Introduction to Python.

- Python is an open-source, interpreted, high-level, general-purpose programming language.
- Python's design philosophy emphasizes code readability with its notable use of significant whitespace.
- Python is probably the easiest-to-learn and nicest-to-use programming language in widespread use.
- Python is a very expressive language, which means that we can usually write far fewer lines of Python code than would be required for an equivalent application written in, say, C++ or Java.
- Python can be used to program in procedural, object-oriented, and to a lesser extent, in a functional style.
- Python was conceived in the late 1980s as a successor to the ABC language.
- Python was created by Guido van Rossum and first released in 1991.
- Python 2.0, released in 2000, introduced features like list comprehensions and a garbage collection system with reference counting.
- Python 3.0 was released in 2008 and the current version of python is 3.8.3 (as of June-2020).
- The Python 2 language was officially discontinued in 2020.

Advantages of Python

- Easy Syntax
  - Python's syntax is easy to learn.
- Readability
  - Python's syntax is very clear, so it is easy to understand program code.
  - Python is often referred to as "executable pseudo-code" because its syntax mostly follows the conventions used by programmers to outline their ideas without the formal verbosity of code in most programming languages.
  - In other words, the syntax of Python is almost identical to the simplified "pseudo-code" used by many programmers to prototype and describe their solution to other programmers.
  - Thus Python can be used to prototype and test code which is later to be implemented in other programming languages.
- High-Level Language
  - Python looks more like a readable, human language than like a low-level language.
  - This gives you the ability to program at a faster rate.
- It's Free & Open Source
  - Python is both free and open-source.
  - The Python Software Foundation distributes pre-made binaries that are freely available for use on all major operating systems called CPython.

  - You can get CPython's source-code, too. Plus, you can modify the source code and distribute it as allowed by CPython's license.
- Cross-platform
  - Python runs on all major operating systems like Microsoft Windows, Linux, and Mac OS X.
- Widely Supported
  - Python has an active support community with many websites, mailing lists, and USENET "netnews" groups that attract a large number of knowledgeable and helpful contributors.
- It's Safe
  - Python doesn't have pointers like other C-based languages, making it much more reliable.
  - Along with that, errors never pass silently unless they're explicitly silenced.
  - This allows you to see and read why the program crashed and where to correct your error.
- Huge amount of additional open-source libraries
  - There are over 300 standard library modules that contain modules and classes for a wide variety of programming tasks.
  - Some libraries are listed below.
    - **matplotib** for plotting charts and graphs
    - **BeautifulSoup** for HTML parsing and XML
    - **NumPy** for scientific computing
    - **pandas** for performing data analysis
    - **SciPy** for engineering applications, science, and mathematics
    - **Scikit** for machine learning
    - **Django** for server-side web development
  - And many more...
- Extensible
  - In addition to the standard libraries, there are extensive collections of freely available add-on modules, libraries, frameworks, and tool-kits.

Installing Python

- For Windows & Mac:
  - To install python in windows you need to download installable file from https://www.python.org/downloads/
  - After downloading the installable file you need to execute the file.
- For Linux :
  - For ubuntu 16.10 or newer
    - sudo apt-get update
    - sudo apt-get install python3.8
- To verify the installation
  - Windows :

    - python --version
  - Linux:
    - **python3** --version (Linux might have python2 already installed, you can check python 2 using **python --version**)
- Alternatively, we can use anaconda distribution for the python installation
  - http://anaconda.com/downloads
  - Anaconda comes with many useful inbuilt libraries.

Hello World (first program) in Python

- To write Python programs, we can use any text editors or IDE (Integrated Development Environment).
- Create new file in editor, save it as first**.py** (Extensions for python programs will be .py)
- Write below code in the first.py file:

```
print("Hello World from python")
```

  Note: Python lines do **not** end with a semicolon (**;**) like most other programming languages.
- To run the python file open command prompt and change directory to where your python file is

```
D:\>cd B.E
D:\B.E>cd 5th
D:\B.E\5th>cd "Phython 2020"
D:\B.E\5th\Phython 2020>cd Demo
```

- Next, run python command (python filename.py)

```
D:\B.E\5th\Phython 2020\Demo>python first.py
Hello World from python
```

