Python Basics Tutorial

- Source: python.org And w3school.com

Why Python ?

Python is an Open Source. For more than 20 years, Python has been cross-platform and open source. You can code on Linux, Windows and Mac OS.

Data is the base in this subject, doesn't matter what field you want to be in, it's going to be there. Python is one of the commonly used programming languages for Data Engineering , Data Science, Deep Learning and Machine Learning. Considering the growing Machine Learning is having, you should give it a try

If you are working on DATA You should know python and python is commonly for

For Data Engineering

For Machine Learning

For Data Science

For Deep Learning

Benefits Python?

Reduce development time

Object Oriented Language

No compile

Supports dynamic data type

Reduce code length

Easy to learn and use as developers

Easy to understand codes

Easy to do team projects

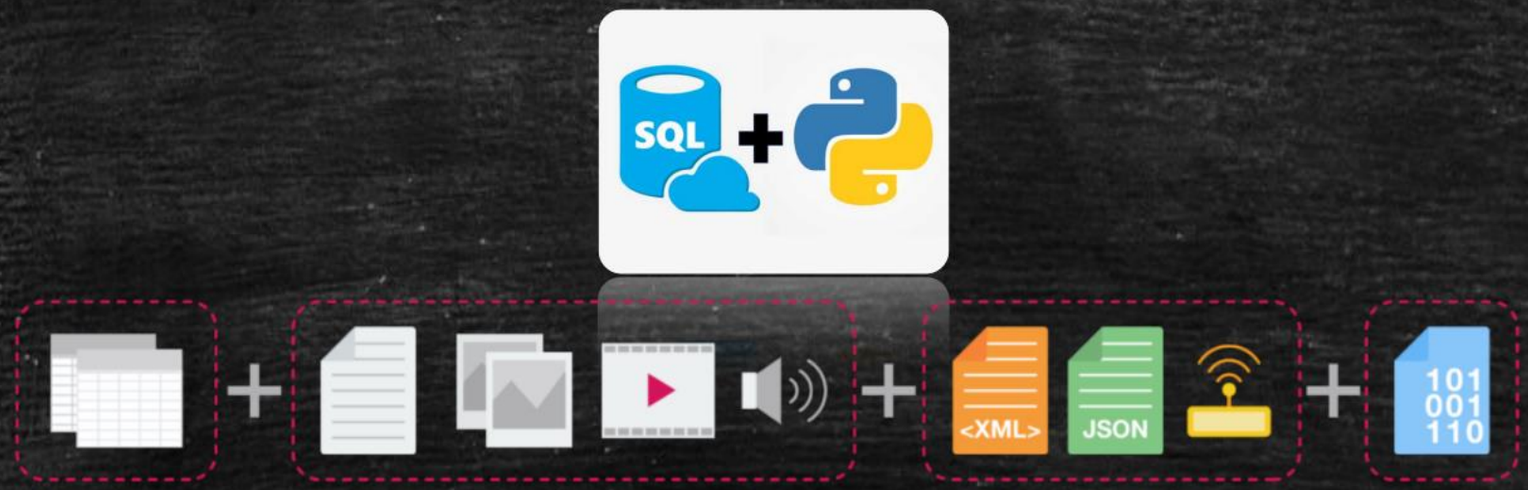
Easy to extend to other languages

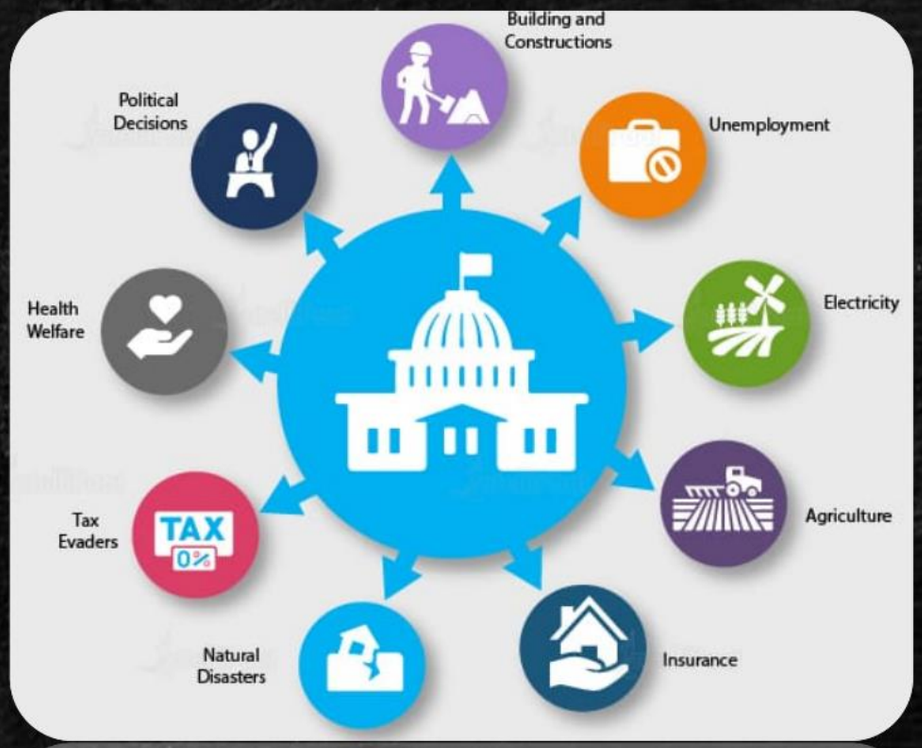
Automatic memory management

It’s free (open source)

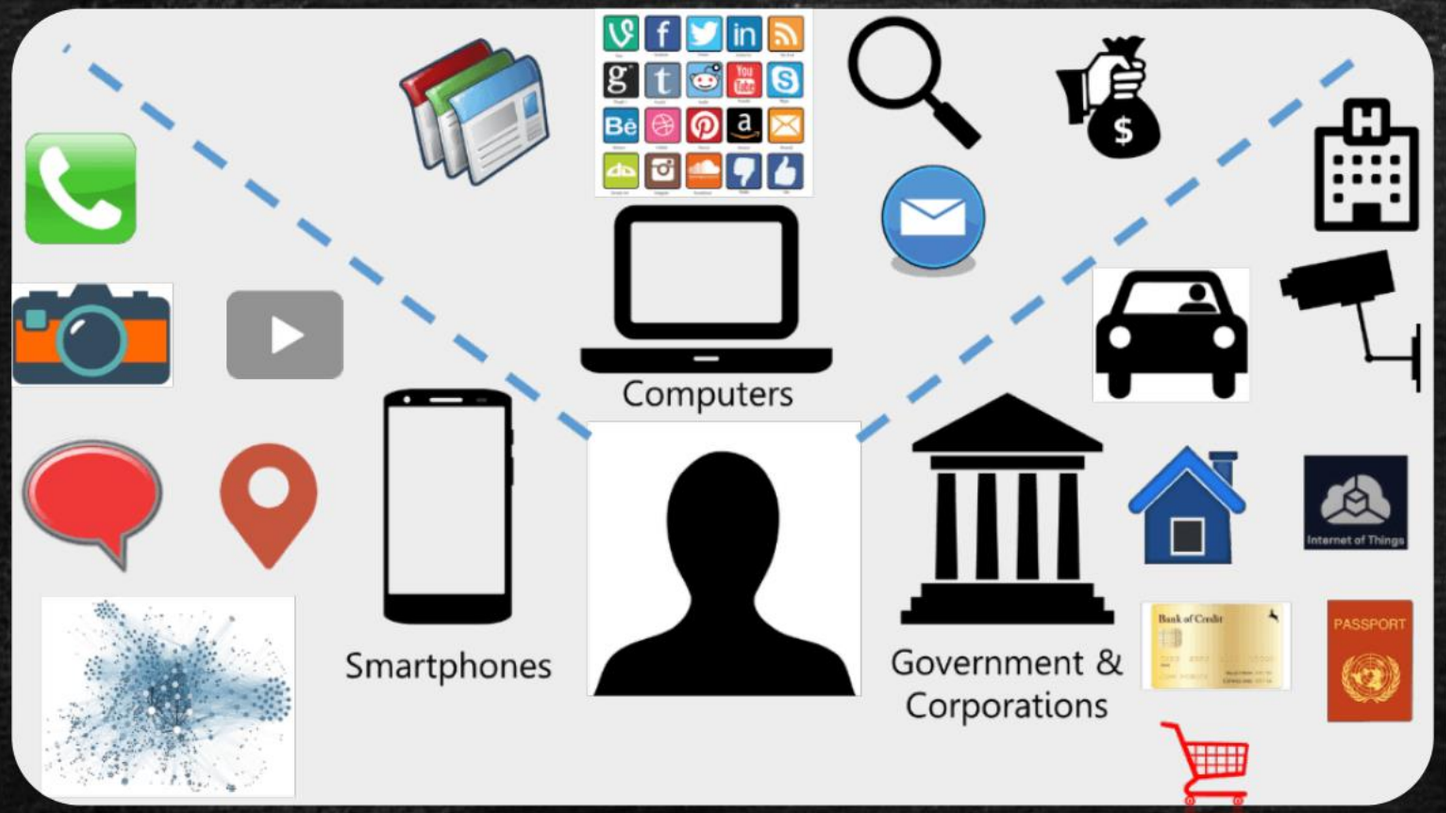
Why Python? Why Not SQL?

