



AI FOUNDATION

DEDICATED TO MY BELOVED PARENTS



Pt. Mahavir Prasad Kaushik
{ Knowledge is that
which liberates || }

Smt. Sunita Kaushik
{ विद्या ददाति विनयम }



AUTHORS

Trilok Kaushik
{ सा विद्या या विमुक्तये । }



Aditya Aggarwal
{ One illuminated mind
can ignite many others }



Khushi Jhavar
{ Programming is
thinking not typing }
SHE_CODE



DISCLAIMER :-

- Geeksman owns all the right's of this content.
- Any distribution of the content without permission is illegal.
- Although great care has been taken in designing the content but still there is a scope of improvement so we welcome all the suggestions regarding the content.

INDEX

- Lesson 1 :-
<https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%201%20AI%20Foundation.ipynb>
- Lesson 2 :-
<https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%202%20AI%20Foundation.ipynb>
- Lesson 3 :-
<https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%204%20AI%20Foundation.ipynb>
- Lesson 4:-
<https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%203%20AI%20Foundation.ipynb>
- Lesson 5 :-
Numpy:-
[https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%205%20\(%20AI%20Foundation%20Numpy\).ipynb](https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%205%20(%20AI%20Foundation%20Numpy).ipynb)

Pandas:-

[https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%205%20\(AI%20Foundation\).ipynb#scrollTo=RgSWR1NQI58K](https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%205%20(AI%20Foundation).ipynb#scrollTo=RgSWR1NQI58K)

- Lesson 6 :-

[https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%206%20\(AI%20Foundation\).ipynb](https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%206%20(AI%20Foundation).ipynb)

- Lesson 7 :-

[https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%207%20\(AI%20Foundation\).ipyn](https://colab.research.google.com/github/pttrilok/courses/blob/master/AI%20Foundation/Lesson%207%20(AI%20Foundation).ipyn)

INDEX

1. Lesson One

- 1.1. An Informal Introduction to Python
 - 1.1.1. Comments
 - 1.1.2. Introduction to Python Indentation
 - 1.1.3. Some Shortcuts
- 1.2. Using Python as a Calculator
 - 1.2.1. Numbers
- 1.3. Strings
- 1.4. Lists

2. Lesson Two

- 2.1. If Statement
- 2.2. Loops
 - 2.2.1. While Loop
 - 2.2.2. For Loop
- 2.3. Range() Function
- 2.4. Nested Loops
 - 2.4.1. Patterns using nested loops
- 2.5. Break, Continue and Pass Statement

3. Lesson Three

- 3.1. Del Statement
- 3.2. Sets
- 3.3. Dictionary
- 3.4. Looping Techniques
- 3.5. Bold Text

4. Lesson Four

- 4.1. Functions
- 4.2. Arguments
 - 4.2.1. Default Arguments
 - 4.2.2. Number of Arguments
 - 4.2.3. Arbitrary Arguments
 - 4.2.4. Passing List as Argument

- 4.3. Keyword Arguments
 - 4.3.1. Arbitrary Keyword Arguments
- 4.4. Return Value
- 4.5. Pass Statement
- 4.6. Recursion
- 4.7. Lambda Expression
 - 4.7.1. Filter()
 - 4.7.2. Map()

5. Lesson Five

- 5.1. NumPy
 - 5.1.1. NumPy Arrays
- 5.2. Pandas
 - 5.2.1. Series
 - 5.2.2. DataFrames

6. Lesson Six

- 6.1. OS Library
- 6.2. Shutil Library
- 6.3. %%time
- 6.4. Zip and Unzip Folders
- 6.5. %cd Command
- 6.6. Deleting Files and Folders

7. Lesson Seven

- 7.1. Googletrans Library
- 7.2. GTTS Library
- 7.3. TextBlob Library

GOOGLE COLAB

Colab is a free notebook environment that runs in the browser entirely using the google cloud. It provides free CPUs, GPUs and TPUs on your fingertips and it does not depend on your computer configuration. All you need is a google account and a web browser. It allows you and your team members to edit your notebook, the way you work with Google Docs. Colab supports many popular machine learning libraries which can easily be loaded in your notebook.

To start working on your colab notebook :

Firstly you need to connect your notebook to the runtime



In the above snapshot, in the rightmost side there is an option “Connect” so click on it and when it shows information about RAM and DISK then your notebook is successfully connected.

Hurray !! You are ready to work on the colab notebook.



NOTE :- If the runtime gets disconnected then all the variables will be lost and hence you will have to run each cell again to load the data.

LESSON ONE

- **An Informal Introduction to Python**

In the Python Interactive Mode, the lines following the prompts (i.e. `>>>` or `...`) are the input and the lines that do not begin with a prompt are output. The secondary prompt (`...`) is used for multi-line input and you must type a blank line to end the multi-line command.

```
>>> if(1):  
...     print("Hello") # This is a multi-line command  
...                     # To end multi-line command type a blank line  
...  
Hello
```

1) Comments

How to create comments ?

- (i) Comments in Python start with the hash character `#` and extend to the end of the physical line.
- (ii) A comment may appear at the start of a line or following whitespace or code, but not within a string literal.
- (iii) A hash character within a string literal is just a hash character.
- (iv) To add multi-line comment just put `#` before each line.

```
▶ #comment 1  
number = 67 #comment 2 following a code  
           #comment 3 following whitespaces  
text = "#This is not a comment as it's inside quotes"
```

Why do we need comments ?

- (i) Comments can be used to make the code more readable.
- (ii) Comments can be used to prevent execution when testing code.

```
[2] # This
    # is
    # a
    # multi_line
    # comment
```

2) Introduction to Python Indentation

Indentation refers to spaces at the beginning of a line. In other programming languages like C or C++, indentation is used to improve readability but in Python indentation is very important, it indicates a block of code.

```
[5] if (1):
    print("Hello")    #indented line indicates the block of "if statement"
```

Hello

Python gives error if we miss the indentation.

```
[ ] #python gives error if we miss the indentation

if(1):
print("hello")    #indentation is missing
```

File "<ipython-input-2-82a32e808c93>", line 4
print("hello") #indentation is missing
^
IndentationError: expected an indented block

3) Some Shortcuts

- *Tab* - For indentation
- *Shift + Tab* - To delete indentation
- *Ctrl + Shift + L* - To edit a name at all places in a cell

● Using Python as a Calculator

1) Numbers

I. Python supports :-

- Integer numbers (5, 0, -10) having `int` type
- Numbers with fractional parts (3.4, 24.0, 5.25) having `float` type
- Numbers with real and imaginary parts (8+3j , 7-4.5j) having `complex` type (the real and imaginary part of complex variables are floating point numbers). The imaginary part is suffixed by `j` or `J` .

In addition to these data types, Python also supports `Decimal` and `Fraction` .

```
[ ] x = 45    # int type
    type(x)  # The type function returns the datatype of the variable

int

[ ] y = -37.25 # float type
    type(y)

float

[ ] z = 3+5j   # complex type
    type(z)

complex
```

- ## II.
- In a Python interpreter , the operators like `+`, `-`, `*` and `/` can be used for general calculations and parentheses `(())` for grouping

```
[ ] 5+5

10

[ ] 50 - 20 * 3

-10

[ ] 70 * (138 - 133)

350

[ ] 359/5    #Division always returns floating point number

71.8
```

- III. The value that the operator operates on is called the **operand**. Python supports mixed operand type operation. Operators with mixed type operands convert the integer operand to floating point.

```
(3*2.5)+6 # 3 and 6 are integers and 2.5 is floating so the answer is a floating point number
13.5
```

- IV. Division `/` always returns a float. To calculate an integer result perform floor division using `//` *floor operator* and to calculate remainder use `%` *modulus operator*.

```
47 / 8 #Division always returns floating point number
5.875

[ ] 47 // 8 #Floor division discards the fractional part
5

[ ] 47 % 8 # % operator returns the remainder
7
```

- V. To calculate power, use the `**` *power operator* . It returns the value of the first number raised to the power of second.

```
[ ] 5**4 # 5^4 or 5 raised to the power 4
49

[ ] 2**6 # 2 raised to the power 6
64
```

- VI. *Assignment operator* `=` is used to assign value to a variable. If we call the variable then it returns the assigned value.

```
[ ] altitude = 9 # 9 is assigned to altitude variable

[ ] base = 10 # 10 is assigned to base variable

[ ] (base*altitude)//2 #the variables are used to perform calculations
45
```

- VII. When trying to use a variable that is not defined or is not assigned a value, you receive an error (**NameError**)

```
[ ] s

-----
NameError                                Traceback (most recent call last)
<ipython-input-1-ded5ba42480f> in <module>()
----> 1 s

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

- VIII. In Python interactive mode, the last printed expression is assigned to the variable `_` (underscore) . This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations.

Eg:- 1)

```
[ ] 82+44

126

[ ] _ * 2 # 126 is assigned to _

252

[ ] 270 - _ # now 252 is assigned to _

18
```

Eg:- 2)

```

[ ] tax = 10.58/100

[ ] price = 210

[ ] price*tax
22.218

[ ] price + _ # the last printed expression i.e. 22.218 is assigned to _
232.218

[ ] round( _ , 2) # 232.218 is assigned to _ variable
                  # round function is used to round-off the number to given places of decimal(here it is 2)
232.22

```

NOTE :- This variable should be treated as *read-only* by the user.

If you explicitly assign a value to the variable `_` then you are actually creating a local variable and masking the built-in variable with these special behaviour.

```

[ ] 33*2
66

[ ] _ # using _ variable to access last printed expression
66

[ ] _ = 34 # explicitly assigning a value to _

[ ] _
34

```

`_` variable should print 66 because 66 is the last printed expression but since a local variable `_` is created so the built-in variable loses its special behaviour at this point.

• Strings

String literals in python are surrounded by either `' '` *single quotation marks* { ex: 'Hello' } or `" "` *double quotation marks*. { ex: "Hello" }

You can display a string literal with the `print()` function:

```
[ ] 'Hey there!'
[ ] "Hey there!"
```

Above both syntax are same so you can use any one of them

I. Now learn how to print the *Apostrophe* ' in our text

Let's check what happened if we simply put ' in our normal text

```
[ ] 'Which is correct, Chris's chair or Chris' chair?'
File "<ipython-input-5-9fb21ef5fed2>", line 1
    'Which is correct, Chris's chair or Chris' chair?'
                                ^
SyntaxError: invalid syntax
```

Hence it shows error so to avoid this we have 2 ways : -

→ By putting *backslash* before the ' { ex : \ } .Hence we escape the single quote.

```
[ ] a='Which is correct, Chris\'s chair or Chris\' chair?'
    print(a)
Which is correct, Chris's chair or Chris' chair?
```

→ By using *double quotes* and enclosing text in *single quotes* and vice versa.


```
[ ] a="Yes," she said.  
    print(a)
```

```
⦿ "Yes," she said.
```

```
[ ] a="Yes," she said.  
    print(a)
```

```
⦿ "Yes," she said.
```

If you are confused with the starting and ending quotes use print function what is basically do is print the text or anything which we pass to it

```
[ ] print("'Yes,' she said.")
```

```
⦿ 'Yes,' she said.
```

NOTE :- Don't use backslash `\` in the double and single quote format it will print it as it is

```
[ ] 'Isn\'t," they said.'
```

```
⦿ 'Isn\'t," they said.'
```

II. Printing the continued text in the new line

By using `\n` it will print the text ahead of it in the next line

```
[ ] 'First line.\nSecond line'
```

```
⦿ 'First line.\nSecond line'
```

If we write like in the above case, `\n` is included in the text so to use `\n` to print the text in the next line use print function

```
[ ] y='First line.\nSecond line.'
    print(y)
```

First line.
Second line.

Another way is by using triple-quotes: `"""..."""` or `'''...'''`. End of lines are automatically considered as the next line

```
print("""First line
      Second line { Writng text after 2 spaces from the begining }
      Third line

""")
```

First line
Second line { Writng text after 2 spaces from the begining }
Third line

We can even prevent the automatically next line syntax by adding a `\` at the end of the line. Look at the following example:

```
[ ] print("""First line \
      First line continued
      Third line

""")
```

First line First line continued
Third line

III. Printing raw strings

Hence `\n` will be read as the operator for printing the value in the next line

```
[ ] print('C:\helo\desktop\name')
```

C:\helo\desktop
ame

If you don't want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:

To avoid this we will add an extra argument `r` at the beginning of our text

```
[ ] y=r'C\helo\desktop\name'
    print(y)

C\user\desktop\name
```

IV. Operations on Strings

→ Strings can be concatenated (glued together) with the `+` operator and repeated with `*` operator.

- `*` Multiplication operator multiplies the same text into the number

```
[ ] c=4 * 'hi'
    c

'hihihihi'
```

- `+` Addition operator works as to stick 2 broken things together

```
[ ] d=c + 'haha'
    d

'hihihihahaha'
```

→ There a lot of other methods which help to concatenate 2 strings

- Two or more string literals (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```

▶ 'Hi ' ',how ' 'are ' 'You ?'
👤 'Hi ,how are You ?'

```

Well the above method only works with two literals, not with variables or expressions:

```

[ ] a='Hi '
    b=a 'how'
    b
👤 File "<ipython-input-56-f496d829943d>", line 2
    b=a 'how'
      ^
SyntaxError: invalid syntax

```

To avoid the above error just simply use the **+** operator

```

[ ] a='Hi '
    b=a + 'how ' + 'are ' + 'u ?'
    b
👤 'Hi how are u ?'

[ ] 3 * 'hu' ' j'
👤 'hu jhu jhu j'

```

V. Indexing

P	y	t	h	o	n
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

```
[ ] word = "Python"
```

```
[ ] word[0]
```

```
'P'
```

```
[ ] word[4]
```

```
'o'
```

Indices may also be negative numbers:

To print the values from the last we put -ve sign and the value beginning from the right end. Let's look at some examples:

```
[ ] word[-2] # returns the 2nd character from the right
```

```
'o'
```

```
[ ] word[-5] # returns the 5th character from the right
```

```
'y'
```

VI. Slicing

Slicing, as the name suggests, means to cut or divide into slices or segments.

Slicing in strings means to create parts out of the original string or to create substring. While indexing is used to obtain individual characters, slicing allows you to obtain substring.

