Chapter-1 Revision Tour -I

Python Revision Tour –I

In this tutorial we will discuss the following topics

S.No.	Topics
1	Python Character Set
2	Tokens
3	Keywords
4	Identifiers
5	Literals / Values
6	Delimiters
7	Operators
8	Data Types
9	Variables
10	print () function
11	input () function
12	if-else statement
13	Ladder if – else statement (if – elif – else)
14	Nested if-else statements
15	while loop
16	for loop
17	range () function
18	break statement
19	continue statement
20	pass statement

Chapter-1 Revision Tour -I

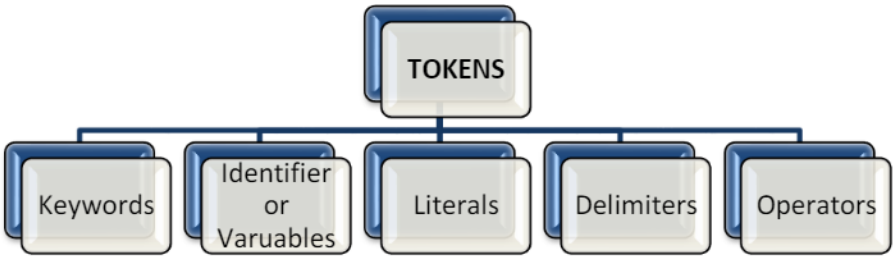
Python Character Set:

A character can represents any letter, digit, or any other sign. Following are some of the python character set.

LETTERS	A to Z and a to z
DIGITS	0 -9
SPECIAL SYMBOLS	Space, + -* ^ \ [] {} =! = < > . _ = = ; : & #, under score(_) etc.
WHITE SPACE	Blank space, horizontal tab (- >), carriage return , Newline, Form feed.
OTHER CHARACTERS	Python can process all ASCII and Unicode characters as part of data or literals.

Tokens:

The smallest individual unit in a program is known as token or lexical unit.



1) Keywords:

Keywords are special identifiers with predefined meanings that cannot change. As these words have specific meaning for interpreter, they cannot be used for any other purpose.

Chapter-1 Revision Tour -I

Some Commonly Used Keywords:

and	exec	not	continue	if	return	except	else
as	finally	or	def	import	try	class	lambda
assert	for	Pass	del	in	while	global	yield
break	from	print	elif	is	with	raise	

2) Identifiers / Variable:

Identifiers are names given to identify something. Identifiers are fundamental building blocks of a program and are used as general terminology for the names given to different part of the program that is **variables, objects, classes, functions, lists, dictionaries etc.** **NOTE: An identifier must not be a keyword of Python**

There are some rules you have to follow for naming identifiers:

* An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
* Python does not allow special characters
* Identifier must not be a keyword of Python.
* Python is a case sensitive programming language.

Thus, **Sname** and **sname** are two different identifiers in Python.

Some valid identifiers: **Mybook, file123, z2td, date_2, _no**

Some invalid identifier: **2rno,break,my.book,data-cs**

Chapter-1 Revision Tour -I

3) Literals / Values:

Literals in Python can be defined as number, text, or other data that represent values to be stored in variables. The data items which never change their value throughout the program run. There are several kinds of literals:

Type	Used For	Example
Numeric Literals : The numeric literals in Python can belong to any of the following different numerical types:		
int	Used to integer value	Age = 20
float	Used for real value	Perc = 98.5
complex	Used to part of complex number	x = 1 + 0i
bool : It is logical literal	Used to logical /Boolean	Result = True
Text Literals: is a sequence of letters surrounded by either by single or double or triple quotes.		
String Literal	Used to Unicode/Text/String	Name = 'Johni', fname="Johny"
bytes	ASCII test/string	"learnpython4cbse.com"
None Type : It is special	No - value	None(and no other object)

4) Delimiters:

Used to implement the grammatical and structure of Syntax.

Following are the python punctuators.