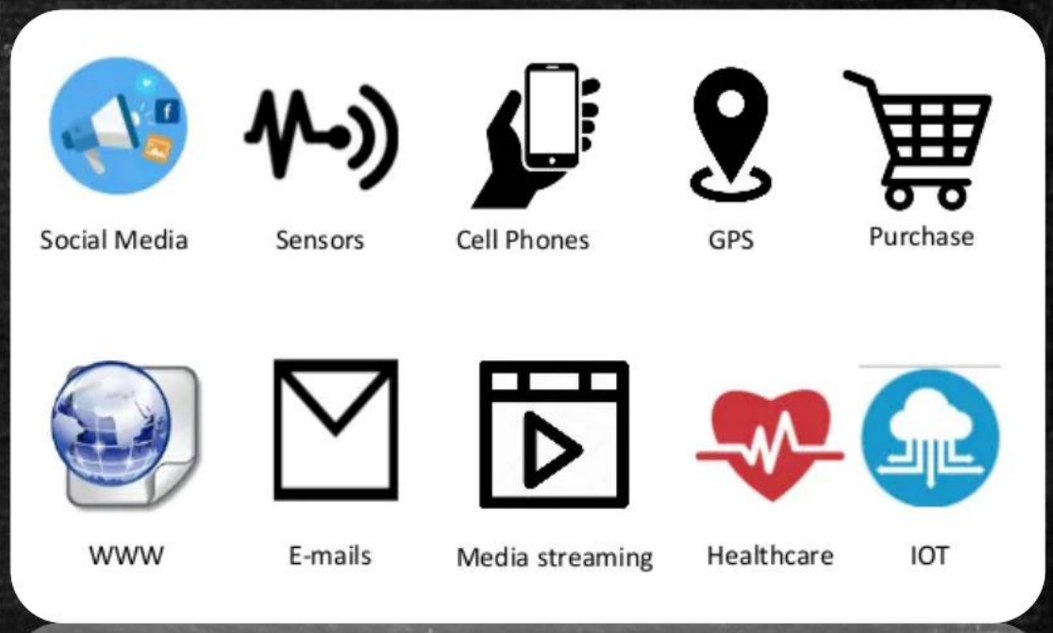
- SQL For Processing, Storing and analysing **Structured data**
- Python for Processing , Storing and Analysing Any Type of data (**Structured data, Semi Structured Data and Un-Structured Data**)





data generation sources





Python Tutorial Content

Basics

- Python Data Types
  - Numbers ,Strings....
- Data Structures (collections)
  - Lists, and Tuples ,Dictionaries and Sets
- Conditionals and Loop Control Statements
  - if , for, while,pass,break,continue...
- Functions

Advanced

Files and Input/Output
Exception Handling
List Comprehension
Lambda Expressions
Regular Expressions
Modules and Logging

What are Python Identifiers?

Python Identifier is the name we give to identify a variable, list, tuple, sets, dictionary, function, class, module or other object. That means whenever we want to give an entity a name, that's called identifier. Sometimes variable and identifier are often misunderstood as same but they are not. Well for clarity, let's see what is a variable?

What is a Variable in Python?

A variable, as the name indicates is something whose value is changeable over time. In fact a variable is a memory location where a value can be stored. Later we can retrieve the value to use. But for doing it we need to give a nickname to that memory location so that we can refer to it. That's identifier, the nickname.

What is print()?

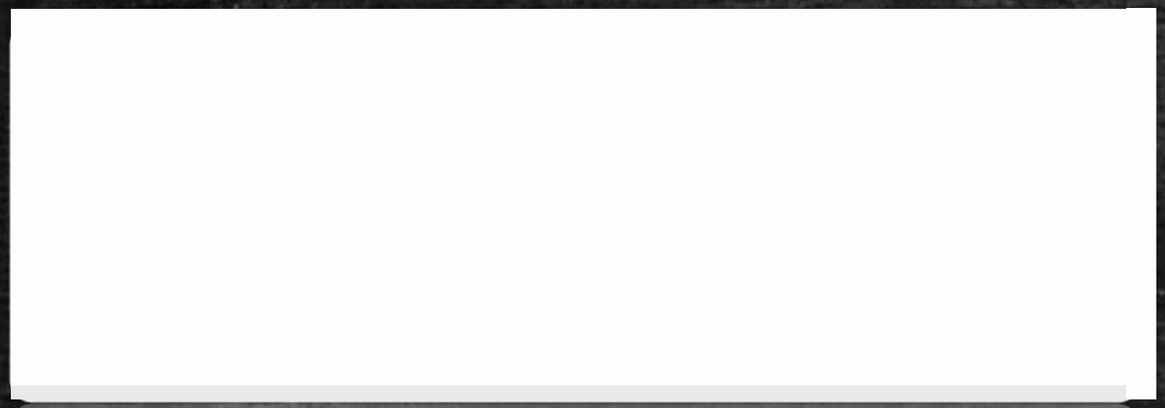
The print() function prints the specified message to the screen, or other standard output device. The message can be a string, or any other object, the object will be converted into a string before written to the screen.

```
1 print("this is sample print string printing on screen using print() function ")
```

```
this is sample print string printing on screen using print() function
```

Keywords

The following identifiers are used as reserved words, or keywords of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here (case-sensitive):



Variables :A variable, as the name indicates is something whose value is changeable over time. In fact a variable is a memory location where a value can be stored

```
1  age=33   # int variable
2  print(age)
3  name='ravi' # String variable
4  print(name)
5  # Get the variable type using type() function
6  print('age Variable Type is :',type(age))
7  print('name Variable Type is :',type(name))
```

```
33
ravi
age Variable Type is : <class 'int'>
name Variable Type is : <class 'str'>
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously

```
a = b = c =55
print('a variable Value is : ',a)
print('b variable Value is : ',b)
print('c variable Value is : ',c)
```

```
a variable Value is :  55
b variable Value is :  55
c variable Value is :  55
```

Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result 2. \ can be used to escape quotes:

```
print('spam eggs')  # single quotes
print('doesn\'t') # use \' to escape the single quote...
print("doesn't") # ...or use double quotes instead
print('"Yes," they said.')
print("\"Yes,\" they said.")
print('"Isn\'t," they said.')
```

```
spam eggs
doesn't
doesn't
"Yes," they said.
"Yes," they said.
"Isn't," they said.
```

Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Pascal or C); parentheses (()) can be used for grouping. For example:

```
print(2 + 2)
print(50 - 5*6)
print((50 - 5*6) / 4)
print(8 / 5)  # division always returns a floating point number
```

```
4
20
5.0
1.6
```

Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

```
print(17 / 3)  # classic division returns a float
print(17 // 3)  # floor division discards the fractional part
print(17 % 3) # the % operator returns the remainder of the division
print(5 * 3 + 2)  # result * divisor + remainder
```

```
5.666666666666667
5
2
17
```

it is possible to use the ** operator to calculate powers

```
print (5 ** 2) # 5 Squared
print (2 ** 7) # 2 to the power of 7
```

```
25
128
```

The equal sign (=) is used to assign a value to a variable.

'+' for Concatenating two strings for string data type variables And Addition for Integer data type variables

```
a='Vikranth'
b='Reddy'
c= a + b # + working as Concatenation operator
print(c)
```

```
VikranthReddy
```

```
a = 10
b = 111
c = a+b  # + working as adition operator
print(c)
```

```
121
```

Variable Type Change (Casting)

Python Variables data type created at dynamically based on data. If we want to convert different datatype we ca use

Casting functions.

int() - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)

float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)

str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
print(x,y,z)
```

```
1 2 3
```

```
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
print(x,y,z,w)
```

```
1.0 2.8 3.0 4.2
```

```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
x,y,z
```

```
Out[8]: ('s1', '2', '3.0')
```

```
# Declare a variable and initialize it
A = "55"
B = 66
C = int(A)+B
C
```

```
Out[9]: 121
```

F-strings

F-Strings provide a way to embed expressions inside string literals, using a minimal syntax. It should be noted that an f-string is really an expression evaluated at run time, not a constant value. In Python source code, an f-string is a literal string, prefixed with 'f', which contains expressions inside braces. The expressions are replaced with their values.

```
1  name='Sai Ram'
2  age = 28
3  f_string = f'My Name is : {name} and My Age is : {age}'
4  print(f_string)
```

```
My Name is : Sai Ram and My Age is : 28
```

```
1  a=55
2  b='ravi'
3  print(f'a value is : {a} , b variable value is : {b}')
4  fstring=f'Hi.. sample F String text A value: {a} And B Value is : {b}'
5  print(fstring)
```

```
a value is : 55 , b variable value is : ravi
Hi.. sample F String text A value: 55 And B Value is : ravi
```

```
1  x=55
2  y=66
3  str = f"X variable value is: {x} test for f-strings Y variable value is : {y}"
4  str
```

```
Out[14]: 'X variable value is: 55 test for f-strings Y variable value is : 66'
```

format() function

* `str.format()` is one of the string formatting methods in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.
* _Syntax : `{ } .format(value)`
  Parameters : `(value)` : Can be an integer, floating point numeric constant, string, characters or even variables.
  Returntype : Returns a formatted string with the value passed as parameter in the placeholder position.

