```

Out[53]: 160

We can also perform operations on `x` and save the result to a new variable:

In [54]:
```
# Use another variable to store the result of the operation between variable and value

y = x / 60
y
```

Out[54]: 2.6666666666666665

If we save a value to an existing variable, the new value will overwrite the previous value:

In [55]:
```
# Overwrite variable with new value

x = x / 60
x
```

Out[55]: 2.6666666666666665

It's a good practice to use meaningful variable names, so you and others can read the code and understand it more easily:

In [56]:
```
# Name the variables meaningfully

total_min = 43 + 42 + 57 # Total length of albums in minutes
total_min
```

Out[56]: 142

In [57]:
```
# Name the variables meaningfully

total_hours = total_min / 60 # Total length of albums in hours
total_hours
```

Out[57]: 2.3666666666666667

In the cells above we added the length of three albums in minutes and stored it in `total_min`. We then divided it by 60 to calculate total length `total_hours` in hours. You can also do it all at once in a single expression, as long as you use parenthesis to add the albums length before you divide, as shown below.

In [58]:
```
# Complicate expression

total_hours = (43 + 42 + 57) / 60  # Total hours in a single expression
total_hours
```

Out[58]: 2.3666666666666667

If you'd rather have total hours as an integer, you can of course replace the floating point division with integer division (i.e., `//`).

Exercise: Expression and Variables in Python

What is the value of `x` where `x = 3 + 2 * 2`

In [59]:
```
x = 3+2*2
x
```

Out[59]: 7

Double-click **here** for the solution.

What is the value of `y` where `y = (3 + 2) * 2`?

In [60]:
```
y = (3+2)*2
y
```

Out[60]: 10

Double-click **here** for the solution.

What is the value of `z` where `z = x + y`?

In [61]:
```
z = x+y
```

Double-click **here** for the solution.

The last exercise!

Congratulations, you have completed your first lesson and hands-on lab in Python. However, there is one more thing you need to do. The Data Science community encourages sharing work. The best way to share and showcase your work is to share it on GitHub. By sharing your notebook on GitHub you are not only building your reputation with fellow data scientists, but you can also show it off when applying for a job. Even though this was your first piece of work, it is never too early to start building good habits. So, please read and follow this article to learn how to share your work.

Author

Author

Joseph Santarcangelo

Other contributors

Mavis Zhou

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-08-26	2.0	Lavanya	Moved lab to course repo in GitLab

String Operations

Estaimted time needed: **15** minutes

Objectives

After compliting this lab you will be able to:

- Work with Strings
- Perform operations on String
- Manipulate Strings using indexing and escape sequences

Table of Contents

- What are Strings?
- Indexing
  - Negative Indexing
  - Slicing
  - Stride
  - Concatenate Strings
- Escape Sequences
- String Operations
- Quiz on Strings

Estimated time needed: **15 min**

What are Strings?

The following example shows a string contained within 2 quotation marks:

In [1]:
```
# Use quotation marks for defining string

"Michael Jackson"
```

Out[1]: 'Michael Jackson'

We can also use single quotation marks:

In [2]:
```
# Use single quotation marks for defining string

'Michael Jackson'
```

Out[2]: 'Michael Jackson'

A string can be a combination of spaces and digits:

In [3]:
```
# Digitals and spaces in string

'1 2 3 4 5 6 '
```

Out[3]: '1 2 3 4 5 6 '

A string can also be a combination of special characters :

In [4]:
```
# Special characters in string

'@#2_#]&*^%$'
```

Out[4]: '@#2_#]&*^%$'

We can print our string using the print statement:

In [5]:
```
# Print the string

print("hello!")
```

```
hello!
```

We can bind or assign a string to another variable:

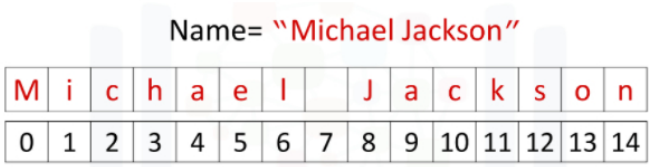
In [6]:
```
# Assign string to variable
```

```
Name = "Michael Jackson"
Name
```

Out[6]: 'Michael Jackson'

Indexing

It is helpful to think of a string as an ordered sequence. Each element in the sequence can be accessed using an index represented by the array of numbers:



The first index can be accessed as follows:

> [Tip]: Because indexing starts at 0, it means the first index is on the index 0.

In [7]:
```
# Print the first element in the string

print(Name[0])
```

```
M
```

We can access index 6:

In [8]:
```
# Print the element on index 6 in the string

print(Name[6])
```

```
l
```

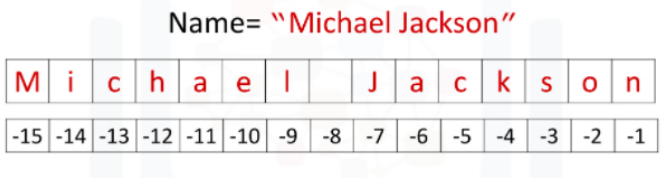
Moreover, we can access the 13th index:

In [9]:
```
# Print the element on the 13th index in the string

print(Name[13])
```

```
o
```

Negative Indexing

We can also use negative indexing with strings:



Negative index can help us to count the element from the end of the string.

The last element is given by the index -1:

In [10]:
```
# Print the last element in the string

print(Name[-1])
```

```
n
```

The first element can be obtained by index -15:

In [11]:
```
# Print the first element in the string

print(Name[-15])
```

```
M
```

We can find the number of characters in a string by using `len`, short for length:

In [12]:
```
# Find the length of string

len("Michael Jackson")
```

Out[12]: 15

Slicing

We can obtain multiple characters from a string using slicing, we can obtain the 0 to 4th and 8th to the 12th element:

[Tip]: When taking the slice, the first number means the index (start at 0), and the second number means the length from the index to the last element you want (start at 1)

In [13]:
```
# Take the slice on variable Name with only index 0 to index 3

Name[0:4]
```

Out[13]: 'Mich'

In [14]:
```
# Take the slice on variable Name with only index 8 to index 11

Name[8:12]
```

Out[14]: 'Jack'

Stride

We can also input a stride value as follows, with the '2' indicating that we are selecting every second variable:

In [17]:
```
# Get every second element. The elments on index 1, 3, 5 ...

Name[::2]
Name[::-1]
```

Out[17]: 'noskcaJ leahciM'

We can also incorporate slicing with the stride. In this case, we select the first five elements and then use the stride:

In [18]:
```
# Get every second element in the range from index 0 to index 4

Name[0:5:2]
```

Out[18]: 'Mca'

Concatenate Strings

We can concatenate or combine strings by using the addition symbols, and the result is a new string that is a combination of both:

In [19]:
```
# Concatenate two strings

Statement = Name + "is the best"
Statement
```

Out[19]: 'Michael Jacksonis the best'

To replicate values of a string we simply multiply the string by the number of times we would like to replicate it. In this case, the number is three. The result is a new string, and this new string consists of three copies of the original string:

In [20]:
```
# Print the string for 3 times

3 * "Michael Jackson"
```

```
Out[20]: 'Michael JacksonMichael JacksonMichael Jackson'
```

You can create a new string by setting it to the original variable. Concatenated with a new string, the result is a new string that changes from Michael Jackson to "Michael Jackson is the best".

```
In [21]: # Concatenate strings

Name = "Michael Jackson"
Name = Name + " is the best"
Name
```

```
Out[21]: 'Michael Jackson is the best'
```

Escape Sequences

Back slashes represent the beginning of escape sequences. Escape sequences represent strings that may be difficult to input. For example, back slash "n" represents a new line. The output is given by a new line after the back slash "n" is encountered:

```
In [22]: # New line escape sequence

print(" Michael Jackson \n is the best" )
```

```
 Michael Jackson
 is the best
```

Similarly, back slash "t" represents a tab:

```
In [23]: # Tab escape sequence

print(" Michael Jackson \t is the best" )
```

```
 Michael Jackson         is the best
```

If you want to place a back slash in your string, use a double back slash:

```
In [24]: # Include back slash in string

print(" Michael Jackson \\ is the best" )
```

```
 Michael Jackson \ is the best
```

We can also place an "r" before the string to display the backslash:

```
In [25]: # r will tell python that string will be display as raw string

print(r" Michael Jackson \ is the best" )
```

```
 Michael Jackson \ is the best
```

String Operations

There are many string operation methods in Python that can be used to manipulate the data. We are going to use some basic string operations on the data.

Let's try with the method `upper`; this method converts lower case characters to upper case characters:

```
In [26]: # Convert all the characters in string to upper case

A = "Thriller is the sixth studio album"
print("before upper:", A)
B = A.upper()
print("After upper:", B)
```

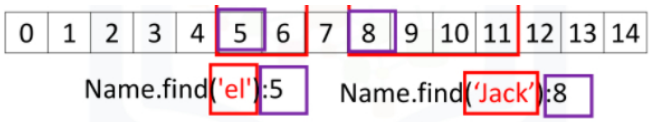
```
before upper: Thriller is the sixth studio album
After upper: THRILLER IS THE SIXTH STUDIO ALBUM
```

The method `replace` replaces a segment of the string, i.e. a substring with a new string. We input the part of the string we would like to change. The second argument is what we would like to exchange the segment with, and the result is a new string with the segment changed:

```
In [27]: # Replace the old substring with the new target substring is the segment has been found in the string

A = "Michael Jackson is the best"
B = A.replace('Michael', 'Janet')
B
```

```
Out[27]: 'Janet Jackson is the best'
```

The method `find` finds a sub-string. The argument is the substring you would like to find, and the output is the first index of the sequence. We can find the sub-string `jack` or `el .`



In [28]:
```
# Find the substring in the string. Only the index of the first elment of substring in string will be the output

Name = "Michael Jackson"
Name.find('el')
```

Out[28]: 5

In [29]:
```
# Find the substring in the string.

Name.find('Jack')
```

Out[29]: 8

If the sub-string is not in the string then the output is a negative one. For example, the string 'Jasdfasdasdf' is not a substring:

In [30]:
```
# If cannot find the substring in the string

Name.find('Jasdfasdasdf')
```

Out[30]: -1

Quiz on Strings

What is the value of the variable `A` after the following code is executed?

In [32]:
```
# Write your code below and press Shift+Enter to execute

A = "1"
A
```

Out[32]: '1'

Double-click **here** for the solution.

What is the value of the variable `B` after the following code is executed?

In [34]:
```
# Write your code below and press Shift+Enter to execute

B = "2"
B
```

Out[34]: '2'

Double-click **here** for the solution.

What is the value of the variable `C` after the following code is executed?

In [36]:
```
# Write your code below and press Shift+Enter to execute

C = A + B
C
```

Out[36]: '12'

Double-click **here** for the solution.

Consider the variable `D` use slicing to print out the first three elements:

In [37]:
```
# Write your code below and press Shift+Enter to execute

D = "ABCDEFG"
D[0:4]
```

Out[37]: 'ABCD'

Double-click **here** for the solution.

Use a stride value of 2 to print out every second character of the string `E` :

In [38]:
```
# Write your code below and press Shift+Enter to execute

E = 'clocrkr1e1c1t'
E[::2]
```

Out[38]: 'correct'

Double-click **here** for the solution.

Print out a backslash:

In [40]:
```
# Write your code below and press Shift+Enter to execute
print("\\")
```

```
\
```

Double-click **here** for the solution.

Convert the variable `F` to uppercase:

In [41]:
```
# Write your code below and press Shift+Enter to execute

F = "You are wrong"
F.upper()
```

Out[41]:
```
'YOU ARE WRONG'
```

Double-click **here** for the solution.

Consider the variable `G`, and find the first index of the sub-string `snow`:

In [42]:
```
# Write your code below and press Shift+Enter to execute