Delimiters	Classification
( ) [ ] { }	Grouping
. , : ; @ ' " / #	Punctuation
&= != ^= <<= >>=	Bit-wise assignment
= += -= *= /= //= %= **=	Arthmetic assignment

5) Operators:

Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

Chapter-1 Revision Tour -I

Unary Operator

Description	Operator	Example 1	Example 2
unary plus	+	If x=10, then +x means 10	If x=-10, then +x means -10
unary minus	–	If x=10, then -x means -10	If x=-10, then +x means 10

Arithmetic Operator

Addition	+	>>>10+5 => 15	>>>'Hello'+'Amjad' => HelloAmjad
Subtraction	–	>>>10-5 => 5	>>>30-70 => -40
Multiplication	*	>>>10*5 => 50	>>>'Hello'*3 => HelleoHelloHello
Division	/	>>>17/5.0 => 3.4 >>>17.0/5 => 3.4	>>28/3 => 9.3 >>28/7.0 => 4.0
Remainder / Modulo	%	>>>16%5 => 1	>>>13%5 => 3
Exponent	**	>>>2**3 => 8	>>>16**0.5 => 4
Floor division	//	>>>7.0//2 => 3 (integer division)	>>>5//2 => 2

Bitwise Operator

We assume the operation1 is X and Operation2 is Y for better understanding of these operators

Operation	Operator	Use	Description
Bitwise AND	&	X & Y	The AND operator compare two bits and generate a result of 1 if both bits are 1; otherwise, it return 0.
Bitwise exclusive OR (XOR)	^	X^Y	The EXCLUSIVE – OR operator compare two bits and returns 1 if either of the bits are 1 and gives 0 if both bits are 0 and 1.
Bitwise OR	|	X | Y	The OR operator compare two bits and generate a result of 1 if both bits are complementary; otherwise, it return 0.
Bitwise Complement	~	~X	The COMPLEMENT operator is used to invert all of the bits of the operands.

Chapter-1 Revision Tour -I

Identity operator			
Is the identity same?	is	X ix Y	Return **True** if both its operands are pointing to same object (i.e, both referring to same memory location), returns **false** otherwise
Is the identity not same?	is not	X is not Y	Return **True** if both its operands are pointing to different object (i.e, both referring to different memory location), returns **false** otherwise

Relational Operator			

Less than	<	>>>5<7 => TRUE
		>>>7<5 => FALSE
		>>> 5<7<10 => TRUE
		>>> 5<7 and 7<10 => TRUE
			>>>'Hello' < 'Amjad' => FALSE
			>>>'Amjad' < 'Hello' => TRUE

Greater than	<	>>>7>5 => TRUE >>>10<10 => FALSE	>>>'Hello' > 'Amjad' => TRUE >>>'Amjad' > 'Hello' => FALSE
Less than or equal to	>=	>>> 2<=6 => TRUE >>> 6<=4 => FALSE	<<< 'Hello' <='Amjad' => FALSE >>>'Amjad'<='Hello' => TRUE
Greater than or equal to	>=	>>> 2>=6 => FALSE >>> 6>=4 => TRUE	<<< 'Hello' >='Amjad' => TRUE >>>'Amjad'>='Hello' => FLASE
Equal to	==	>>> 5==5 => TRUE >>>10==11 => FALSE	>>>'Hello'=='Hello => TRUE >>>'Hello'=='Good Bye' => FALSE
Not equal to	!= , < >	>>>10!=11 => TRUE >>>10!=10 => FALSE	>>>'Hi!='HI' => TRUE >>>'HI'!='HI' => FALSE

Assignment Operators: and Shorthand Assignment Operators				
We assume the value of variable p as 10 for better understanding of these operators				
Assignment	=	It is use to assign the value to the variable	a=6	A will become 6
Assign quotient	/=	Divided and assign back the result to left operand	>>> p/=2	p will become 5
Assign sum	+=	Added and assign back the result to left operand	>>> p+=2	p will become 12

Chapter-1 Revision Tour -I

Assign product	*=	multiplied and assign back the result to left operand	>>> p*=2	p will become 20
Assign remainder	%=	Taken modulus using two operands and assign the result to left operand	>>> p%=2	p will become 0
Assign difference	-=	subtracted and assign back the result to left operand	>>> p-=2	p will become 8
Assign Exponent	**=	Performed exponential(power) calculation on operators and assign value to the left operand	>>> p**=2	p will become 100
Assign floor division	//=	Performed floor division on operators	>>> p//=2	p will become 5

Logical Operators

Logical AND	And	X and Y	If both the operand is true, then the condition becomes **True**
Logical OR	Or	X or Y	If any one of the operand is true, then the condition becomes **True**
Logical NOT	Not	not X	Reverses the state of the operand/ condition.