The placeholders can be identified using named indexes {price}, numbered indexes {0}, or even empty placeholders {}.

```
1 name='Reshwanth'
2 age=25
3 sal=2000
4 commission=200
5 total_salary= sal+commission
6 print('My Name is : {} And My Age is : {} , my total salary is : {}'.format(name,age,total_salary))
```

```
My Name is : Reshwanth And My Age is : 25 , my total salary is : 2200
```

```
1 # empty place holders and it will occupy based on position based left to right
2 print('My Name is {name} and I am living in {loc} '.format(name='Ravi',loc='BAngalore'))
```

```
My Name is Ravi and I am living in BAngalore
```

```
1 #index value based placeholders
2 print("this is a {2} sample {1} format {0} function usage example ".format(55,66,77))
```

```
this is a 77 sample 66 format 55 function usage example
```

String Formatting

Python uses C-style string formatting to create new, formatted strings. The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like "%s" and "%d".

s – strings
d – decimal integers (base-10)
f – floating point display
c – character
b – binary
o – octal
x – hexadecimal with lowercase letters after 9
X – hexadecimal with uppercase letters after 9
e – exponent notation

```
print("{} , {}".format(1, 1.23))
print("%d , %f"%(1, 1.23))
```

```
1 , 1.23
1 , 1.230000
```

```
print("{!s}".format("this"))
print("{:s}".format("that"))
```

```
this
that
```

```
loc='Bangalore'
name='Ravi'
print("my name is : %s And i am living in : %s "%(name,loc))
```

```
my name is : Ravi And i am living in : Bangalore
```

```
x=55
y=44.4443
print("this is integer %d and this is float value : %f "%(x,y))
```

```
this is integer 55 and this is float value : 44.444300
```

Python Data Types

There are different types of data types in Python. Some built-in Python data types are:

Numeric data types: int, float, complex

String data types: str

Sequence types: list, tuple, range

Binary types: bytes, bytearray, memoryview

Mapping data type: dictionary

Boolean type: bool

Set data types: set, frozenset

- Data Structures (collections): list, tuple, range, set, frozenset,

  dictionary

```python
#create a variable with integer value.
a=100
print("The type of variable having value", a, " is ", type(a))
#create a variable with float value.
b=10.2345
print("The type of variable having value", b, " is ", type(b))
#create a variable with complex value. (real part + img part)
c=100+3j
print("The type of variable having value", c, " is ", type(c))
```

```
The type of variable having value 100  is  <class 'int'>
The type of variable having value 10.2345  is  <class 'float'>
The type of variable having value (100+3j)  is  <class 'complex'>
```

Complex integer data type

Complex numbers have a real and imaginary part, which are each implemented using double in C.
To extract these parts from a complex number z, use z.real and z.imag.

Returns a complex number constructed from arguments
Complex numbers are specified as `<real part>+<imaginary part>j`.
J (or j) represents the square root of -1 (which is an imaginary number)
Syntax `complex(re,im) ` a complex number with real part re, imaginary part im. im defaults to zero.
Complex numbers have a real and imaginary part, which are each implemented using double in C. To extract these parts from a complex number z, use z.real and z.imag.

```
x = 10+55j  #complex    (method or function )
print('Complex Data Type : ',type(x))
print(x)
print('real value  : ' ,x.real)
print('image value : ',x.imag)
```

```
Complex Data Type :  <class 'complex'>
(10+55j)
real value  :  10.0
image value :  55.0
```

Range data Type

syntax: range(start, stop, step)

The range() function is used to generate a sequence of numbers over time. At its simplest, it accepts an integer and returns a range object (a type of Iterable)

- start: Optional. An integer number specifying at which position to start. Default is 0
- Stop: Required. An integer number specifying at which position to stop (not included).
- Step: Optional. An integer number specifying the incrementation. Default is 1

```
1 range_v = range(10)
2 print(type(range_v))
3 print(tuple(range_v))
```

```
<class 'range'>
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
1 list(range(-100, -95))
```

```
Out[5]: [-100, -99, -98, -97, -96]
```

```
1 v_range = range(3,10)
2 print('Range values are generated : ',list(v_range))
3 print('v_range data type is : ', type(v_range))
```

```
Range values are generated :  [3, 4, 5, 6, 7, 8, 9]
v_range data type is :  <class 'range'>
```

```
1 list(range(1, 20, 5))  # Increment by 5
```

```
Out[7]: [1, 6, 11, 16]
```

LIST []

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets []. Lists might contain items of different types, but usually the items all have the same type.

List is a collection which is ordered and changeable. Allows duplicate members.

In Python lists are written with square brackets.

Note: Python Lists replace Arrays (from most programming languages)

```
squares = [1, 4, 9, 16, 25]
print(squares)
type(squares)
```

```
[1, 4, 9, 16, 25]
Out[3]: list
```

```
mylist = ["Green","Green", "Red", "Yello",1,1,2,3,4,True,False]
print(mylist)
type(mylist)
```

```
['Green', 'Green', 'Red', 'Yello', 1, 1, 2, 3, 4, True, False]
Out[4]: list
```

lists can be indexed and sliced

Range of Indexes

- You can specify a range of indexes by specifying where to start and where to end the range.
- When specifying a range, the return value will be a new list with the specified items.

```
1  mylist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
2  mylist[0:3]
```

```
Out[8]: ['apple', 'banana', 'cherry']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

Slicing starting from minimum index 0

```
1  mylist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
2  mylist[:6]
```

```
Out[10]: ['apple', 'banana', 'cherry', 'orange', 'kiwi', 'melon']
```

Slicing Ending with maximum index value available in list

```
1  mylist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
2  mylist[2:]
```

```
Out[11]: ['cherry', 'orange', 'kiwi', 'melon', 'mango']
```

List methods

Python provides methods that operate on lists. For example, append adds a new element to the end of a list

Inserting or appending new item or value into LIST using `append`

```
1  mylist = ["Jan", "Feb", "Mar","Apr"]
2  print('before adding new value :',mylist)
3  mylist.append("May")
4  print('After adding new value : ',mylist)
```

```
before adding new value : ['Jan', 'Feb', 'Mar', 'Apr']
After adding new value :  ['Jan', 'Feb', 'Mar', 'Apr', 'May']
```

if we want add any value in between we can go with `insert` specifying index value

```
1  mylist = ["Jan", "Mar", "Apr"]
2  print('Before inserting :',mylist)
3  mylist.insert(1, "Feb")
4  print('After inserting : ',mylist)
```

```
Before inserting : ['Jan', 'Mar', 'Apr']
After inserting :  ['Jan', 'Feb', 'Mar', 'Apr']
```

extend takes a list as an argument and appends all of the elements

```
1  mylist = ["Jan", "Feb", "Mar","Apr"]
2  monthlist = ["May","June","Jul"]
3  mylist.extend(monthlist)
4  mylist
```

```
Out[16]: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'Jul']
```

sort arranges the elements of the list from **low** to **high**

```
1  unsortlist = ['a','d','e','c','f','h','g','b']
2  unsortlist.sort()
3  unsortlist
```

```
Out[20]: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

sort with **reverse** arranges the elements of the list from **high** to **low**

```
1  unsortlist = ['a','d','e','c','f','h','g','b']
2  unsortlist.sort(reverse=True)
3  unsortlist
```

```
Out[22]: ['h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
```

List Length

To determine how many items a list has, use the len() function

```
1  lenlist = [1,2,3,4,5,6,7,8,9,10]
2  len(lenlist)

Out[21]: 10
```

Deleting elements from LIST

Removing individual items from LIST we ca use `remove` method with value

```
1  mylist = ["Jan", "Feb", "Mar","Apr"]
2  print('Before Removing :',mylist)
3  mylist.remove("Feb")
4  print('After removing :',mylist)

Before Removing : ['Jan', 'Feb', 'Mar', 'Apr']
After removing : ['Jan', 'Mar', 'Apr']
```

Remove last item from list is `pop()` it will be removed last item from list

Removed item can be returned to variable.

```
1  mylist = ["jan", "feb", "mar","apr"]
2  removed_var=mylist.pop()
3  print(mylist)
4  print(removed_var)
```

```
['jan', 'feb', 'mar']
apr
```