Data types in Python

- A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.
- Python provides various standard data types that define the storage method on each of them.
- The data types defined in Python are given below.
  - Numbers
    - Integer
    - Complex Number
    - Float
  - Boolean
  - Sequence Type
    - String
    - List
    - Tuple

- Set
- Dictionary

- **Numbers**
  - Number stores numeric values.
  - The integer, float, and complex values belong to a Python Numbers data-type.
  - Python provides the type() function to know the data-type of the variable.
  - Python creates Number objects when a number is assigned to a variable.
  - Python supports three types of numeric data.
    - **Int** - Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to **int**
    - **Float** - Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate up to 15 decimal points.
    - **complex** - A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc. (Note: imaginary part in python will be denoted with **j** suffix)

      Example:

```
a = 5
print("The type of a", type(a))

b = 40.5
print("The type of b", type(b))

c = 1+3j
print("The type of c", type(c))
print(" c is a complex number", isinstance(1+3j,complex))
```

      Output:

```
The type of a <class 'int'>
The type of b <class 'float'>
The type of c <class 'complex'>
c is complex number: True
```

- **Boolean**
  - Boolean type provides two built-in values, True and False, these values are used to determine within the given statement is true or false.
  - It is denoted by bool.
  - True can be represented by any non-zero value or 'True' whereas false can be represented by the 0 or 'False'.
  - Example:

```
# Python program to check the boolean type
print(type(True))
print(type(False))
print(false)
```

Output:

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

- **Sequence Type, Set and Dictionary** are discussed in subsequent topics

Variables in Python

- A Python variable is a reserved memory location to store values.
- Unlike other programming languages, Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- Python uses Dynamic Typing so,
  - We need not specify the data types to the variable as it will internally assign the data type to the variable according to the value assigned.
  - we can also reassign the different data types to the same variable, the variable data type will change to a new data type automatically.
  - We can check the current data type of the variable with the **type(variablename)** in-built function.
- Rules for variable name
  - Name cannot start with digit
  - Space not allowed
  - Cannot contain special character
  - Python keywords not allowed
  - **Should** be in lower case

Strings in Python

- The stringsOrdered Sequence of character such as “darshan”, ‘college’, “**રાજકોટ**” etc...
- Strings are arrays of bytes representing Unicode characters.
- A string can be represented as single, double, or triple quotes.
- String with triple Quotes allows multiple lines.
- Square brackets can be used to access elements of the string, Ex. “Darshan”[1] = a, characters can also be accessed with a reverse index like “Darshan”[-1] = n.
- String in python is immutable.
- Square brackets can be used to access elements of the string, Ex. “Darshan”[1] = a, characters can also be accessed with a reverse index like “Darshan”[-1] = n.

```
            x = "  D  a  r  s  h  a  n  "
        index =    0  1  2  3  4  5  6
Reverse index =    0 -6 -5 -4 -3 -2 -1
```

String slicing in Python.

- We know string is a sequence of individual characters, and therefore individual characters in a string can be extracted using the item access operator ([])
- Access operator ([]) is much more versatile and can be used to extract not just one item or character, but an entire slice (subsequence).
- We can get the substring in python using string slicing, we can specify start index, end index, and steps (colon-separated) to slice the string.
- Syntax :

```
x = 'our string'
subx1 = x[startindex]
subx2 = x[startindex:endindex]
subx3 = x[startindex:endindex:steps]
```

- The startindex, endindex, and steps values must all be integers, It extracts the substring from startindex till endindex (not including endindex) with steps defining the increment of index, If we specify steps to be -1 it will extract the reversed string.
- Example:

```
x = 'darshan institute of engineering and technology, rajkot,
gujarat, INDIA'
subx1 = x[0:7]
subx2 = x[49:55]
subx3 = x[66:]
subx4 = x[::2]
subx5 = x[::-1]
print(subx1)
print(subx2)
print(subx3)
print(subx4)
print(subx5)
```

Output:

```
darshan
rajkot
INDIA
drhnisiueo niern n ehooy akt uaa,IDA
AIDNI ,tarajug ,tokjar ,ygolonhcet dna gnireenigne fo etutitsni nahsrad
```

String functions in Python

Python has lots of built-in methods that you can use on strings, we are going to cover some frequently used methods for string like

- **len()** is **not a string function** but we can use this method to get the length of string.

```
x = "Darshan"
print(len(x))
```