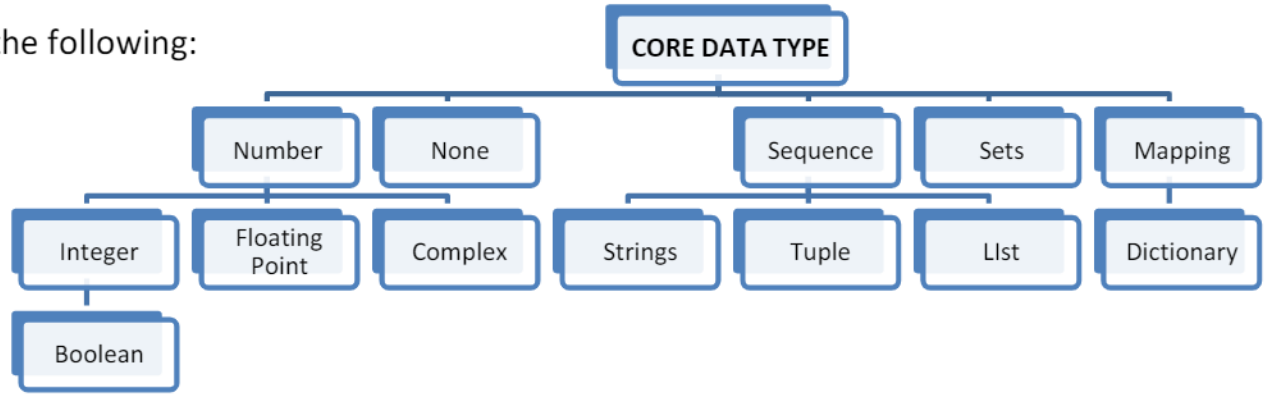
Membership Operators

Whether variable in sequence	In	The **in operator** tests if a given value is contained in a sequence or not and return True or False Accordingly	>>>3 in [1,2,3,4] **TRUE**
Whether variable not in sequence	not in	The **not in operator** tests if a given value is contained in a sequence or not and return True or False Accordingly	>>>6 not in [1,2,3,4,5] **True** (Because the value 6 is not in the sequence.) >>>4 not in [1,2,3,4,5,6] **FALSE**

Chapter-1 Revision Tour -I

DATA TYPES:

It is a set of values, and the allowable operations on those values. It can be one of the following:



Type	Data Type	Description	Range	Example
Number	Number data type stores Numerical Values. This data type is immutable i.e. value of its object cannot be changed			

	Int	Integers are whole numbers such as 5, 30, 1981,0 etc. They have no fractional parts. Integers can be positive or negative.	-2^{31} to $+2^{31} - 1$ that is -2147483648 to 2147483647 - 1 **for long integer** An unlimited range, subject to available (virtual) memory only	>>> a = 10
				>>> b = 5192L
				#example of supplying a very long value to a variable
				>>> c= 4298114

	Float	A floating point number will consist of sign (+, -) sequence of decimals digits and a dot such as 0.0, -21.9, 0.98333328.	An unlimited range, subject to available (virtual) memory on underlying machine architecture.	y= 12.36 >>>print(y) output : 12.36 >>>type(y) <type 'float'>

	Complex	Complex number in python is made up of two floating point values, one each for real and imaginary part.	Same as floating point numbers because the real and imaginary parts represented as floats.	>>> x = 1+0j
				>>> print (x.real,x.imag)
				output: 1.0 0.0

	Bool	It is a unique data type, consisting of two constants, True & False. A Boolean True value is Non-Zero, Non-Null and Non-empty.	Two values True (1) and False (0)	>>> flag = True
				>>> type(flag)
				<type 'bool'>