Removing item using index

```
1  mylist = ["jan", "feb", "mar","apr"]
2  removed_var=mylist.pop(1)
3  print(mylist)
4  print(removed_var)
```

```
['jan', 'mar', 'apr']
feb
```

Removing item using del method based on index

```python
mylist = ["jan", "feb", "mar"]
del mylist[1]
print(mylist)
```

```
['jan', 'mar']
```

Deleting entire object

```python
mylist = ["jan", "jan", "jan"]
del mylist
print(mylist)
```

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

Clearing all values using clear method and making empty list []

```python
mylist = ["jan", "feb", "mar"]
mylist.clear()
print(mylist)
```

```
[]
```

COPY

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.
There are ways to make a copy, one way is to use the built-in List method copy().

```
1  thislist = ["vikranth", "reshwanth", "Pragna","Ravi","Raj"]
2  mylist = thislist.copy()
3  print(mylist)
4  type(mylist)
```

```
['vikranth', 'reshwanth', 'Pragna', 'Ravi', 'Raj']
Out[32]: list
```

copylist will only be a reference to thislist. If we add a value in thislist and same value will be available in copylist. Bcz its referring
Original list.

```
1  thislist = ["apple", "banana", "cherry"]
2  copylist=thislist
3  thislist.append("kiwi")
4  print('thislist values : ',thislist)
5  print('copylist values : ',copylist)
```

```
thislist values :  ['apple', 'banana', 'cherry', 'kiwi']
copylist values :  ['apple', 'banana', 'cherry', 'kiwi']
```

Tuple

A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets ().
Python tuple is much like a list except that it is immutable or unchangeable once created.
Tuples use parentheses and creating them is as easy as putting different items separated by a comma between parentheses.

```
1  a_tuple = (1,2,3,4,4,5,'Sample','Python','Pyspark')
2  print(type(a_tuple))
3  print(a_tuple)
```

```
<class 'tuple'>
(1, 2, 3, 4, 4, 5, 'Sample', 'Python', 'Pyspark')
```

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.
When specifying a range, the return value will be a new tuple with the specified items.

```
1  thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
2  print(thistuple[2:5])
```

```
('cherry', 'orange', 'kiwi')
```

Negative Indexing

Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

```
mytuple = ("apple", "banana", "cherry")
print('Negative Index -1 Value: ',mytuple[-1])
```

```
Negative Index -1 Value:  cherry
```

Slicing

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
mytuple = ("apple", "banana", "cherry","kiwi","mango","orange","dragen")
print(mytuple)
print(mytuple[:4])
```

```
('apple', 'banana', 'cherry', 'kiwi', 'mango', 'orange', 'dragen')
('apple', 'banana', 'cherry', 'kiwi')
```

```
mytuple = ("apple", "banana", "cherry","kiwi","mango","orange","dragen")
print(mytuple)
print(mytuple[3:])
```

```
('apple', 'banana', 'cherry', 'kiwi', 'mango', 'orange', 'dragen')
('kiwi', 'mango', 'orange', 'dragen')
```

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

```
1  x = ("apple", "banana", "cherry")
2  x = list(x)  # Converting TUPLE into LIST
3  x[2] = "kiwi"  # Updating a value based index in LIST
4  x = tuple(x)  # Converting back to TUPLE From LIST
5  print(x)
6  print('X type is : ',type(x))
```

```
('apple', 'banana', 'kiwi')
X type is :  <class 'tuple'>
```

Remove Items

- Tuples are unchangeable, so you cannot remove items from it, but you can delete the tuple completely
- The del keyword can delete the tuple completely

```
1  thistuple = ("apple", "banana", "cherry")
2  del thistuple
3  print(thistuple) #this will raise an error NameError: name 'thistuple' is not defined
```

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

Join Two Tuples

To join two or more tuples you can use the + operator

```
tuple1 = ("a", "b" , "c","d") # its similar to append option. just it will add end of the tuple.
tuple2 = ("e","f","g")
tuple3 = tuple1 + tuple2
print(tuple3)
type(tuple3)
```

```
('a', 'b', 'c', 'd', 'e', 'f', 'g')
Out[21]: tuple
```

Nested Tuples

It is also possible to create a tuple of tuples or tuple of lists.

```
list1 = ['Python', 'pyspark', 1, 3.1415]
list2 = [('a', 'b'), ('c', 'd')]  # List of tuples is possible too!
tuple1 = (1, 2, 3, 4, 5)
tuple2 = tuple(list1 + list2)+ tuple1  # Concatenating the list and converting to tuple.
#Then adding two tuples and appening it in another tuple
tuple2
```

```
Out[25]: ('Python', 'pyspark', 1, 3.1415, ('a', 'b'), ('c', 'd'), 1, 2, 3, 4, 5)
```

Sets

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the set() function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary, a data structure that we discuss in the next section.

```
1  basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
2  basket    # show that duplicates have been removed
```

```
Out[1]: {'apple', 'banana', 'orange', 'pear'}
```

Access Sets Items

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

```
1  thisset = {"apple", "banana", "cherry"}
2  for x in thisset:
3    print(x)
```

```
apple
cherry
banana
```

Add Items

To add one item to a set use the add() method.

```
1  thisset = {"apple", "banana", "cherry"}
2  thisset.add("range")
3  print(thisset)
```

```
{'range', 'apple', 'cherry', 'banana'}
```

To add more than one item to a set use the update() method.

```
1  thisset = {"apple", "banana", "cherry"}
2  thisset.update(["abc","test1","test2"])
3  print(thisset)
```

```
{'test2', 'test1', 'banana', 'apple', 'cherry', 'abc'}
```

Get the Length of a Set using len() method.

```
1  a_set ={1,2,3,4,5,6,7,8,9,10}
2  len(a_set)
```

```
Out[9]: 10
```

Remove Item from Sets

To remove an item in a set, use the remove() or the discard() method.

```
1  thisset = {"apple", "banana", "cherry"}
2  thisset.remove("banana")
3  print(thisset)
```

```
{'apple', 'cherry'}
```

```
1  thisset = {"apple", "banana", "cherry","kiwi"}
2  thisset.discard("banana")
3  print(thisset)
```

```
{'kiwi', 'apple', 'cherry'}
```

You can also use the pop(), method to remove an item, but this method will remove the last item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the pop() method is the removed item.

```
1  thisset = {"apple", "banana", "cherry","kiwi"}
2  x = thisset.pop()
3  print(x)
4  print(thisset)
```

```
kiwi
{'apple', 'cherry', 'banana'}
```

The clear() method empties the set

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
thisset.clear()
print(thisset)
```

```
{'apple', 'cherry', 'banana'}
set()
```

The del keyword will delete the set completely

```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset)
```

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

Join Two Sets

There are several ways to join two or more sets in Python.

You can use the union() method that returns a new set containing all items from both sets.

update() method that inserts all the items from one set into another

```
set1 = {"a", "b" , "c",1,2,3,4,5,6,66,99}
set2 = {1, 2, 3,"c",23,45,"test11"}
set3 = set2.union(set1)
set2.update(set1)
print('union set : ',set3)
print('Updating existing set : ',set2)
```

```
union set :  {1, 2, 3, 4, 5, 6, 66, 'a', 'c', 23, 99, 'b', 45, 'test11'}
Updating existing set :  {1, 2, 3, 'b', 4, 5, 6, 66, 'a', 99, 45, 'c', 23, 'test11'}
```

Intersection : will get common matching data items from both datasets.

A union B, B union A and A intersection B and B intersection A will get same results

```
A = {'a', 'b', 'c', 'd','e'}
B = {'c','d','e', 'f', 'g'}
# Equivalent to common matching values
print(A.intersection(B))
```

```
{'e', 'c', 'd'}
```

Difference (Minus or Subtract) Set Operator

Subtracting right side data set values in left dataset and displaying remaining left dataset values.

Note: A Difference B and B Difference A will get different result set

```
A = {'a', 'b', 'c', 'd'}
B = {'c', 'f', 'g'}
print(A.difference(B)) # Equivalent to A-B ( it is equal to minus operator)
print(B.difference(A)) # Equivalent to B-A ( it is equal to minus operator)
```

```
{'a', 'b', 'd'}
{'g', 'f'}
```

Dictionaries

A dictionary is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type.
You can think of a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.

The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.
A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

```
1  thisdict = {"brand": "Mahindra","model": "XUV300","year": 2019}
2  print(thisdict)
3  type(thisdict)
```

```
{'brand': 'Mahindra', 'model': 'XUV300', 'year': 2019}
Out[2]: dict
```

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets

```
1  thisdict = {"brand": "Mahindra","model": "XUV300","year": 2019}
2  thisdict['model']
```

```
Out[9]: 'XUV300'
```

There is also a method called get() that will give you the same result

```
1  thisdict = {"brand": "Mahindra","model": "XUV300","year": 2019}
2  print(thisdict.get("brand"))
```

```
Mahindra
```

Change Values

You can change the value of a specific item by referring to its key name

```
1  thisdict = {"brand": "Mahindra","model": "XUV300","year": 2019}
2  thisdict["year"] = "2021"
3  print('Updated Year key value Is : ',thisdict["year"])
4  print(thisdict)
```

```
Updated Year key value Is :  2021
{'brand': 'Mahindra', 'model': 'XUV300', 'year': '2021'}
```

Loop Through a Dictionary

You can loop through a dictionary by using a for loop.