Output:

```
7 (length of "Darshan")
```

- **count()** method will return the number of times a specified value occurs in a string.

```
x = "Darshan"
ca = x.count('a')
print(ca)
```

Output:

```
2 (occurrence of 'a' in "Darshan")
```

- **title()**, **lower()**, **upper()** will returns capitalized, lower case and upper case string respectively

```
x = "darshan Institute, rajkot"
c = x.title()
l = x.lower()
u = x.upper()
print(c)
print(l)
print(u)
```

Output:

```
Darshan Institute, Rajkot
darshan institute, rajkot
DARSHAN INSTITUTE, RAJKOT
```

- **istitle()**, **islower()**, **isupper()** will returns True if the given string is capitalized, lower case and upper case respectively.

```
x = 'darshan institute, rajkot'
c = x.istitle()
l = x.islower()
u = x.isupper()
print(c)
print(l)
print(u)
```

Output:

```
False
True
False
```

- **find(), rfind(), replace()**
  - **find**() method will search the string and returns the index at which the specified value is found.

```
x = 'darshan institute, rajkot, india'
f = x.find('in')
print(f)
```

Output:

```
8 (index of 'in' in x)
```

  - **rfind**() will search the string and returns the last index at which the specified value is found.

```
x = 'darshan institute, rajkot, india'
r = x.rfind('in')
print(r)
```

Output:

```
27 (last index of 'in'  in x)
```

  - **replace**() will replace str1 with str2 from our string and return the updated string

```
x = 'darshan institute, rajkot, india'
r = x.replace('india','INDIA')
print(r)
```

Output:

```
darshan institute, rajkot, INDIA
```

- **index(), rindex()**
  - **index**() method will search the string and returns the index at which the specified value is found, but if unable to find the string it will raise an exception.

```
x = 'darshan institute, rajkot, india'
f = x.index('in')
print(f)
```

Output:

```
8 (index of 'in' in x)
```

  - **rindex**() will search the string and returns the last index at which the specified value is found, but if unable to find the string it will raise an exception.

```
x = 'darshan institute, rajkot, india'
r = x.rindex('in')
print(r)
```

Output:

```
27 (last index of 'in' in x)
```

  - Note : **find**() and **index**() are almost the same, the only difference is if **find**() is unable to find then it will return -1 and if **index**() is unable to find, it will raise an exception.

- **isalpha(), isalnum(), isdecimal(), isdigit()**
  - **isalnum**() method will return true if all the characters in the string are alphanumeric (i.e either alphabets or numeric).

```
x = 'darshan123'
f = x.isalnum()
print(f)
```

Output:

```
True
```

  - **isalpha**() and **isnumeric**() will return true if all the characters in the string are only alphabets and numeric respectively.
  - **isdecimal**() will return true if all the characters in the string are decimal.
  - **Note:** str.isdigit only returns True when strings containing solely the digits 0-9, by contrast, str.isnumeric returns True if it contains any numeric characters.
- **strip(), lstrip(), rstrip()**
  - **strip**() method will remove whitespaces from both sides of the string and returns the string.
  - **rstrip**() and **lstrip**() will remove whitespaces from right and left side respectively.
- **split()** method splits a string into a list.

String Formatting with the str.format() Method

- The str.format() method provides a very flexible and powerful way of creating strings.
- The str.format() method returns a new string with the replacement fields in its string replaced by its arguments formatted suitably, for example:

```
x = '{0} institute, rajkot, INDIA'
r = x.format('darshan')
print(r)
```

Output:

```
        darshan institute, rajkot, INDIA
```

- There can be more than one argument in the format function, for example:

```
x = '{0} institute, {1}, {2}'
r = x.format('darshan', 'rajkot', 'INDIA')
print(r)
```

Output:

```
        darshan institute, rajkot, INDIA
```

- Each replacement field is identified by a field name in braces, if the field name is a simple integer it is taken to be the index position. {0} field name will be replaced by first argument, {1} field name will be replaced by the second argument, and this will continue till the last argument.