G = "Mary had a little lamb Little lamb, little lamb Mary had a little lamb \
Its fleece was white as snow And everywhere that Mary went Mary went, Mary went \
Everywhere that Mary went The lamb was sure to go"
G.find("snow")
```

Out[42]:
```
95
```

Double-click **here** for the solution.

In the variable `G`, replace the sub-string `Mary` with `Bob`:

In [43]:
```
# Write your code below and press Shift+Enter to execute
G.replace("Mary", "Bob")
```

Out[43]:
```
'Bob had a little lamb Little lamb, little lamb Bob had a little lamb Its fleece was white as snow And everywhere that Bob went Bob went, Bob went Everywhere that Bob went The lamb was sure to go'
```

Double-click **here** for the solution.

The last exercise!

Congratulations, you have completed your first lesson and hands-on lab in Python. However, there is one more thing you need to do. The Data Science community encourages sharing work. The best way to share and showcase your work is to share it on GitHub. By sharing your notebook on GitHub you are not only building your reputation with fellow data scientists, but you can also show it off when applying for a job. Even though this was your first piece of work, it is never too early to start building good habits. So, please read and follow this article to learn how to share your work.

Author

Joseph Santarcangelo

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-08-26	2.0	Lavanya	Moved lab to course repo in GitLab

In [ ]:
```

```

Lists in Python

Estimated time needed: **15** minutes

Objectives

After completing this lab you will be able to:

- Perform list operations in Python, including indexing, list manipulation and copy/clone list.

Table of Contents

- About the Dataset
- Lists
  - Indexing
  - List Content
  - List Operations
  - Copy and Clone List
- Quiz on Lists

Estimated time needed: **15 min**

About the Dataset

Imagine you received album recommendations from your friends and compiled all of the recommandations into a table, with specific information about each album.

The table has one row for each movie and several columns:

- **artist** - Name of the artist
- **album** - Name of the album
- **released_year** - Year the album was released
- **length_min_sec** - Length of the album (hours,minutes,seconds)
- **genre** - Genre of the album
- **music_recording_sales_millions** - Music recording sales (millions in USD) on SONG://DATABASE
- **claimed_sales_millions** - Album's claimed sales (millions in USD) on SONG://DATABASE
- **date_released** - Date on which the album was released
- **soundtrack** - Indicates if the album is the movie soundtrack (Y) or (N)
- **rating_of_friends** - Indicates the rating from your friends from 1 to 10

The dataset can be seen below:

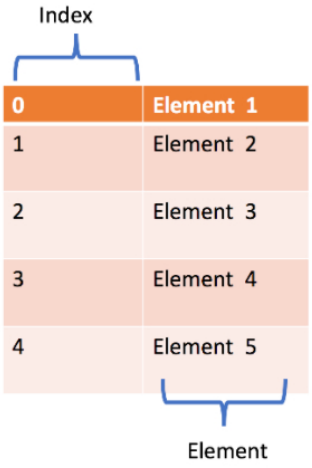
Artist	Album	Released	Length	Genre	Music recording sales (millions)	Claimed sales (millions)	Released	Soundtrack	Rating (friends)
Michael Jackson	Thriller	1982	00:42:19	Pop, rock, R&B	46	65	30-Nov-82		10.0
AC/DC	Back in Black	1980	00:42:11	Hard rock	26.1	50	25-Jul-80		8.5
Pink Floyd	The Dark Side of the Moon	1973	00:42:49	Progressive rock	24.2	45	01-Mar-73		9.5
Whitney Houston	The Bodyguard	1992	00:57:44	Soundtrack/R&B, soul, pop	26.1	50	25-Jul-80	Y	7.0
Meat Loaf	Bat Out of Hell	1977	00:46:33	Hard rock, progressive rock	20.6	43	21-Oct-77		7.0
Eagles	Their Greatest Hits (1971-1975)	1976	00:43:08	Rock, soft rock, folk rock	32.2	42	17-Feb-76		9.5
Bee Gees	Saturday Night Fever	1977	1:15:54	Disco	20.6	40	15-Nov-77	Y	9.0
Fleetwood Mac	Rumours	1977	00:40:01	Soft rock	27.9	40	04-Feb-77		9.5

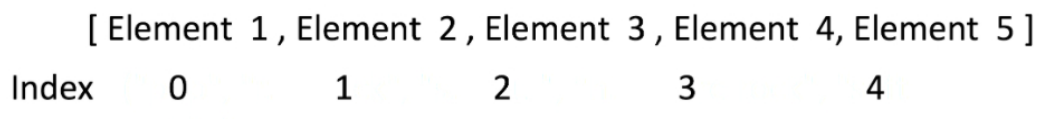
Lists

Indexing

We are going to take a look at lists in Python. A list is a sequenced collection of different objects such as integers, strings, and other lists as well. The address

of each element within a list is called an **index**. An index is used to access and refer to items within a list.



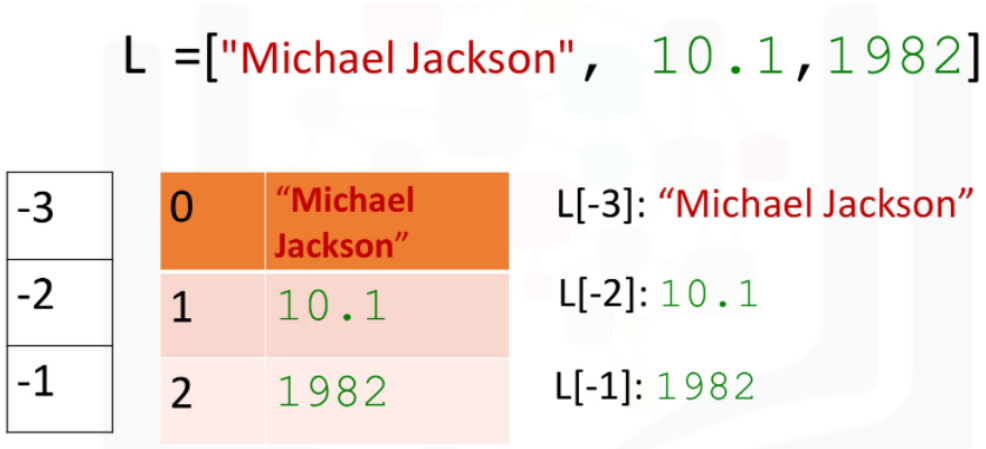


To create a list, type the list within square brackets **[ ]**, with your content inside the parenthesis and separated by commas. Let's try it!

In [1]:
```
# Create a list

L = ["Michael Jackson", 10.1, 1982]
L
```

Out[1]:
```
['Michael Jackson', 10.1, 1982]
```

We can use negative and regular indexing with a list :



In [2]:
```
# Print the elements on each index

print('the same element using negative and positive indexing:\n Postive:',L[0],
'\n Negative:' , L[-3]  )
print('the same element using negative and positive indexing:\n Postive:',L[1],
'\n Negative:' , L[-2]  )
print('the same element using negative and positive indexing:\n Postive:',L[2],
'\n Negative:' , L[-1]  )
```

```
the same element using negative and positive indexing:
 Postive: Michael Jackson
 Negative: Michael Jackson
the same element using negative and positive indexing:
 Postive: 10.1
 Negative: 10.1
the same element using negative and positive indexing:
 Postive: 1982
 Negative: 1982
```

List Content

Lists can contain strings, floats, and integers. We can nest other lists, and we can also nest tuples and other data structures. The same indexing conventions apply for nesting:

In [3]:
```
# Sample List
```

```
["Michael Jackson", 10.1, 1982, [1, 2], ("A", 1)]
```

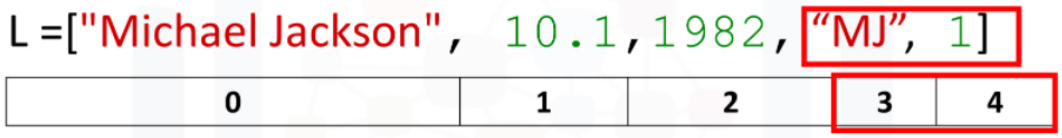
Out[3]:
```
['Michael Jackson', 10.1, 1982, [1, 2], ('A', 1)]
```

List Operations

We can also perform slicing in lists. For example, if we want the last two elements, we use the following command:

In [4]:
```
# Sample List

L = ["Michael Jackson", 10.1,1982,"MJ",1]
L
```

Out[4]:
```
['Michael Jackson', 10.1, 1982, 'MJ', 1]
```



In [5]:
```
# List slicing

L[3:5]
```

Out[5]:
```
['MJ', 1]
```

We can use the method `extend` to add new elements to the list:

In [6]:
```
# Use extend to add elements to list

L = [ "Michael Jackson", 10.2]
L.extend(['pop', 10])
L
```

Out[6]:
```
['Michael Jackson', 10.2, 'pop', 10]
```

Another similar method is `append`. If we apply `append` instead of `extend`, we add one element to the list:

In [7]:
```
# Use append to add elements to list

L = [ "Michael Jackson", 10.2]
L.append(['pop', 10])
L
```

Out[7]:
```
['Michael Jackson', 10.2, ['pop', 10]]
```

Each time we apply a method, the list changes. If we apply `extend` we add two new elements to the list. The list `L` is then modified by adding two new elements:

In [8]:
```
# Use extend to add elements to list

L = [ "Michael Jackson", 10.2]
L.extend(['pop', 10])
L
```

Out[8]:
```
['Michael Jackson', 10.2, 'pop', 10]
```

If we append the list `['a','b']` we have one new element consisting of a nested list:

In [9]:
```
# Use append to add elements to list

L.append(['a','b'])
L
```

Out[9]:
```
['Michael Jackson', 10.2, 'pop', 10, ['a', 'b']]
```

As lists are mutable, we can change them. For example, we can change the first element as follows:

In [10]:
```
# Change the element based on the index

A = ["disco", 10, 1.2]
print('Before change:', A)
A[0] = 'hard rock'
print('After change:', A)
```

```
Before change: ['disco', 10, 1.2]
After change: ['hard rock', 10, 1.2]
```

We can also delete an element of a list using the `del` command:

In [11]:
```
# Delete the element based on the index

print('Before change:', A)
del(A[0])
print('After change:', A)
```

```
Before change: ['hard rock', 10, 1.2]
After change: [10, 1.2]
```

We can convert a string to a list using `split`. For example, the method `split` translates every group of characters seperated by a space into an element in

We can convert a string to a list using `split`. For example, the method `split` translates every group of characters separated by a space into an element in a list:

In [12]:
```
# Split the string, default is by space

'hard rock'.split()
```

Out[12]:
```
['hard', 'rock']
```

We can use the split function to separate strings on a specific character. We pass the character we would like to split on into the argument, which in this case is a comma. The result is a list, and each element corresponds to a set of characters that have been separated by a comma:

In [13]:
```
# Split the string by comma

'A,B,C,D'.split(',')
```

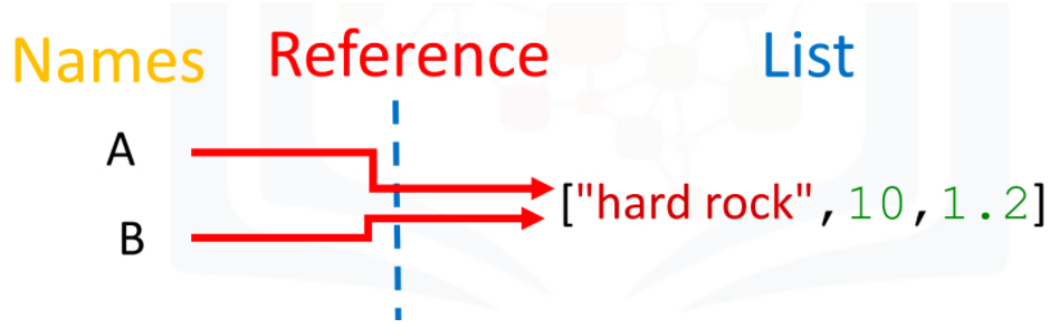
Out[13]:
```
['A', 'B', 'C', 'D']
```

Copy and Clone List

When we set one variable **B** equal to **A**; both **A** and **B** are referencing the same list in memory:

In [14]:
```
# Copy (copy by reference) the list A

A = ["hard rock", 10, 1.2]
B = A
print('A:', A)
print('B:', B)
```

```
A: ['hard rock', 10, 1.2]
B: ['hard rock', 10, 1.2]
```

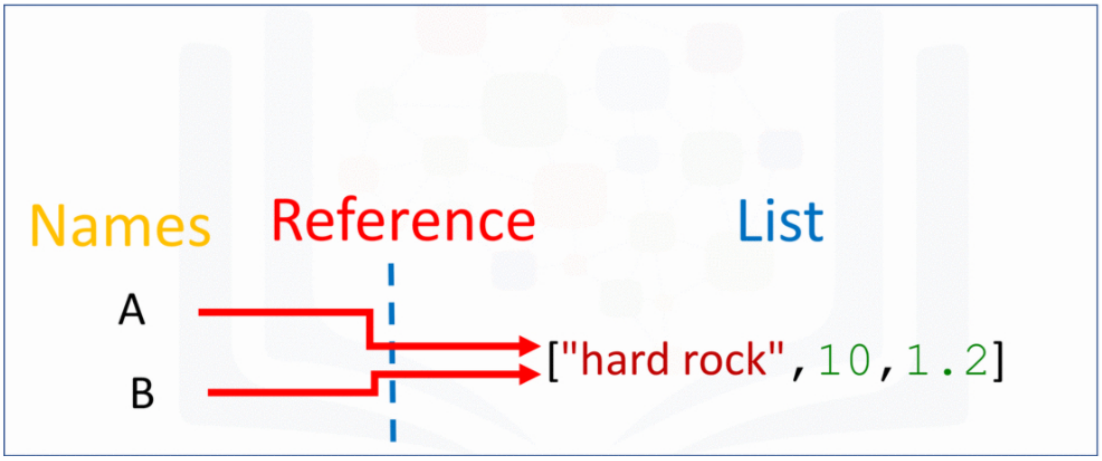


Initially, the value of the first element in **B** is set as hard rock. If we change the first element in **A** to **banana**, we get an unexpected side effect. As **A** and **B** are referencing the same list, if we change list **A**, then list **B** also changes. If we check the first element of **B** we get banana instead of hard rock:

In [15]:
```
# Examine the copy by reference

print('B[0]:', B[0])
A[0] = "banana"
print('B[0]:', B[0])
```

```
B[0]: hard rock
B[0]: banana
```

This is demonstrated in the following figure:



You can clone list **A** by using the following syntax:

In [16]:
```
# Clone (clone by value) the list A

B = A[:]
B
```

Out[16]:
```
['banana', 10, 1.2]
```

Variable **B** references a new copy or clone of the original list; this is demonstrated in the following figure:

Now if you change **A**, **B** will not change:

In [17]:
```
print('B[0]:', B[0])
A[0] = "hard rock"
print('B[0]:', B[0])
```