When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

```python
thisdict = {"brand": "Mahindra","model": "XUV300","year": 2019}
#Print all key names in the dictionary, one by one:
for a in thisdict:
  print("Dictinary Key Name : ",a)
  print('Dictionary Value  :',thisdict[a])
```

```
Dictinary Key Name :  brand
Dictionary Value  : Mahindra
Dictinary Key Name :  model
Dictionary Value  : XUV300
Dictinary Key Name :  year
Dictionary Value  : 2019
```

You can also use the values() method to return values of a dictionary

```python
for a in thisdict.values():
  print('Keys :',a)
```

```
Keys : Mahindra
Keys : XUV300
Keys : 2019
```

You can also use the items() method to return keys and values of a dictionary

```
thisdict = {"brand": "Mahindra","model": "XUV300","year": 2019}
for a,b in thisdict.items():
  print('Keys :',a)
  print('Values : ',b)
```

```
Keys : brand
Values :  Mahindra
Keys : model
Values :  XUV300
Keys : year
Values :  2019
```

Getting key names using keys() method

```
for x in thisdict.keys():
  print('Dictionary Key is : ',x)
```

```
Dictionary Key is :  brand
Dictionary Key is :  model
Dictionary Key is :  year
```

Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword

```
thisdict = {"brand": "Mahindra","model": "XUV300","year": 2020}
v_col="model"
if  v_col in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
else:
  print("NO Value is not available in Dict")
```

```
Yes, 'model' is one of the keys in the thisdict dictionary
```

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it

```
thisdict = {"brand": "Mahindra","model": "XUV300","year": 2020}
thisdict["color"] = "red"
thisdict["year"] = 2021
print(thisdict)
```

```
{'brand': 'Mahindra', 'model': 'XUV300', 'year': 2021, 'color': 'red'}
```

Removing Items

There are several methods to remove items from a dictionary

```
1  thisdict = {"brand": "Mahindra","model": "XUV300","year": 2020}
2  #The pop() method removes the item with the specified key name
3  print("Before removing Dict items :",thisdict)
4  thisdict.pop("year")
5  print('After removing item : ',thisdict)
```

```
Before removing Dict items : {'brand': 'Mahindra', 'model': 'XUV300', 'year': 2020}
After removing item :  {'brand': 'Mahindra', 'model': 'XUV300'}
```

The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead)

```
1  thisdict = {"brand": "Mahindra","model": "XUV300","year": 2020}
2  print("Before removing Dict items :",thisdict)
3  thisdict.popitem()
4  print('after removing dit items : ',thisdict)
```

```
Before removing Dict items : {'brand': 'Mahindra', 'model': 'XUV300', 'year': 2020}
after removing dit items :  {'brand': 'Mahindra', 'model': 'XUV300'}
```

The del keyword removes the item with the specified key name

```
thisdict = {"brand": "Mahindra","model": "XUV300","year": 2020}
print(thisdict)
del thisdict["year"]
print(thisdict)
```

```
{'brand': 'Mahindra', 'model': 'XUV300', 'year': 2020}
{'brand': 'Mahindra', 'model': 'XUV300'}
```

The del keyword can also delete the dictionary completely

```
thisdict = {"brand": "Mahindra","model": "XUV300","year": 2020}
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

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

The clear() method empties the dictionary

```
thisdict = {"brand": "Mahindra","model": "XUV300","year": 2020}
thisdict.clear()
print(thisdict)
type(thisdict)
```

```
{}
Out[26]: dict
```

Copy a Dictionary

* You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.
There are ways to make a copy, one way is to use the built-in Dictionary method copy()

```
1  thisdict = {"brand": "Mahindra","model": "XUV300","year": 2020}
2  print(thisdict)
3  mydict = thisdict.copy()
4  print(mydict)
5  type(mydict)
```

```
{'brand': 'Mahindra', 'model': 'XUV300', 'year': 2020}
{'brand': 'Mahindra', 'model': 'XUV300', 'year': 2020}
Out[27]: dict
```

Nested Dictionaries

A dictionary can also contain many dictionaries, this is called nested dictionaries.

```
1  myFriends = {"friend1" : {"name" : "Raj","year" : 1981},
2               "friend2" : {"name" : "Prasad","year" : [1984,1955]},
3               "friend3" : {"name" : "Mahesh","year" : 1985},
4               "friend4" : {"name" : "Sridhar","year" : 1986}
5              }
6  print(myFriends['friend2']["year"])
```

```
[1984, 1955]
```

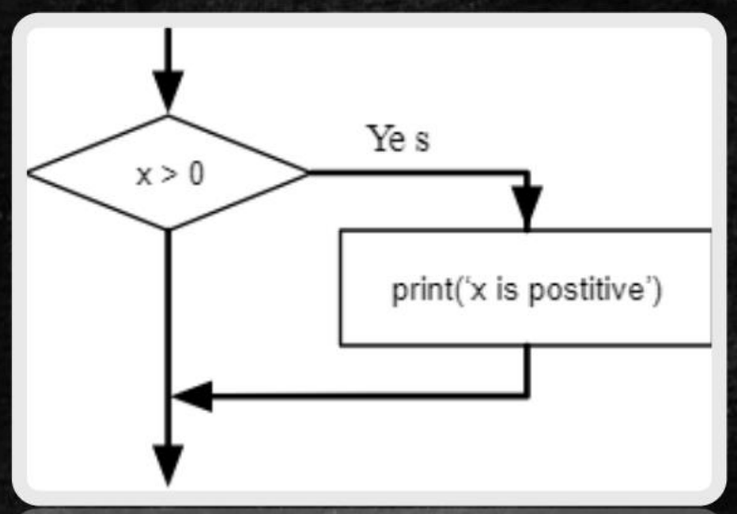
Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behaviour of the program accordingly. Conditional statements give us this ability. The simplest form is the if statement:

```
1  x=1
2  if x > 0 :
3      print('x is positive')