- We can use field name more than one time in the same string, for example

```
x = '{0} institute, {1}, {2}, welcome to {0}'
r = x.format('darshan', 'rajkot', 'INDIA')
print(r)
```

Output:

```
        darshan institute, rajkot, INDIA, welcome to darshan
```

- If we need to include braces inside format strings, we can do so by writing three brackets, for example:

```
x = '{{{0}}} institute'
r = x.format('darshan')
print(r)
```

Output: {darshan} institute

- If we try to concatenate a string and a number, Python will quite rightly raise a TypeError, But we can do this using str.format(), for example:

```
x = 'The price for {0} is ₹{1}'
r = x.format('iphone',50000)
print(r)
```

Output:

```
The price for iphone is ₹50000
```

- The default formatting of integers, floating-point numbers, and strings are often perfectly satisfactory, but if we want to fine control, we can easily do so by using format specifications.
- Below is the general form of a format specification

:	fill	align	sign	#	0	width	. precision	type
	Any character except }	< left > right ^ center = pad between sign and digits	+ force sign; - sign if needed; " " space or – as appropriate	Prefix ints with 0b,0o 0r	0-pad numbers	Minimum field width	Maximum field width for strings; number of decimal places for floating-point numbers.	ints b,c,d,n,o ,x,X; floats e,E,f,g, G,n,%

- String values formatting,

```
s = '{0} institute of engineering'
print(s.format('darshan')) #default formatting
```

Output:

```
darshan institute of engineering
```

```
s = '{0:25} institute of engineering'
print(s.format('darshan')) #minimum width
```

Output:

```
darshan                   institute of engineering
```

```
s = '{0:>25} institute of engineering'
print(s.format('darshan')) #right align, minimum width
```

Output:

```
                  darshan institute of engineering
```

```
s = '{0:^25} institute of engineering'
print(s.format('darshan')) #center align, minimum width
```

Output:

```
         darshan          institute of engineering
```

```
s = '{0:-^25} institute of engineering'
print(s.format('darshan')) #fill center align, minimum width
```

Output:

```
---------darshan--------- institute of engineering
```

```
s = '{0:.3} institute of engineering' #maximum width of 3
print(s.format('darshan'))
```

Output:

```
dar institute of engineering
```

- Integer values formatting,

```
s = 'amount = {0}' # default formatting
print(s.format(123456))
```

Output:

```
123456
```

```
s1 = 'amount = {0:0=10}' # 0 fill, minimum width 12 (technique 1)
print(s1.format(123456))

s2 = 'amount = {0:010}' # 0 pad, minimum width 12 (technique 2)
print(s2.format(123456))
```

Output:

```
amount = 0000123456
amount = 0000123456
```

```
s = 'amount = {0: }' # space or - sign
print(s.format(123456))
print(s.format(-123456))
```

Output: (Note: single quote is just to demonstrate the space in the positive numbers)

```
‘ 123456’
‘-123456’
```

```
s = 'amount = {0:+}' # force sign
print(s.format(123456))
print(s.format(-123456))
```

Output:

```
+123456
-123456
```

```
s1 = '{0:b} {0:o} {0:x} {0:X}'
s1 = '{0:#b} {0:#o} {0:#x} {0:#X}'
print(s1.format(12))
print(s2.format(12))
```

Output:

```
1100 14 c C
0b1100 0o14 0xc 0XC
```

- Decimal values formatting,

```
s = 'amount = {0}' # default formatting
print(s.format(12.3456789))
```

Output:

```
12.3456789
```

```
s1 = 'amount = {0:10.2e}' # exponential formatting with two precession
s2 = 'amount = {0:10.2f}' # floating formatting with two precession
print(s1.format(12345.6789))
print(s2.format(12345.6789))
```

Output:

```
amount =   1.23e+04
amount =   12345.68
```

Operators in Python

- Operators are used to performing operations on variables and values.
- We can segregate python operators in following groups,
  - Arithmetic operators
  - Assignment operators
  - Comparison operators
  - Logical operators
  - Identity operators
  - Membership operators
  - Bitwise operators
- We are going to explore some of the operators in this section.