Chapter-1 Revision Tour -I

NONE	**None**	This is special data type with single value. It is used to signify the absence of value/false in a situation. It is represented by **None**.	
Sequence	A sequence is an ordered collection of items, indexed by positive integers. It is combination of mutable and non mutable data types.		
	Data Type	**Description**	**Example**
	String	They can have any character or sign, including space in them. These are immutable data types	Example: 'Boy', 'Navin', "ZIG 1981"

	List	A list is a mutable (can change) sequence data type and it may contained mixed data types elements	Lst= [1,2,3,4,"xyz",2.34,[10,11]]
			>>> Lst[0]
			1
			>>> Lst[4]
			'xyz'
			>>> Lst[6]
			[10, 11]
			>>> Lst[6][0]
			10

	Tuple	A tuple in Python is much like a list except
		that it is immutable (unchangeable) once created. Tuples are enclosed in ().
			t = (1,2,3,4,'abc',2.34,(10,11))
			>>> t[0]
			1
			>>> t[4]
			'abc'
			>>> t[6]
			(10,11)

Mapping	This data type is unordered and mutable. Dictionaries fall under Mappings.		
	Dictionaries	Can store any number of python objects. What they store is a key – value pairs, which are accessed using key. Dictionary is enclosed in curly brackets.	>>>d = {1:'a',2:'b',3:'c'} # here, 1,2 & 3 are the keys >>> print(d) {1: 'a', 2: 'b', 3: 'c'}

Chapter-1 Revision Tour -I

Variables:

Named labels, whose values can be manipulated during program run, are called Variables.

Creating a Variable: Python variables are created by assigning value of desired type to them

Example:

X=10.8	# variable created of numeric (floating point) type
Y = 90	# variable created of numeric (integer) type
Name = "My Name"	# variable created of string type

Multiple Assignments:

1. Assigning same value to multiple variables:

x = y = z = 100

It will assign value 100 to all three variables x, y and z.

2. Assigning multiple value to multiple variables

p, q, r = 10, 20, 30

It will assign the value order wise that is value 10 assign to variable p, value 20 assign to variable q and value 30 assign to variable r.

INPUT AND OUTPUT FUNCTION

The print () function:`

The print() function is a built-in function. It prints/outputs a specified message to the screen/consol window.

The print() function statement has the following rules:

- The print() function allows you to print out the value of a variable and strings in parenthesis.

Chapter-1 Revision Tour -I

- The print() function will print as strings everything in a comma-separated sequence of expressions, and it will separate the results with single blanks by default.
- The print() function prints strings in between quotes (either single or double). If a print statement has quotes around text, the computer will print it out just as it is written.
- To print multiple items, separate them with commas. The print() function inserts a blank between objects.
- The print() function automatically appends a newline to output. To print without a newline, use end separator after the last object.
- The ***sep*** separator is used between the values. It defaults into a space character. For example:

  >>> print (10,20,30,40, sep='*') # prints a separator *

  10*20*30*40

- This **'end'** is also optional and it allows us to specify any string to be appended after the last value. For example:

  >>> print (10,20,30,40, sep='*', end='@') # A new end is assigned @

  10*20*30*40@

  In the above example @ printed at the end of line.

The input ():

The input() function is able to read data entered by the user and to return the same data to the running program.

Chapter-1 Revision Tour -I

- The input() function is invoked with one argument - it's a string containing a message.
- The message will be displayed on the console before the user is given an opportunity to enter anything.
- The **result of the** input() function is a **string**.

Examples of input ( ) functions:

1)

```
rollno = input("Enter your roll no. ")
sname = input('Enter your name: ')
print ('My roll no. is:', rollno)
print ('My name is:', sname)
```

2) '''Addition of two number from input () function.'''

```
Num1 =int(input("Enter the  first number: "))
Num2 =int(input("Enter the second number: ")))
print('The sum of ', Num1, ' and ', Num2, ' is ',
Num1+Num2, '.', sep='')
```

When we execute the above program, it will produce the following output:

```
Enter the first number: 20
Enter the second number: 40
The sum of 20 and 40 is 60.
```

Comments:

Comments are any text to the right of the # symbol and are mainly useful as notes for the reader of the program.