```
B[0]: banana
B[0]: banana
```

Quiz on List

Create a list `a_list`, with the following elements `1`, `hello`, `[1,2,3]` and `True`.

In [19]:
```
# Write your code below and press Shift+Enter to execute
a_list = [1, "hello", [1,2,3] ,"True"]
```

Double-click **here** for the solution.

Find the value stored at index 1 of `a_list`.

In [21]:
```
# Write your code below and press Shift+Enter to execute
a_list[1]
```

Out[21]:
```
'hello'
```

Double-click **here** for the solution.

Retrieve the elements stored at index 1, 2 and 3 of `a_list`.

In [23]:
```
# Write your code below and press Shift+Enter to execute
a_list[1:4]
```

Out[23]:
```
['hello', [1, 2, 3], 'True']
```

Double-click **here** for the solution.

A = [1, 'a']Concatenate the following lists `A = [1, 'a']` and `B = [2, 1, 'd']`:

In [24]:
```
# Write your code below and press Shift+Enter to execute
A = [1, 'a']
B = [2, 1, 'd']
A + B
```

Out[24]:
```
[1, 'a', 2, 1, 'd']
```

Double-click **here** for the solution.

The last exercise!

Congratulations, you have completed your first lesson and hands-on lab in Python. However, there is one more thing you need to do. The Data Science community encourages sharing work. The best way to share and showcase your work is to share it on GitHub. By sharing your notebook on GitHub you are not only building your reputation with fellow data scientists, but you can also show it off when applying for a job. Even though this was your first piece of work, it is never too early to start building good habits. So, please read and follow this article to learn how to share your work.

Author

Joseph Santarcangelo

Other contributors

Mavis Zhou

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-08-26	2.0	Lavanya	Moved lab to course repo in GitLab

Tuples in Python

Estaimted time needed: **15** minutes

Objectives

After compliting this lab you will be able to:

- Perform the basics tuple operations in Python, including indexing, slicing and sorting

Table of Contents

- About the Dataset
- Tuples
  - Indexing
  - Slicing
  - Sorting
- Quiz on Tuples

Estimated time needed: **15 min**

About the Dataset

Imagine you received album recommendations from your friends and compiled all of the recommendations into a table, with specific information about each album.

The table has one row for each movie and several columns:

- **artist** - Name of the artist
- **album** - Name of the album
- **released_year** - Year the album was released
- **length_min_sec** - Length of the album (hours,minutes,seconds)
- **genre** - Genre of the album
- **music_recording_sales_millions** - Music recording sales (millions in USD) on SONG://DATABASE
- **claimed_sales_millions** - Album's claimed sales (millions in USD) on SONG://DATABASE
- **date_released** - Date on which the album was released
- **soundtrack** - Indicates if the album is the movie soundtrack (Y) or (N)
- **rating_of_friends** - Indicates the rating from your friends from 1 to 10

The dataset can be seen below:

Artist	Album	Released	Length	Genre	Music recording sales (millions)	Claimed sales (millions)	Released	Soundtrack	Rating (friends)
Michael Jackson	Thriller	1982	00:42:19	Pop, rock, R&B	46	65	30-Nov-82		10.0
AC/DC	Back in Black	1980	00:42:11	Hard rock	26.1	50	25-Jul-80		8.5
Pink Floyd	The Dark Side of the Moon	1973	00:42:49	Progressive rock	24.2	45	01-Mar-73		9.5
Whitney Houston	The Bodyguard	1992	00:57:44	Soundtrack/R&B, soul, pop	26.1	50	25-Jul-80	Y	7.0
Meat Loaf	Bat Out of Hell	1977	00:46:33	Hard rock, progressive rock	20.6	43	21-Oct-77		7.0
Eagles	Their Greatest Hits (1971-1975)	1976	00:43:08	Rock, soft rock, folk rock	32.2	42	17-Feb-76		9.5
Bee Gees	Saturday Night Fever	1977	1:15:54	Disco	20.6	40	15-Nov-77	Y	9.0
Fleetwood Mac	Rumours	1977	00:40:01	Soft rock	27.9	40	04-Feb-77		9.5

Tuples

In Python, there are different data types: string, integer and float. These data types can all be contained in a tuple as follows:

Now, let us create your first tuple with string, integer and float.

In [1]:
```
# Create your first tuple

tuple1 = ("disco",10,1.2 )
tuple1
```

Out[1]:
```
('disco', 10, 1.2)
```

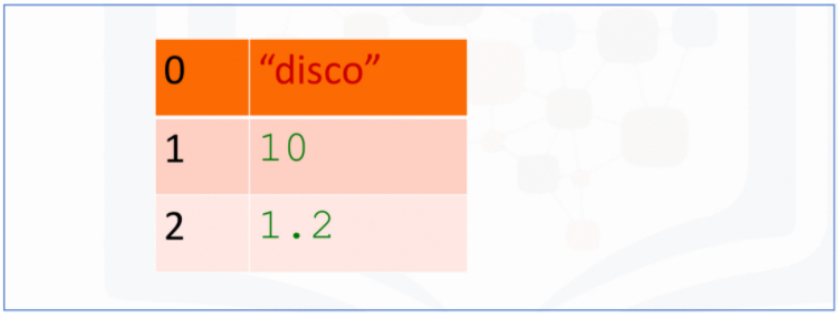
The type of variable is a **tuple**.

In [2]:
```
# Print the type of the tuple you created

type(tuple1)
```

Out[2]:
```
tuple
```

Indexing

Each element of a tuple can be accessed via an index. The following table represents the relationship between the index and the items in the tuple. Each element can be obtained by the name of the tuple followed by a square bracket with the index number:



We can print out each value in the tuple:

In [3]:
```
# Print the variable on each index

print(tuple1[0])
print(tuple1[1])
print(tuple1[2])
```

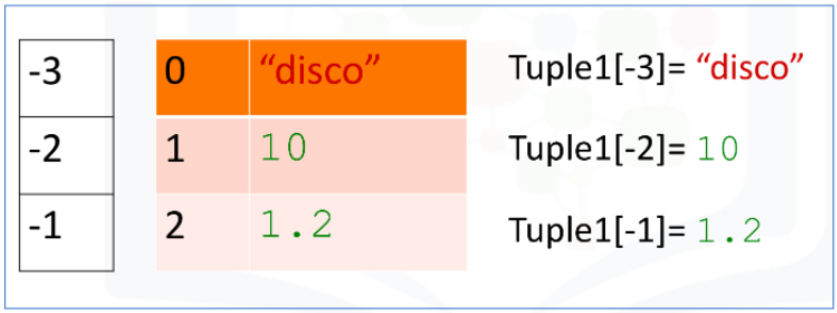
```
disco
10
1.2
```

We can print out the **type** of each value in the tuple:

In [4]:
```
# Print the type of value on each index

print(type(tuple1[0]))
print(type(tuple1[1]))
print(type(tuple1[2]))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
```

We can also use negative indexing. We use the same table above with corresponding negative values:



We can obtain the last element as follows (this time we will not use the print statement to display the values):

In [5]:
```
# Use negative index to get the value of the last element

tuple1[-1]
```

Out[5]:
```
1.2
```

We can display the next two elements as follows:

In [6]:
```
# Use negative index to get the value of the second last element

tuple1[-2]
```

Out[6]:
```
10
```

In [7]:
```
# Use negative index to get the value of the third last element

tuple1[-3]
```

Out[7]:
```
'disco'
```

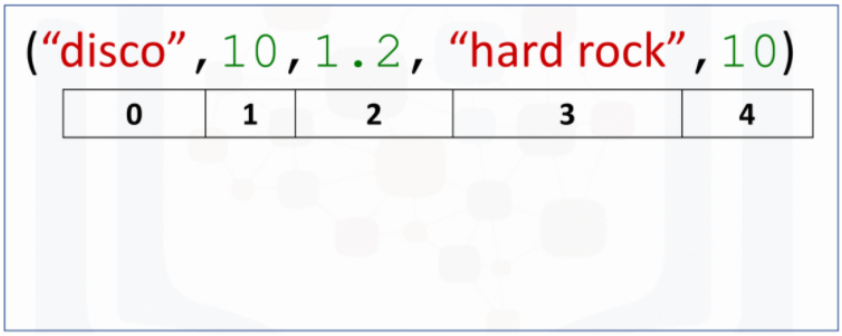
Concatenate Tuples

We can concatenate or combine tuples by using the **+** sign:

In [8]:
```
# Concatenate two tuples

tuple2 = tuple1 + ("hard rock", 10)
tuple2
```

Out[8]:
```
('disco', 10, 1.2, 'hard rock', 10)
```

We can slice tuples obtaining multiple values as demonstrated by the figure below:



Slicing

We can slice tuples, obtaining new tuples with the corresponding elements:

In [9]:
```
# Slice from index 0 to index 2

tuple2[0:3]
```

Out[9]:
```
('disco', 10, 1.2)
```

We can obtain the last two elements of the tuple:

In [10]:
```
# Slice from index 3 to index 4

tuple2[3:5]
```

Out[10]:
```
('hard rock', 10)
```

We can obtain the length of a tuple using the length command:

In [11]:
```
# Get the length of tuple

len(tuple2)
```

Out[11]:
```
5
```

This figure shows the number of elements:

Sorting

Consider the following tuple:

In [12]:
```
# A sample tuple

Ratings = (0, 9, 6, 5, 10, 8, 9, 6, 2)
```

We can sort the values in a tuple and save it to a new tuple:

In [13]:
```
# Sort the tuple

RatingsSorted = sorted(Ratings)
RatingsSorted
```

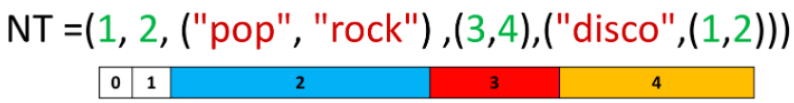
Out[13]:
```
[0, 2, 5, 6, 6, 8, 9, 9, 10]
```

Nested Tuple

A tuple can contain another tuple as well as other more complex data types. This process is called 'nesting'. Consider the following tuple with several elements:

In [14]:
```
# Create a nest tuple

NestedT =(1, 2, ("pop", "rock") ,(3,4),("disco",(1,2)))
```

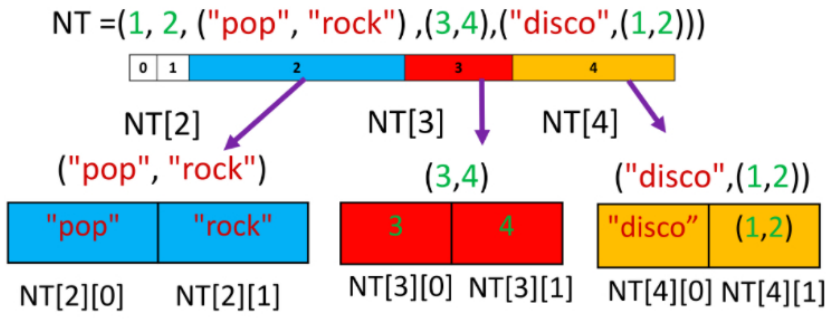
Each element in the tuple including other tuples can be obtained via an index as shown in the figure:



In [15]:
```
# Print element on each index

print("Element 0 of Tuple: ", NestedT[0])
print("Element 1 of Tuple: ", NestedT[1])
print("Element 2 of Tuple: ", NestedT[2])
print("Element 3 of Tuple: ", NestedT[3])
print("Element 4 of Tuple: ", NestedT[4])
```

```
Element 0 of Tuple:  1
Element 1 of Tuple:  2
Element 2 of Tuple:  ('pop', 'rock')
Element 3 of Tuple:  (3, 4)
Element 4 of Tuple:  ('disco', (1, 2))
```

We can use the second index to access other tuples as demonstrated in the figure:



We can access the nested tuples :

In [16]:
```
# Print element on each index, including nest indexes

print("Element 2, 0 of Tuple: ",   NestedT[2][0])
print("Element 2, 1 of Tuple: ",   NestedT[2][1])
print("Element 3, 0 of Tuple: ",   NestedT[3][0])
print("Element 3, 1 of Tuple: ",   NestedT[3][1])
print("Element 4, 0 of Tuple: ",   NestedT[4][0])
print("Element 4, 1 of Tuple: ",   NestedT[4][1])
```

```
Element 2, 0 of Tuple:  pop
Element 2, 1 of Tuple:  rock
Element 3, 0 of Tuple:  3
Element 3, 1 of Tuple:  4
Element 4, 0 of Tuple:  disco
Element 4, 1 of Tuple:  (1, 2)
```

We can access strings in the second nested tuples using a third index:

In [17]:
```
# Print the first element in the second nested tuples

NestedT[2][1][0]
```

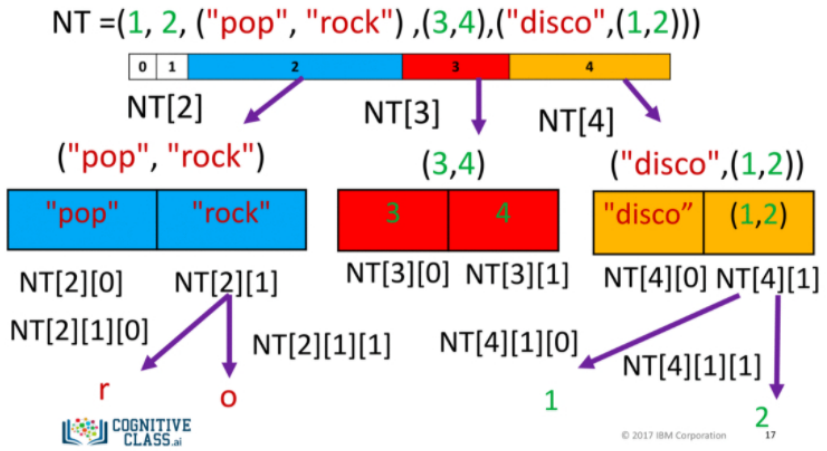
Out[17]:
```
'r'
```

In [18]:
```
# Print the second element in the second nested tuples
```

```
NestedT[2][1][1]
```

Out[18]: 'o'

We can use a tree to visualise the process. Each new index corresponds to a deeper level in the tree:



Similarly, we can access elements nested deeper in the tree with a fourth index:

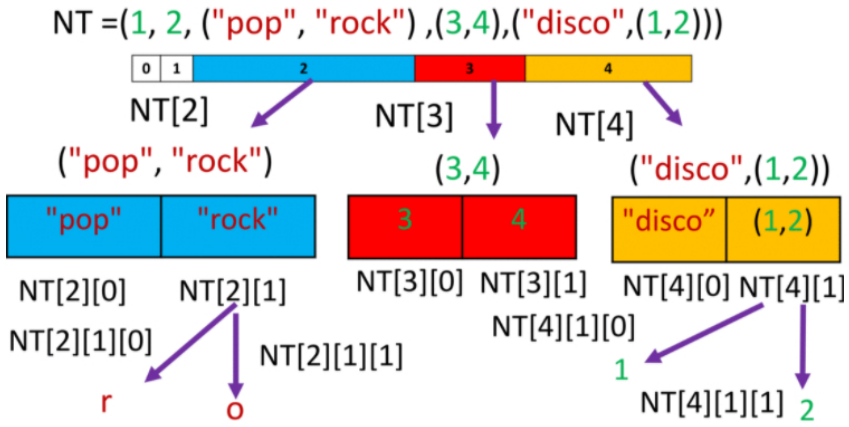
In [19]:
```
# Print the first element in the second nested tuples

NestedT[4][1][0]
```

Out[19]: 1

In [20]:
```
# Print the second element in the second nested tuples