For better understanding let's look at some examples:

- Specify the start index and the end index, separated by a colon `:` to return a part of string.
- The first value is included and the last value is excluded `[start,stop)` {start and stop are the values you have entered}

```
[ ] word = "Python"

[ ] word[2:5] # slicing starts at index 2 and ends at index 5 excluding the 5th character
'cho'

[ ] word [0:4]
'Pyth'
```

- The absence of start index defaults to zero and absence of end index defaults to the size of the string.

```
[ ] word1 = "Geeksman"

[ ] word1[:5] #omitted start index is default to 0
'Geeks'

[ ] word1[5:] #omitted end index is default to the size of string
'man'

[ ] word1[-3:]
'man'

[ ] word1[:]
'Geeksman'
```

- Using too large an index results in error.

```
[ ] word[50]

-----
IndexError                                Traceback (most recent call last)
<ipython-input-6-22e03385725e> in <module>()
----> 1 word[50]

IndexError: string index out of range

SEARCH STACK OVERFLOW

[ ] word[-30]

-----
IndexError                                Traceback (most recent call last)
<ipython-input-11-67244c14c9b6> in <module>()
----> 1 word[-30]

IndexError: string index out of range
```

- Passing out of range slice indices is very well handled by Python. If we pass the index which is greater than the size of string it doesn't give any error it just simply prints values upto last index.

```
[ ] word[3:23]

'hon'

[ ] word[23:4]

''
```

- Python strings are immutable i.e. they cannot be changed. So assigning value to an indexed position in string results in an error.

```
[ ] word[0] = "K"

-----
TypeError                                Traceback (most recent call last)
<ipython-input-16-35c5c818a9af> in <module>()
----> 1 word[0] = "K"
```

- A new string can be created by using **+** operator.

```
[ ] "K" + word[1:]
'Kython'

[ ] word1[:-3] + "woman"
'Geekswoman'
```

- The built-in `len()` function is used to return the length of string.

```
[ ] s = "pneumonoultramicroscopicsilicovolcanoconiosis" #This is the longest word of english language and it refers to a lung disease.
[ ] len(s)
45
```

• Lists

To store similar values together like integers , floats and strings and to access them easily we use lists in python. There are a number of data types available but list is the most commonly used. As we have the concept of arrays in C and C++, similarly we have lists in Python . Most of the functionalities are same but we'll experience few differences ...

I. Initializing a list and printing values

```
[ ] a=[1,2,3,4,5,6]
a
[1, 2, 3, 4, 5, 6]
```

II. Like strings list can also be indexed and sliced

→ Indexing

Square brackets `[]` can be used to access elements of the list.

```
[9] a=[1,2,3,4,5,6]
    a
    a[4] # prints the value at 4th index
    5
```

→ Slicing

Slicing of the list meaning printing the particular values in a list in a given range.

1) Printing all the values in the list.

- Firstly by simply passing the list name
- Secondly with the help of slicing

```
[ ] a
[1, 2, 3, 4, 5, 6]

[ ] a[:]
[1, 2, 3, 4, 5, 6]
```

2) Printing part of a list.

- Specify the start index and the end index, separated by a colon `:` to return a part of the list.
- The first value is included and the last value is excluded [start,stop)
{start and stop are the values you have entered}

```
[10] a=[1,2,3,4,5,6]
     a[1:5]
     [2, 3, 4, 5]

[11] a[0:3]
     [1, 2, 3]
```

→ If we pass the index which is greater than the size of list it doesn't give any error it just simply print values upto last index

```
[ ] a[1:78]
```

```
[2, 3, 4, 5, 6]
```

```
[ ] a[89:]
```

```
[ ]
```

3) Printing the values from the last or negative slicing

To print the values from the last we put negative sign and the value beginning from the right end. Remember value starts from 1 not from 0 because { -0 & +0 } have no meaning. Let's look at examples to get a clear view on this

Eg:-

1. `a[-1]` Print's the last value i.e 6
2. `a[-4]` Print the 4th value from the end i.e 3
3. `a[-2:]` Here we are printing values from second last to end
4. `a[0:-2]` Prints all values except the last two

```
[12] print(a)  
a[-1]
```

```
[1, 2, 3, 4, 5, 6]  
6
```

```
a[-4]
```

```
3
```

```
[ ] a[-2:]
```

```
[5, 6]
```


```
[ ] a[0:-2]
```

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

III. Lists also support operations like concatenation


Remember that the result of concatenation has to be stored in another list.

```
[ ] a=[1,2,3,4,5,6]
    b= a + [7,8,9]
    print(a)
    print(b)
```

 [1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7, 8, 9]

The initial list {a,b} doesn't get updated until you store the result in it. You can see the result in the below example :-


```
[ ] a=[1,2,3,4,5,6]
    b=[7,8,9,10]
    a=a+b
    a
```

 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


IV. Lists are mutable

- Unlike strings, which are immutable, list is a mutable data type i.e value at particular index can be changed.

```
[ ] a
```

 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```
[ ] a[0]=10
    a
```

 [10, 2, 3, 4, 5, 6, 7, 8, 9, 10]

- You can also add new items at the end of the list, by using the append() method

```
[ ] a.append(7)
a
```

```
[ ] [10, 2, 3, 4, 5, 6, 7, 8, 9, 10, 7]
```

- Assignment to slices is also possible, and this can even change the size of the list or clear it entirely

- Replacing values

```
[ ] a
```

```
[ ] [10, 2, 3, 4, 5, 6, 7, 8, 9, 10, 7]
```

```
[ ] a[2:4]=[15,12]
a
```

```
[ ] [10, 2, 15, 12, 5, 6, 7, 8, 9, 10, 7]
```

- Removing values in a given range

```
[ ] a[2:4]=[]
a
```

```
[ ] [10, 2, 5, 6, 7, 8, 9, 10, 7]
```

- Clear the list by replacing all the elements with an empty list

```
[ ] a[:] = []  
a  
[ ]
```

V. **len()** function can be used to return length of List

```
[ ] b  
[1, 2, 3, 4, 5, 6, 7, 8, 9]  
[ ] len(b)  
9
```

VI. Nested Lists

It is possible to nest lists (create lists containing other lists) similar to 2-D & 3-D array

```
[ ] c=['a','b','c']  
d=[1,2,3]  
e=[c,d]  
e  
[['a', 'b', 'c'], [1, 2, 3]]  
[ ] e[0]  
['a', 'b', 'c']  
[ ] e[0][0]  
['a']
```

LESSON TWO

- **If statement**

In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value or any sequence; anything with a non-zero length is true, empty sequences are false.

The standard comparison operators are the same as in C,

< → (*less than*)

> → (*greater than*)

== → (*equal to*)

<= → (*less than equal to*)

>= → (*greater than equal to*)

!= → (*not equal to*)

→ If

- An “if statement” starts with **if** keyword.
- The body of the loop is indented. To add indentation in the interactive prompt you have to type a tab. (Here in colab notebook we have auto indent facility)


→ Elif

The elif keyword in Python is way of saying "if the previous conditions were not true, then try this condition".

→ Else

If no condition is satisfied then statements below else are executed.

```
[ ] a = 20
    b = 5
    if a > b:
        print("a is greater.")
```

 a is greater.

```
[ ] x = 32      #try with other values

    if x < 0:
        x = 0
        print('Negative changed to zero')
    elif x == 0:      #elif is short for else if
        print('Zero')
    elif x == 1:
        print('Single')
    else:
        print('More')
```


 More

- **Loops**

→ **While Loop**

- With the *while loop* we can execute a set of statements as long as a condition is true.
- The “while loop” starts with **while** keyword.

```
[ ] i = 1      #initialising the variable
    while i < 6: #condition
        print(i)
        i = i + 1 #incrementing i
```

 1
2
3
4
5

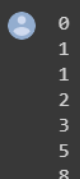
NOTE :- Don't forget to increment i or else the loop will continue forever.

Fibonacci series :

0, 1, 1, 2, 3, 5, 8, 13, 21,

Each number is the sum of the two preceding ones, starting from 0 and 1.

```
[ ] # Fibonacci Series
a, b = 0, 1      # multiple assignment
while a < 10:
    print (a)
    a, b = b, a+b
```



First and last line contain multiple assignments :

On the first line, the variables **a** and **b** simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side of the assignment operator(**=**) are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right. Then the values are assigned to the left-hand side variables simultaneously.

We can also print the series in the same line :

We can use the **end** parameter of the print function to end a print statement using a string or character. By default the value is **\n** which means the text is printed in new line.


```
[ ] # Fibonacci Series
a, b = 0, 1
while a < 10:
    print (a, end = ", ") #end the print statement using by a ", "
    a, b = b, a+b
```

```
0, 1, 1, 2, 3, 5, 8,
```

```
[ ] # Fibonacci Series
a, b = 0, 1
while a < 10:
    print (a, end = " : ")
    a, b = b, a+b
```

```
0 : 1 : 1 : 2 : 3 : 5 : 8 :
```

→ For Loop

For loop in Python, is different from that in C or C++ . A *for loop* is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string), in the order of their appearance in the sequence.

- Iterating over list

```
[ ] fruits = ["banana", "apple", "cherry", "strawberry", "kiwi"]
for fruit in fruits:
    print(fruit, len(fruit))
```

```
banana 6
apple 5
cherry 6
strawberry 10
kiwi 4
```

- Iterating over string

```
[ ] name = "banana"
    for ch in name:
        print(ch)
```

```
b
a
n
a
n
a
```

- **range() Function**

Range (start, stop, skip)

This will be well understood through following examples:

- range(9, 15)

--> 9, 10, 11, 12, 13, 14

- range(5)

--> 0, 1, 2, 3, 4

- range(2, 19, 2)

--> 2, 4, 6, 8, 10, 12, 14, 16, 18

- range(5, 20, 5)

--> 5, 10, 15

```
[ ] for i in range(3, 10):  
    print(i, i + 2)
```

```
3 5  
4 6  
5 7  
6 8  
7 9  
8 10  
9 11
```

Observing these examples, we feel that the object returned by `range()` behaves as if it is a list but in fact it isn't.

We say such an object is iterable, that is, suitable as a target for *functions* and *constructs* that expect something from which they can obtain successive items until the supply is exhausted.

```
[ ] #lets see what happens if we print a range  
  
print(range(15))
```

```
range(0, 15)
```

```
[ ] # we can use range in sum() function  
  
sum(range(10))    # 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
```

```
45
```

- Later we'll see more functions that take iterables as argument and return iterables.

```
[ ] # To get a list from range  
  
list(range(10))    #list() function creates list from iterables
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

• Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Looking at some examples will make things clear:

```

▶ count = ["two", "three", "four"]
  fruits = ["apples", "bananas", "oranges"]

  for x in count:
    for y in fruits:
      print(x, y , end = " ---- ")
    print()

  two apples ---- two bananas ---- two oranges ----
  three apples ---- three bananas ---- three oranges ----
  four apples ---- four bananas ---- four oranges ----

```

● Printing Various Pattern using nested for loops

I. Printing a square of size 4 * 4

```

[ ] l=4 # intializing length to 4
    for i in range(0,l):
      for j in range(0,l):
        print('1',end='')
      print('\n')

  1111
  1111
  1111
  1111

```

II. Rectangle of size 4 * 3

```
▶ l=4 # intializing length to 4
  b=3 # intializing breadth to 4
  for i in range(0,l):
    for j in range(0,b):
      print('1',end='')
    print('\n')
```

```
111
111
111
111
```

III. Triangle 1

```
[ ] l=5 # intializing length to 5
    for i in range(1,l+1):
      for j in range(1,i+1):
        print(j,end='')
      print('\n')
```

```
1
12
123
1234
12345
```

NOTE: In this pattern, the count of numbers on each row is equal to the current row number. In each row, every next number is incremented by 1

IV. Triangle 2

```
l=5
for i in range(1,l+1):
    for j in range(1,i):
        print(j,end='')
    for j in range(i,0,-1):
        print(j,end='')
    print('\n')
```

1

121

12321

1234321

123454321

NOTE: In this pattern, the count of numbers on each row is $(2n - 1)$ where n is the current row number. In each row, starting from 1, every next number is incremented by 1 upto the current row number and then decremented by 1 upto the number 1.

```
l=5
for i in range(1,l+1):
    for j in range(1,i+1):
        if j==i or j==1:
            print(j,end='')
        else:
            print('*',end='')
    print('\n')
```

1

12

1*3

1**4

1***5

V. Hollow Triangle

NOTE: In this pattern, only the first and the last number of the row is printed and the intermediate elements are * .

VI. Equilateral Triangle

```
[ ] l=10
    for i in range(1,l+1):
        for j in range(i,l):
            print(' ',end='')
        for j in range(1,2*i):
            print('*',end='')
        print('')
```

```

      *
    ***
  *****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****

```

VII. Double the number

```
[ ] l=8
for i in range(1,l+1):
    p=2**(i-1)
    for j in range(0,i):
        print(p,end=' ')
        p=p//2
    print("")
```

```

1
2 1
4 2 1
8 4 2 1
16 8 4 2 1
32 16 8 4 2 1
64 32 16 8 4 2 1
128 64 32 16 8 4 2 1

```

NOTE: In each column, every number is double the previous number.

VIII. Reverse number from 10 to 1

```
[ ] current=1
    count=1
    s=0
    for i in range(1,6):
        current=count
        count=count+i+1
        for j in range(0,i):
            print(current,end=' ')
            current=current-1
        print(' ')

1
3 2
6 5 4
10 9 8 7
15 14 13 12 11
```

- **Break, Continue and Pass Statements**

In a coding event you are solving a question and it has 21 test cases and now to get awarded the marks for it you have to solve at least 15 test cases.

So once you solved at least 15 test cases you want to jump to the next question so in this case break statement is used.

Also for a particular value you want to execute half your code statement and proceed to the next test case continue statement is used.

If you have solved more than 15 test cases and it doesn't matter whether the remaining test cases are correct or not in this case, a pass statement is used.

- **Break Statement**

When a **break** statement is encountered the current loop is ended at once.

- Writing a code which will break the execution of the loop if the number is divisible by 7.


```
for i in range(1,1+1):
    print(i)
    if i%7==0:
        print('The current number is divisible by 7')
        break
print('Mission accomplished')
```

1
2
3
4
5
6
7
The current number is divisible by 7
Mission accomplished

- Print all the characters in a string unless 'a' is encountered.

```
s='hey how are u ?'
for i in range(0,len(s)):
    if s[i]=='a':
        break
    else:
        print(s[i])
```

h
e
y

h
o
w

→ Continue Statement

When a **continue** statement is encountered the current iteration is skipped and the next iteration continues.

- Print values from 0 to 10 excluding 5,6,7

```
[ ] 1=10
    for i in range(0,1+1):
        if i>= 5 and i<= 7:
            print('Found a number between 5 & 7')
            continue
        print(i)
```

0
1
2
3
4
Found a number between 5 & 7
Found a number between 5 & 7
Found a number between 5 & 7
8
9
10

- Print only odd values

```
[ ] 1=15
    for i in range(0,1+1):
        if i%2==0:
            print('Given value is even')
            continue
        print(i)
```

Given value is even
1
Given value is even
3
Given value is even
5
Given value is even
7
Given value is even
9
Given value is even
11
Given value is even
13
Given value is even
15

→ Pass Statement

Pass statement is a null statement i.e. nothing happens on its execution.

```
[ ] s='Pass Statement'
    for i in range(0,len(s)):
        if s[i]=='S':
            pass
            print("I'm here for nothing ",end=' ')
        print(s[i])
```

```
P
a
s
s

I'm here for nothing S
t
a
t
e
m
e
n
t
```

Understanding difference between **continue** and **pass** statement with the help of this example:

1) continue

```
[ ] l=5
    for i in range(0,l+1):
        if i==2:
            pass
            print('asdf')
        else:
            print(i)
```

```
0
1
asdf
3
4
5
```

2) pass

```
l=5
for i in range(0,l+1):
    if i==2:
        continue
    print('asdf')
else:
    print(i)
```

0
1
3
4
5

LESSON THREE

- **Del Statement**

The **del** keyword is used to delete objects. So the del keyword can be used to delete variables, lists, parts of a list etc.