Arithmetic Operators

Arithmetic operators can be used with numeric values or variables to perform common mathematical operations:

Operator	Description	Example	Output
+	Addition	A + B	13
-	Subtraction	A - B	7
/	Division	A / B	3.33333..33335
*	Multiplication	A * B	30
%	Modulus return the remainder	A % B	1
//	Floor division returns the quotient	A // B	3
**	Exponentiation	A ** B	10 * 10 * 10 = 1000

Note: Consider A = 10 and B = 3 in above table

Logical Operators

Logical operators can be used to merge conditional statements:

Operator	Description	Example	Output
and	Returns True if both statements are true	A > 5 **and** B < 5	True

or	Returns True if one of the statements is true	A > 5 **or** B > 5	True
not	Negate the result, returns True if the result is False	not(A>5)	False

Note: Consider A = 10 and B = 3 in above table

Identity Operators

Identity operators can be used to compare the objects, not if they are equal, but if they are the same object, with the same memory location:

Operator	Description	Example	Output
is	Returns True if both variables are the same object	A is B A is C	False True
is not	Returns True if both variables are **different** object	A is not B	True

Note: Consider A = [1,2], B = [1,2] and C = A in above table

Member Operators

Membership operators can be used to check if a sequence is presented in an object:

Operator	Description	Example	Output
in	Returns True if a sequence with the specified value is present in the object	A in B	True
not in	Returns True if a sequence with the specified value is not present in the object	A not in B	False

Note: Consider A = 2 and B = [1,2,3] in above table

Conditional statements in Python

Python supports 3 types of conditional statements,

1. if statements
2. if..else statements
3. if…elif…else statements

   python also supports nested if statements

if statement

- It is one of the powerful conditional statement.

- If statement is responsible for modifying the flow of execution of a program.
- If statement is always used with a condition.
- The condition is evaluated first before executing any statement inside the body of If.
- if statement is written using the **if** keyword followed by **condition** and **colon(:)**
- The syntax for if statement is as follows:

```
if some_condition :
   # Code to execute when condition is true
```

- Code to execute when the condition is true will be ideally written in the next line with Indentation (white space).
- Python relies on indentation to define the scope in the code (Other programming languages often use curly-brackets for this purpose).
- Python programming language assumes any non-zero and non-null values as True, and if it is either zero or null, then it is assumed as False value.
- Example:

```
x = 10

if x > 5 :
    print("X is greater than 5")
```

Output :

```
X is greater than 5
```

if else statements

- An else statement can be combined with an if statement.
- An else statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a False value.
- The else statement is an optional statement and there could be at most only one else statement following if.
- Example:

```
x = 3

if x > 5 :
    print("X is greater than 5")
else :
    print("X is less than 5")
```

Output:

```
X is less than 5
```

if elif else statements

- The elif is short for else if. It allows us to check for multiple expressions.
- If the condition for if is False, it checks the condition of the next elif block and so on.
- If all the conditions are False, the body of else is executed.
- Only one block among the several if...elif...else blocks is executed according to the condition.
- The if a block can have only one else block, but it can have multiple elif blocks.
- Example:

```
x = 10

if x > 12 :
    print("X is greater than 12")
elif x > 5 :
    print("X is greater than 5")
else :
    print("X is less than 5")
```

Output:

```
X is greater than 5
```

Looping statements in python

- There are two looping statements in python,
  - while loop
  - for loop

while loop

- While loop will continue to execute a block of code until specified condition remains True.
- For example
  - while hungry keep eating
  - while internet pack available, keep watching videos
  - etc..
- Syntax:

```
while some_condition :
    # Code to execute in loop
```

- Example:

```
x = 0
while x < 3 :
    print(x)
    x += 1    # x++ is invalid in python
```

Output:

```
0
1
2
```

- We can use an else statement with a while loop if we want to execute some block when the condition specified in the while statement is False.
- Example:

```
x = 5
while x < 3 :
    print(x)
    x += 1    # x++ is invalid in python
else :
    print("X is greater than 3")
```

Output:

```
X is greater than 3
```

For loop

- Many objects in python are iterable, meaning we can iterate over every element in the object, such as every element from the List, every character from the string, etc..
- Syntax:

```
for temp_item in iterable_object :
    # Code to execute for each object in iterable
```

- Example 1 (number list):

```
my_list = [1, 2, 3, 4]
for list_item in my_list :
    print(list_item)
```

Output:

```
1
2
3
4
```

- Example 2 (number list from the range):
  range() function will create a list from start till (not including) the value specified as the argument.

```
my_list = range(0,5)
for list_item in my_list :
    print(list_item)
```

Output:

```
0
1
2
3
4
```

- Example 3 (string list):

```
my_list = ["darshan","institute","rajkot","gujarat"]
for list_item in my_list :
    print(list_item)
```

Output:

```
darshan
institute
rajkot
gujarat
```

Break, continue and pass keywords

- **break** statement will breaks out of the closest enclosing loop, for example:

```
for temp in range(5) :
    if temp == 2 :
        break
    print(temp)
```