NestedT[4][1][1]
```

Out[20]: 2

The following figure shows the relationship of the tree and the element `NestedT[4][1][1]`:



Quiz on Tuples

Consider the following tuple:

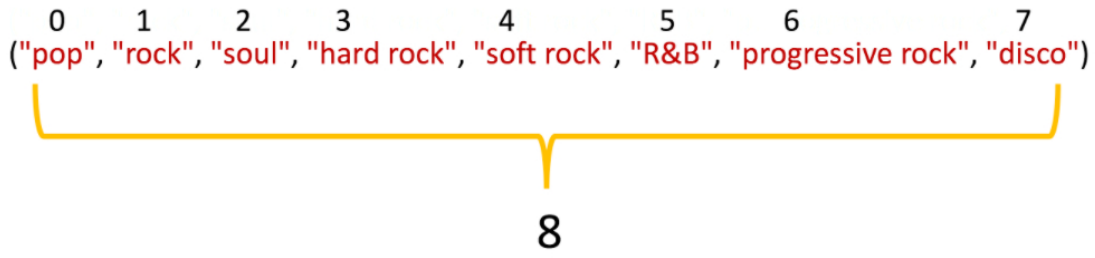
In [21]:
```
# sample tuple

genres_tuple = ("pop", "rock", "soul", "hard rock", "soft rock", \
                "R&B", "progressive rock", "disco")
genres_tuple
```

Out[21]:
```
('pop',
 'rock',
 'soul',
 'hard rock',
 'soft rock',
 'R&B',
 'progressive rock',
 'disco')
```

Find the length of the tuple, `genres_tuple`:

In [22]:
```
# Write your code below and press Shift+Enter to execute
len(genres_tuple)
```

Out[22]: 8



Double-click **here** for the solution.

Access the element, with respect to index 3:

In [24]:
```
# Write your code below and press Shift+Enter to execute
genres_tuple[3]
```

Out[24]:
```
'hard rock'
```

Double-click **here** for the solution.

Use slicing to obtain indexes 3, 4 and 5:

In [25]:
```
# Write your code below and press Shift+Enter to execute
genres_tuple[3:6]
```

Out[25]:
```
('hard rock', 'soft rock', 'R&B')
```

Double-click **here** for the solution.

Find the first two elements of the tuple `genres_tuple`:

In [28]:
```
# Write your code below and press Shift+Enter to execute
genres_tuple[0:2]
```

Out[28]:
```
('pop', 'rock')
```

Double-click **here** for the solution.

Find the first index of `"disco"`:

In [31]:
```
# Write your code below and press Shift+Enter to execute
genres_tuple.index("disco")
```

Out[31]:
```
7
```

Double-click **here** for the solution.

Generate a sorted List from the Tuple `C_tuple=(-5, 1, -3)`:

In [35]:
```
# Write your code below and press Shift+Enter to execute
C_tuple=(-5, 1, -3)
sorted_list = sorted(C_tuple)
sorted_list
```

Out[35]:
```
[-5, -3, 1]
```

Double-click **here** for the solution.

The last exercise!

Congratulations, you have completed your first lesson and hands-on lab in Python. However, there is one more thing you need to do. The Data Science community encourages sharing work. The best way to share and showcase your work is to share it on GitHub. By sharing your notebook on GitHub you are not only building your reputation with fellow data scientists, but you can also show it off when applying for a job. Even though this was your first piece of work, it is never too early to start building good habits. So, please read and follow this article to learn how to share your work.

Author

Joseph Santarcangelo

Other contributors

Mavis Zhou

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-08-26	2.0	Lavanya	Moved lab to course repo in GitLab

Dictionaries in Python

Estaimted time needed: **20** minutes

Objectives

After compliting this lab you will be able to:

- Work with libraries in Python, including operations

Table of Contents

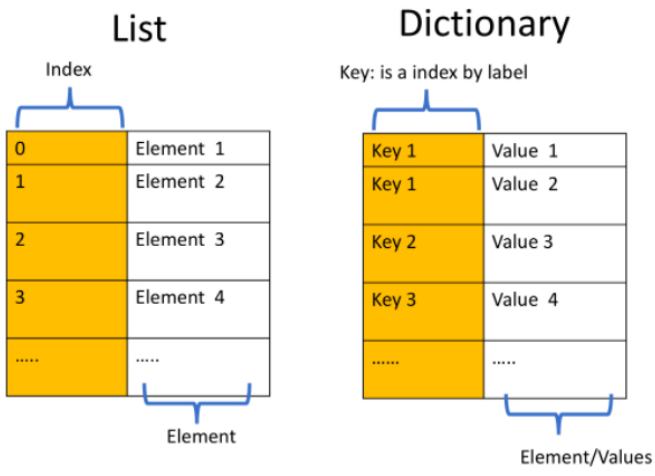
- Dictionaries
  - What are Dictionaries?
  - Keys
- Quiz on Dictionaries

Estimated time needed: **20 min**

Dictionaries

What are Dictionaries?

A dictionary consists of keys and values. It is helpful to compare a dictionary to a list. Instead of the numerical indexes such as a list, dictionaries have keys. These keys are the keys that are used to access values within a dictionary.



An example of a Dictionary `Dict`:

In [1]:
```
# Create the dictionary

Dict = {"key1": 1, "key2": "2", "key3": [3, 3, 3], "key4": (4, 4, 4), ('key5'): 5, (0, 1): 6}
Dict
```

Out[1]:
```
{'key1': 1,
 'key2': '2',
 'key3': [3, 3, 3],
 'key4': (4, 4, 4),
 'key5': 5,
 (0, 1): 6}
```

The keys can be strings:

In [2]:
```
# Access to the value by the key

Dict["key1"]
```

Out[2]:
```
1
```

Keys can also be any immutable object such as a tuple:

In [3]:
```
# Access to the value by the key

Dict[(0, 1)]
```

Out[3]: 6

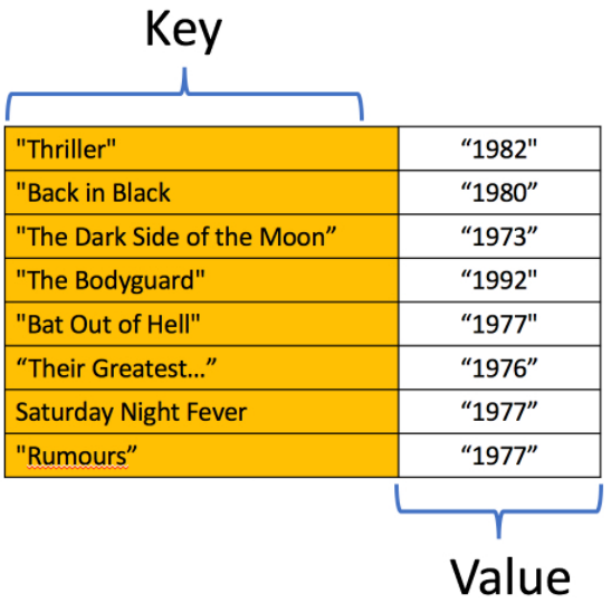
Each key is separated from its value by a colon " `:` ". Commas separate the items, and the whole dictionary is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this " `{}` ".

In [4]:
```
# Create a sample dictionary

release_year_dict = {"Thriller": "1982", "Back in Black": "1980", \
                     "The Dark Side of the Moon": "1973", "The Bodyguard": "1992", \
                     "Bat Out of Hell": "1977", "Their Greatest Hits (1971-1975)": "1976", \
                     "Saturday Night Fever": "1977", "Rumours": "1977"}
release_year_dict
```

Out[4]:
```
{'Thriller': '1982',
 'Back in Black': '1980',
 'The Dark Side of the Moon': '1973',
 'The Bodyguard': '1992',
 'Bat Out of Hell': '1977',
 'Their Greatest Hits (1971-1975)': '1976',
 'Saturday Night Fever': '1977',
 'Rumours': '1977'}
```

In summary, like a list, a dictionary holds a sequence of elements. Each element is represented by a key and its corresponding value. Dictionaries are created with two curly braces containing keys and values separated by a colon. For every key, there can only be one single value, however, multiple keys can hold the same value. Keys can only be strings, numbers, or tuples, but values can be any data type.

It is helpful to visualize the dictionary as a table, as in the following image. The first column represents the keys, the second column represents the values.



Keys

You can retrieve the values based on the names:

In [5]:
```
# Get value by keys

release_year_dict['Thriller']
```

Out[5]: '1982'

This corresponds to:

"Thriller"	"1982"
"Back in Black	"1980"
"The Dark Side of the Moon"	"1973"
"The Bodyguard"	"1992"
"Bat Out of Hell"	"1977"
"Their Greatest..."	"1976"
"Saturday Night Fever"	"1977"
"Rumours"	"1977"

Similarly for **The Bodyguard**

In [6]:
```
# Get value by key

release_year_dict['The Bodyguard']
```

Out[6]:
```
'1992'
```

"Thriller"	“1982"
"Back in Black	“1980”
"The Dark Side of the Moon”	“1973”
"The Bodyguard"	“1992"
"Bat Out of Hell"	“1977"
“Their Greatest...”	“1976”
“Saturday Night Fever”	“1977”
"Rumours”	“1977”

Now let you retrieve the keys of the dictionary using the method `release_year_dict()`:

In [7]:
```
# Get all the keys in dictionary

release_year_dict.keys()
```

Out[7]:
```
dict_keys(['Thriller', 'Back in Black', 'The Dark Side of the Moon', 'The Bodyguard', 'Bat Out of Hell', 'Their Greatest Hits
(1971-1975)', 'Saturday Night Fever', 'Rumours'])
```

You can retrieve the values using the method `values()`:

In [8]:
```
# Get all the values in dictionary

release_year_dict.values()
```

Out[8]:
```
dict_values(['1982', '1980', '1973', '1992', '1977', '1976', '1977', '1977'])
```

We can add an entry:

In [9]:
```
# Append value with key into dictionary

release_year_dict['Graduation'] = '2007'
release_year_dict
```

Out[9]:
```
{'Thriller': '1982',
 'Back in Black': '1980',
 'The Dark Side of the Moon': '1973',
 'The Bodyguard': '1992',
 'Bat Out of Hell': '1977',
 'Their Greatest Hits (1971-1975)': '1976',
 'Saturday Night Fever': '1977',
 'Rumours': '1977',
 'Graduation': '2007'}
```

We can delete an entry:

In [10]:
```
# Delete entries by key

del(release_year_dict['Thriller'])
del(release_year_dict['Graduation'])
release_year_dict
```

Out[10]:
```
{'Back in Black': '1980',
 'The Dark Side of the Moon': '1973',
 'The Bodyguard': '1992',
 'Bat Out of Hell': '1977',
 'Their Greatest Hits (1971-1975)': '1976',
 'Saturday Night Fever': '1977',
 'Rumours': '1977'}
```

We can verify if an element is in the dictionary:

In [11]:
```
# Verify the key is in the dictionary

'The Bodyguard' in release_year_dict
```

Out[11]:
```
True
```

Quiz on Dictionaries

You will need this dictionary for the next two questions:

In [12]:
```
# Question sample dictionary
```

```
soundtrack_dic = {"The Bodyguard":"1992", "Saturday Night Fever":"1977"}
soundtrack_dic
```

Out[12]:
```
{'The Bodyguard': '1992', 'Saturday Night Fever': '1977'}
```

a) In the dictionary `soundtrack_dic` what are the keys ?

In [14]:
```
soundtrack_dic.keys()
```

Out[14]:
```
dict_keys(['The Bodyguard', 'Saturday Night Fever'])
```

Double-click **here** for the solution.

b) In the dictionary `soundtrack_dic` what are the values ?

In [15]:
```
soundtrack_dic.values()
```

Out[15]:
```
dict_values(['1992', '1977'])
```

Double-click **here** for the solution.

You will need this dictionary for the following questions:

The Albums **Back in Black**, **The Bodyguard** and **Thriller** have the following music recording sales in millions 50, 50 and 65 respectively:

a) Create a dictionary `album_sales_dict` where the keys are the album name and the sales in millions are the values.

In [17]:
```
album_sales_dict = {"Back in Black":50, "The Bodyguard": 50, "Thriller":65}
```

Double-click **here** for the solution.

b) Use the dictionary to find the total sales of **Thriller**:

In [18]:
```
album_sales_dict["Thriller"]
```

Out[18]:
```
65
```

Double-click **here** for the solution.

c) Find the names of the albums from the dictionary using the method `keys` :

In [19]:
```
album_sales_dict.keys()
```

Out[19]:
```
dict_keys(['Back in Black', 'The Bodyguard', 'Thriller'])
```

Double-click **here** for the solution.

d) Find the names of the recording sales from the dictionary using the method `values` :

In [21]:
```
album_sales_dict.values()
```

Out[21]:
```
dict_values([50, 50, 65])
```

Double-click **here** for the solution.

The last exercise!

Congratulations, you have completed your first lesson and hands-on lab in Python. However, there is one more thing you need to do. The Data Science community encourages sharing work. The best way to share and showcase your work is to share it on GitHub. By sharing your notebook on GitHub you are not only building your reputation with fellow data scientists, but you can also show it off when applying for a job. Even though this was your first piece of work, it is never too early to start building good habits. So, please read and follow this article to learn how to share your work.

Author

Joseph Santarcangelo

Other contributors

Mavis Zhou

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description

2020-09-09	2.1	Malika Singla	Updated the variable soundtrack_dict to soundtrack_dic in Questions
2020-08-26	2.0	Lavanya	Moved lab to course repo in GitLab

In [ ]:

Sets in Python

Estaimted time needed: **20** minutes

Objectives

After compliting this lab you will be able to:

- Work with sets in Python, including operations and logic operations.

Table of Contents

- Sets
  - Set Content
  - Set Operations
  - Sets Logic Operations
- Quiz on Sets

Estimated time needed: **20 min**

Sets

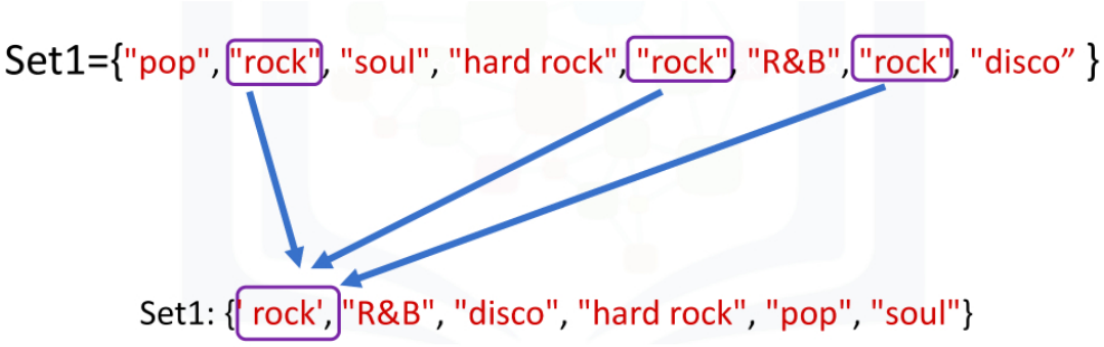
Set Content

A set is a unique collection of objects in Python. You can denote a set with a curly bracket **{}**. Python will automatically remove duplicate items:

In [1]:
```
# Create a set

set1 = {"pop", "rock", "soul", "hard rock", "rock", "R&B", "rock", "disco"}
set1
```

Out[1]:
```
{'R&B', 'disco', 'hard rock', 'pop', 'rock', 'soul'}
```

The process of mapping is illustrated in the figure:



You can also create a set from a list as follows:

In [2]:
```
# Convert list to set

album_list = [ "Michael Jackson", "Thriller", 1982, "00:42:19", \
               "Pop, Rock, R&B", 46.0, 65, "30-Nov-82", None, 10.0]
album_set = set(album_list)
album_set
```

Out[2]:
```
{'00:42:19',
 10.0,
 1982,
 '30-Nov-82',
 46.0,
 65,
 'Michael Jackson',
 None,
 'Pop, Rock, R&B',
 'Thriller'}
```

Now let us create a set of genres:

In [3]:
```
# Convert list to set

music_genres = set(["pop", "pop", "rock", "folk rock", "hard rock", "soul", \
                    "progressive rock", "soft rock", "R&B", "disco"])
music_genres
```

Out[3]:
```
{'R&B',
 'disco',
 'folk rock',
 'hard rock',
 'pop',
 'progressive rock',
 'rock',
 'soft rock',
 'soul'}
```

Set Operations

Let us go over set operations, as these can be used to change the set. Consider the set **A**:

In [4]:
```
# Sample set