x is positive
```

The Boolean expression after the if statement is called the condition. We end the if statement with a colon character (:) and the line(s) after the if statement are indented.



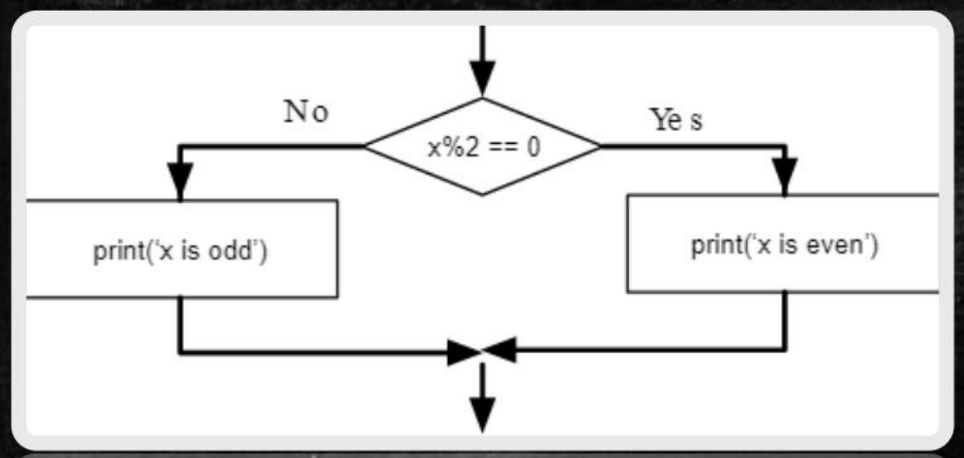
Alternative execution (If Else)

A second form of the if statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
x=55
if x%2 == 0 :
    print('x is even')
else :
    print('x is odd')
```

```
x is odd
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed.



Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

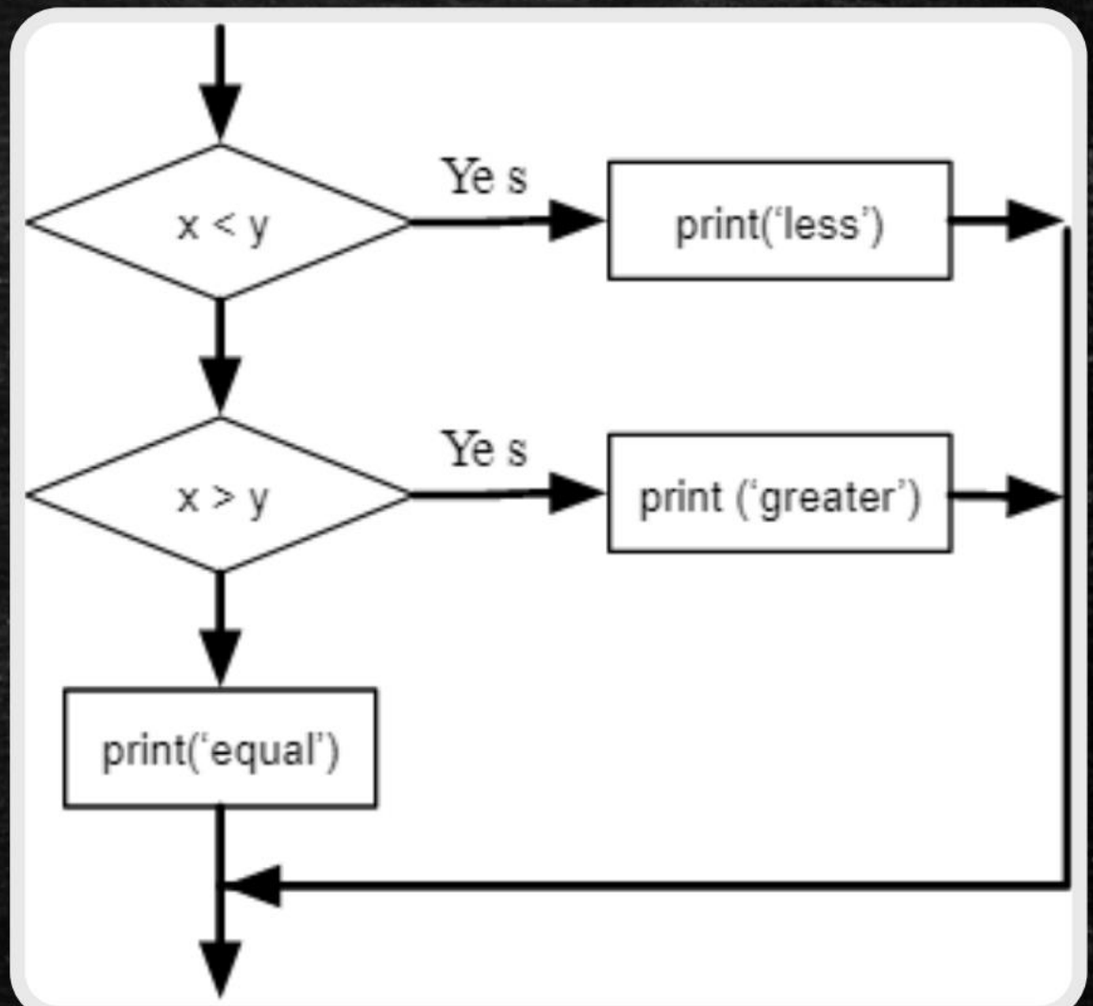
If-Then-Else Logic

Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

```
1  x=55
2  y=66
3  if x < y:
4      print('x is less than y')
5  elif x > y:
6      print('x is greater than y')
7  else:
8      print('x and y are equal')
```

```
x is less than y
```

elif is an abbreviation of “else if.” Again, exactly one branch will be executed.

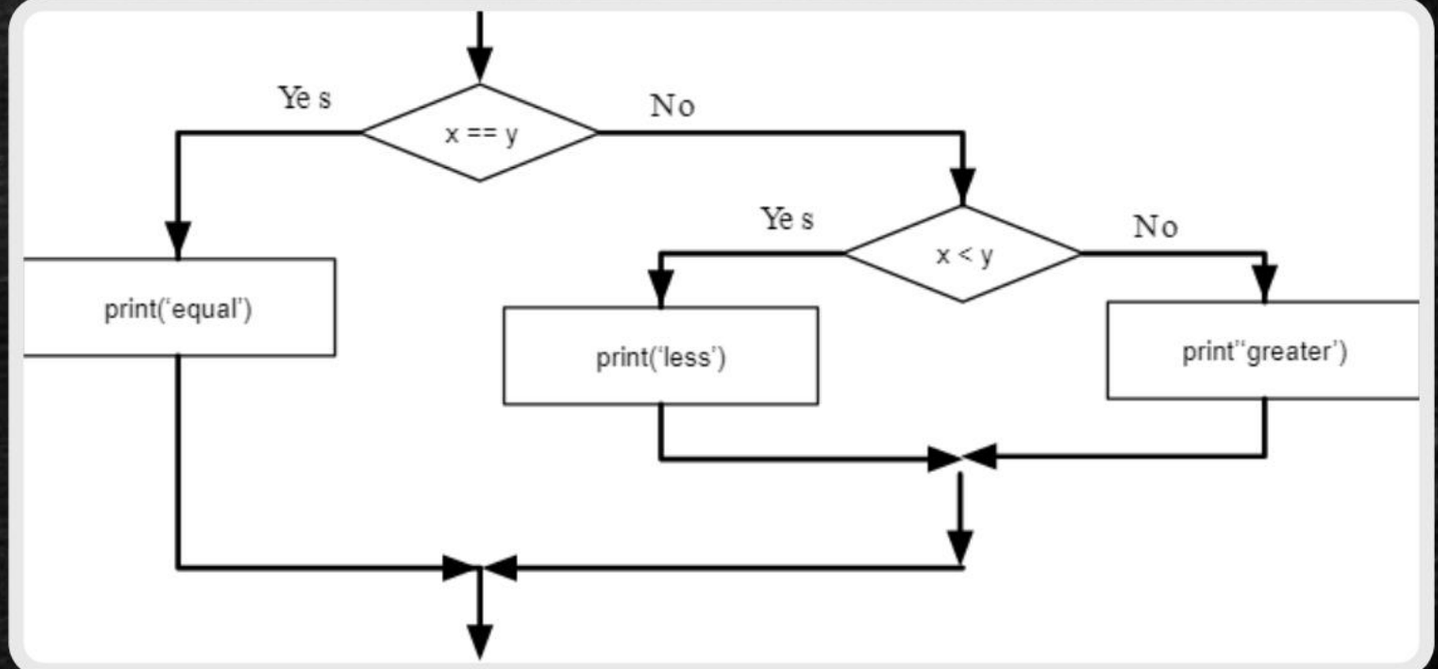


Nested if conditionals

One conditional can also be nested within another. We could have written the three-branch example like this:

```
1  x=55
2  y=66
3  if x == y:
4      print('x and y are equal')
5  else:
6      if x < y:
7          print('x is less than y')
8      else:
9          print('x is greater than y')


x is less than y
```



The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

```
1  a=99
2  b=88
3  if a > b: print("a is greater than b")


a is greater than b
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

```
1  a = 980
2  b = 550
3  print('A is greater than b Statement',a) if a > b else print('a is less than b Statement',b)


A is greater than b Statement 980
```

One line if else statement, with 3 conditions:

This technique is known as Ternary Operators, or Conditional Expressions.
Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being true or false. It was added to Python in version 2.5. It simply allows to test a condition in a single line replacing the multiline if-else making the code compact.

```
a = 990
b = 990
print("A Greater Than B : ",b) if a > b else print("its a equal to B =") if a == b else print("B is Greater Than A",a)
```

```
its a equal to B =
```

```
a = 990
b = 880
print("A Greater Than B : ",b) if a > b else print("its a equal to B =") if a == b else print("B is Greater Than A",a)
```

```
A Greater Than B :  880
```

```
a = 770
b = 880
print("A Greater Than B : ",b) if a > b else print("its a equal to B =") if a == b else print("B is Greater Than A",a)
```

```
B is Greater Than A 770
```

Nested If

You can have if statements inside if statements, this is called nested if statements.

```
x = 50

if x < 10:
  print("below ten,")
  if x > 20:
    print("and also below 10!")
  else:
    print("but not above 10.")
else:
  print('actual X value is : ',x)
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

```
actual X value is :  50
and also above 20!
```

The pass Statement

if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

```
a = 33
b = 200

if b > a:
  pass
```

Python Loops

There are two types of loops available in python. Those are for loops and while loops.

- for loop
- while loop

For Loops with range()

A for loop is a programming structure where a user-defined block of code runs a specified number of times.

Another method to print the same statement three times is to use a for loop.

The basic structure of a for loop in Python is below:

`for var in range(num):`

`code`

```
for a in range(5):
  print(a)
```

```
0
1
2
3
4
```

For loops with lists

For loops can also be run using Python lists.
If a list is used, the loop will run as many times as there are items in the list.
The general structure is:

`for <var> in <list>:`

`  <statements>`

```
my_list = ['raj','prasad','mahesh','sridhar']
for item in my_list:
    print('My Name is : ',item)
```

```
My Name is :  raj
My Name is :  prasad
My Name is :  mahesh
My Name is :  sridhar
```

The break Statement In For Loop

With the break statement we can stop the loop before it has looped through all the items:

```
fruits = ["apple", "banana", "cherry","orange","mango","kiwi"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

```
apple
banana
```