→ Delete a variable

```
[1] a='Hello'

[2] del a

[3] a
-----
NameError                                Traceback (most recent call last)
<ipython-input-3-3f786850e387> in <module>()
----> 1 a

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

→ Delete an element of a list

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

[ ] del(a[2])

[ ] a
[1, 2, 4, 5]

[ ] del(a[0])

[ ] a
[2, 4, 5]
```

→ Delete a slice from a list

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

```
[ ] del(a[:])
```

```
[ ] a
```

```
[ ] []
```

→ Delete all elements of a list

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

```
[ ] del(a[:])
```

```
[ ] a
```

```
[ ] []
```

→ Delete the entire list

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

```
[ ] del(a)
```

```
[ ] a
```

```
[ ] -----  
NameError                                Traceback (most recent call last)  
<ipython-input-90-3f786850e387> in <module>()  
----> 1 a  
  
NameError: name 'a' is not defined
```

• Sets

A **set** is an unordered collection with no duplicate elements.

A set is similar to a list but in a set there are no duplicate values.

For eg:-

Defining syntax

```
a = {value_1, value_2, value_3, value_4}
```

In set **a** , if value_1 is the same as value_2 . The output will be
a = {value_1,value_3,value_4}

→ Define and print a set

```
[ ] a={'Lesson1','Lesson2','Lesson3','Lesson4','Lesson1'}  
  
[ ] a  
[ ] {'Lesson1', 'Lesson2', 'Lesson3', 'Lesson4'}
```

→ Check whether a particular value is present or not

It will return a boolean value i.e. either true{ if present} or false{ if not present}

```
[ ] 'Lesson1' in a  
[ ] True  
[ ] 'Lesson5' in a  
[ ] False
```

→ Pass a string to a set and store results in a new variable.

It will basically break each character of the string and keep only unique characters of the given string

1. Printing all the unique characters in a

```
[ ] a=set('Lesson 1')  
[ ] a  
[ ] {' ', '1', 'L', 'e', 'n', 'o', 's'}
```

2. Printing all the unique characters in b

```
[ ] b=set('Lesson 2')  
[ ] b  
[ ] {' ', '2', 'L', 'e', 'n', 'o', 's'}
```

→ What happen on using **+** operator to combine two sets

```
[ ] a + b

-----
TypeError                                Traceback (most recent call last)
<ipython-input-99-bd58363a63fc> in <module>()
----> 1 a + b

TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

Oops! It shows an error

→ Operations on sets

- Characters present in both a and b

```
a | b

{' ', '1', '2', 'L', 'e', 'n', 'o', 's'}
```

- Characters which are present in a but not in b

```
a - b

{'1'}
```

- Characters which are present in both a and b

```
a & b

{' ', 'L', 'e', 'n', 'o', 's'}
```

- Characters that are not common in both a & b

```
a ^ b

{'1', '2'}
```

→ Iterate through set

```
[ ] c=set('Lesson 1')

[ ] c

{' ', '1', 'L', 'e', 'n', 'o', 's'}
```



```
▶ for i in c:  
    print(i,end=' ')
```

▶ s n L 1 e o

→ Add values to set

```
▶ c.add('Hello')  
c
```

▶ {' ', '1', 'Hello', 'L', 'e', 'n', 'o', 's'}

```
[ ] for i in range(0,5):  
    c.add(i)
```

[] c

▶ {' ', 0, 1, '1', 2, 3, 4, 'Hello', 'L', 'e', 'n', 'o', 's'}

→ Remove element from set

- **remove()** :

This function removes the element from the set

```
[ ] c
```

▶ {' ', 0, 1, '1', 2, 3, 4, 'Hello', 'L', 'e', 'n', 'o', 's'}

```
[ ] c.remove(0)
```

[] c

▶ {' ', 1, '1', 2, 3, 4, 'Hello', 'L', 'e', 'n', 'o', 's'}

```
[ ] c.remove('o')
```

[] c

▶ {' ', 1, '1', 2, 3, 4, 'Hello', 'L', 'e', 'n', 's'}

Removing value which are not present in the set will give an error

```
c.remove('a')

-----
KeyError                                Traceback (most recent call last)
<ipython-input-114-79ba6b4e5a55> in <module>()
----> 1 c.remove('a')

KeyError: 'a'
```

- **discard()** :

This function removes the element from the set

```
[ ] c
{' ', 1, '1', 2, 3, 4, 'Hello', 'L', 'e', 'n', 's'}

[ ] c.discard(4)

[ ] c
{' ', 1, '1', 2, 3, 'Hello', 'L', 'e', 'n', 's'}

[ ] c.discard(4)
```

Removing a non existing item using discard(), will not raise an error

```
[ ] c
{' ', '1', 'Hello', 'L', 'e', 'n', 'o', 's'}

[ ] c.discard('f')

[ ] c
{' ', '1', 'Hello', 'L', 'e', 'n', 'o', 's'}
```

→ Set does not support indexing

This is because Sets are unordered, so you cannot be sure in which order the items will appear.

- **Dictionary**

It is best to think of a dictionary as a set of **key : value** pairs, with the requirement that the keys are unique (within one dictionary). A pair of curly braces **{ }** creates an empty dictionary. Placing a comma-separated list of key : value pairs within the braces adds the initial key : value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key.

syntax:- a = { key:value , key:value }

for eg:- a={ 'A' : 0, 'B' :1 }

Here A is the key and 0 is the value for it and B is another key and 1 is the value for it.

→ Defining a dictionary

```
[ ] a={'Chetan': 1 , 'Ankit': 2 , 'Vaibhav':3}
```

```
[ ] a={'Chetan': 1 , 'Ankit': 2 , 'Vaibhav':3}
```

→ Deleting a key-value pair from the dictionary

```
[ ] del a['Ankit']
```

```
[ ] a
```

```
{'Chetan': 1, 'Vaibhav': 3}
```

→ Deleting the complete dictionary

```
[ ] del(a)

[ ] a

-----
NameError                                Traceback (most recent call last)
<ipython-input-141-3f786850e387> in <module>()
----> 1 a

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

→ Adding a new key-value in the dictionary

```
[ ] a['Arnav']=6

[ ] a

{'Ankit': 2, 'Arnav': 6, 'Chetan': 1, 'Vaibhav': 3}
```

→ Changing the value for a particular key

```
[ ] a['Ankit']=4

[ ] a

{'Ankit': 4, 'Arnav': 6, 'Chetan': 1, 'Vaibhav': 3}

[ ] a['Chetan']=5

[ ] a

{'Ankit': 4, 'Arnav': 6, 'Chetan': 5, 'Vaibhav': 3}
```

→ Listing the keys of a dictionary

```
[ ] b=list(a)
    b

['Chetan', 'Ankit', 'Vaibhav', 'Arnav']
```

→ Checking if a particular key is present in the dictionary or not

It will return boolean value i.e. True { if_value_is_present } and False{if_value_is_not_present }

```
[ ] 'Vaibhav' in a
True

[ ] 'Cheeta' in a
False

[ ] 'Cheeta' not in a
True
```

→ A shortcut to create a dictionary

```
b={x: x**2 for x in (2, 4, 6)}

[ ] b
{2: 4, 4: 16, 6: 36}
```

→ Creating dictionary using `dict()` simply by specifying pairs using keyword arguments

```
[ ] a=dict(Vipul=1,Anshul=2,Vaibhav=3)

[ ] a
{'Anshul': 2, 'Vaibhav': 3, 'Vipul': 1}
```

→ Iterating over a dictionary :

- Printing the keys

```
[ ] for i in a:
    print(i)

Vipul
Anshul
Vaibhav
```

- Printing the values

```
[ ] for i in a:  
    print(i,a[i],end=' ')  
    print('')
```

```
Vipul 1  
Anshul 2  
Vaibhav 3
```

- Printing key-value pairs

```
▶ a=['tic', 'tac', 'toe']  
for i,j in enumerate(a):  
    print(i,j,end=' ')  
    print('')
```

```
0 tic  
1 tac  
2 toe
```

• Looping Techniques

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

→ List

```
▶ a=['tic', 'tac', 'toe']  
for i,j in enumerate(a):  
    print(i,j,end=' ')  
    print('')
```

```
0 tic  
1 tac  
2 toe
```

→ Set

```
[ ] a={'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}  
for i,j in enumerate(a):  
    print(i,j,end=' ')  
    print('')
```

```
0 orange  
1 pear  
2 banana  
3 apple
```

→ Dictionary

- Using `items()` :

We can Loop through both keys and values in dictionary by using the items() keyword

For eg :-

```
new_dict = { "A" : 1, "B" : 2, "C" : 3}
```

```
new_dict.items() returns [ ( "A", 1 ), ( "B", 2 ), ( "C", 3 ) ]
```

```
[ ] a = {'Coding': 'Sites', 'Bread': 'Butter', 'Pen': 'Pencil'}
    for k, v in a.items():
        print(k,v)
```

```
Coding Sites
Bread Butter
Pen Pencil
```

- Using `enumerate()` :

```
▶ a=['tic', 'tac', 'toe']
  for i,j in enumerate(a):
      print(i,j,end=' ')
      print('')
```

```
0 tic
1 tac
2 toe
```

By using the enumerate function and passing the dictionary with the items() keyword it will return a tuple and we can access the key value pair by using the indexing.

→ Looping over two or more sequences

To loop over two or more sequences at the same time, the entries can be paired with the zip() function

```
[ ] a = [1,2,3,4,5]
    b = [1,4,9,16,25]
    c = [1,8,27,64,125]
    for x,y,z in zip(a,b,c):
        print(x,y,z)
```

```
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
```

- **Bold Text**

In this we will learn how to output bold text in python. Well we are having a lot of different ways to print bold text in python you can choose any of them which suits you better

→ Method one

```
[6] a_string = "End of lesson 3"
    bolded_string = "\033[1m" + a_string + "\033[0m"

[7] print(bolded_string )
```

```
End of lesson 3
```

→ Method two

```
[ ] !pip install termcolor
```

```
Requirement already satisfied: termcolor in /usr/local/lib/python3.6/dist-packages (1.1.1)
```

```
[8] from termcolor import colored
    print(colored('End of leeson 3', attrs=['bold']))
```

```
End of leeson 3
```


LESSON FOUR

- **Function**

A function is a block of code that performs a specific task only when it is called.
You can also pass data in function, it is called parameters.
And the function can also return a value as a result.

→ Defining a function:

The **def** keyword is used to define the function followed by the name of the function and then the parenthesized formal parameters. The function body starts at the next line which is indented.

```
[ ] def my_function():  
    print("Hello guyzzz")
```

→ Calling a function:

```
▶ my_function()  
⬆ Hello guyzzz
```

Just simply write the name of the function followed by parenthesis.

- **Arguments**

They are the information that we can pass to the function.
We can add as many arguments as we wish.
{ informally we can say that arguments actually create personalised result }

What if we don't pass argument to a function which is defined to have one :

```
[ ] #defining function
def my_function1(country):
    print("Welcome to "+country)

[ ] #calling function for various values of country
my_function1("India")
my_function1("England")
my_function1("Germany")

Welcome to India
Welcome to England
Welcome to Germany

[ ] #lets see what happens if we do not pass any values to the function

my_function1()

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-24-358011bd65ea> in <module>()
      1 #lets see what happens if we do not pass any values to the function
      2
----> 3 my_function1()

TypeError: my_function1() missing 1 required positional argument: 'country'
```

What is the difference between *parameters* and *arguments* :

Parameters are what we declare in the function

Arguments are what we pass while calling the function

For example: In `my_function1()` :

- `country` is the parameter.
- values like `"India"` , `"England"` , `"Germany"` are arguments.

• Default argument values

As the name suggests, the function will take some arguments and will initialize the values of those parameters irrespective of the fact that these parameters are receiving values or not.

This method creates a function that can be called with fewer arguments than it is defined to allow. If a particular argument is not passed by the user the function uses the default values.

```
[ ] def my_function2(lang = "Python"):
    print("I am working on " + lang)

    my_function2("C")
    my_function2("C++")
    my_function2()
    my_function2("Java")
```

```
I am working on C
I am working on C++
I am working on Python
I am working on Java
```

→ Another form of setting default parameters initially

NOTE :- The default values are set permanently once we define the function and after that any changes made to the value will not affect it.

```
[ ] lang='Python'

def f(arg=lang):
    print(arg)

lang="C"

f("C++")
f()
```

```
C++
Python
```

→ Although we can make changes to the values before the function declaration statement

```
[ ] lang='Python'
    lang='C'

    def f(arg=lang):
        print(arg)

    f("C++")
    f()
```

C++
C

● Number of Arguments

We can add any number of arguments separated by commas

```
[ ] def name(fname, lname):
    print(fname+" "+lname)
```

```
[ ] name("Harry", "Potter") # You can try adding your own name
```

Harry Potter

```
[ ] # lets see what happens if we pass only one argument instead of two
```

```
name("Harry")
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-25-2751dece01b0> in <module>()
      1 # lets see what happens if we pass only one parameter instead of two
      2
----> 3 name("Harry")

TypeError: name() missing 1 required positional argument: 'lname'
```

In the above example the function expected 2 arguments and received just 1

● Arbitrary Arguments (***args**)

In a similar situation, if we are unaware of the number of arguments that will be passed. Then we simply put ***** before the parameter name in the function definition . What actually happens is that the arguments that are passed are accepted as a tuple of arguments and hence the items are accessed accordingly.

NOTE: A tuple is an immutable collection i.e. you cannot append, remove, insert or alter any values in it. eg: a = (3, 5, 2)

```
[ ] def my_function3(*fruits):  
    for fruit in fruits:  
        print("I love to eat "+fruit)  
  
[ ] my_function3("bananas","apples","oranges")  
  
I love to eat bananas  
I love to eat apples  
I love to eat oranges
```

● Passing a List as an Argument

→ You can send any data type as an argument to a function (string, number, list, dictionary and many more) and it will be treated as the same data type inside the function.

For eg:- If you pass a string as an argument, it will still be a string when it reaches the function. In the same way list will be treated as list.

```
[ ] def my_function4(coding_sites):  
    for i in coding_sites:  
        print(i)  
  
Coding_Sites=['Hackerrank','Hackerearth','Geeksforgeeks','Codeshef','Codeforces','Leetcode']  
  
my_function4(Coding_Sites)  
  
Hackerrank  
Hackerearth  
Geeksforgeeks  
Codeshef  
Codeforces  
Leetcode
```

→ As we know list is mutable so we can make changes to it like we can append elements or remove elements from it. Let's see how we can implement it

```
[ ] def my_function5(coding_sites):
    coding_sites.append('Spoj') # Adding a value at the end
    for i in coding_sites:
        print(i)

    print(len(coding_sites))

    coding_sites=coding_sites[0:6] # Removing the last element in the list
    for i in coding_sites:
        print(i)

coding_sites=['Hackerrank','Hackerearth','Geeksforgeeks','Codeshef','Codeforces','Leetcode']

my_function5(coding_sites)
```

```
Hackerrank
Hackerearth
Geeksforgeeks
Codeshef
Codeforces
Leetcode
Spoj
7
Hackerrank
Hackerearth
Geeksforgeeks
Codeshef
Codeforces
Leetcode
```

● Keyword Arguments

Functions can also be called using keyword arguments of the form key = value.