A = set(["Thriller", "Back in Black", "AC/DC"])
A
```

Out[4]:
```
{'AC/DC', 'Back in Black', 'Thriller'}
```

We can add an element to a set using the `add()` method:

In [5]:
```
# Add element to set

A.add("NSYNC")
A
```

Out[5]:
```
{'AC/DC', 'Back in Black', 'NSYNC', 'Thriller'}
```

If we add the same element twice, nothing will happen as there can be no duplicates in a set:

In [6]:
```
# Try to add duplicate element to the set

A.add("NSYNC")
A
```

Out[6]:
```
{'AC/DC', 'Back in Black', 'NSYNC', 'Thriller'}
```

We can remove an item from a set using the `remove` method:

In [7]:
```
# Remove the element from set

A.remove("NSYNC")
A
```

Out[7]:
```
{'AC/DC', 'Back in Black', 'Thriller'}
```

We can verify if an element is in the set using the `in` command:

In [8]:
```
# Verify if the element is in the set

"AC/DC" in A
```

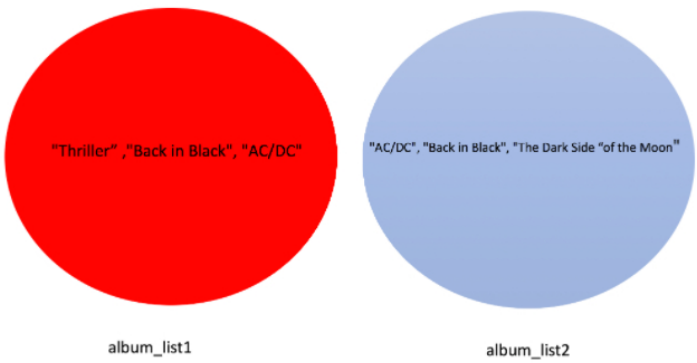
Out[8]:
```
True
```

Sets Logic Operations

Remember that with sets you can check the difference between sets, as well as the symmetric difference, intersection, and union:

Consider the following two sets:

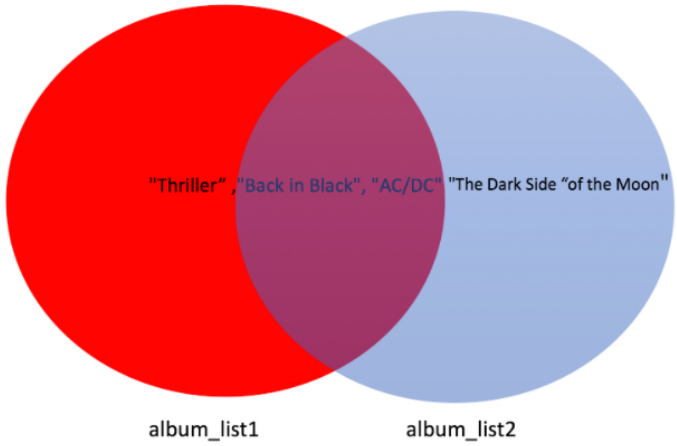
In [9]:
```
# Sample Sets

album_set1 = set(["Thriller", 'AC/DC', 'Back in Black'])
album_set2 = set([ "AC/DC", "Back in Black", "The Dark Side of the Moon"])
```



In [10]:
```
# Print two sets

album_set1, album_set2
```

Out[10]:
```
({'AC/DC', 'Back in Black', 'Thriller'},
 {'AC/DC', 'Back in Black', 'The Dark Side of the Moon'})
```

As both sets contain **AC/DC** and **Back in Black** we represent these common elements with the intersection of two circles.



You can find the intersect of two sets as follow using `&`:

In [11]:
```
# Find the intersections

intersection = album_set1 & album_set2
intersection
```

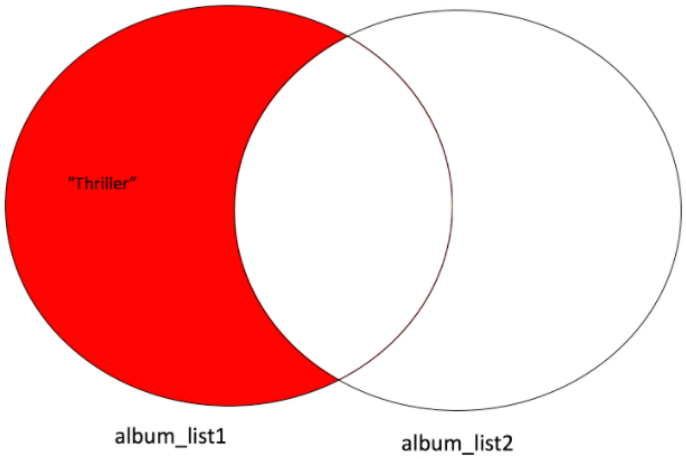
Out[11]:
```
{'AC/DC', 'Back in Black'}
```

You can find all the elements that are only contained in `album_set1` using the `difference` method:

In [12]:
```
# Find the difference in set1 but not set2

album_set1.difference(album_set2)
```

Out[12]:
```
{'Thriller'}
```

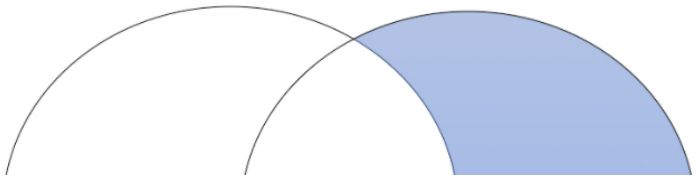
You only need to consider elements in `album_set1`; all the elements in `album_set2`, including the intersection, are not included.

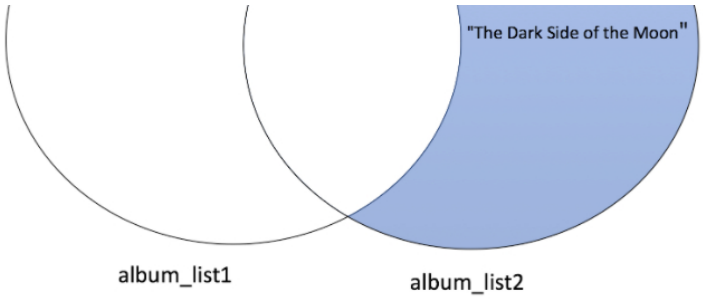


The elements in `album_set2` but not in `album_set1` is given by:

In [13]:
```
album_set2.difference(album_set1)
```

Out[13]:
```
{'The Dark Side of the Moon'}
```



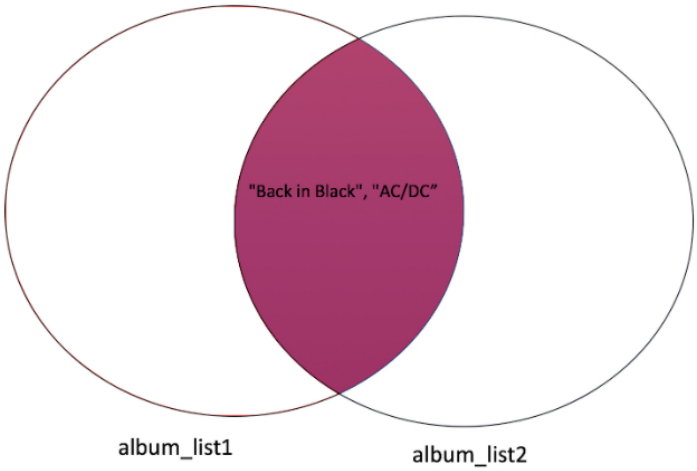


You can also find the intersection of `album_list1` and `album_list2`, using the `intersection` method:

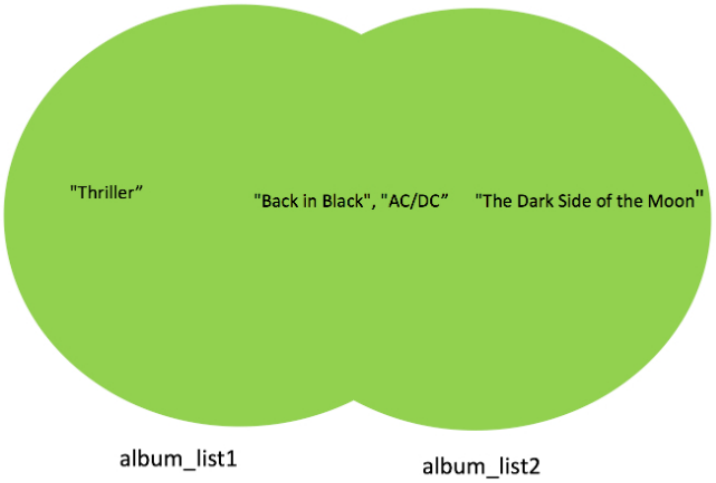
In [14]:
```
# Use intersection method to find the intersection of album_list1 and album_list2

album_set1.intersection(album_set2)
```

Out[14]:
```
{'AC/DC', 'Back in Black'}
```

This corresponds to the intersection of the two circles:



The union corresponds to all the elements in both sets, which is represented by coloring both circles:



The union is given by:

In [15]:
```
# Find the union of two sets

album_set1.union(album_set2)
```

Out[15]:
```
{'AC/DC', 'Back in Black', 'The Dark Side of the Moon', 'Thriller'}
```

And you can check if a set is a superset or subset of another set, respectively, like this:

In [16]:
```
# Check if superset

set(album_set1).issuperset(album_set2)
```

Out[16]: False

In [17]:
```
# Check if subset

set(album_set2).issubset(album_set1)
```

Out[17]: False

Here is an example where `issubset()` and `issuperset()` return true:

In [18]:
```
# Check if subset

set({"Back in Black", "AC/DC"}).issubset(album_set1)
```

Out[18]: True

In [19]:
```
# Check if superset

album_set1.issuperset({"Back in Black", "AC/DC"})
```

Out[19]: True

Quiz on Sets

Convert the list `['rap','house','electronic music', 'rap']` to a set:

In [21]:
```
# Write your code below and press Shift+Enter to execute
list = ['rap','house','electronic music', 'rap']
set_list = set(list)
```

Double-click **here** for the solution.

Consider the list `A = [1, 2, 2, 1]` and set `B = set([1, 2, 2, 1])`, does `sum(A) = sum(B)`

In [23]:
```
# Write your code below and press Shift+Enter to execute
A = [1, 2, 2, 1]
B = set([1, 2, 2, 1])
print("the sum of A is:", sum(A))
print("the sum of B is:", sum(B))
```