Output:

```
0
1
```

- **continue** statement will send execution control to top of the current closest enclosing loop, for example:

```
for temp in range(5) :
    if temp == 2 :
        continue
    print(temp)
```

Output:

```
0
1
3
4
```

- **pass** does nothing at all !!!!!, it can be used as a placeholder in conditions where we don't write anything, for example:

```
for temp in range(5) :
    pass
```

Functions in Python

- Creating clean repeatable code is a key part of becoming an effective programmer.
- In Python, a function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks, as our program grows larger and larger, functions make it more organized and manageable.
- A function is a block of code that only runs when it is called.
- Python provides built-in functions like print(), etc. but we can also create our functions, these functions are called **user-defined functions**.
- We can pass arguments/parameters into function and function can return some value as a result.
- In python user-defined functions can be created using def keyword followed by the name of the function and then its argument in the round brackets.
- General Syntax for the UDF in Python:

```
def function_name(arguments) :
    ''' docstring ''' #optional
    #code to execute when function is called
```

- Example:

```
def seperator() :
    print('================================')

print("hello world")
seperator()
print("from darshan college")
seperator()
print("rajkot")
```

Output:

```
hello world
================================
from darshan college
================================
rajkot
```

Arguments in Python Functions

- Data can be passed into functions as parameters/arguments.
- We can specify any number of arguments in the function inside parenthesis just after the function name as comma-separated values.

- Example (passing parameter)

```
def seperator(type) :
    if(type=='=') :
        print('==============================')
    elif(type=='*') :
        print('******************************')
    else :
        print('&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&')

print("hello world")
seperator('=')
print("from darshan college")
seperator('*')
print("rajkot")
```

Output:

```
hello world
==============================
from darshan college
******************************
rajkot
```

- All parameters in the Python language are passed by reference, which means if we change what parameter refers it will change the actual parameter value (if the parameter is mutable), below example uses List which is mutable. (we will discuss List later in this chapter)

```
def setName(name) :
    name[0] = "arjun bala"

name = ["Default"]
setName(name)
print(name)
```

Output:

```
['arjun bala']
```

- We can use **keyword arguments** in Python which gives freedom to specify arguments in any sequence.
- Example (keyword arguments):

```
def collegeDetails(collegeName, state) :
    print(collegeName + " is located in "+ state)

collegeDetails(state='Gujarat', collegeName='Darshan')
```

Output:

```
Darshan is located in Gujarat
```

- It is also possible to specify the **default value** to some arguments in the Python functions, such arguments are optional to pass when calling the function.
- Example (Default value):

```
def collegeDetails(collegeName, state='Gujarat') :
    print(collegeName + " is located in "+ state)

collegeDetails(collegeName='Darshan')
```

Output:

```
Darshan is located in Gujarat
```

- We can specify the dynamic argument using **Arbitrary Arguments (*args)** in python, which can help to accept any number of arguments while calling the function.
- Example (Arbitrary Arguments):

```
def collegeList(*colleges):
    print(colleges[0])
    # we can also loop through all the arguments
    # for c in colleges :
    #    print(c)

collegeList('darshan','nirma','vvp')
```

Output:

```
darshan
```

- We can also specify the dynamic keyword argument using **Arbitrary Keyword Arguments (**kwargs)** in python, which can help to accept any number of keyword arguments while calling the function.
- Example (Arbitrary Keyword Arguments):

```
def collegeDetail(**college) :
    print('College Name is '+college['name'])

collegeDetail(city='rajkot', state='gujarat', name='Darshan')
```

Output:

```
College Name is Darshan
```

- The **return** statement exits a function and passes the caller of the function of some value (result).