The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next

```
fruits = ["apple", "banana", "cherry","orange","mango","kiwi"]
for x in fruits:
  if x == "orange":
    continue
  print(x)
```

```
apple
banana
cherry
mango
kiwi
```

```
fruits = ["apple", "banana", "cherry","orange","mango"]
for x in fruits:
  if x in ("banana","cherry"):
    print('Inside If Condition',x)
    continue
  print('Out side if condition ',x)
```

```
Out side if condition  apple
Inside If Condition banana
Inside If Condition cherry
Out side if condition  orange
Out side if condition  mango
```

The while Loop

With the while loop we can execute a set of statements as long as a condition is true.

syntax: `while condition:`
`statements`

```
#Print i as long as i is less than 10 with increment by 2
i = 0
while i < 10:
  print(i)
  i += 2
```

```
0
2
4
6
8
```

The break Statement

With the break statement we can stop the loop even if the while condition is true:

```
i = 0
while i < 5:
  i +=1
  if i == 4:
    break
  print(i)
```

```
1
2
3
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next

```
i = 0
while i < 8:
  i += 1
  if i == 5:
    continue
  print(i)
```

```

```

The else Statement

With the else statement we can run a block of code once when the condition no longer is true

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

```
1
2
3
4
5
i is no longer less than 6
```

Creating list using `For Loop` , `If Condition ` And `Range()`

```
my_list = [x for x in range(20) if x%2==0]
print(my_list)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

To find sum from 0 to 1000

```
v_num = 0
target_value =1000
for i in range(target_value+1):
    v_num = v_num+i
print('Sum of 1000 Values : ',v_num)
```

```
Sum of 1000 Values :  500500
```

To find sum from 0 to 1000 (only even)

```
v_even = 0
var_list = []
for i in range(1001):
    if i%2 ==0:
        var_list.append(i)
        v_even = v_even+i

print('Sum of Even Numbers from 1 To 1000 using variable : ', v_even)
print('Sum Of Even Numbers From 1 to 1000 using list : ',sum(var_list))
```

```
Sum of Even Numbers from 1 To 1000 using variable :  250500
Sum Of Even Numbers From 1 to 1000 using list :  250500
```

Functions

In the context of programming, a function is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements.
The keyword def introduces a function definition.
It must be followed by the function name and the parenthesized list of formal parameters.
The statements that form the body of the function start at the next line, and must be indented.
A function is a block of code which only runs when it is called.
You can pass data, known as parameters, into a function.
A function can return data as a result.

```
#Creating Function
def my_func():
  print("Hello from a function")
  print("Statement two")
  print("Statement Three")
#Calling or Executing function
my_func()
```

```
Hello from a function
Statement two
Statement Three
```

```
#Creating a function with two arguments  a and b
def my_sum(a,b):
  print('Sum of a and b value is : ',a+b)
  return a+b
#calling or executing a function
my_sum(90,20)
```

```
Sum of a and b value is :  110
Out[3]: 110
```

Arguments (parameters)

Information can be passed into functions as arguments.

Function arguments: positional, keyword

A function is most useful when arguments are passed to the function.
New values for times are processed inside the function.
This function is also a ```'positional' argument, vs a keyword` argument.
Positional arguments are processed in the order they are created in.

```
def my_func(fname,age):
  print("My Name is : ",fname)
  print("MY Age is : ",age)
```

```
#Positional Arguments are processed in order
my_func("Raveendra",35)
```

```
My Name is :  Raveendra
MY Age is :  35
```

Keyword Arguments are processed by key, value and can have default values
You can also send arguments with the key = value syntax.
One handy feature of keyword arguments is that you can set defaults and only change the defaults you want to change.

```
my_func(age=55,fname="Rajkumar")
# optional
```

```
My Name is :  Rajkumar
MY Age is :  55
```

Default Parameter Value

The following example shows how to use a default parameter value.
If we call the function without argument, it uses the default value:

```
def my_function(name,age,loc = "India"):
  print("My Name is  : ",name)
  print("My Age is : ",age)
  print("I am from : ",loc)
# Calling a function without 3rd argument. it will considar default value.
my_function("Ravi",33)
```

```
My Name is  :  Ravi
My Age is :  33
I am from :  India
```

Variable Number of Arguments (`*args`)

In cases where you don't know the exact number of arguments that you want to pass to a function,

you can use the following syntax with *args:

```
def my_sum(*args):
    return sum(args)
my_sum(2,3,4,5,23,54,7,43,23,23,23,2354,34,34,23,23,43)
```

```
Out[15]: 2721
```

Nested functions

```
def func_outside():
  print('outside')
  def func_inside():  # Local objects
    print('inside')
  func_inside()
func_outside()
```

```
outside
inside
```

```
def function1(): # outer function
    x = 2 # A variable defined within the outer function
    print('x value is before updating : ',x)
    x = 8
    print('x value is after update ',x)
    def function2(a): # inner function
       # Let's define a new variable within the inner function
       # rather than changing the value of x of the outer function
        x = 6
        print('inner function x value is : ',x)
        print ('Inner Function Value : ',a+x)
        return x
    print('Before executing func x : ' ,x)
    y=function2(3)
    print('inner function x value is : ',y)
    print ('after executing Func x : ',x)
    # to display the value of x of the outer function
function1()
```

```
x value is before updating :  2
x value is after update  8
Before executing func x :  8
inner function x value is :  6
Inner Function Value :  9
inner function x value is :  6
after executing Func x :  8
```

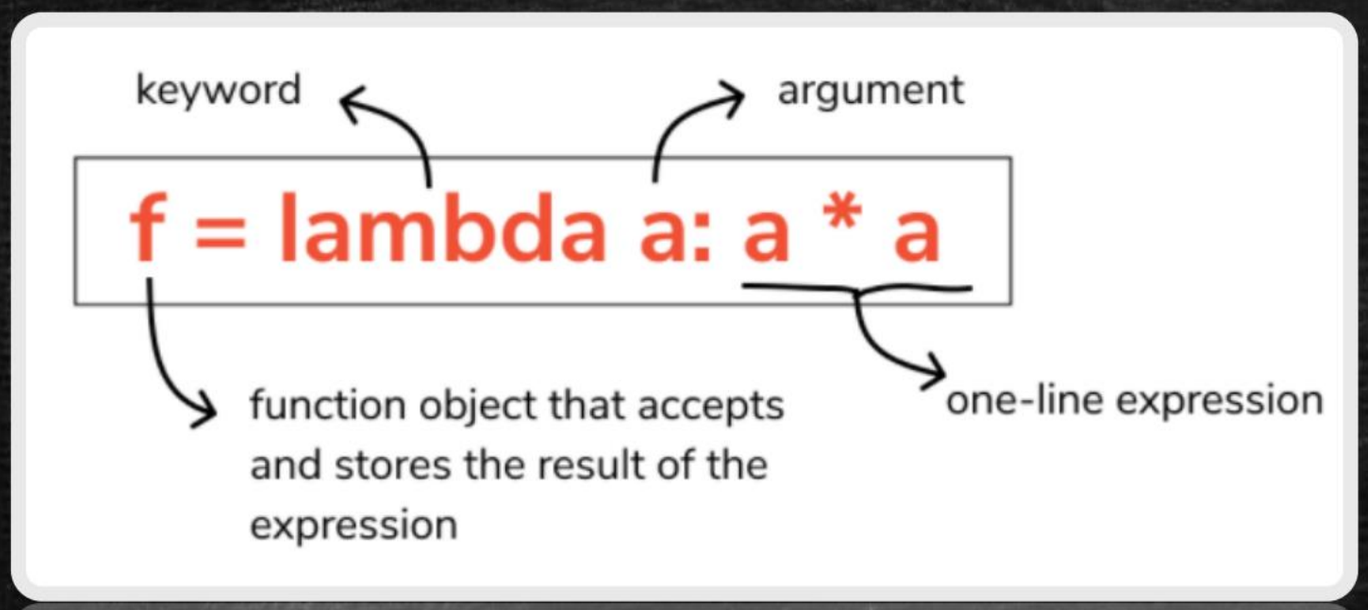
Python Lambda Function

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can have one line expression.

Syntax

`lambda arguments : expression`




```
x = lambda a,b : a * b
print(x(50,40))
```

```
2000
```

```
x = lambda a, b : a + b
print(x(10, 50))
```

```
60
```

```
divide = lambda val : val/2
print(list(map(divide,[11,12,13,14])))
```

```
[5.5, 6.0, 6.5, 7.0]
```

Local Vs Global Variables

if we want to change the Global Variable value inside functions Use `GLOBAL` Keyword..

With GLOBAL keyword making global.