```
the sum of A is: 6
the sum of B is: 3
```

Double-click **here** for the solution.

Create a new set `album_set3` that is the union of `album_set1` and `album_set2`:

In [25]:
```
# Write your code below and press Shift+Enter to execute

album_set1 = set(["Thriller", 'AC/DC', 'Back in Black'])
album_set2 = set([ "AC/DC", "Back in Black", "The Dark Side of the Moon"])
album_set3 = album_set1.union(album_set2)
```

Double-click **here** for the solution.

Find out if `album_set1` is a subset of `album_set3`:

In [26]:
```
# Write your code below and press Shift+Enter to execute
album_set1.issubset(album_set3)
```

Out[26]: True

Double-click **here** for the solution.

The last exercise!

Congratulations, you have completed your first lesson and hands-on lab in Python. However, there is one more thing you need to do. The Data Science community encourages sharing work. The best way to share and showcase your work is to share it on GitHub. By sharing your notebook on GitHub you are not only building your reputation with fellow data scientists, but you can also show it off when applying for a job. Even though this was your first piece of work, it is never too early to start building good habits. So, please read and follow this article to learn how to share your work.

Author

Joseph Santarcangelo

Other contributors

Mavis Zhou

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-08-26	2.0	Lavanya	Moved lab to course repo in GitLab

In [ ]:

Conditions in Python

Estaimted time needed: **20** minutes

Objectives

After compliting this lab you will be able to:

- work with condition statements in Python, including operators, and branching.

Table of Contents

- Condition Statements
  - Comparison Operators
  - Branching
  - Logical operators
- Quiz on Condition Statement

Estimated time needed: **20 min**

Condition Statements

Comparison Operators

Comparison operations compare some value or operand and, based on a condition, they produce a Boolean. When comparing two values you can use these operators:

- equal: **==**
- not equal: **!=**
- greater than: **>**
- less than: **<**
- greater than or equal to: **>=**
- less than or equal to: **<=**

Let's assign `a` a value of 5. Use the equality operator denoted with two equal **==** signs to determine if two values are equal. The case below compares the variable `a` with 6.

In [1]:
```
# Condition Equal

a = 5
a == 6
```

Out[1]:
```
False
```

The result is **False**, as 5 does not equal to 6.

Consider the following equality comparison operator `i > 5` . If the value of the left operand, in this case the variable **i**, is greater than the value of the right operand, in this case 5, then the statement is **True**. Otherwise, the statement is **False**. If **i** is equal to 6, because 6 is larger than 5, the output is **True**.

In [2]:
```
# Greater than Sign

i = 6
i > 5
```

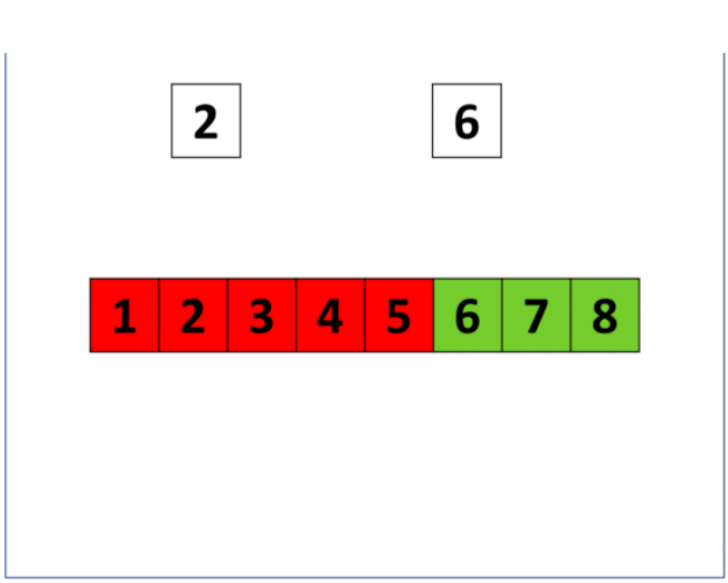
Out[2]:
```
True
```

Set `i = 2` . The statement is false as 2 is not greater than 5:

In [3]:
```
# Greater than Sign

i = 2
i > 5
```

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

Let's display some values for `i` in the figure. Set the values greater than 5 in green and the rest in red. The green region represents where the condition is **True**, the red where the statement is **False**. If the value of `i` is 2, we get **False** as the 2 falls in the red region. Similarly, if the value for `i` is 6 we get a **True** as the condition falls in the green region.



The inequality test uses an exclamation mark preceding the equal sign, if two operands are not equal then the condition becomes **True**. For example, the following condition will produce **True** as long as the value of `i` is not equal to 6:

In [4]:
```
# Inequality Sign

i = 2
i != 6
```

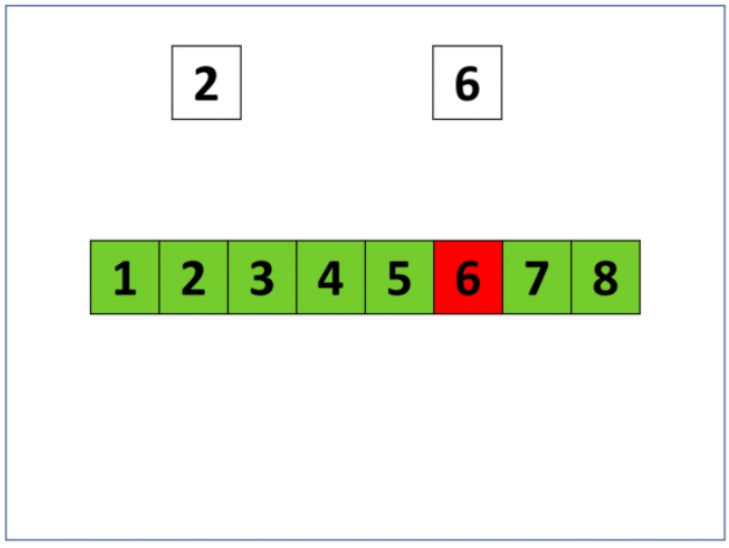
Out[4]: True

When `i` equals 6 the inequality expression produces **False**.

In [5]:
```
# Inequality Sign

i = 6
i != 6
```

Out[5]: False

See the number line below. when the condition is **True** the corresponding numbers are marked in green and for where the condition is **False** the corresponding number is marked in red. If we set `i` equal to 2 the operator is true as 2 is in the green region. If we set `i` equal to 6, we get a **False** as the condition falls in the red region.



We can apply the same methods on strings. For example, use an equality operator on two different strings. As the strings are not equal, we get a **False**.

In [6]:
```
# Use Equality sign to compare the strings

"ACDC" == "Michael Jackson"
```

Out[6]: False

If we use the inequality operator, the output is going to be **True** as the strings are not equal.

In [7]:
```
# Use Inequality sign to compare the strings

"ACDC" != "Michael Jackson"
```

Out[7]: True

Inequality operation is also used to compare the letters/words/symbols according to the ASCII value of letters. The decimal value shown in the following table represents the order of the character:

For example, the ASCII code for **!** is 21, while the ASCII code for **+** is 43. Therefore **+** is larger than **!** as 43 is greater than 21.

Similarly, the value for **A** is 101, and the value for **B** is 102 therefore:

In [8]:
```
# Compare characters

'B' > 'A'
```

Out[8]:
```
True
```

When there are multiple letters, the first letter takes precedence in ordering:

In [9]:
```
# Compare characters

'BA' > 'AB'
```

Out[9]:
```
True
```

Note: Upper Case Letters have different ASCII code than Lower Case Letters, which means the comparison between the letters in python is case-sensitive.

Branching

Branching allows us to run different statements for different inputs. It is helpful to think of an **if statement** as a locked room, if the statement is **True** we can enter the room and your program will run some predefined tasks, but if the statement is **False** the program will ignore the task.

For example, consider the blue rectangle representing an ACDC concert. If the individual is older than 18, they can enter the ACDC concert. If they are 18 or younger than 18 they cannot enter the concert.

Use the condition statements learned before as the conditions need to be checked in the **if statement**. The syntax is as simple as `if condition statement :`, which contains a word `if`, any condition statement, and a colon at the end. Start your tasks which need to be executed under this condition in a new line with an indent. The lines of code after the colon and with an indent will only be executed when the **if statement** is **True**. The tasks will end when the line of code does not contain the indent.

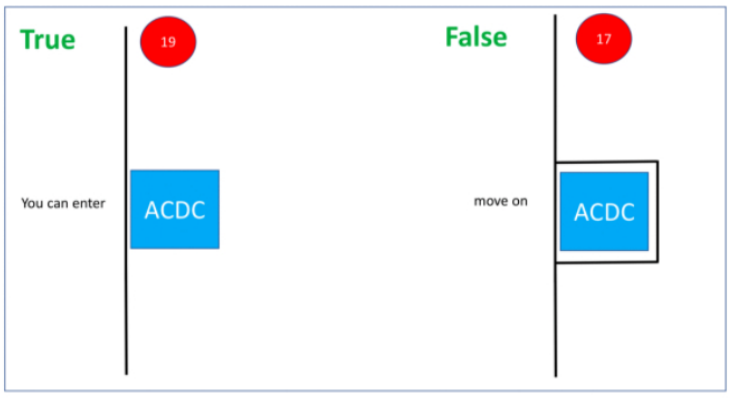
In the case below, the tasks executed `print("you can enter")` only occurs if the variable `age` is greater than 18 is a True case because this line of code has the indent. However, the execution of `print("move on")` will not be influenced by the if statement.

In [10]:
```
# If statement example

age = 19
#age = 18

#expression that can be true or false
if age > 18:

    #within an indent, we have the expression that is run if the condition is true
    print("you can enter" )

#The statements after the if statement will run regardless if the condition is true or false
print("move on")
```

```
you can enter
move on
```

Try uncommenting the age variable

It is helpful to use the following diagram to illustrate the process. On the left side, we see what happens when the condition is **True**. The person enters the ACDC concert representing the code in the indent being executed; they then move on. On the right side, we see what happens when the condition is **False**; the person is not granted access, and the person moves on. In this case, the segment of code in the indent does not run, but the rest of the statements are run.



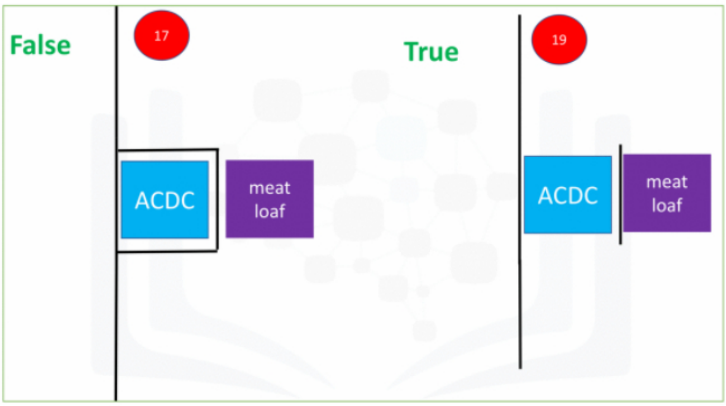
The `else` statement runs a block of code if none of the conditions are **True** before this `else` statement. Let's use the ACDC concert analogy again. If the user is 17 they cannot go to the ACDC concert, but they can go to the Meatloaf concert. The syntax of the `else` statement is similar as the syntax of the `if` statement, as `else :`. Notice that, there is no condition statement for `else`. Try changing the values of `age` to see what happens:

In [11]:
```
# Else statement example

age = 18
# age = 19

if age > 18:
    print("you can enter" )
```

```
else:
    print("go see Meat Loaf" )

print("move on")
```

```
go see Meat Loaf
move on
```

The process is demonstrated below, where each of the possibilities is illustrated on each side of the image. On the left is the case where the age is 17, we set the variable age to 17, and this corresponds to the individual attending the Meatloaf concert. The right portion shows what happens when the individual is over 18, in this case 19, and the individual is granted access to the concert.

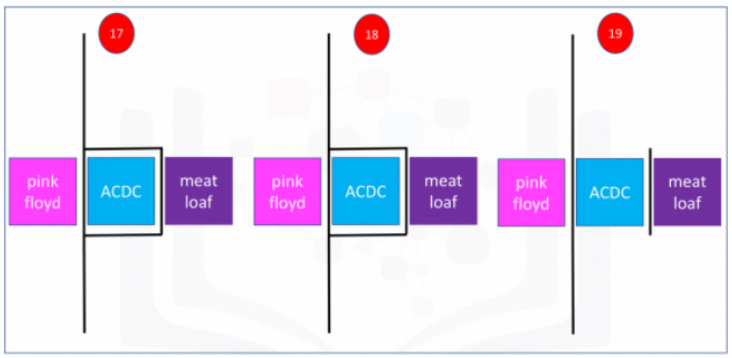


The `elif` statement, short for else if, allows us to check additional conditions if the condition statements before it are **False**. If the condition for the `elif` statement is **True**, the alternate expressions will be run. Consider the concert example, where if the individual is 18 they will go to the Pink Floyd concert instead of attending the ACDC or Meat-loaf concert. The person of 18 years of age enters the area, and as they are not older than 18 they can not see ACDC, but as they are 18 years of age, they attend Pink Floyd. After seeing Pink Floyd, they move on. The syntax of the `elif` statement is similar in that we merely change the `if` in `if` statement to `elif`.

In [12]:
```
# Elif statment example

age = 18

if age > 18:
    print("you can enter" )
elif age == 18:
    print("go see Pink Floyd")
else:
    print("go see Meat Loaf" )

print("move on")
```

```
go see Pink Floyd
move on
```

The three combinations are shown in the figure below. The left-most region shows what happens when the individual is less than 18 years of age. The central component shows when the individual is exactly 18. The rightmost shows when the individual is over 18.



Look at the following code:

In [13]:
```
# Condition statement example

album_year = 1983
album_year = 1970

if album_year > 1980:
    print("Album year is greater than 1980")

print('do something..')
```

```
do something..
```

Feel free to change `album_year` value to other values -- you'll see that the result changes!

Notice that the code in the above **indented** block will only be executed if the results are **True**.

As before, we can add an `else` block to the `if` block. The code in the `else` block will only be executed if the result is **False**.

Syntax:

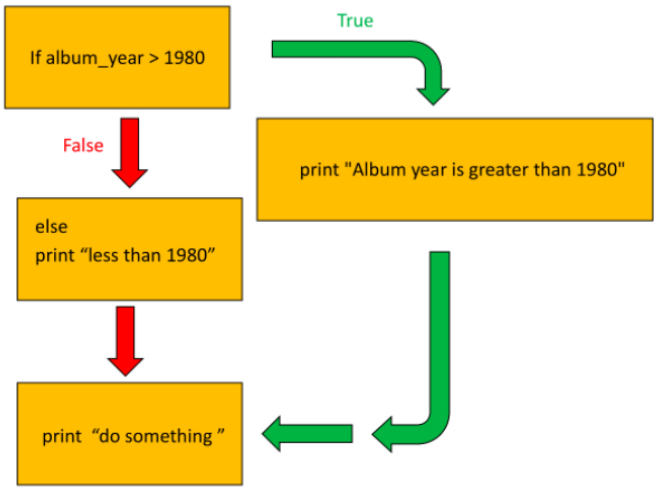
if (condition):

```
    # do something
```

else:

```
    # do something else
```

If the condition in the `if` statement is **False**, the statement after the `else` block will execute. This is demonstrated in the figure:



In [14]:
```
# Condition statement example

album_year = 1983
#album_year = 1970

if album_year > 1980:
    print("Album year is greater than 1980")
else:
    print("less than 1980")

print('do something..')
```

```
Album year is greater than 1980
do something..
```

Feel free to change the `album_year` value to other values -- you'll see that the result changes based on it!

Logical operators

Sometimes you want to check more than one condition at once. For example, you might want to check if one condition and another condition is **True**. Logical operators allow you to combine or modify conditions.

- `and`
- `or`
- `not`

These operators are summarized for two variables using the following truth tables:

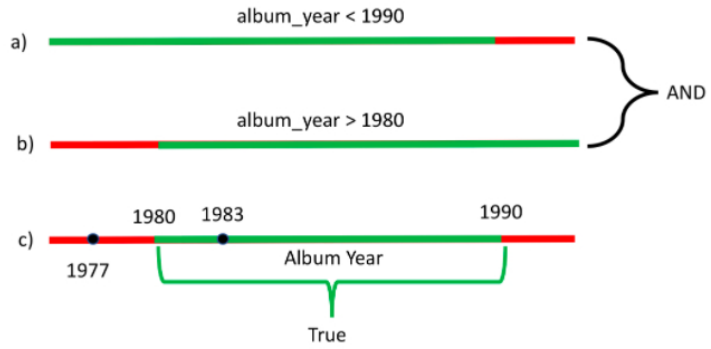
A	B	A & B
False	False	False
False	True	False
True	False	False
True	True	True

A	B	A or B
False	False	False
False	True	True
True	False	Ture
True	True	True

A	A!
False	True
True	False

The `and` statement is only **True** when both conditions are true. The `or` statement is true if one condition is **True**. The `not` statement outputs the opposite truth value.

Let's see how to determine if an album was released after 1979 (1979 is not included) and before 1990 (1990 is not included). The time periods between 1980 and 1989 satisfy this condition. This is demonstrated in the figure below. The green on lines **a** and **b** represents periods where the statement is **True**. The green on line **c** represents where both conditions are **True**, this corresponds to where the green regions overlap.



The block of code to perform this check is given by:

In [15]:
```
# Condition statement example

album_year = 1980

if(album_year > 1979) and (album_year < 1990):
    print ("Album year was in between 1980 and 1989")

print("")
print("Do Stuff..")
```

```
Album year was in between 1980 and 1989

Do Stuff..
```

To determine if an album was released before 1980 (~ - 1979) or after 1989 (1990 - ~) satisfy this condition. This is demonstrated in the following figure, the color green in **a** and **b** represents periods where the statement is true. The color green in **c** represents where at least one of the conditions are true.

The block of code to perform this check is given by:

In [16]:
```
# Condition statement example

album_year = 1990

if(album_year < 1980) or (album_year > 1989):
    print ("Album was not made in the 1980's")
else:
    print("The Album was made in the 1980's ")
```

```
Album was not made in the 1980's
```

The `not` statement checks if the statement is false:

In [17]:
```
# Condition statement example

album_year = 1983

if not (album_year == '1984'):
    print ("Album year is not 1984")
```

```
Album year is not 1984
```

```
Album year is not 1984
```

Quiz on Conditions

Write an if statement to determine if an album had a rating greater than 8. Test it using the rating for the album **"Back in Black"** that had a rating of 8.5. If the statement is true print "This album is Amazing!"

In [3]:
```
# Write your code below and press Shift+Enter to execute
rating = 8.5
if rating > 8:
    print("This album is Amazing!")
```

```
This album is Amazing!
```

Double-click **here** for the solution.

Write an if-else statement that performs the following. If the rating is larger then eight print "this album is amazing". If the rating is less than or equal to 8 print "this album is ok".

In [4]:
```
# Write your code below and press Shift+Enter to execute
rating = 8.5
if rating > 8:
    print ("this album is amazing")
else:
    print ("this album is ok")
```

```
this album is amazing
```

Double-click **here** for the solution.

Write an if statement to determine if an album came out before 1980 or in the years: 1991 or 1993. If the condition is true print out the year the album came out.

In [7]:
```
# Write your code below and press Shift+Enter to execute
album_year = 1979

if album_year < 1980 or album_year == 1991 or album_year == 1993:
    print ("This album came out in year %d" %album_year)
```

```
This album came out in year 1979
```

Double-click **here** for the solution.

The last exercise!

Congratulations, you have completed your first lesson and hands-on lab in Python. However, there is one more thing you need to do. The Data Science community encourages sharing work. The best way to share and showcase your work is to share it on GitHub. By sharing your notebook on GitHub you are not only building your reputation with fellow data scientists, but you can also show it off when applying for a job. Even though this was your first piece of work, it is never too early to start building good habits. So, please read and follow this article to learn how to share your work.

Author

Joseph Santarcangelo

Other contributors

Mavis Zhou

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-08-26	2.0	Lavanya	Moved lab to course repo in GitLab

Loops in Python

Estimated time needed: **20** minutes

Objectives

After completing this lab you will be able to:

- work with the loop statements in Python, including for-loop and while-loop.

Loops in Python

Welcome! This notebook will teach you about the loops in the Python Programming Language. By the end of this lab, you'll know how to use the loop statements in Python, including for loop, and while loop.

Table of Contents

- Loops
  - Range
  - What is `for` loop?
  - What is `while` loop?
- Quiz on Loops

Estimated time needed: **20 min**

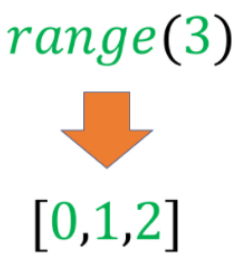
Loops

Range

Sometimes, you might want to repeat a given operation many times. Repeated executions like this are performed by **loops**. We will look at two types of loops, `for` loops and `while` loops.

Before we discuss loops lets discuss the `range` object. It is helpful to think of the range object as an ordered list. For now, let's look at the simplest case. If we would like to generate a sequence that contains three elements ordered from 0 to 2 we simply use the following command:

In [ ]:
```
# Use the range

range(3)
```



What is `for` loop?

The `for` loop enables you to execute a code block multiple times. For example, you would use this if you would like to print out every element in a list. Let's try to use a `for` loop to print all the years presented in the list `dates`:

This can be done as follows:

In [1]:
```
# For loop example

dates = [1982,1980,1973]
N = len(dates)

for i in range(N):
```

```
    print(dates[i])
```

```
1982
1980
1973
```

The code in the indent is executed `N` times, each time the value of `i` is increased by 1 for every execution. The statement executed is to `print` out the value in the list at index `i` as shown here:

In this example we can print out a sequence of numbers from 0 to 7:

In [2]:
```
# Example of for loop

for i in range(0, 8):
    print(i)
```

```
0
1
2
3
4
5
6
7
```

In Python we can directly access the elements in the list as follows:

In [3]:
```
# Exmaple of for loop, loop through list

for year in dates:
    print(year)
```

```
1982
1980
1973
```

For each iteration, the value of the variable `years` behaves like the value of `dates[i]` in the first example:

We can change the elements in a list:

In [4]:
```
# Use for loop to change the elements in list

squares = ['red', 'yellow', 'green', 'purple', 'blue']

for i in range(0, 5):
    print("Before square ", i, 'is', squares[i])
```

```
    print("Before square ", i, 'is ', squares[i])
    squares[i] = 'weight'
    print("After square ", i, 'is',  squares[i])
```

```
Before square  0 is red
After square  0 is weight
Before square  1 is yellow
After square  1 is weight
Before square  2 is green
After square  2 is weight
Before square  3 is purple
After square  3 is weight
Before square  4 is blue
After square  4 is weight
```

We can access the index and the elements of a list as follows:

In [ ]:
```
# Loop through the list and iterate on both index and element value

squares=['red', 'yellow', 'green', 'purple', 'blue']

for i, square in enumerate(squares):
    print(i, square)
```

What is `while` loop?

As you can see, the `for` loop is used for a controlled flow of repetition. However, what if we don't know when we want to stop the loop? What if we want to keep executing a code block until a certain condition is met? The `while` loop exists as a tool for repeated execution based on a condition. The code block will keep being executed until the given logical condition returns a **False** boolean value.

Let's say we would like to iterate through list `dates` and stop at the year 1973, then print out the number of iterations. This can be done with the following block of code:

In [ ]:
```
# While Loop Example

dates = [1982, 1980, 1973, 2000]

i = 0
year = 0

while(year != 1973):
    year = dates[i]
    i = i + 1
    print(year)

print("It took ", i ,"repetitions to get out of loop.")
```

A while loop iterates merely until the condition in the argument is not met, as shown in the following figure:

Quiz on Loops

Write a `for` loop the prints out all the element between **-5** and **5** using the range function.

In [7]:
```
# Write your code below and press Shift+Enter to execute
for i in range(-5,6):
    print(i)
```

```
-5
-4
-3
-2
-1
0
1
2
3
4
5
```

Double-click **here** for the solution.

Print the elements of the following list: `Genres=[ 'rock', 'R&B', 'Soundtrack', 'R&B', 'soul', 'pop']` Make sure you follow Python conventions.

In [9]:
```
# Write your code below and press Shift+Enter to execute
Genres = ['rock', 'R&B', 'Soundtrack', 'R&B', 'soul', 'pop']
for genre in Genres:
    print(genre)
```

```
rock
R&B
Soundtrack
R&B
soul
pop
```

Double-click **here** for the solution.

Write a for loop that prints out the following list: `squares=['red', 'yellow', 'green', 'purple', 'blue']`

In [10]:
```
# Write your code below and press Shift+Enter to execute
squares=['red', 'yellow', 'green', 'purple', 'blue']

for square in squares:
    print(square)
```

```
red
yellow
green
purple
blue
```

Double-click **here** for the solution.

Write a while loop to display the values of the Rating of an album playlist stored in the list `PlayListRatings`. If the score is less than 6, exit the loop. The list `PlayListRatings` is given by: `PlayListRatings = [10, 9.5, 10, 8, 7.5, 5, 10, 10]`

In [14]:
```
# Write your code below and press Shift+Enter to execute
PlayListRatings = [10, 9.5, 10, 8, 7.5, 5, 10, 10]
i = 1
Ratings = PlayListRatings[0]
while(Ratings >= 6):
    print(Ratings)
    Ratings = PlayListRatings[i]
    i = i+1
```

```
10
9.5
10
8
7.5
```

Double-click **here** for the solution.

Write a while loop to copy the strings `'orange'` of the list `squares` to the list `new_squares`. Stop and exit the loop if the value on the list is not `'orange'`:

In [18]:
```
# Write your code below and press Shift+Enter to execute

squares = ['orange', 'orange', 'purple', 'blue ', 'orange']
new_squares = []
i = 0
while(squares[i] == 'orange'):
    new_squares.append(squares[i])
    i = i+1

print(new_squares)
```

```
['orange', 'orange']
```

Double-click **here** for the solution.

The last exercise!

Congratulations, you have completed your first lesson and hands-on lab in Python. However, there is one more thing you need to do. The Data Science community encourages sharing work. The best way to share and showcase your work is to share it on GitHub. By sharing your notebook on GitHub you are not only building your reputation with fellow data scientists, but you can also show it off when applying for a job. Even though this was your first piece of work, it is never too early to start building good habits. So, please read and follow this article to learn how to share your work.

Author

Joseph Santarcangelo

Other contributors

Mavis Zhou

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-08-26	2.0	Lavanya	Moved lab to course repo in GitLab

Functions in Python

Estimated time needed: **40** minutes

Objectives

After completing this lab you will be able to:

- Understand functions and variables
- Work with functions and variables

Functions in Python

Welcome! This notebook will teach you about the functions in the Python Programming Language. By the end of this lab, you'll know the basic concepts about function, variables, and how to use functions.

Table of Contents

- Functions
  - What is a function?
  - Variables
  - Functions Make Things Simple
- Pre-defined functions
- Using `if` / `else` Statements and Loops in Functions
- Setting default argument values in your custom functions
- Global variables
- Scope of a Variable
- Collections and Functions
- Quiz on Loops

Estimated time needed: **40 min**

Functions

A function is a reusable block of code which performs operations specified in the function. They let you break down tasks and allow you to reuse your code in different programs.

There are two types of functions :

- **Pre-defined functions**
- **User defined functions**

What is a Function?

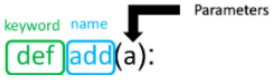
You can define functions to provide the required functionality. Here are simple rules to define a function in Python:

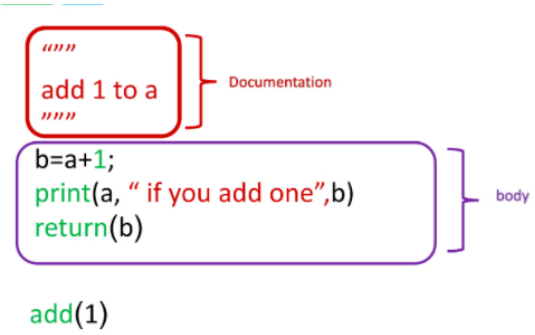
- Functions blocks begin `def` followed by the function `name` and parentheses `()`.
- There are input parameters or arguments that should be placed within these parentheses.
- You can also define parameters inside these parentheses.
- There is a body within every function that starts with a colon (`:`) and is indented.
- You can also place documentation before the body.
- The statement `return` exits a function, optionally passing back a value.

An example of a function that adds on to the parameter `a` prints and returns the output as `b`:

In [ ]:
```
# First function example: Add 1 to a and store as b

def add(a):
    b = a + 1
    print(a, "if you add one", b)
    return(b)
```

The figure below illustrates the terminology:





We can obtain help about a function :

In [ ]:
```
# Get a help on add function

help(add)
```

We can call the function:

In [ ]:
```
# Call the function add()

add(1)
```

If we call the function with a new input we get a new result:

In [ ]:
```
# Call the function add()

add(2)
```

We can create different functions. For example, we can create a function that multiplies two numbers. The numbers will be represented by the variables `a` and `b`:

In [2]:
```
# Define a function for multiple two numbers

def Mult(a, b):
    c = a * b
    return(c)
    print('This is not printed')

result = Mult(12,2)
print(result)
```

```
24
```

The same function can be used for different data types. For example, we can multiply two integers:

In [ ]:
```
# Use mult() multiply two integers

Mult(2, 3)
```

Note how the function terminates at the `return` statement, while passing back a value. This value can be further assigned to a different variable as desired.

The same function can be used for different data types. For example, we can multiply two integers:

Two Floats:

In [ ]:
```
# Use mult() multiply two floats

Mult(10.0, 3.14)
```

We can even replicate a string by multiplying with an integer:

In [3]:
```
# Use mult() multiply two different type values together