While calling functions using arguments the order of passing arguments matters

But while calling functions through keyword arguments order does not matter.

```
[ ] # lets look at the name() function we created earlier

def name(fname, lname):
    print(fname+" "+lname)

[ ] # calling function using arguments

name ("Daniel", "Radcliffe") # fname="Daniel" and lname="Radcliffe"

Daniel Radcliffe

[ ] name ("Radcliffe","Daniel") # fname="Radcliffe" and lname="Daniel"

Radcliffe Daniel

[ ] # calling function using keyword arguments

name(lname = "Radcliffe", fname = "Daniel")

Daniel Radcliffe

[ ] def dog(count, colour = "brown", breed = "German Shepherd", action = "barking"):
    print("There are",count,colour +" "+ breed, end = ".")
    print("They are "+ action)
```

Let's look at the following example of a **dog()** function:

→ Valid calling statements:

```
# the dog function can be called in the following ways

print("# 1", end = " ")

dog(5) # 1 positional argument

print("# 2", end = " ")

dog(count = 5) # 1 keyword argument

print("# 3", end = " ")

dog(count = 5, action = "sleeping") # 2 keyword arguments

print("# 4", end = " ")

dog(breed = "St. Bernard", colour = "white", count = "2") # 3 keyword arguments

print("# 5", end = " ")

dog(10, "black", "Cane Corso") # 3 positional arguments

print("# 6", end = " ")

dog(6, action = "eating") # 1 positional argument and 1 keyword argument

# 1 There are 5 brown German Shepherd.They are barking
# 2 There are 5 brown German Shepherd.They are barking
# 3 There are 5 brown German Shepherd.They are sleeping
# 4 There are 2 white St. Bernard.They are barking
# 5 There are 10 black Cane Corso.They are barking
# 6 There are 6 brown German Shepherd.They are eating
```

→ Invalid calling statements:

```
[ ] dog() # 1 required argument count is missing

-----
TypeError                                Traceback (most recent call last)
<ipython-input-27-26a4184126ac> in <module>()
----> 1 dog() # 1 required argument count is missing

TypeError: dog() missing 1 required positional argument: 'count'
```

```
[ ] dog(count = 5, "Poodle") # positional argument follows keyword argument

File "<ipython-input-29-a0efc3131b01>", line 1
    dog(count = 5, "Poodle") # positional argument follows keyword argument
              ^
SyntaxError: positional argument follows keyword argument
```

```
[ ] dog(9, count = 3) # duplicate value for the same argument

-----
TypeError                                Traceback (most recent call last)
<ipython-input-31-28873430fdf0> in <module>()
----> 1 dog(9, count = 3) # duplicate value for the same argument

TypeError: dog() got multiple values for argument 'count'
```


- **Arbitrary Keyword Arguments (**kwargs)**

In a similar situation, if we are unaware of the number of keyword arguments that will be passed. Then we simply put ****** before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly:

NOTE: A dictionary is a collection which is unordered, mutable and indexed using keys. In Python dictionaries are written with curly brackets, and they have keys and values.

ex: { 1 : "first" , 2 : "second" , 3 : "third" }

```
def myfunction4(**kwargs):  
    for key, value in kwargs.items(): # looping through all the keys and values of the dictionary  
        print (key + " = " + value)  
  
[ ] myfunction4(first = "apple", second = "banana", third = "orange")  
  
first = apple  
second = banana  
third = orange
```

- **Return Value**

As the name suggests a function will return a value; it may be a string , list , number , boolean value or any other.

→ Returning Number

```
def sqre(x):  
    return x * x  
  
l=sqre(3)  
p=sqre(l)  
print(p)  
  
81
```

→ Returning Boolean value

```
def check(x):  
    if x%2==0:  
        return True  
    else:  
        return False  
  
print(check(4))  
print(check(5))
```

True
False

→ Returning List

```
def ar(x):  
    a=[]  
    for i in range(0,x):  
        a.append(i+1)  
    return a  
  
p=ar(5)  
print(p)
```

[1, 2, 3, 4, 5]

→ Returning string

```
def str(x):  
    x= x + ' Concept'  
    return x  
  
p=str("Welcome to the Return")  
print(p)
```

Welcome to the Return Concept

- **Pass Statement**

Function definition cannot be empty, but if for some reason we have a function definition with no statements, put in the pass statement to avoid getting an error.

```
[ ] def myfunction():  
    pass
```

- **Recursion**

Recursion is similar to loops :)


Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

NOTE:- Remember while using recursion you must put a terminating condition to exit from the program or it will result in a never ending process and will display an error :(

→ Printing factorial using recursion

- With terminating condition

```
[ ] def fib(a,b,k):  
    if k-2>0:  
        print(a+b,end=' ')  
        fib(b,a+b,k-1)  
    else:  
        return  
print('Printing the terms of series upto 8')  
k=8  
print('Required series :- 0 1 ',end='')  
fib(0,1,k)
```

```
 Printing the terms of series upto 8  
Required series :- 0 1 1 2 3 5 8 13
```

- Without terminating condition

```
[ ] def fact(x):
    return x*fact(x-1)
print("factorial of 5")
k=5
print(fact(k))
```

factorial of 5

RecursionError Traceback (most recent call last)

<ipython-input-5-de445e67b33c> in <module>()

3 print("factorial of 5")

4 k=5

----> 5 print(fact(k))

----- 1 frames -----

... last 1 frames repeated, from the frame below ...

<ipython-input-5-de445e67b33c> in fact(x)

1 def fact(x):

----> 2 return x*fact(x-1)

3 print("factorial of 5")

4 k=5

5 print(fact(k))

RecursionError: maximum recursion depth exceeded

→ Printing Fibonacci series { 0, 1, 1, 2, 3, 5, 8, . . . }

```
[ ] def fib(a,b,k):
    if k-2>0:
        print(a+b,end=' ')
        fib(b,a+b,k-1)
    else:
        return
print('Printing the terms of series upto 8')
k=8
print('Required series :- 0 1 ',end='')
fib(0,1,k)
```

Printing the terms of series upto 8

Required series :- 0 1 1 2 3 5 8 13

• Lambda Expression

A **Lambda Function** is a small anonymous function. It can take any number of arguments, but can only have one expression.

→ A lambda function that multiply 10 to the number passed in as an argument

```
x = lambda a : a * 10  
print(x(3))
```

30

The explanation of the above Programme is :-

1. What does `a : a * 10` does ? It returns `a` multiplied by 10 and stores the result in a
2. You can think of it as $f(x) = x^2 + x + 1$ { Any function }
3. We can call the function $f(x)$ i.e { x in our question } by passing suitable value to it for getting our desired result

→ A lambda function multiplying 3 values

```
[ ] x = lambda a, b, c : a * b * c  
print(x(5, 6, 2))
```

60

→ A lambda function to perform addition of 2 variables

```
def make_incrementor(n):  
    return lambda x: x + n  
f = make_incrementor(42)  
f(42)
```

84

Why Lambda Function ?

Lambda function is used along with built-in functions like `filter()` , `map()` and many more...

Let's see what they do ?

- `filter()` :

As the name suggests it will filter out values according to the condition applied

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

The filter() function in Python takes in a function and a list as arguments.

Syntax:- `new_list = list (filter (lambda x : condition, list_on which_operation))`

→ Print only those values which are divisible by 2

◆ Using loop

```
[ ] a = [1, 5, 4, 6, 8, 11, 3, 12]

[ ] new_list= list(filter(lambda x: (x%2 == 0),a))
    print(new_list)

[ ] [4, 6, 8, 12]
```

◆ Using lambda function

```
[ ] b=[]
    for i in range(0,len(a)):
        if a[i]%2==0:
            b.append(a[i])
    print(b)

[ ] [4, 6, 8, 12]
```

- **map()** :

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item


The map() function in Python takes in a function and a list as arguments.


Syntax:- `new_list = list (map (lambda x : condition, list_on which_operation))`

→ Print only those values which are divisible by 2

◆ Using loop

```
[ ] a=[1,2,3,4,5,6,7,8,9,10]
```

```
 new_list = list(map(lambda x: x * 2 , a))  
print(new_list)
```

```
 [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

◆ Using lambda function

```
[ ] for i in range(0,len(a)):  
    a[i]=2*a[i]  
    print(a)
```

```
 [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

LESSON FIVE

- **NumPy**

NumPy (or Numpy) is a Linear Algebra Library for Python, the reason it is so important for Data Science with Python is that almost all of the libraries in the PyData Ecosystem rely on NumPy as one of their main building blocks.

Numpy is also incredibly fast, as it has bindings to C libraries. For more info on why you would want to use Arrays instead of lists, check out this great [StackOverflow post](#).

We will only learn the basics of NumPy, to get started we need to install it!

→ **Importing `numpy` library**

```
[ ] import numpy as np
```

Numpy has many built-in functions and capabilities. We won't cover them all but instead we will focus on some of the most important aspects of Numpy: vectors, arrays, matrices, and number generation. Let's start by discussing arrays.

Numpy Arrays

NumPy arrays are the main way we will use Numpy throughout our course. Numpy arrays essentially come in two flavors: vectors and matrices. Vectors are strictly 1-d arrays and matrices are 2-d (but you should note a matrix can still have only one row or one column).

- **Creating NumPy Arrays**

- 1. From a Python List***

We can create an array by directly converting a list or list of lists:

2. Built-in Methods

There are lots of built-in ways to generate Arrays

```
[ ] my_list = [1,2,3]
    my_list

[ ] [1, 2, 3]

[ ] np.array(my_list)

[ ] array([1, 2, 3])

[ ] my_matrix = [[1,2,3],[4,5,6],[7,8,9]]
    my_matrix

[ ] [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

[ ] np.array(my_matrix)

[ ] array([[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]])
```

- arange

Return evenly spaced values within a given interval.

```
[ ] np.arange(0,10)

[ ] array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

[ ] np.arange(0,11,2)

[ ] array([ 0,  2,  4,  6,  8, 10])
```

- zeros and ones

Generate arrays of zeros or ones

```
[ ] np.zeros(3)
↳ array([0., 0., 0.])

[ ] np.zeros((5,5))
↳ array([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])

[ ] np.ones(3)
↳ array([1., 1., 1.])

[ ] np.ones((3,3))
↳ array([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
```

- linspace

Return evenly spaced numbers over a specified interval.

```
[ ] np.linspace(0,10,3)
↳ array([ 0.,  5., 10.])

[ ] np.linspace(0,10,50)
↳ array([ 0.,  0.20408163,  0.40816327,  0.6122449 ,  0.81632653,
        1.02040816,  1.2244898 ,  1.42857143,  1.63265306,  1.83673469,
        2.04081633,  2.24489796,  2.44897959,  2.65306122,  2.85714286,
        3.06122449,  3.26530612,  3.46938776,  3.67346939,  3.87755102,
        4.08163265,  4.28571429,  4.48979592,  4.69387755,  4.89795918,
        5.10204082,  5.30612245,  5.51020408,  5.71428571,  5.91836735,
        6.12244898,  6.32653061,  6.53061224,  6.73469388,  6.93877551,
        7.14285714,  7.34693878,  7.55102041,  7.75510204,  7.95918367,
        8.16326531,  8.36734694,  8.57142857,  8.7755102 ,  8.97959184,
        9.18367347,  9.3877551 ,  9.59183673,  9.79591837, 10.      ])
```

- eye

Creates an identity matrix

```
[ ] np.eye(4)

↳ array([[1., 0., 0., 0.],
         [0., 1., 0., 0.],
         [0., 0., 1., 0.],
         [0., 0., 0., 1.]])
```

3. *Random number array*

Numpy also has lots of ways to create random number arrays:

- `rand`

Create an array of the given shape and populate it with random samples from a uniform distribution over $[0, 1)$.

```
[ ] np.random.rand(2)

↳ array([0.60452511, 0.96348445])

[ ] np.random.rand(5,5)

↳ array([[0.03706106, 0.20121708, 0.43783284, 0.42680886, 0.02910492],
        [0.50689372, 0.90558677, 0.37877526, 0.78446399, 0.6625466 ],
        [0.73476794, 0.78279171, 0.38555063, 0.59559387, 0.13541721],
        [0.54818959, 0.59674716, 0.78568783, 0.08412492, 0.36005211],
        [0.33714127, 0.31071422, 0.23986093, 0.73942881, 0.32237019]])
```

- `randn`

Return a sample (or samples) from the "standard normal" distribution. Unlike `rand` which is uniform:

```
↳ np.random.randn(2)

↳ array([-0.59463209, 0.23905196])

[ ] np.random.randn(5,5)

↳ array([[ 0.97667587,  0.32440485, -1.57531587, -0.83655875, -0.06697723],
        [-0.23190846,  0.23319799,  0.21997686,  0.21922616,  2.66524901],
        [ 1.56858627,  1.54675016, -0.40417972, -0.31535543,  1.95878655],
        [-0.52796245, -3.77431214,  0.29885402, -0.44400648,  0.58508331],
        [-1.29222612,  0.91072417, -0.59116667, -1.57213122, -2.21795473]])
```

- `randint`

```
[ ] np.random.randint(1,100)

↳ 29

[ ] np.random.randint(1,100,10)

↳ array([18, 75, 75, 59, 31, 33, 83, 89, 14, 34])
```

Return random integers from [low, high) ; low (inclusive) to high (exclusive).