For example:

print('Hello World') # Note that print is a function

OR

Note that print is a function

print ('Hello World')

Chapter-1 Revision Tour -I

Block and Indentation:

A group of statements which are part of another statement or a function are called **block or code – block** or suite in Python.

Consider the following Example:

```
if n1<n2:
        Tmp =n1
        n1=n2
        n2=Tmp
print ( "I Understand Block")
```

Whitespace is important in Python. Actually, **whitespace at the beginning of the line is important. This is called indentation.**

Flow of control

1) The if-else statement:

Selection if – else structures choose among alternative courses of action

If student's grade is greater than or equal to 60 Print "Passed "otherwise print "Not Qualified"

Syntax:

```
if true_or_false_condition:
    perform_if_condition_true
else:
    perform_if_condition_fails
```

The *if-else* execution goes as follows:

- if the condition evaluates to **True** (its value is not equal to zero), the `perform_if_condition_true` statement is executed, and the conditional statement comes to an end;
- if the condition evaluates to **False** (it is equal to zero), the `perform_if_condition_false` statement is executed, and the conditional statement comes to an end.

Chapter-1 Revision Tour -I

Example:

```
# Created By Amjad Khan
'''Program to print whether student
Qualified(percentage>=40)or Not Qualified'''

percentage = float(input("Enter percentage:  "))

if(percentage>=40):
    print("Qualified")
else:
    print("Not Qualified")
```

- In the above example, `percentage >= 40` is the test expression.
- In the above example, when `percentage` is greater or equal to 40, the test expression is true and the body of `if` is executed and the `body` of else is skipped.
- If `percentage` is less than to 40, the test expression is false and the body of `else` is executed and the body of `if` is skipped.

OUTPUT:

```
====================== RESTART: D:/Amjad_CS
/if-else.py ======================
Enter percentage:  39
Not Qualified
>>>
====================== RESTART: D:/Amjad_CS
/if-else.py ======================
Enter percentage:  45
Qualified
```

Chapter-1 Revision Tour -I

Ladder if – else statement (if – elif – else)

If-elif-else is used to **check more than just one condition**, and to **stop** when the first statement which is true is found.

Syntax of if...elif...else:

```
if true_or_false_condition:
    perform_if_condition_true
elif:
    perform_elif_condition_true
else:
    perform_if_condition_fails
```

- you **mustn't use** *else* **without a** preceding *if;*
- *else* is always the **last branch of the cascade**, regardless of whether you've used *elif* or not;
- *else* is an **optional** part of the cascade, and may be omitted;
- if there is an else branch in the cascade, only one of all the branches is executed;
- if there is no else branch, it's possible that none of the available branches is executed.

```
'''check if the number is positive or
negative or zero and display
an appropriate message
by using ladder if statement'''

number = float(input("Enter a number: "))
if number >= 0:
    if number == 0:
        print("Zero")
    else:
        print("+ive number")
else:
    print("-ive number")
```

```
========== RESTART: D:/Amjad_
Ladder_if-else.py ==========
Enter a number: 0.0
Zero
>>>
========== RESTART: D:/Amjad_
Ladder_if-else.py ==========
Enter a number: 28
+ive number
>>>
========== RESTART: D:/Amjad_
Ladder_if-else.py ==========
Enter a number: -1000
-ive number
```

Chapter-1 Revision Tour -I

Nested if-else statements

There may be a situation when you want to check for **another condition after a condition resolves to true**. In such a situation, you can use the *nested if* construct.

In nested *if – else* instruction placed after *if* is another *if*.

For Example:

```
check if the number is positive or
negative or zero and display
an appropriate message
by using nested if statement'''

number = float(input("Enter a number: "))
if number >= 0:
    if number == 0:
        print("Zero")
    else:
        print("+ive number")
else:
    print("-ive number")
```

Here is important point:

- this use of the if statement is known as nesting; remember that every else refers to the if which lies at the same indentation level

```
========== RESTART: D:/Amjad_CS
/nested_if-else.py ==========
Enter a number: 10
+ive number
>>>
========== RESTART: D:/Amjad_CS
/nested_if-else.py ==========
Enter a number: 0.0
Zero
>>>
========== RESTART: D:/Amjad_CS
/nested_if-else.py ==========
Enter a number: -20
-ive number
```

Chapter-1 Revision Tour -I

Loops/Iterative Statement

Performing a certain part of the code more than once is called a **loop**.