Mult(2, "Michael Jackson ")
```

Out[3]:
```
'Michael Jackson Michael Jackson '
```

Variables

The input to a function is called a formal parameter.

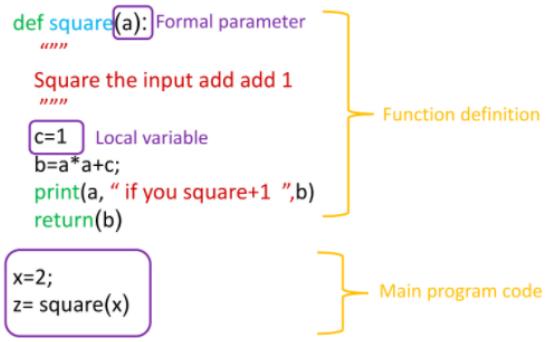
A variable that is declared inside a function is called a local variable. The parameter only exists within the function (i.e. the point where the function starts and stops).

A variable that is declared outside a function definition is a global variable, and its value is accessible and modifiable throughout the program. We will discuss more about global variables at the end of the lab.

In [ ]:
```
# Function Definition

def square(a):
```

```
    # Local variable b
    b = 1
    c = a * a + b
    print(a, "if you square + 1", c)
    return(c)
```

The labels are displayed in the figure:



We can call the function with an input of **3**:

In [ ]:
```
# Initializes Global variable

x = 3
# Makes function call and return function a y
y = square(x)
y
```

We can call the function with an input of **2** in a different manner:

In [ ]:
```
# Directly enter a number as parameter

square(2)
```

If there is no `return` statement, the function returns `None`. The following two functions are equivalent:

In [4]:
```
# Define functions, one with return value None and other without return value

def MJ():
    print('Michael Jackson')

def MJ1():
    print('Michael Jackson')
    return(None)
```

In [5]:
```
# See the output

MJ()
```

```
Michael Jackson
```

In [6]:
```
# See the output

MJ1()
```

```
Michael Jackson
```

Printing the function after a call reveals a **None** is the default return statement:

In [7]:
```
# See what functions returns are

print(MJ())
print(MJ1())
```

```
Michael Jackson
None
Michael Jackson
None
```

Create a function `con` that concatenates two strings using the addition operation:

In [8]:
```
# Define the function for combining strings

def con(a, b):
    return(a + b)
```

In [9]:
```
# Test on the con() function

con("This ", "is")
```

Out[9]:
```
'This is'
```

[Tip] How do I learn more about the pre-defined functions in Python?

We will be introducing a variety of pre-defined functions to you as you learn more about Python. There are just too many functions, so there's no way we

can teach them all in one sitting. But if you'd like to take a quick peek, here's a short reference card for some of the commonly-used pre-defined functions: Reference

Functions Make Things Simple

Consider the two lines of code in **Block 1** and **Block 2**: the procedure for each block is identical. The only thing that is different is the variable names and values.

Block 1:

In [ ]:
```
# a and b calculation block1

a1 = 4
b1 = 5
c1 = a1 + b1 + 2 * a1 * b1 - 1
if(c1 < 0):
    c1 = 0
else:
    c1 = 5
c1
```

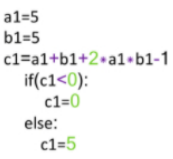
Block 2:

In [ ]:
```
# a and b calculation block2

a2 = 0
b2 = 0
c2 = a2 + b2 + 2 * a2 * b2 - 1
if(c2 < 0):
    c2 = 0
else:
    c2 = 5
c2
```

We can replace the lines of code with a function. A function combines many instructions into a single line of code. Once a function is defined, it can be used repeatedly. You can invoke the same function many times in your program. You can save your function and use it in another program or use someone else's function. The lines of code in code **Block 1** and code **Block 2** can be replaced by the following function:

In [ ]:
```
# Make a Function for the calculation above

def Equation(a,b):
    c = a + b + 2 * a * b - 1
    if(c < 0):
        c = 0
    else:
        c = 5
    return(c)
```

This function takes two inputs, a and b, then applies several operations to return c. We simply define the function, replace the instructions with the function, and input the new values of `a1`, `b1` and `a2`, `b2` as inputs. The entire process is demonstrated in the figure:



Code **Blocks 1** and **Block 2** can now be replaced with code **Block 3** and code **Block 4**.

Block 3:

In [ ]:
```
a1 = 4
b1 = 5
c1 = Equation(a1, b1)
c1
```

Block 4:

In [ ]:
```
a2 = 0
b2 = 0
c2 = Equation(a2, b2)
c2
```

Pre-defined functions

There are many pre-defined functions in Python, so let's start with the simple ones.

The `print()` function:

In [ ]:
```
# Build-in function print()

album_ratings = [10.0, 8.5, 9.5, 7.0, 7.0, 9.5, 9.0, 9.5]
print(album_ratings)
```

The `sum()` function adds all the elements in a list or tuple:

In [ ]:
```
# Use sum() to add every element in a list or tuple together

sum(album_ratings)
```

The `len()` function returns the length of a list or tuple:

In [ ]:
```
# Show the length of the list or tuple

len(album_ratings)
```

Using `if`/`else` Statements and Loops in Functions

The `return()` function is particularly useful if you have any IF statements in the function, when you want your output to be dependent on some condition:

In [ ]:
```
# Function example

def type_of_album(artist, album, year_released):

    print(artist, album, year_released)
    if year_released > 1980:
        return "Modern"
    else:
        return "Oldie"

x = type_of_album("Michael Jackson", "Thriller", 1980)
print(x)
```

We can use a loop in a function. For example, we can `print` out each element in a list:

In [ ]:
```
# Print the list using for loop

def PrintList(the_list):
    for element in the_list:
        print(element)
```

In [ ]:
```
# Implement the printlist function

PrintList(['1', 1, 'the man', "abc"])
```

Setting default argument values in your custom functions

You can set a default value for arguments in your function. For example, in the `isGoodRating()` function, what if we wanted to create a threshold for what we consider to be a good rating? Perhaps by default, we should have a default rating of 4:

In [ ]:
```
# Example for setting param with default value

def isGoodRating(rating=4):
    if(rating < 7):
        print("this album sucks it's rating is",rating)

    else:
        print("this album is good its rating is",rating)
```

In [ ]:
```
# Test the value with default value and with input

isGoodRating()
isGoodRating(10)
```

Global variables

So far, we've been creating variables within functions, but we have not discussed variables outside the function. These are called global variables. Let's try to see what `printer1` returns:

In [5]:
```
# Example of global variable

artist = "Michael Jackson"
def printer1(artist):
    internal_var1 = artist
    print(artist, "is an artist")

printer1(artist)
# try runningthe following code
#printer1(internal_var1)
```

```
Michael Jackson is an artist
```

We got a Name Error: `name 'internal_var' is not defined`. Why?

It's because all the variables we create in the function is a **local variable**, meaning that the variable assignment does not persist outside the function.

But there is a way to create **global variables** from within a function as follows:

In [ ]:
```
artist = "Michael Jackson"

def printer(artist):
    global internal_var
    internal_var= "Whitney Houston"
    print(artist,"is an artist")

printer(artist)
printer(internal_var)
```

Scope of a Variable

The scope of a variable is the part of that program where that variable is accessible. Variables that are declared outside of all function definitions, such as the `myFavouriteBand` variable in the code shown here, are accessible from anywhere within the program. As a result, such variables are said to have global scope, and are known as global variables. `myFavouriteBand` is a global variable, so it is accessible from within the `getBandRating` function, and we can use it to determine a band's rating. We can also use it outside of the function, such as when we pass it to the print function to display it:

In [ ]:
```
# Example of global variable

myFavouriteBand = "AC/DC"

def getBandRating(bandname):
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is:", getBandRating("AC/DC"))
print("Deep Purple's rating is:",getBandRating("Deep Purple"))
print("My favourite band is:", myFavouriteBand)
```

Take a look at this modified version of our code. Now the `myFavouriteBand` variable is defined within the `getBandRating` function. A variable that is defined within a function is said to be a local variable of that function. That means that it is only accessible from within the function in which it is defined. Our `getBandRating` function will still work, because `myFavouriteBand` is still defined within the function. However, we can no longer print `myFavouriteBand` outside our function, because it is a local variable of our `getBandRating` function; it is only defined within the `getBandRating` function:

In [ ]:
```
# Example of local variable

def getBandRating(bandname):
    myFavouriteBand = "AC/DC"
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is: ", getBandRating("AC/DC"))
print("Deep Purple's rating is: ", getBandRating("Deep Purple"))
print("My favourite band is", myFavouriteBand)
```

Finally, take a look at this example. We now have two `myFavouriteBand` variable definitions. The first one of these has a global scope, and the second of them is a local variable within the `getBandRating` function. Within the `getBandRating` function, the local variable takes precedence. **Deep Purple** will receive a rating of 10.0 when passed to the `getBandRating` function. However, outside of the `getBandRating` function, the `getBandRating` s local variable is not defined, so the `myFavouriteBand` variable we print is the global variable, which has a value of **AC/DC**:

In [ ]:
```
# Example of global variable and local variable with the same name

myFavouriteBand = "AC/DC"

def getBandRating(bandname):
    myFavouriteBand = "Deep Purple"
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is:",getBandRating("AC/DC"))
print("Deep Purple's rating is: ",getBandRating("Deep Purple"))
print("My favourite band is:",myFavouriteBand)
```

Collections and Functions

When the number of arguments are unknown for a function, They can all be packed into a tuple as shown:

In [ ]:
```
def printAll(*args): # All the arguments are 'packed' into args which can be treated like a tuple
    print("No of arguments:", len(args))
    for argument in args:
        print(argument)
#printAll with 3 arguments
printAll('Horsefeather','Adonis','Bone')
#printAll with 4 arguments
printAll('Sidecar','Long Island','Mudslide','Carriage')
```

Similarly, The arguments can also be packed into a dictionary as shown:

In [10]:
```
def printDictionary(**args):
    for key in args:
        print(key + " : " + args[key])

printDictionary(Country='Canada',Province='Ontario',City='Toronto')
```

```
Country : Canada
Province : Ontario
City : Toronto
```

Functions can be incredibly powerful and versatile. They can accept (and return) data types, objects and even other functions as arguements. Consider the example below:

In [8]:
```
def addItems(list):
    list.append("Three")
    list.append("Four")

myList = ["One","Two"]

addItems(myList)

myList
```

Out[8]:
```
['One', 'Two', 'Three', 'Four']
```

Note how the changes made to the list are not limited to the functions scope. This occurs as it is the lists **reference** that is passed to the function - Any changes made are on the orignal instance of the list. Therefore, one should be cautious when passing mutable objects into functions.

Quiz on Functions

Come up with a function that divides the first input by the second input:

In [11]:
```
# Write your code below and press Shift+Enter to execute
def div(a,b):
    return(a/b)
```

Double-click **here** for the solution.

Use the function `con` for the following question.

In [12]:
```
# Use the con function for the following question

def con(a, b):
    return(a + b)
```

Can the `con` function we defined before be used to add to integers or strings?

In [ ]:
```
# Write your code below and press Shift+Enter to execute
#Yes, Function can take any values
```

Double-click **here** for the solution.

Can the `con` function we defined before be used to concatenate a list or tuple?

In [ ]:
```
# Write your code below and press Shift+Enter to execute
Yes
```

Double-click **here** for the solution.

Probability Bag

You have been tasked with creating a lab that demonstrates the basics of probability by simulating a bag filled with colored balls. The bag is represented using a dictionary called "bag", where the key represents the color of the ball and the value represents the no of balls. The skeleton code has been made for you, do not add or remove any functions. Complete the following functions -

- fillBag - A function that packs it's arguments into a global dictionary "bag".
- totalBalls - returns the total no of balls in the bucket
- probOf - takes a color (string) as argument and returns probability of drawing the selected ball. Assume total balls are not zero and the color given is a valid key.
- probAll - returns a dictionary of all colors and their corresponding probability

In [13]:
```
def fillBag(**balls):
    pass


def totalBalls():
    pass

def probOf(color):
    pass

def probAll():
    pass
```

Run this snippet of code to test your solution. Note: This is not a comprehensive test.

In [14]:
```
testBag = dict(red = 12, blue = 20, green = 14, grey = 10)
total =  sum(testBag.values())
prob={}
for color in testBag:
    prob[color] = testBag[color]/total;

def testMsg(passed):
    if passed:
       return 'Test Passed'
    else :
       return ' Test Failed'

print("fillBag : ")
try:
    fillBag(**testBag)
    print(testMsg(bag == testBag))
except NameError as e:
    print('Error! Code: {c}, Message: {m}'.format(c = type(e).__name__, m = str(e)))
except:
    print("An error occured. Recheck your function")



print("totalBalls : ")
try:
    print(testMsg(total == totalBalls()))
except NameError as e:
    print('Error! Code: {c}, Message: {m}'.format(c = type(e).__name__, m = str(e)))
except:
    print("An error occured. Recheck your function")

print("probOf")
try:
    passed = True
    for color in testBag:
           if probOf(color) != prob[color]:
                passed = False

    print(testMsg(passed) )
except NameError as e:
    print('Error! Code: {c}, Message: {m}'.format(c = type(e).__name__, m = str(e)))
except:
    print("An error occured. Recheck your function")

print("probAll")
try:
    print(testMsg(probAll() == prob))
except NameError as e:
    print('Error! Code: {c}, Message: {m}'.format(c = type(e).__name__, m = str(e)))
except:
    print("An error occured. Recheck your function")
```

```
fillBag :
Error! Code: NameError, Message: name 'bag' is not defined
totalBalls :
 Test Failed
probOf
 Test Failed
probAll
 Test Failed
```

Double-click **here** for the solution.

The last exercise!

Congratulations, you have completed your first lesson and hands-on lab in Python. However, there is one more thing you need to do. The Data Science community encourages sharing work. The best way to share and showcase your work is to share it on GitHub. By sharing your notebook on GitHub you are not only building your reputation with fellow data scientists, but you can also show it off when applying for a job. Even though this was your first piece of work,

it is never too early to start building good habits. So, please read and follow this article to learn how to share your work.

Author

Joseph Santarcangelo

Other contributors

Mavis Zhou

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-08-26	0.2	Lavanya	Moved lab to course repo in GitLab
2020 -09 -04	0.2	Arjun	Under What is a function, added code/text to further demonstrate the functionality of the return statement
2020 -09 -04	0.2	Arjun	Under Global Variables, modify the code block to try and print 'internal_var' - So a nameError message can be observed
2020 -09 -04	0.2	Arjun	Added section Collections and Functions
2020 -09 -04	0.2	Arjun	Added exercise "Probability Bag"