- Example (return statement):

```
def addition(n1,n2):
    return n1 + n2

ans = addition(10,5)
print(ans)
```

Output:

```
15
```

Data Structures in Python

- There are many types of Data Structures available in python, in this section we will discuss 4 inbuilt Data Structures,
  - **List**: Ordered Mutable sequence of objects
  - **Set**: Unordered collection of unique objects
  - **Tuple**: Ordered Immutable sequence of objects
  - **Dictionary**: Unordered key:value pair of objects

List

- A **List** is a data structure that holds an **ordered** collection of items i.e. you can store a sequence of items in a list.
- Once we have created a list, we can add, remove or search for items in the list.
- Since we can add and remove items, we say that a list is a **mutable** data type i.e. this type can be altered.
- A List can hold duplicate values and can be used similar to Arrays.
- A List will be represented with the Square Brackets '[**list**]'
- Example:

```
my_list = ['darshan', 'institute', 'rkot']
print(my_list[1])
print(len(my_list))
my_list[2] = "rajkot"
print(my_list)
print(my_list[-1])
```

Output:

```
institute (List index starts with 0)
3 (length of the List)
['darshan', 'institute', 'rajkot'] (Note :  spelling of rajkot is updated)
rajkot (-1 represent last element)
```

- We can use slicing similar to string to get the sublist from the list.

- Example (Slicing):

```
my_list = ['darshan', 'institute', 'rajkot','gujarat','INDIA']
print(my_list[1:3])
```

Output:

```
['institute', 'rajkot']  (Note: end index is not included)
```

Methods of List

- There are many built-in methods of List in Python, some important are discussed here,
  - list.**append**(element) will append the element at the end of the list.

    Example:

    ```
    my_list = ['darshan', 'institute', 'rajkot']
    my_list.append('gujarat')
    print(my_list)
    ```

    Output:

    ```
    ['darshan', 'institute', 'rajkot', 'gujarat']
    ```

  - list.**insert**() method will add element at the specified index in the list.

    Example:

    ```
    my_list = ['darshan', 'institute', 'rajkot']
    my_list.insert(2,'of')
    my_list.insert(3,'engineering')
    print(my_list)
    ```

    Output:

    ```
    ['darshan', 'institute', 'of', 'engineering', 'rajkot']
    ```

  - list.**extend**() method will add one data structure (List or any) to the current List.

    Example:

    ```
    my_list1 = ['darshan', 'institute']
    my_list2 = ['rajkot','gujarat']
    my_list1.extend(my_list2)
    print(my_list1)
    ```

    Output:

    ```
    ['darshan', 'institute', 'rajkot', 'gujarat']
    ```

  - list.**pop**() method will remove the last element from the list and return it.

Example:

```
my_list = ['darshan', 'institute','rajkot']
temp = my_list.pop()
print(temp)
print(my_list)
```

Output:

```
rajkot
['darshan', 'institute']
```

- list.**remove**() method will remove first element from the list (it doesn't return anything).

  Example:

```
my_list = ['darshan', 'institute', 'darshan','rajkot']
my_list.remove('darshan')
print(my_list)
```

  Output:

```
['institute', 'darshan', 'rajkot']
```

- list.**clear**() method will remove all elements from the list.

  Example:

```
my_list = ['darshan', 'institute', 'darshan','rajkot']
my_list.clear()
print(my_list)
```

  Output:

```
[]  (Empty List)
```

- list.**index**() method will return the first index of the specified element from the list.

  Example:

```
my_list = ['darshan', 'institute', 'darshan','rajkot']
id = my_list.index('institute')
print(id)
```

  Output:

```
1
```

- list.**count**() method will return a number of occurrences for the specified element in the list.

Example:

```
my_list = ['darshan', 'institute', 'darshan','rajkot']
c = my_list.count('darshan')
print(c)
```

Output:

```
2
```

- list.**reverse**() method will reverse the elements of the list.

  Example:

```
my_list = ['darshan', 'institute','rajkot']
my_list.reverse()
print(my_list)
```

  Output:

```
['rajkot', 'institute','darshan']
```

- list.**sort**() method will sort the elements of the list in ascending order, if we want to sort the elements in descending order we can specify the **reverse** parameter to be **True** while calling the sort method.

  Example:

```
my_list = ['darshan', 'college','of','enginnering','rajkot']
my_list.sort()
print(my_list)
my_list.sort(reverse=True)
print(my_list)
```

  Output:

```
['college', 'darshan', 'enginnering', 'of', 'rajkot']
['rajkot', 'of', 'enginnering', 'darshan', 'college']
```

Set

- A **Set** is a **mutable** data structure that holds an **unordered** collection of **unique** items.
- A Set will be represented with the Curly Brackets '**{ Set }**'
- It has similar methods as List, the only difference between List and Set is
  - List is **ordered** collection of items, whereas
  - Set is an **unordered** collection of **unique** items.
- Example:

```
my_set = {1,1,1,2,2,5,3,9}
print(my_set)
```

Output:

```
{1, 2, 3, 5, 9}
```

- Set also has a **union, intersection,** and **different** methods.