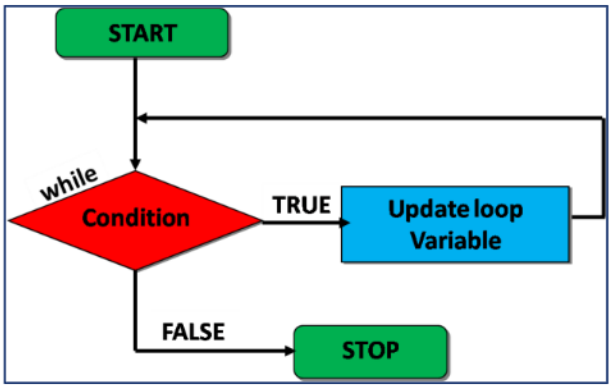
Two types of loops:

1) *while* loop

2) *for* loop

1) while loop:

- The ***while*** loop continues to run as long as the condition is still ***True***. In other words, it runs *while* the condition is ***True.***
- Once the condition becomes **False** and if the **else** clause is written in while, and then **else** will get executed.



Syntax:

```
while conditional_expression:
    instruction_one
    instruction_two
    instruction_three
    :
    instruction_n
else:
    Instructions of else block
```

- if you want to execute **more than one statement inside one** while, you must **indent** all the instructions in the same way;
- an instruction or set of instructions executed inside the while loop is called the **loop's body**;
- if the condition is False (equal to zero) as early as when it is tested for the first time, the body is not executed even once
- the body should be able to change the condition's value, because if the condition is True at the beginning, the body might run continuously to infinity - notice that doing a thing usually decreases the number of things to do).
- The **else** part is executed if the condition in the while loop evaluates to **False**.

Chapter-1 Revision Tour -I

Example 2: The following program fragment prints the squares of all integers from 1 to 10.

```
'''
The following program fragment
prints the squares of all
integers from 1 to 10
'''

i = 1
while i <= 10:
    print(i ** 2)
    i += 1
```

```
Amjad_CS/whileLoo
p.py ============
=========
1
4
9
16
25
36
49
64
81
100
```

In this example, the variable ***i*** inside the loop iterates from 1 to 10. Note that after executing this fragment the value of the variable ***i*** is defined and is equal to 11, because when ***i == 11*** the condition ***i <= 10*** is ***False*** for the first time.

Example 2:

```
'''
The following program prints
the squares of all integers
from 1 to 10 with else
'''

i = 1
while i <= 10:
    print(i)
    i += 1
else:
    print('Loop ended, i =', i)
```

```
RESTART: D:/Amjad_
CS/whileLoopwithEl
se.py ============
=====
1
2
3
4
5
6
7
8
9
10
Loop ended, i = 11
```

In the above when ***i == 11*** the condition ***i <= 10*** is ***False*** for the first time then the ***else*** part is executed and print the last value of ***i*** i.e. **11**

Chapter-1 Revision Tour -I

for loop

The for loop in Python is used to iterate over a sequence (***list, tuple, string***) or other iterable objects.

Its Syntax is:

for <variable> in <sequence>:

Statement Block 1

else: # optional block

Statement Block 2

- The *for* loop ends when the loop is repeated for the last value of sequence.
- The *for* loop repeats n number of times, where n is the length of sequence given in for loop's header.

Example 1: Print all numbers from 0 to 4

```
'''
The following program prints
the all integers
from 1 to 4
'''

for i in range(1,5):
    print('inside for loop, i =', i)
```

```
======================
 RESTART: D:/Amjad_CS/
forLoop.py ===========
==========
inside for loop, i = 1
inside for loop, i = 2
inside for loop, i = 3
inside for loop, i = 4
```

Example 2: Print all numbers from 0 to 5, and print a message when the loop has ended:

```
'''
The following program prints
the all integers
from 1 to 5 with else
'''

for i in range(1,6):
    print('inside for loop, i =', i)
else:
    print('Lopp ended i =',i)
```

```
TART: D:/Amjad_CS/forL
oopwithelse.py =======
==========
inside for loop, i = 1
inside for loop, i = 2
inside for loop, i = 3
inside for loop, i = 4
inside for loop, i = 5
Lopp ended i = 5
```

Chapter-1 Revision Tour -I

The range () function

Python has a built-in function called **range()** that can generate a list of numbers.