- **Array Attributes and Methods**

```
[ ] arr = np.arange(25)
    ranarr = np.random.randint(0,50,10)

[ ] arr

↳ array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24])

[ ] ranarr

↳ array([41,  7, 48, 14,  8, 28,  6,  8, 24, 24])
```

1. **Reshape**

Returns an array containing the same data with a new shape.

```
[ ] arr.reshape(5,5)

↳ array([[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24]])
```

2. **max,min,argmax,argmin**

These are useful methods for finding maximum , minimum values and their index locations using argmax or argmin.

```
[ ] ranarr
↳ array([41,  7, 48, 14,  8, 28,  6,  8, 24, 24])

[ ] ranarr.max()
↳ 48

[ ] ranarr.argmax()
↳ 2

[ ] ranarr.min()
↳ 6

[ ] ranarr.argmin()
↳ 6
```

3. Shape

Shape is an attribute of arrays (not a method). It returns the shape of the array.

```
[ ] arr
↳ array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24])

[ ] # Vector
    arr.shape
↳ (25,)

[ ] # Notice the two sets of brackets
    arr.reshape(1,25)

↳ array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
        16, 17, 18, 19, 20, 21, 22, 23, 24]])

[ ] arr.reshape(1,25).shape
↳ (1, 25)
```

```
▶ arr.reshape(25,1)

array([[ 0],
       [ 1],
       [ 2],
       [ 3],
       [ 4],
       [ 5],
       [ 6],
       [ 7],
       [ 8],
       [ 9],
      [10],
      [11],
      [12],
      [13],
      [14],
      [15],
      [16],
      [17],
      [18],
      [19],
      [20],
      [21],
      [22],
      [23],
      [24]])

[ ] arr.reshape(25,1).shape

▶ (25, 1)
```

4. *dtype*

You can also grab the data type of the object in the array:

```
[ ] arr.dtype

▶ dtype('int64')
```

- **NumPy Operations**

1. *Arithmetic*

You can easily perform array with array arithmetic, or scalar with array arithmetic. Let's see some examples:

```
[ ] import numpy as np
    arr = np.arange(0,10)

[ ] arr + arr
↳ array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

[ ] arr * arr
↳ array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])

[ ] arr - arr
↳ array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

[ ] # Warning on division by zero, but not an error!
    # Just replaced with nan
    arr/arr
↳ /usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: invalid value encountered in true_divide
    This is separate from the ipykernel package so we can avoid doing imports until
    array([nan,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])

[ ] # Also warning, but not an error instead infinity
    1/arr
↳ /usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:2: RuntimeWarning: divide by zero encountered in true_divide
    array([      inf,  1.,          0.5,      0.33333333,  0.25,
           0.2,      0.16666667,  0.14285714,  0.125,      0.11111111])

[ ] arr**3
↳ array([ 0,  1,  8, 27, 64, 125, 216, 343, 512, 729])
```

2. Universal Array Functions

Numpy comes with many universal array functions , which are essentially just mathematical operations you can use to perform the operation across the array. Let's show some common ones:

```
[ ] #Taking Square Roots
np.sqrt(arr)

array([0.          , 1.          , 1.41421356, 1.73205081, 2.
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.
       ])

[ ] #Calculating exponential (e^)
np.exp(arr)

array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03])

[ ] np.max(arr) #same as arr.max()

9

[ ] np.sin(arr)

array([ 0.          ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])

[ ] np.log(arr)

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: RuntimeWarning: divide by zero encountered in log
"""Entry point for launching an IPython kernel.
array([      -inf,  0.          ,  0.69314718,  1.09861229,  1.38629436,
       1.60943791,  1.79175947,  1.94591015,  2.07944154,  2.19722458])
```

- **NumPy Indexing and Selection**

```
[ ] import numpy as np

#Creating sample array
arr = np.arange(0,11)

[ ] #Show
arr

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

1. **Bracket Indexing and Selection**

The simplest way to pick one or some elements of an array looks very similar to python lists:


```
[ ] #Get a value at an index
arr[8]
```

```
↳ 8
```

```
[ ] #Get values in a range
arr[1:5]
```

```
↳ array([1, 2, 3, 4])
```

```
[ ] #Get values in a range
arr[0:5]
```

```
↳ array([0, 1, 2, 3, 4])
```

2. Indexing a 2D array (matrices)

The general format is `arr_2d [row] [col]` or `arr_2d [row,col]` . I recommend using the comma notation for clarity.

```
▶ arr_2d = np.array([[5,10,15],[20,25,30],[35,40,45]])
#Show
arr_2d
```

```
↳ array([[ 5, 10, 15],
        [20, 25, 30],
        [35, 40, 45]])
```

```
[ ] #Indexing row
arr_2d[1]
```

```
↳ array([20, 25, 30])
```

```
[ ] # Format is arr_2d[row][col] or arr_2d[row,col]

# Getting individual element value
arr_2d[1][0]
```

```
↳ 20
```

```
[ ] # Getting individual element value
arr_2d[1,0]
```

```
↳ 20
```

```
[ ] # Getting individual element value
arr_2d[1,0]
```

```
↳ 20
```

```
[ ] # 2D array slicing

#Shape (2,2) from top right corner
arr_2d[:2,1:]
```

```
↳ array([[10, 15],
        [25, 30]])
```

```
[ ] #Shape bottom row
arr_2d[2]
```

```
↳ array([35, 40, 45])
```

```
[ ] #Shape bottom row
arr_2d[2,:]
```

```
↳ array([35, 40, 45])
```

3. Fancy Indexing

Fancy indexing allows you to select entire rows or columns out of order.

```
[ ] #Set up matrix
arr2d = np.zeros((10,10))
```

```
[ ] arr2d
```

```
↳ array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

```
[ ] #Length of array
arr_length = arr2d.shape[1]
```

```
[ ] arr_length
```

```
↳ 10
```

4. Selection

```
[ ] #Set up array

for i in range(arr_length):
    arr2d[i] = i

arr2d

[ ] array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
          [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
          [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
          [3., 3., 3., 3., 3., 3., 3., 3., 3., 3.],
          [4., 4., 4., 4., 4., 4., 4., 4., 4., 4.],
          [5., 5., 5., 5., 5., 5., 5., 5., 5., 5.],
          [6., 6., 6., 6., 6., 6., 6., 6., 6., 6.],
          [7., 7., 7., 7., 7., 7., 7., 7., 7., 7.],
          [8., 8., 8., 8., 8., 8., 8., 8., 8., 8.],
          [9., 9., 9., 9., 9., 9., 9., 9., 9., 9.]])

[ ] arr2d[[2,4,6,8]]

[ ] array([[2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
          [4., 4., 4., 4., 4., 4., 4., 4., 4., 4.],
          [6., 6., 6., 6., 6., 6., 6., 6., 6., 6.],
          [8., 8., 8., 8., 8., 8., 8., 8., 8., 8.]])

[ ] #Allows in any order
arr2d[[6,4,2,7]]

[ ] array([[6., 6., 6., 6., 6., 6., 6., 6., 6., 6.],
          [4., 4., 4., 4., 4., 4., 4., 4., 4., 4.],
          [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
          [7., 7., 7., 7., 7., 7., 7., 7., 7., 7.]])
```

5. Using brackets for selection based on comparison operators.

```
[ ] arr = np.arange(1,11)
arr

↳ array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

[ ] arr > 4

↳ array([False, False, False, False,  True,  True,  True,  True,  True,
        True])

[ ] bool_arr = arr>4

[ ] bool_arr

↳ array([False, False, False, False,  True,  True,  True,  True,  True,
        True])

[ ] arr[bool_arr]

↳ array([ 5,  6,  7,  8,  9, 10])

[ ] arr[arr>2]

↳ array([ 3,  4,  5,  6,  7,  8,  9, 10])

[ ] x = 2
arr[arr>x]

↳ array([ 3,  4,  5,  6,  7,  8,  9, 10])
```

- **Pandas**

In this section, we will learn how to use pandas for data analysis. You can think of pandas as an extremely powerful version of Excel, with a lot more features.

Series

The first main data type we will learn about for pandas is the Series data type. First import `pandas` and explore the Series object.

A Series is very similar to a NumPy array (in fact it is built on top of the NumPy array object). What differentiates the NumPy array from a Series, is that a Series can have axis labels, meaning it can be indexed by a label, instead of just a number location. It also doesn't need to hold numeric data, it can hold any arbitrary Python Object. Let's explore this concept through some examples:

```
[ ] import numpy as np
    import pandas as pd
```

1. Creating a Series

You can convert a list, numpy array, or dictionary to a Series:

- Using Lists

```
▶ labels = ['a','b','c']
  my_list = [10,20,30]
  arr = np.array([10,20,30])
  d = {'a':10,'b':20,'c':30}
```

```
[ ] pd.Series(data=my_list)
```

```
0    10
1    20
2    30
dtype: int64
```

```
[ ] pd.Series(data=my_list,index=labels)
```

```
a    10
b    20
c    30
dtype: int64
```

```
[ ] pd.Series(my_list,labels)
```

```
a    10
b    20
c    30
dtype: int64
```

- Using NumPy Arrays

```
[ ] pd.Series(arr)
```

```
0    10
1    20
2    30
dtype: int64
```

```
[ ] pd.Series(arr,labels)
```

```
a    10
b    20
c    30
dtype: int64
```

- Using Dictionary

```
pd.Series(d)
```

```
a    10
b    20
c    30
dtype: int64
```

2. Data in a Series

A pandas Series can hold a variety of object types:

```
[ ] pd.Series(data=labels)
```

```
0    a
1    b
2    c
dtype: object
```

```
[ ] # Even functions (although unlikely that you will use this)
    pd.Series([sum,print,len])
```

```
0    <built-in function sum>
1    <built-in function print>
2    <built-in function len>
dtype: object
```

3. Using an Index

The key to using a Series is understanding its index. Pandas makes use of these index names or numbers by allowing for fast lookups of information (works like a hashtable or dictionary).

Let's see some examples of how to grab information from a Series. Let us create two series, `ser1` and `ser2` :

```
[ ] ser1 = pd.Series([1,2,3,4],index = ['USA', 'Germany', 'USSR', 'Japan'])

[ ] ser1

USA      1
Germany  2
USSR     3
Japan    4
dtype: int64

[ ] ser2 = pd.Series([1,2,5,4],index = ['USA', 'Germany', 'Italy', 'Japan'])

[ ] ser2

USA      1
Germany  2
Italy    5
Japan    4
dtype: int64

[ ] ser1['USA']

1
```

→ Operations can also be done based on index:

```
ser1 + ser2

Germany    4.0
Italy      NaN
Japan      8.0
USA        2.0
USSR       NaN
dtype: float64
```

DataFrames

DataFrames are the workhouse of pandas and are directly inspired by the R programming language. We can think of a DataFrame as a bunch of Series objects put together to share the same index. Let's use pandas to explore this topic!


```
[ ] from numpy.random import randn

[ ] np.random.seed(11)
    df = pd.DataFrame(randn(5,4),index='A B C D E'.split(),columns='W X Y Z'.split())

[ ] df
```

	W	X	Y	Z
A	1.749455	-0.286073	-0.484565	-2.653319
B	-0.008285	-0.319631	-0.536629	0.315403
C	0.421051	-1.065603	-0.886240	-0.475733
D	0.689682	0.561192	-1.305549	-1.119475
E	0.736837	1.574634	-0.031075	-0.683447

1. Selection and Indexing

Various methods to grab data from a DataFrame

```
[ ] df['W']
```

A	1.749455
B	-0.008285
C	0.421051
D	0.689682
E	0.736837

Name: W, dtype: float64

```
[ ] # Pass a list of column names
    df[['W','Z']]
```

	W	Z
A	1.749455	-2.653319
B	-0.008285	0.315403
C	0.421051	-0.475733
D	0.689682	-1.119475
E	0.736837	-0.683447


```
[ ] # SQL Syntax (NOT RECOMMENDED!)
    df.W
```

A	1.749455
B	-0.008285
C	0.421051
D	0.689682
E	0.736837

Name: W, dtype: float64

DataFrame Columns are just Series :


```
[ ] type(df['W'])
```

 pandas.core.series.Series

- Creating a new column:

```
[ ] df['new'] = df['W'] + df['Y']
```


```
[ ] df
```



	W	X	Y	Z	new
A	1.749455	-0.286073	-0.484565	-2.653319	1.264890
B	-0.008285	-0.319631	-0.536629	0.315403	-0.544914
C	0.421051	-1.065603	-0.886240	-0.475733	-0.465189
D	0.689682	0.561192	-1.305549	-1.119475	-0.615866
E	0.736837	1.574634	-0.031075	-0.683447	0.705762

- Removing Columns

```
[ ] df.drop('new',axis=1)
```



	W	X	Y	Z
A	1.749455	-0.286073	-0.484565	-2.653319
B	-0.008285	-0.319631	-0.536629	0.315403
C	0.421051	-1.065603	-0.886240	-0.475733
D	0.689682	0.561192	-1.305549	-1.119475
E	0.736837	1.574634	-0.031075	-0.683447


```
[ ] # Not inplace unless specified!  
df
```



	W	X	Y	Z	new
A	1.749455	-0.286073	-0.484565	-2.653319	1.264890
B	-0.008285	-0.319631	-0.536629	0.315403	-0.544914
C	0.421051	-1.065603	-0.886240	-0.475733	-0.465189
D	0.689682	0.561192	-1.305549	-1.119475	-0.615866
E	0.736837	1.574634	-0.031075	-0.683447	0.705762

```
[ ] df.drop('new',axis=1,inplace=True)
```

 df



	W	X	Y	Z
A	1.749455	-0.286073	-0.484565	-2.653319
B	-0.008285	-0.319631	-0.536629	0.315403
C	0.421051	-1.065603	-0.886240	-0.475733
D	0.689682	0.561192	-1.305549	-1.119475
E	0.736837	1.574634	-0.031075	-0.683447

- Removing Rows


```
[ ] df.drop('E',axis=0)
```



	W	X	Y	Z
A	1.749455	-0.286073	-0.484565	-2.653319
B	-0.008285	-0.319631	-0.536629	0.315403
C	0.421051	-1.065603	-0.886240	-0.475733
D	0.689682	0.561192	-1.305549	-1.119475


- Selecting Rows
 - By passing index

```
[ ] df.iloc[2]
```



W 0.421051
X -1.065603
Y -0.886240
Z -0.475733
Name: C, dtype: float64

```
[ ] df.iloc[0:2]
```



	W	X	Y	Z
A	1.749455	-0.286073	-0.484565	-2.653319
B	-0.008285	-0.319631	-0.536629	0.315403

- By passing position

```
[ ] df.loc['A']
```

```
W    1.749455  
X   -0.286073  
Y   -0.484565  
Z   -2.653319  
Name: A, dtype: float64
```

```
df.loc[:, 'W']
```

```
A    1.749455  
B   -0.008285  
C    0.421051  
D    0.689682  
E    0.736837  
Name: W, dtype: float64
```

- Printing column values by passing index

```
df.iloc[:,1]
```

```
A   -0.286073  
B   -0.319631  
C   -1.065603  
D    0.561192  
E    1.574634  
Name: X, dtype: float64
```

```
[ ] df.iloc[:,1:3]
```

	X	Y
A	-0.286073	-0.484565
B	-0.319631	-0.536629
C	-1.065603	-0.886240
D	0.561192	-1.305549
E	1.574634	-0.031075

- Selecting subset of rows and columns

```
df.loc['B','Y']
```

-0.53662936223473

```
[ ] df.loc[['A','B'],['W','Y']]
```

	W	Y
A	1.749455	-0.484565
B	-0.008285	-0.536629

- Selecting particular rows and column

```
[ ] df.iloc[[1,3],[1,2,3]]
```

	X	Y	Z
B	-0.319631	-0.536629	0.315403
D	0.561192	-1.305549	-1.119475

2. Conditional Selection

An important feature of pandas is conditional selection using bracket notation, very similar to numpy:

For two conditions you can use `||` and `&` with parenthesis:

```
[ ] df
```

	W	X	Y	Z
A	1.749455	-0.286073	-0.484565	-2.653319
B	-0.008285	-0.319631	-0.536629	0.315403
C	0.421051	-1.065603	-0.886240	-0.475733
D	0.689682	0.561192	-1.305549	-1.119475
E	0.736837	1.574634	-0.031075	-0.683447

```
[ ] df>0
```