Tuple

- The tuple is an **immutable ordered** sequence of zero or more objects; it allows duplicate value similar to the List.
- Tuple will be represented with the Round Brackets '**(Tuple)**'
- Tuple supports the same slicing syntax as String/List.
- Similar to String, Tuple are immutable, so we cannot replace or delete any of their items.
- The only difference between List and Tuple is
  - List is **mutable,** whereas
  - Tuple is **immutable.**
- Example:

```
my_tuple = ('darshan','institute','of','engineering','of')
print(my_tuple)
print(my_tuple.index('engineering'))
print(my_tuple.count('of'))
print(my_tuple[-1])
```

Output:

```
('darshan', 'institute', 'of', 'engineering', 'of')
3
2
of
```

Dictionaries

- A dictionary (dict) is an unordered collection of zero or more key-value pairs whose keys are object references that refer to hashable objects, and whose values are object references referring to objects of any type.
- Dictionaries are mutable, so we can easily add, edit, or remove items, but as it is unordered we cannot slice the dictionary.
- Dictionary will be represented with the Curly Brackets '**{ key: value }**'

```
my_dict = { 'key1':'value1', 'key2':'value2' }
```

Note: key-value are separated by colon (**:**) and pairs of key-value are separated by comma (**,**)

- Example:

```
my_dict = {'college':"darshan", 'city':"rajkot",'type':"enginee
ring"}
print(my_dict['college'])
print(my_dict.get('city'))
```

Output:

```
darshan
rajkot
```

Note: Values can be accessed using key inside square brackets as well as using get() method

Methods of Dictionary

- dict.**keys**() method will return a list of all the keys associated with the Dictionary.

  Example:

```
my_dict = {'college':"darshan",
           'city':"rajkot",
           'type':"engineering"}
print(my_dict.keys())
```

  Output:

```
['college', 'city', 'type']
```

- dict.**values**() method will return list of all the values associated with the Dictionary.

  Example:

```
my_dict = {'college':"darshan",
           'city':"rajkot",
           'type':"engineering"}
print(my_dict.values())
```

  Output:

```
['darshan', 'rajkot', 'engineering']
```

- dict.**items**() method will return a list of tuples for each key-value pair associated with the Dictionary..

  Example:

```
my_dict = {'college':"darshan",
           'city':"rajkot",
           'type':"engineering"}
print(my_dict.items())
```

Output:

```
[('college', 'darshan'), ('city', 'rajkot'), ('type', 'engineering')]
```

List Comprehension

- List comprehensions offer a way to create lists based on existing iterable. When using list comprehensions, lists can be built by using any iterable, including strings, list, tuples.
- For example, if we want to create a list of characters from the string, we can use for loop like below example,
- Example (Using for loop):

```
mystr = 'darshan'
mylist = []
for c in mystr:
    mylist.append(c)

print(mylist)
```

Output:

```
['d', 'a', 'r', 's', 'h', 'a', 'n']
```

- The same output can be achieved using list comprehension with just one line of code.
- Syntax:

```
[ expression for item in iterable ]
                OR
[ expression for item in iterable if condition ]
```

- Example (Using List Comprehension):

```
mylist = [c for c in 'darshan']
print(mylist)
```

Output:

```
['d', 'a', 'r', 's', 'h', 'a', 'n']
```

- Similarly, we can use list comprehensions in many cases where we want to create a list out of other iterable, let's see another example of the use of List Comprehension.
- Example (Using for loop):

```
# list of square from 1 to 10
mylist = []
for i in range(1,11):
    mylist.append(i**2)

print(mylist)
```

Output

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Example (Using List Comprehension):

```
# list of square from 1 to 10
mylist = [ i**2 for i in range(1,11) ]
print(mylist)
```

Output

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- We can also use conditions in List Comprehension, for example, if we want a list of number from 1980 to 2020 which are divisible by 4 then we can use list comprehension.
- Example (Using List Comprehension):

```
leapYears = [i for i in range(1980,2021) if i%4==0]
print(leapYears)
```

Output

```
[1980, 1984, 1988, 1992, 1996, 2000, 2004, 2008, 2012, 2016, 2020]
```