Without global keyword default variable value is inside function (scope is within function and its wont update outside
Variable)

```python
y = 55
print('first value :',y)
y=66
print('Updated value : ',y)
def func():
  global y
  y=100
  print('inside function Y variable value is : ',y)
func()
print('after function out side Y value is : ',y)
```

```
first value : 55
Updated value :  66
inside function Y variable value is :  100
after function out side Y value is :  100
```

```python
y = 55
print('first value :',y)
y=66
print('Updated value : ',y)
def func():
  y=100
  print('inside function Y variable value is : ',y)
func()
print('after function out side Y value is : ',y)
```

```
first value : 55
Updated value :  66
inside function Y variable value is :  100
after function out side Y value is :  66
```

Exception and Error Handling

- Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some.
- There are (at least) two distinguishable kinds of errors: syntax errors and exceptions.

Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.

Errors detected during execution are called exceptions and are not unconditionally fatal:

you will soon learn how to handle them in Python programs.

Most exceptions are not handled by programs, however, and result in error messages as shown here

Try Block

here we can write our code - main execution code we can write in try block

Exception blob

here we can handle whichever exception or error throwing by try block

```
a = 4
b = '0'
try :
    c = a+b
    print('Try Blob c value is  : ' ,c)
except Exception as c:
    print("Exception block Sum of A  + B :",int(a)+int(b))
    print("exception raised and its Handled : => ",a)
```

```
Exception block Sum of A  + B : 4
exception raised and its Handled : =>  4
```

Multiple Exception handling

```python
def divbyzero(x,y):
    try:
        return x/y
    except ZeroDivisionError as e:
        print('ZeroDivisionError and error message :',e)
    except Exception as e:
        print('Others Exception  error message : ',e)
#Calling function with second argument with zero (0)
divbyzero(10,0)
#Calling function with first argument with string value instead of int.
divbyzero('10',0)
```

```
ZeroDivisionError and error message : division by zero
Others Exception  error message :  unsupported operand type(s) for /: 'str' and 'int'
```

Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur. For example:

```
try:
     raise NameError('HiThere')
except NameError:
     print('Manually Excpetion raised by using raise ')
```

```
Manually Excpetion raised by using raise
```

If a finally clause is present, the finally clause will execute as the last task before the try statement completes. The finally clause runs whether or not the try statement produces an exception. The following points discuss more complex cases when an exception occurs:

- ✓ If an exception occurs during execution of the try clause, the exception may be handled by an except clause. If the exception is not handled by an except clause, the exception is re-raised after the finally clause has been executed.
- ✓ An exception could occur during execution of an except or else clause. Again, the exception is re-raised after the finally clause has been executed.
- ✓ If the try statement reaches a break, continue or return statement, the finally clause will execute just prior to the break, continue or return statement's execution.
- ✓ If a finally clause includes a return statement, the returned value will be the one from the finally clause's return statement, not the value from the try clause's return statement.

```python
a = 88
b = 0
d = [1,2,3,4,5,6,6]
try :
    c = a/b
    e = d[5]
    print('inside Try Block : ',c)
except ZeroDivisionError:
    print("Handle ZeroDivisionError")
except IndexError:
    print("Exception is caught is list index out of range")
else:
    print('in else block :' ,c)
finally:
    print("Finally will always exuted if we get any exception or not")
```

```
Handle ZeroDivisionError
Finally will always exuted if we get any exception or not
```

The try ... except statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

Reading and Writing Files (file I/O)

open() returns a file object, and is most commonly used with two arguments: open(filename, mode).

The key function for working with files in Python is the open() function.

The open() function takes two parameters; filename, and mode.
There are four different methods (modes) for opening a file

"r" - Read - Default value. Opens a file for reading, error if the file does not exist
"a" - Append - Opens a file for appending, creates the file if it does not exist
"w" - Write - Opens a file for writing, creates the file if it does not exist
"x" - Create - Creates the specified file, returns an error if the file exists

- Open a file
- Read or write (perform operation)
- Close the file

```
f = open("demofile2.txt", "w")
f.write("Now the file has more content! its append method or mode  \n")
f.write("this is another line writing into file using append mode")
f.close()
```

```
#open and read the file after the appending:
f = open("demofile2.txt","r")
print(f.read())
f.close()
```

```
Now the file has more content! its append method or mode
this is another line writing into file using append mode
```

f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

```
#open and read the file after the appending:
f = open("demofile2.txt","r")
print(f.readline())
f.close()
```

```
Now the file has more content! its append method or mode
```

File Handling

Reading and Writing Files

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:
"a" - Append - will append to the end of the file
"w" - Write - will overwrite any existing content

```python
filename='ravi.txt'

try:
    with open(filename) as file_object:
        contents=file_object.read()
        print(contents)
except FileNotFoundError:
    print(f"sorry ,the file {filename} does not exist")
```

```
sorry ,the file ravi.txt does not exist
```

Regular expressions

Regular expressions are almost their own little programming language for searching and parsing strings. As a matter of fact, entire books have been written on the topic of regular expressions. In this chapter, we will only cover the basics of regular expressions. For more detail on regular expressions, see:

The regular expression library re must be imported into your program before you can use it. The simplest use of the regular expression library is the search() function. The following program demonstrates a trivial use of the search function.

```
import re

pattern = '^a...s$'
test_string = 'abyss'
result = re.match(pattern, test_string)
if result:
  print("Search successful.")
else:
  print("Search unsuccessful.")
```

```
Search successful.
```

```
# Program to extract numbers from a string
import re

string = 'hello 12 hi 89. Howdy 34'
pattern = '\d+'
result = re.findall(pattern, string)
print(result)
```

```
['12', '89', '34']
```

findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

```python
import re

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
x
```

```
Out[30]: <re.Match object; span=(0, 17), match='The rain in Spain'>
```

```python
import re

txt = "The rain in Spain"
x = re.search("\s", txt)
print("The first white-space character is located in position:", x.start())
```

```
The first white-space character is located in position: 3
```

```python
import re

txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

```
['ai', 'ai']
```

```python
import re

txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

```
['The', 'rain', 'in', 'Spain']
```

```python
import re

txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)
```

```
The9rain9in9Spain
```

Examples

```python
# Python code to get the Cumulative sum of a list
def Cumulative(lists):
    cum_list = []
    length = len(lists)
    cum_list = [sum(lists[0:x]) for x in range(0, length+1)]
    return cum_list[1:]

lists = [10, 20, 30, 40, 50]
print (Cumulative(lists))
```

```
[10, 30, 60, 100, 150]
```

```python
# Get sum of values from 1 To N without using  sum function
# Using For loop and Range we can achive this.
def sum_all(n):
  a=0
  for i in range(1,n+1):
    a+=i # keep adding i into the sum - same as sum = sum +i
  return a
sum_all(10)
```

```
Out[5]: 55
```

```python
# 1. Python program to check if a string is palindrome or not
# Input : malayalam
# Output : Yes
# Input : KITIKI
# Output : No
def palindrome(word):
    for i in range(0, int(len(word)/2)):
            if word[i] != word[len(word) - i - 1]:
                return False
    else:
        return True


word = 'malayalam'
output = palindrome(word)

if output:
    print("Yes")
else:
    print("No")
```

```
Yes
```

```python
# Reverse words in a given String in Python
def reverseString(str):
    words = str.split(" ")

    rev_sentence = (" ").join(reversed(words))

    return rev_sentence

str = "string reverse sample is this"
reverseString(str)
```

```
Out[9]: 'this is sample reverse string'
```

```python
#Find length of a string in python without using len function
def lengthStr(word):
    count = 0
    for i in word:
        count += 1
    return count

word = 'hello world !'
lengthStr(word)
```

```
Out[12]: 13
```

```python
rows = 5
for i in range(rows, 0, -1):
    num = i
    for j in range(0, i):
        print(num, end=' ')
    print("\r")
```

```
5 5 5 5 5
4 4 4 4
3 3 3
2 2
1
```

```python
rows = 5
num = rows
for i in range(rows, 0, -1):
    for j in range(0, i):
        print(num, end=' ')
    print("\r")
```

```
5 5 5 5 5
5 5 5 5
5 5 5
5 5
5
```

```python
rows = 5
b = 0
for i in range(rows, 0, -1):
    b += 1
    for j in range(1, i + 1):
        print(b, end=' ')
    print('\r')
```

```
1 1 1 1 1
2 2 2 2
3 3 3
4 4
5
```

```python
n = 5
for i in range(1, n+1):
    for j in range(1, i + 1):
        print(j, end=' ')
    print("")
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```python
rows = 9
for i in range(1, rows):
    for i in range(0, i, 1):
        print(format(2 ** i, "4d"), end=' ')
    for i in range(-1 + i, -1, -1):
        print(format(2 ** i, "4d"), end=' ')
    print("")
```

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

```python
rows = 6
for num in range(rows):
    for i in range(num):
        print(num, end=" ")  # print number
    # line after each row to display pattern correctly
    print(" ")
```

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

```python
a=5
for i in range(a,-1,-1):
  for j in range(0,i+1):
    print("*",end="")
  print("\r")
```

```
******
*****
****
***
**
*
```

```python
a=5
for i in range(0,a):
  for j in range(0,i+1):
    print("*",end="")
  print("\r")
```

```
*
**
***
****
*****
```