	W	X	Y	Z
A	True	False	False	False
B	False	False	False	True
C	True	False	False	False
D	True	True	False	False
E	True	True	False	False

```
[ ] df[df>0]
```

	W	X	Y	Z
A	1.749455	NaN	NaN	NaN
B	NaN	NaN	NaN	0.315403
C	0.421051	NaN	NaN	NaN
D	0.689682	0.561192	NaN	NaN
E	0.736837	1.574634	NaN	NaN

```
[ ] df[df['W']>0]
```

	W	X	Y	Z
A	1.749455	-0.286073	-0.484565	-2.653319
C	0.421051	-1.065603	-0.886240	-0.475733
D	0.689682	0.561192	-1.305549	-1.119475
E	0.736837	1.574634	-0.031075	-0.683447

```
[ ] df[df['W']>0]['Y']
```

A -0.484565
C -0.886240
D -1.305549
E -0.031075
Name: Y, dtype: float64

```
[ ] df[df['W']>0][['Y','X']]
```

	Y	X
A	-0.484565	-0.286073
C	-0.886240	-1.065603
D	-1.305549	0.561192
E	-0.031075	1.574634

```
[ ] df[(df['W']>0) & (df['Y'] > 1)]
```

	W	X	Y	Z
--	---	---	---	---

- More index details


```
[ ] df
```



	W	X	Y	Z
A	1.749455	-0.286073	-0.484565	-2.653319
B	-0.008285	-0.319631	-0.536629	0.315403
C	0.421051	-1.065603	-0.886240	-0.475733
D	0.689682	0.561192	-1.305549	-1.119475
E	0.736837	1.574634	-0.031075	-0.683447

```
[ ] newind = 'CA NY WY OR CO'.split()
```

```
[ ] df['States'] = newind
```

```
[ ] df
```



	W	X	Y	Z	States
A	1.749455	-0.286073	-0.484565	-2.653319	CA
B	-0.008285	-0.319631	-0.536629	0.315403	NY
C	0.421051	-1.065603	-0.886240	-0.475733	WY
D	0.689682	0.561192	-1.305549	-1.119475	OR
E	0.736837	1.574634	-0.031075	-0.683447	CO

```
[ ] df.set_index('States')
```



	W	X	Y	Z
States				
CA	1.749455	-0.286073	-0.484565	-2.653319
NY	-0.008285	-0.319631	-0.536629	0.315403
WY	0.421051	-1.065603	-0.886240	-0.475733
OR	0.689682	0.561192	-1.305549	-1.119475
CO	0.736837	1.574634	-0.031075	-0.683447

```
[ ] df
```



	W	X	Y	Z	States
A	1.749455	-0.286073	-0.484565	-2.653319	CA
B	-0.008285	-0.319631	-0.536629	0.315403	NY
C	0.421051	-1.065603	-0.886240	-0.475733	WY
D	0.689682	0.561192	-1.305549	-1.119475	OR
E	0.736837	1.574634	-0.031075	-0.683447	CO

3. Multi-Index and Index Hierarchy

- Creating Multi-Index DataFrame :

```
[ ] # Index Levels
    outside = ['G1','G1','G1','G2','G2','G2']
    inside = [1,2,3,1,2,3]
    hier_index = list(zip(outside,inside))
    hier_index = pd.MultiIndex.from_tuples(hier_index)

[ ] hier_index

MultiIndex([( 'G1', 1),
            ( 'G1', 2),
            ( 'G1', 3),
            ( 'G2', 1),
            ( 'G2', 2),
            ( 'G2', 3)],
           )

[ ] df = pd.DataFrame(np.random.randn(6,2),index=hier_index,columns=['A','B'])
    df
```

		A	B
G1	1	1.095630	-0.309577
	2	0.725752	1.549072
	3	0.630080	0.073493
G2	1	0.732271	-0.642575
	2	-0.178093	-0.573955
	3	-0.204375	-0.486495

- Index Hierarchy :

For index hierarchy we use `df.loc[]` , if this was on the columns axis, you would just use normal bracket notation `df[]` . Calling one level of the index returns the sub-dataframe:

```
[ ] df.loc['G1']

      A      B
1  1.095630 -0.309577
2  0.725752  1.549072
3  0.630080  0.073493

[ ] df.loc['G1'].loc[1]

A    1.095630
B   -0.309577
Name: 1, dtype: float64

[ ] df.index.names

FrozenList([None, None])
```

```
[ ] df.index.names = ['Group', 'Num']
```

df

		A	B
Group Num			
G1	1	1.095630	-0.309577
	2	0.725752	1.549072
	3	0.630080	0.073493
G2	1	0.732271	-0.642575
	2	-0.178093	-0.573955
	3	-0.204375	-0.486495

4. Missing Data

```
[ ] df = pd.DataFrame({'A':[1,2,np.nan],  
                        'B':[5,np.nan,np.nan],  
                        'C':[1,2,3]})
```

df

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3

```
[ ] df.dropna()
```

	A	B	C
0	1.0	5.0	1

```
df.dropna(axis=1)
```

	C
0	1
1	2
2	3

Keep only the rows with at least 1 non-NA values.

Few convenient methods to deal with Missing Data in pandas:

- Keep only the rows with at least 1 non-NA values.

```
[ ] df.dropna(thresh=1)
```

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3

Rows having atleast 2 non - NA values

- Rows having at least 2 non - NA values

```
[ ] df.dropna(thresh=2)
```

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2

```
[ ] df.fillna(value='FILL VALUE')
```

	A	B	C
0	1	5	1
1	2	FILL VALUE	2
2	FILL VALUE	FILL VALUE	3

```
[ ] df['A'].fillna(value=df['A'].mean())
```

	A
0	1.0
1	2.0
2	1.5

Name: A, dtype: float64

5. Groupby

The groupby method allows to group rows of data together and call aggregate functions

Use the `.groupby()` method to group rows together based on column name.

For instance, we can group based on `"Company"` . This will create a

DataFrameGroupBy object :

```
[ ] import pandas as pd
# Create dataframe
data = {'Company': ['GOOG', 'GOOG', 'MSFT', 'MSFT', 'FB', 'FB'],
        'Person': ['Sam', 'Charlie', 'Amy', 'Vanessa', 'Carl', 'Sarah'],
        'Sales': [200, 120, 340, 124, 243, 350]}
```


```
[ ] df = pd.DataFrame(data)
```

```
[ ] df
```



	Company	Person	Sales
0	GOOG	Sam	200
1	GOOG	Charlie	120
2	MSFT	Amy	340
3	MSFT	Vanessa	124
4	FB	Carl	243
5	FB	Sarah	350

- Save this object as a new variable:

```
 df.groupby('Company')
```

```
[ ] by_comp = df.groupby("Company")
```

- Call aggregate methods off the object:

```
[ ] by_comp.all()
```



	Person	Sales
Company		
FB	True	True
GOOG	True	True
MSFT	True	True

```
[ ] by_comp.mean()
```



	Sales
Company	
FB	296.5
GOOG	160.0
MSFT	232.0

```
[ ] by_comp.std()
```



	Sales
Company	
FB	75.660426
GOOG	56.568542
MSFT	152.735065

```
[ ] by_comp.min()
```



	Person	Sales
Company		
FB	Carl	243
GOOG	Charlie	120
MSFT	Amy	124

```
[ ] by_comp.describe()
```

Sales								
	count	mean	std	min	25%	50%	75%	max
Company								
FB	2.0	296.5	75.660426	243.0	269.75	296.5	323.25	350.0
GOOG	2.0	160.0	56.568542	120.0	140.00	160.0	180.00	200.0
MSFT	2.0	232.0	152.735065	124.0	178.00	232.0	286.00	340.0

```
[ ] by_comp.describe().transpose()
```

Company		FB	GOOG	MSFT
Sales	count	2.000000	2.000000	2.000000
	mean	296.500000	160.000000	232.000000
	std	75.660426	56.568542	152.735065
	min	243.000000	120.000000	124.000000
	25%	269.750000	140.000000	178.000000
	50%	296.500000	160.000000	232.000000
	75%	323.250000	180.000000	286.000000
	max	350.000000	200.000000	340.000000

```
[ ] by_comp.describe().transpose()['GOOG']
```

Sales	count	2.000000
	mean	160.000000
	std	56.568542
	min	120.000000
	25%	140.000000
	50%	160.000000
	75%	180.000000
	max	200.000000

Name: GOOG, dtype: float64

6. Merging, Joining, and Concatenating

There are 3 main ways of combining DataFrames together: Merging, Joining and Concatenating.

```
[ ] df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                        'B': ['B0', 'B1', 'B2', 'B3'],
                        'C': ['C0', 'C1', 'C2', 'C3'],
                        'D': ['D0', 'D1', 'D2', 'D3']},
                        index=[0, 1, 2, 3])


[ ] df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                        'B': ['B4', 'B5', 'B6', 'B7'],
                        'C': ['C4', 'C5', 'C6', 'C7'],
                        'D': ['D4', 'D5', 'D6', 'D7']},
                        index=[4, 5, 6, 7])

[ ] df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                        'B': ['B8', 'B9', 'B10', 'B11'],
                        'C': ['C8', 'C9', 'C10', 'C11'],
                        'D': ['D8', 'D9', 'D10', 'D11']},
                        index=[8, 9, 10, 11])
```

- **Concatenation**


Concatenation basically glues together DataFrames. Keep in mind that dimensions should match along the axis you are concatenating on. You can use `pd.concat` and pass in a list of DataFrames to concatenate together.

[] df1




	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

[] df2




	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

[] df3



	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

[] pd.concat([df1,df2,df3])



	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

pd.concat([df1,df2,df3],axis=1)

	A	B	C	D	A	B	C	D	A	B	C	D
0	A0	B0	C0	D0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	A4	B4	C4	D4	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	A5	B5	C5	D5	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	A6	B6	C6	D6	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	A7	B7	C7	D7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A8	B8	C8	D8
9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A9	B9	C9	D9
10	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A10	B10	C10	D10
11	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A11	B11	C11	D11

- **Merging**

The merge function allows you to merge DataFrames together using a similar logic as merging SQL Tables together. For example:

```
[ ] left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                        'A': ['A0', 'A1', 'A2', 'A3'],
                        'B': ['B0', 'B1', 'B2', 'B3']})


right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']})
```

```
[ ] left
```





	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2
3	K3	A3	B3

```
[ ] right
```



	key	C	D
0	K0	C0	D0
1	K1	C1	D1
2	K2	C2	D2
3	K3	C3	D3


```
 pd.merge(left,right,how='inner',on='key')
```



	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3


Looking at a little more complicated example:

```
[ ] pd.merge(left, right, on=['key1', 'key2'])
```




	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

```
[ ] pd.merge(left, right, how='inner', on=['key1', 'key2'])
```




	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

```
[ ] pd.merge(left, right, how='outer', on=['key1', 'key2'])
```




	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN
5	K2	K0	NaN	NaN	C3	D3

```
[ ] pd.merge(left, right, how='left', on=['key1', 'key2'])
```



	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN

```
[ ] pd.merge(left, right, how='outer', on=['key1', 'key2'])
```



	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN
5	K2	K0	NaN	NaN	C3	D3


- **Joining**

Joining is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame.

```
[ ] left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                        'B': ['B0', 'B1', 'B2']},
                        index=['K0', 'K1', 'K2'])


right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                     'D': ['D0', 'D2', 'D3']},
                     index=['K0', 'K2', 'K3'])
```

```
[ ] left
```




	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

```
[ ] right
```




	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

```
[ ] left.join(right)
```



	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

```
[ ] left.join(right, how='outer')
```



	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

7. Operations

There are lots of operations with pandas that will be really useful to you, but don't fall into any distinct category.

```
df = pd.DataFrame({'col1':[1,2,3,4], 'col2':[444,555,666,444], 'col3':['abc','def','ghi','xyz']})
df.head()
```

	col1	col2	col3
0	1	444	abc
1	2	555	def
2	3	666	ghi
3	4	444	xyz

- Unique Values

```
[ ] df['col2'].unique()
```

```
array([444, 555, 666])
```

```
[ ] df['col2'].nunique()
```

```
3
```

```
[ ] df['col2'].value_counts()
```

```
444    2
555    1
666    1
Name: col2, dtype: int64
```

- Selecting Data

```
[ ] #Select from DataFrame using criteria from multiple columns
newdf = df[(df['col1']>2) & (df['col2']==444)]
```

```
[ ] newdf
```

```
col1  col2  col3
3     4   444   xyz
```

- Applying Functions

```
[ ] def times2(x):  
    return x*2  
  
[ ] df['col1'].apply(times2)  
  
0    2  
1    4  
2    6  
3    8  
Name: col1, dtype: int64  
  
df['col1'].sum()  
  
10
```

- Permanently Removing a Column

```
[ ] del df['col1']  
  
[ ] df  
  
   col2 col3  
0   444  abc  
1   555  def  
2   666  ghi  
3   444  xyz
```

- Get column and index names:

```
[ ] df.columns  
Index(['col2', 'col3'], dtype='object')  
  
[ ] df.index  
RangeIndex(start=0, stop=4, step=1)
```

- Sorting and Ordering a DataFrame:

```
[ ] df
```

	col2	col3
0	444	abc
1	555	def
2	666	ghi
3	444	xyz

```
[ ] df.sort_values(by='col2') #inplace=False by default
```

	col2	col3
0	444	abc
3	444	xyz
1	555	def
2	666	ghi

- Find Null Values or Check for Null Values

```
[ ] df.isnull()
```

	col2	col3
0	False	False
1	False	False
2	False	False
3	False	False

```
[ ] 1# Drop rows with NaN Values
df.dropna()
```

	col2	col3
0	444	abc
1	555	def
2	666	ghi
3	444	xyz

- Filling in NaN values with some other element


```
[ ] import numpy as np

[ ] df = pd.DataFrame({'col1':[1,2,3,np.nan],
                        'col2':[np.nan,555,666,444],
                        'col3':['abc','def','ghi','xyz']})

df.head()
```

	col1	col2	col3
0	1.0	NaN	abc
1	2.0	555.0	def
2	3.0	666.0	ghi
3	NaN	444.0	xyz

```
[ ] df.fillna('FILL')
```

	col1	col2	col3
0	1	FILL	abc
1	2	555	def
2	3	666	ghi
3	FILL	444	xyz

8. Data Input and Output

Pandas can read a variety of file types using its `pd.read_` methods. Let's take a look at the most common data types:

Uploading our kaggle json file

```
[ ] from google.colab import files
my_file=files.upload()
```

Setting up our kaggle credentials

```
import os
import pandas as pd
cred = pd.read_json("kaggle.json",lines=True)
os.environ['KAGGLE_USERNAME'] = cred.iloc[0][0]
os.environ['KAGGLE_KEY'] = cred.iloc[0][1]
```

- CSV
 - Unzipping and downloading our data

```
[ ] !kaggle datasets download -d vishnuvarthanrao/windows-store
```

Downloading windows-store.zip to /content
0% 0.00/93.1k [00:00<?, ?B/s]
100% 93.1k/93.1k [00:00<00:00, 27.5MB/s]

Unzipping and downloading our data

```
[ ] !unzip "*.zip"
```

Archive: windows-store.zip
inflating: msft.csv

- CSV Input

```
df = pd.read_csv('/content/msft.csv')
df
```

	Name	Rating	No of people Rated	Category	Date	Price
0	Dynamic Reader	3.5	268	Books	07-01-2014	Free
1	Chemistry, Organic Chemistry and Biochemistry...	3.0	627	Books	08-01-2014	Free
2	BookViewer	3.5	593	Books	29-02-2016	Free
3	Brick Instructions	3.5	684	Books	30-01-2018	Free
4	Introduction to Python Programming by GoLearn...	2.0	634	Books	30-01-2018	Free
...
5317	JS King	1.0	720	Developer Tools	19-07-2018	₹ 269.00
5318	MQTTSniffer	2.5	500	Developer Tools	10-04-2017	₹ 64.00
5319	Dev Utils - JSON, CSV and XML	4.0	862	Developer Tools	18-11-2019	₹ 269.00
5320	Simply Text	4.0	386	Developer Tools	23-01-2014	₹ 219.00
5321	NaN	NaN	948	NaN	NaN	NaN

5322 rows × 6 columns

- CSV Output

```
[ ] df.to_csv('example',index=False)
```

- Excel

Pandas can read and write excel files, keep in mind, this only imports data. Not formulas or images, having images or macros may cause this read_excel method to crash.

- Downloading dataset from kaggle

```
[ ] !kaggle datasets download -d prashdash112/african-ngo-data
```

Downloading african-ngo-data.zip to /content
 0% 0.00/97.8k [00:00<?, ?B/s]
 100% 97.8k/97.8k [00:00<00:00, 29.2MB/s]

```
[ ] !unzip "*.zip"
```

Archive: african-ngo-data.zip
 inflating: Data2 (1).xls

Archive: windows-store.zip
 replace msft.csv? [y]es, [n]o, [A]ll, [N]one, [r]ename: n

2 archives were successfully processed.

- Excel Input

```
[ ] pd.read_excel('/content/Data2 (1).xls')
```

	pid	w3_region	w3_age	w3_gen	w3_b1	w2_region	hhid	w2_age	w2_gen	w2_b1	w1_region	w1_gen	w1_age	w1_smk_type	w2_smk_type	w3_smk_type
0	23	Tbilisi	82	Male	No, I did not	Tbilisi	23	82	Male	No, I did not	Tbilisi	Male	81	Filtered cigarettes	Filtered cigarettes only	Unfiltered cigarettes only
1	26	Kutaisi	45	Male	No, I did not	Kutaisi	26	44	Male	No, I did not	Kutaisi	Male	41	Filtered cigarettes	Filtered cigarettes only	Filtered cigarettes only
2	28	Kutaisi	47	Male	No, I did not	Kutaisi	28	46	Male	No, I did not	Kutaisi	Male	45	Filtered cigarettes	Filtered cigarettes only	Filtered cigarettes only
3	29	Gori	29	Male	Yes I gave up	Gori	29	30	Male	No, I did not	Gori	Male	27	Filtered cigarettes	Filtered cigarettes only	NaN
4	31	Gori	43	Male	No, I did not	Gori	31	42	Male	No, I did not	Gori	Male	41	Filtered cigarettes	Filtered cigarettes only	Filtered cigarettes only
...
1701	4334	Tbilisi	47	Female	No, I did not	Tbilisi	4334	44	Male	No, I did not	Tbilisi	Male	43	Only other tobacco	Filtered cigarettes only	Filtered cigarettes only
1702	4337	Tbilisi	32	Male	No, I did not	Tbilisi	4337	32	Male	No, I did not	Tbilisi	Male	29	Filtered cigarettes	Filtered cigarettes only	Filtered cigarettes only
1703	4343	Akhaltzikhe	41	Male	No, I did not	Akhaltzikhe	4343	40	Male	No, I did not	Akhaltzikhe	Male	38	Filtered cigarettes	Unfiltered cigarettes only	RYO only
1704	4344	Akhaltzikhe	52	Male	No, I did not	Akhaltzikhe	4344	52	Male	No, I did not	Akhaltzikhe	Male	51	Filtered cigarettes	Filtered cigarettes only	RYO only
1705	4345	Akhaltzikhe	50	Female	No, I did not	Akhaltzikhe	4345	50	Female	No, I did not	Akhaltzikhe	Female	48	Both filtered & non-filtered	Filtered cigarettes only	Filtered cigarettes only

1706 rows x 16 columns

- Excel Output

```
[ ] df.to_excel('Excel_Sample.xlsx')
```

- HTML

Pandas read_html function will read tables off of a webpage and return a list of DataFrame objects:

- HTML input

```
[ ] df = pd.read_html('http://www.fdic.gov/bank/individual/failed/banklist.html')
```

```
[ ] df[0]
```

	Bank Name	City	ST	CERT	Acquiring Institution	Closing Date
0	The First State Bank	Barboursville	WV	14361	MVB Bank, Inc.	April 3, 2020
1	Ericson State Bank	Ericson	NE	18265	Farmers and Merchants Bank	February 14, 2020
2	City National Bank of New Jersey	Newark	NJ	21111	Industrial Bank	November 1, 2019
3	Resolute Bank	Maumee	OH	58317	Buckeye State Bank	October 25, 2019
4	Louisa Community Bank	Louisa	KY	58112	Kentucky Farmers Bank Corporation	October 25, 2019
...
556	Superior Bank, FSB	Hinsdale	IL	32646	Superior Federal, FSB	July 27, 2001
557	Malta National Bank	Malta	OH	6629	North Valley Bank	May 3, 2001
558	First Alliance Bank & Trust Co.	Manchester	NH	34264	Southern New Hampshire Bank & Trust	February 2, 2001
559	National State Bank of Metropolis	Metropolis	IL	3815	Banterra Bank of Marion	December 14, 2000
560	Bank of Honolulu	Honolulu	HI	21029	Bank of the Orient	October 13, 2000

561 rows x 6 columns

LESSON SIX

- **OS Library**

→ Importing os library

```
[ ] import os
```

→ **getcwd()** : Returns the Current Working Directory of the file used.

```
[ ] os.getcwd()
```

```
👤 '/content'
```

→ **makedirs()** : Creates a new directory.

```
[ ] os.makedirs("/content/My_new_dir")
```

→ **makedirs()** : Creates directories recursively.

```
[ ] os.makedirs("/content/My_new_dir2/inner_dir1")
```

→ **chdir()** : Change the Current Working directory.

```
[ ] os.chdir("/content/My_new_dir")
```

```
▶ # lets check now which is the current directory
```

```
os.getcwd()
```

```
👤 '/content/My_new_dir'
```

→ We can directly make a new directory in the current working directory without mentioning the preceding path.

```
[ ] os.mkdir("Hello")

[ ] os.getcwd()

👤 '/content/My_new_dir'

[ ] os.chdir("/content/My_new_dir/Hello")

[ ] os.getcwd()

👤 '/content/My_new_dir/Hello'
```

→ **rmdir()** : Removes the specified directory(empty) .

```
[ ] os.rmdir("/content/My_new_dir/Hello")

[ ] os.chdir("/content/My_new_dir")

[ ] os.mkdir("dir1")

[ ] os.mkdir("dir2")
```

→ **removedirs()** : Removes empty directories recursively.

```
[ ] os.removedirs("/content/My_new_dir/dir1")    # only "dir1" removed because the outer directory is still not empty

[ ] os.removedirs("/content/My_new_dir/dir2")    # first "dir2" removed and then "My_new_dir" removed
```

- only "dir1" is removed because the outer directory is still not empty.
- first "dir2" removed and then "My_new_dir" removed because it is empty now.

→ **rename()** : Renames the directory

```
[ ] os.mkdir("/content/My_Dir")

[ ] os.chdir("/content")

[ ] os.rename("My_Dir", "Dir1")

[ ] os.rename("My_Dir", "Dir1")

-----
FileNotFoundError                                Traceback (most recent call last)
<ipython-input-20-eb17e885fb3f> in <module>()
----> 1 os.rename("My_Dir", "Dir1")

FileNotFoundError: [Errno 2] No such file or directory: 'My_Dir' -> 'Dir1'
```

→ **renames()** : renames directories or files recursively.

```
[ ] os.mkdir("/content/Dir1/Hello")

[ ] os.renames("/content/Dir1/Hello", "/content/Dir2/Hello2")
```

→ **listdir()** : Returns the list of directories and files in a specified directory.

```
[ ] os.chdir("/content/Dir2")

[ ] for i in range(5):
    os.makedirs('Hello{}'.format(i), exist_ok = True)

[ ] os.listdir(".")

['Hello1', 'Hello3', 'Hello4', 'Hello2', 'Hello0']
```

● Shutil Library

→ Importing shutil library

```
[ ] import shutil
```

→ **copy()** : copies a file to a specified file in the same or other directory.

Downloading an image and saving as **"Image.jpg"** :

```
[ ] !wget -O "/content/Image.jpg" "https://hackernoon.com/images/w6n3meqS5kSiaK2pnn0qw1keSSd2-ua3fq3x40.jpg"

--2020-09-07 16:57:17-- https://hackernoon.com/images/w6n3meqS5kSiaK2pnn0qw1keSSd2-ua3fq3x40.jpg
Resolving hackernoon.com (hackernoon.com)... 35.224.188.159, 104.197.241.218, 35.193.55.160
Connecting to hackernoon.com (hackernoon.com)|35.224.188.159|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 670086 (654K) [image/jpeg]
Saving to: '/content/Image.jpg'

/content/Image.jpg 100%[=====] 654.38K 3.94MB/s in 0.2s

2020-09-07 16:57:17 (3.94 MB/s) - '/content/Image.jpg' saved [670086/670086]

[ ] shutil.copy("/content/Image.jpg", "/content/My_new_dir2/inner_dir1/sample_img.jpg")

'/content/My_new_dir2/inner_dir1/sample_img.jpg'
```

→ **move()** : Moves files or directories from one directory to another.

```
[ ] shutil.move("/content/Dir2/Hello4", "/content/My_new_dir2/inner_dir1/")

'/content/My_new_dir2/inner_dir1/Hello4'
```

→ **copytree()** : Copies file and subdirectories in one directory to another directory.

```
[ ] shutil.copytree("/content/My_new_dir2/inner_dir1", "/content/Dir2/inner_dir1")

'/content/Dir2/inner_dir1'
```

→ **rmtree()** : Removes files and subdirectories (including the non-empty ones) in a specified directory.

```
[ ] shutil.rmtree("/content/Dir2/inner_dir1")
```

→ **disk_usage()** : Retrieves usage statistics of given directory (in bytes).

```
[ ] shutil.disk_usage("/content/My_new_dir2/inner_dir1/Hello4")

usage(total=115721105408, used=32418349056, free=77380399104)
```

→ **make_archive()** : builds an archive (zip or tar) of files.

```
shutil.make_archive("/content/Dir2arc", "zip", "/content/Dir2")

'/content/Dir2arc.zip'
```


- `unpack_archive()` : Extracts files in the given archive. Second parameter is the directory in which files are to be extracted.

```
[ ] shutil.unpack_archive("/content/Dir2arc.zip", "/content/Dir2ext")
```

- **%%time**

%%time is a magic command.


%%time prints the wall time for the entire cell whereas **%time** gives you the time for first line only

Using **%%time** or **%time** prints 2 values:

→ CPU Times

→ Wall Time


```
[ ] %%time
x =5
```

 CPU times: user 3 µs, sys: 1e+03 ns, total: 4 µs
Wall time: 6.68 µs

```
▶ %%time
if (5>2):
    print("DONE")
```

 DONE
CPU times: user 104 µs, sys: 5 µs, total: 109 µs
Wall time: 97 µs

```
▶ %time
if (5>2):
    print("DONE")
```

 CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 4.53 µs
DONE

- **Zippping and Unzipping Files and Folders**

Setting up kaggle credentials

```
[ ] from google.colab import files
    my_file=files.upload()

[ ] import os
    import pandas as pd
    cred = pd.read_json("kaggle.json",lines=True)
    os.environ['KAGGLE_USERNAME'] = cred.iloc[0][0]
    os.environ['KAGGLE_KEY'] = cred.iloc[0][1]
```

Downloadiing dataset

```
[ ] !kaggle datasets download -d vishnuvarthanrao/windows-store

[ ] !kaggle datasets download -d prashdash112/african-ngo-data
```

→ Unzipping files

We have downloaded 2 or more different zip files

1. To unzip a particular file we just need to type:-

```
!unzip "path_of_zip_file".
```

```
▶ !unzip "/content/african-ngo-data.zip"
✕ Archive: /content/african-ngo-data.zip
  inflating: Data2 (1).xls

[ ] !unzip "/content/windows-store.zip"
👤 Archive: /content/windows-store.zip
  inflating: msft.csv
```

2. To unzip all the zip files in current working directory we just need to type :-
`!unzip " *.zip"`.

```
[>] os.getcwd()
👤 '/content'

[ ] !unzip " *.zip"
👤 Archive: african-ngo-data.zip
  replace Data2 (1).xls? [y]es, [n]o, [A]ll, [N]one, [r]ename: n

  Archive: windows-store.zip
  replace msft.csv? [y]es, [n]o, [A]ll, [N]one, [r]ename: n

  2 archives were successfully processed.
```

→ Zipping files

To zip all the files present in the folder you just need to type this command:-

`!zip -r { name_of_the_zipped_file } "path_of the folder"`.

NOTE :- File will be downloaded in the current working directory.

```
[ ] import os
    os.mkdir("/content/My_Dir")
```

```
[ ] os.chdir("/content/My_Dir")
```

```
[ ] os.getcwd()
```

```
👤 '/content/My_Dir'
```

```
[ ] !kaggle datasets download -d vishnuvarthanrao/windows-store
```

```
👤 Downloading windows-store.zip to /content/My_Dir
    0% 0.00/93.1k [00:00<?, ?B/s]
  100% 93.1k/93.1k [00:00<00:00, 34.2MB/s]
```

```
[ ] !kaggle datasets download -d prashdash112/african-ngo-data
```

```
👤 Downloading african-ngo-data.zip to /content/My_Dir
    0% 0.00/97.8k [00:00<?, ?B/s]
  100% 97.8k/97.8k [00:00<00:00, 36.6MB/s]
```

```
[ ] !unzip "*.zip"
```

```
👤 Archive: african-ngo-data.zip
    inflating: Data2 (1).xls

Archive: windows-store.zip
    inflating: msft.csv

2 archives were successfully processed.
```

```
[ ] !zip -r final "/content/My_Dir"
```

```
👤 adding: content/My_Dir/ (stored 0%)
    adding: content/My_Dir/african-ngo-data.zip (stored 0%)
    adding: content/My_Dir/Data2 (1).xls (deflated 78%)
    adding: content/My_Dir/windows-store.zip (stored 0%)
    adding: content/My_Dir/msft.csv (deflated 69%)
```

```
[ ] !zip -r fin "/content/My_Dir/msft.csv"
```

```
👤 adding: content/My_Dir/msft.csv (deflated 69%)
```

- **%cd command**

%cd command is used to change the current working directory.

→ Moving one folder back from the current directory by using :- **%cd ..**

```
[ ] %cd ..  
[ ] /content  
[ ] os.getcwd()  
[ ] '/content'
```

→ Falling back to the home directory

- ◆ 1st method

```
[ ] %cd  
[ ] /root  
[ ] os.getcwd()  
[ ] '/root'
```

- ◆ 2nd method

```
[ ] %cd ~  
[ ] /root  
[ ] os.getcwd()  
[ ] '/root'
```

→ Jumping to root directory

What is the [root directory](#) ?

Basically it is the beginning of a particular folder structure

For eg :- C:\Windows\System32 . Here C:\ is the root directory.

```
[ ] %cd /  
/br/>[ ] os.getcwd()  
'/'
```

→ Jumping to next folder directory in which we are present

```
[ ] %cd content/  
/content
```

→ **%ls** command is used for generating all the files and folder present in the current directory

```
[ ] %ls  
african-ngo-data.zip  kaggle.json  My_Dir/  windows-store.zip  
'Data2 (1).xls'      msft.csv     sample_data/  
[ ] %cd My_Dir/  
/content/My_Dir  
[ ] os.getcwd()  
'/content/My_Dir'
```

→ Moving to any path just by specifying the location

```
%cd /content/My_Dir  
/content/My_Dir
```

→ Moving inside a folder which contain spaces in between their names. For eg :
“ My Dir ”.

```
[ ] os.mkdir("/content/My Dir")

[ ] os.getcwd()

[ ] %cd /content

[ ] %ls

[ ] %cd My dir

[ ] [Errno 2] No such file or directory: 'My dir'
/content
```

- ◆ By using `\` *backslash*

```
[ ] %cd My\ Dir

[ ] os.getcwd()

[ ] %cd /content

[ ] /content
```

- ◆ By enclosing the directory name in *single quotes* (`' '`) or *double quotes* (`" "`)

```
[ ] %cd "My Dir"

[ ] os.getcwd()

[ ] '/content/My Dir'
```

- **Deleting Files and Folders**

→ Deleting a particular file by passing its name

For deleting a particular file present in a given directory the command for it is:-

`!rm -r "path_of_file"`.

```
[ ] !rm "/content/My_Dir/Data2 (1).xls"
```

```
[ ] !rm "/content/My_Dir/final.zip"
```

→ Removing the entire folder:-

```
▶ !rm -r "/content/My_Dir"
```


LESSON SEVEN

- **googletrans Library**

→ Installing **googletrans** library

```
pip install googletrans
```

→ Creating an object, **translator** that will make translations .

```
[ ] import googletrans  
  
[ ] translator = googletrans.Translator()
```

Language code for supported languages for translation.

```
googletrans.LANGUAGES  
{  
  'ky': 'kyrgyz',  
  'la': 'latin',  
  'lb': 'luxembourgish',  
  'lo': 'lao',  
  'lt': 'lithuanian',  
  'lv': 'latvian',  
  'mg': 'malagasy',  
  'mi': 'maori',  
  'mk': 'macedonian',  
  'ml': 'malayalam',  
  'mn': 'mongolian',  
  'mr': 'marathi',  
  'ms': 'malay',  
  'mt': 'maltese',  
  'my': 'myanmar (burmese)',  
  'ne': 'nepali',  
  'nl': 'dutch',  
  'no': 'norwegian',  
  'ny': 'chichewa',  
  'or': 'odia',  
  'pa': 'punjabi',  
  'pl': 'polish',  
  'ps': 'pashto',  
  'pt': 'portuguese',  
}
```

This returns a dictionary with *language codes* as keys and their corresponding *languages* as values.

I. Translation

.translate is used to make translations

Parameters are text, destination_language, source_language (if the source_language is not specified, the system will attempt to detect it)

```
[ ] txt = "How are you ?"

[ ] translation = translator.translate(txt, dest='hi')
```

NOTE : Default destination language is english.

- **.src** : Returns the source language code
- **.dest** : Returns the destination language code
- **.origin** : Returns the original text
- **.text** : Returns the translated text

```
[ ] translation.src      # source language
👤 'en'

[ ] translation.dest     # destination language
👤 'hi'

[ ] translation.origin   # original text
👤 'How are you ?'

[ ] translation.text     # translated text
👤 'क्या हाल है ?'
```

II. Language Detection

.detect is used to detect the language of the text passed.

Parameter is text.

- **.lang** : Returns the detected language code
- **.confidence** : Returns the confidence(accuracy) of detection within the range [0.00, 1.00]

```
[ ] detect_lang = translator.detect("여보세요")
```

```
▶ detect_lang.lang
```

```
⦿ 'ko'
```

```
[ ] detect_lang.confidence
```

```
⦿ 1.0
```

Let's look at an example :

```
▶ text = ["guten Morgen", "boa tarde", "Добрый вечер", "buenas noches"]
  translations = translator.translate(text)
  for translation in translations:
    print(translation.origin + " --> " + translation.text)

  print("\n")

  detected_langs = translator.detect(text)
  for d_lang in detected_langs:
    print(d_lang.lang, googletrans.LANGUAGES[d_lang.lang], d_lang.confidence)
```

```
⦿ guten Morgen --> good Morning
  boa tarde --> good afternoon
  Добрый вечер --> good evening
  buenas noches --> Goodnight
```

```
de german 1.0
pt portuguese 1.0
ru russian 1.0
es spanish 1.0
```

- **gTTS Library**

- Installing **gTTS** library

```
▶ pip install gTTS
```

- Creating an object, **obj** that will convert text to speech

```
[ ] from gtts import gTTS

[ ] text = "Hello! Welcome to lesson 7."
    obj = gTTS(text, lang = "en")

[ ] obj.save("Hello.mp3")
```

• **textblob** Library

textblob is a library for processing textual data. It has a simple interface

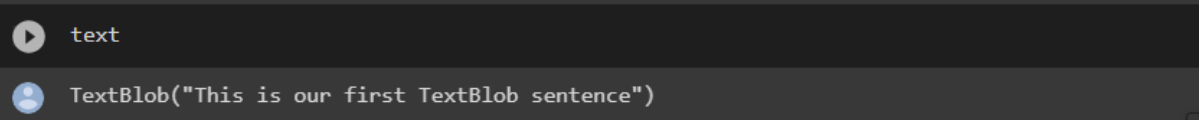
A good thing about TextBlob is that they are just like python strings. So, you can transform and play with it similar to that with strings.

Importing required libraries

```
[ ] from textblob import TextBlob
    from textblob import Word
    import nltk
    nltk.download('wordnet')
    nltk.download('averaged_perceptron_tagger')
    nltk.download('punkt')
    nltk.download('brown')
```

→ Creating TextBlob

```
[ ] text=TextBlob("This is our first TextBlob sentence")
```



The image shows a Jupyter Notebook cell with the code `text=TextBlob("This is our first TextBlob sentence")`. Below the code, there are two output cells. The first output cell shows a play button icon followed by the text `text`. The second output cell shows a play button icon followed by the text `TextBlob("This is our first TextBlob sentence")`.

→ Part-of-speech Tagging

Part-of-speech tagging or grammatical tagging is a method to mark words present in a text on the basis of its definition and context. In simple words, it tells whether a word is a noun, or an adjective, or a verb...

```
[ ] text = TextBlob(" And now for something completely different ")

[ ] text.tags

[ ] [ ('And', 'CC'),
      ('now', 'RB'),
      ('for', 'IN'),
      ('something', 'NN'),
      ('completely', 'RB'),
      ('different', 'JJ')]
```

Here's a list of keyword corresponding to their meaning :-

CC coordinating conjunction

CD cardinal digit

DT determiner

EX existential there (like: "there is" ... think of it like "there exists")

FW foreign word

IN preposition/subordinating conjunction

JJ adjective 'big'

JJR adjective, comparative 'bigger'

JJS adjective, superlative 'biggest'

LS list marker 1)

MD modal could, will

NN noun, singular 'desk'

NNS noun plural 'desks'

NNP proper noun, singular 'Harrison'

NNPS proper noun, plural 'Americans'

PDT predeterminer 'all the kids'

POS possessive ending parent's

PRP personal pronoun I, he, she

PRP\$ possessive pronoun my, his, hers

RB adverb very, silently,

RBR adverb, comparative better

RBS adverb, superlative best

RP particle give up

TO to go 'to' the store.

UH interjection errrrrrrm

VB verb, base form take

VBD verb, past tense took

VBG verb, gerund/present participle taking

VBN verb, past participle taken

VBP verb, sing. present, non-3d take
VBZ verb, 3rd person sing. present takes
WDT wh-determiner which
WP wh-pronoun who, what
WP\$ possessive wh-pronoun whose
WRB wh-adverb where, when

→ Noun Phrase Extraction

Since we extracted the words in the previous section, instead of that we can just extract out the noun phrases from the textblob

```
[ ] wiki = TextBlob("Python is a high-level, general-purpose programming language.")  
wiki.noun_phrases  
WordList(['python'])
```

→ Sentiment Analysis

Used to analyze the sentiment of sentence whether it is in happy mood :) , sad mood :(or neutral mood :

The sentiment property returns a named tuple of the form **Sentiment (polarity, subjectivity)** . The polarity score is a float within the range [-1.0, 1.0]. The subjectivity is a float within the range [0.0, 1.0], where 0.0 is very objective and 1.0 is very subjective.

Subjectivity refer to personal opinion, emotion or judgment

```
[ ] for sentence in text.sentences:  
    print(sentence.sentiment)  
Sentiment(polarity=0.21666666666666667, subjectivity=0.8333333333333334)  
Sentiment(polarity=0.5, subjectivity=0.5)  
Sentiment(polarity=0.06666666666666667, subjectivity=0.41904761904761906)
```

→ Tokenization

You can break TextBlobs into words or sentences.

- Printing out the sentiment for each sentences

```
[ ] text = TextBlob("Beautiful is better than ugly. "
                    "Explicit is better than implicit. "
                    "Simple is better than complex.")

[ ] text.words

WordList(['Beautiful', 'is', 'better', 'than', 'ugly', 'Explicit', 'is', 'better', 'than', 'implicit', 'Simple', 'is', 'better', 'than', 'complex'])

[ ] text.sentences

[Sentence("Beautiful is better than ugly."),
 Sentence("Explicit is better than implicit."),
 Sentence("Simple is better than complex.")]
```

→ Words Inflection and Lemmatization

- Word Inflection

Inflection is a process of word formation in which characters are added to the base form of a word to express grammatical meanings. Word inflection in TextBlob is very simple, i.e., the words we tokenized from a textblob can be easily changed into singular or plural.

```
[ ] sentence = TextBlob('Use 4 spaces per indentations level.')
    sentence.words

WordList(['Use', '4', 'spaces', 'per', 'indentations', 'level'])

[ ] sentence.words[4].singularize()

'indentation'

[ ] sentence.words[-1].pluralize()

'levels'
```

- Word lemmatization

Lemmatization usually refers to reducing different forms of word to a same form , normally aiming to return the base or dictionary form of a word.

```
[ ] w = Word("octopi")
    w.lemmatize()
```

```
'octopus'
```

```
[ ] w = Word("runs")
    w.lemmatize()
```

```
'run'
```

→ Word definition

.definitions returns the definitions of a word.

```
[ ] Word("octopus").definitions
```

```
['tentacles of octopus prepared as food',
 'bottom-living cephalopod having a soft oval body with eight long tentacles']
```

```
[ ] Word("Lion").definitions
```

```
['large gregarious predatory feline of Africa and India having a tawny coat with a shaggy mane in the male',
 'a celebrity who is lionized (much sought after)',
 '(astrology) a person who is born while the sun is in Leo',
 'the fifth sign of the zodiac; the sun is in this sign from about July 23 to August 22']
```

```
[ ] Word("Language").definitions
```

```
['a systematic means of communicating by the use of sounds or conventional symbols',
 '(language) communication by word of mouth',
 'the text of a popular song or musical-comedy number',
 'the cognitive processes involved in producing and understanding linguistic communication',
 'the mental faculty or power of vocal communication',
 'a system of words used to name things in a particular discipline']
```

→ Spelling

- **.correct()** is used to correct the spelling of each word in the sentence.

```
text = TextBlob("My speelling is wrog")
text
```

```
TextBlob("My speelling is wrog")
```

```
[ ] text.correct()
```

```
TextBlob("By spelling is wrong")
```

- Word objects have a **Word.spellcheck()** method that returns a list of (word, confidence) tuples with spelling suggestions.


```
[ ] w = Word('speeeling')
    w.spellcheck()

[ ] [('spelling', 0.5), ('speeding', 0.25), ('peeling', 0.25)]
```

→ **word_counts()** function

- It returns the number of times a string occurs in the text
- Case_sensitive means CAPITAL LETTERS are treated different from small letters

```
[ ] monty = TextBlob("We are no longer the Knights who say Ni. "
                    "We are now the Knights who say Ekki ekki ekki PTANG.")

    monty.word_counts['ekki']

[ ] monty.words.count('ekki')

[ ] monty.words.count('ekki', case_sensitive=True)
```

3

3

2

→ Operations on TextBlobs (similar to python strings)

```
[ ] a=TextBlob("This is the last lesson of the AI foundation")

[ ] a[0:19]

[ ] a.upper()

[ ] a.find('is')
```

TextBlob("This is the last le")

TextBlob("THIS IS THE LAST LESSON OF THE AI FOUNDATION")

2

```
[ ] apple_blob = TextBlob('apples')
    banana_blob = TextBlob('bananas')
    apple_blob < banana_blob
```

 True


```
[ ] apple_blob == 'apples'
```

 True


→ n - grams

The **TextBlob.ngrams()** method returns a list of tuples of n successive words.

```
[ ] a=TextBlob("This is the lesson 7")
    a.ngrams(n=2)
```

 [WordList(['This', 'is']),
WordList(['is', 'the']),
WordList(['the', 'lesson']),
WordList(['lesson', '7'])]

```
[ ] a=TextBlob("This is the lesson 7")
    a.ngrams(n=3)
```

 [WordList(['This', 'is', 'the']),
WordList(['is', 'the', 'lesson']),
WordList(['the', 'lesson', '7'])]

List of some important keywords in english :-

Keyword :- Meaning (Examples)

ADJ :- adjective (new, good, high, special, big, local)

ADP :- adposition (on, of, at, with, by, into, under)

ADV :- adverb (really, already, still, early, now)

CONJ :- conjunction (and, or, but, if, while, although)

DET :- determiner (article the, a, some, most, every, no, which)

NOUN :- noun (year, home, costs, time, Africa)

NUM :- numeral (twenty-four, fourth, 1991, 14:24)

PRT :- particle (at, on, out, over per, that, up, with)

PRON :- pronoun (he, their, her, its, my, I, us)

VERB :- verb (is, say, told, given, playing, would)

. :- punctuation marks (. , ; !)

X :- other (ersatz, esprit, dunno, gr8, university)