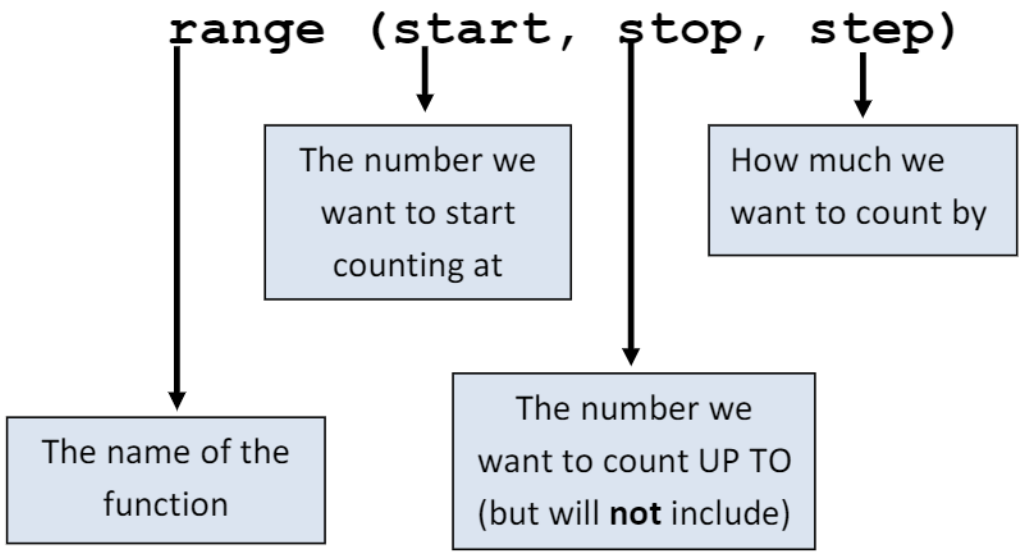
For example:

```
ex = list(range(0, 10))

print (ex)
```

Output: **[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**

Syntax of range () Function:



- **start** is an optional parameter specifying the starting number of the sequence (0 by default)
- **stop** is an optional parameter specifying the end of the sequence generated (it is not included),
- and **step** is an optional parameter specifying the difference between the numbers in the sequence (1 by default.)

Chapter-1 Revision Tour -I

Using **range()** in a **for** Loop:

We can use the **range()** function to control a loop through "counting"

```
for i in range(0, 20):
    print(i + 1)
```

What will this code do?

– Print the numbers 1 through 20 on separate lines

When we use the **range()** function in **for** loops, we don't need to cast it to a list

The **for** loop handles that for us

```
print("Table of five...")
for num in range(5, 51, 5):
    print(num, end=",")
```

Call the **range()** function, but don't need to cast it to a list

```
Output:
Table of five...
5,10,15,20,25,30,35,40,45,50,
```

JUMP STATEMENTS

Jump statements are used to transfer the program's control from one location to another. Means these are used to alter the flow of a loop like-to skip a part of a loop or terminate a loop

There are three types of jump statements used in python.

1. break

2. continue

3. pass

Chapter-1 Revision Tour -I

1) Break statement

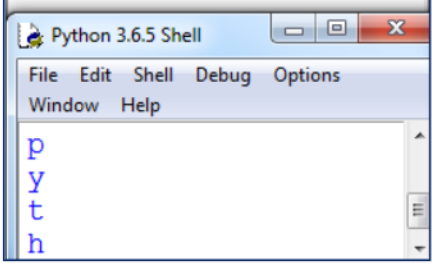
The break statement is used to exits the loop immediately, and unconditionally ends the loop's operation.

```
for u_var in sequence:
    # codes inside loop
    if test_expr:
        break
    # code inside loop
# codes outside for loop
```

```
while test_expr:
    # codes inside loop
    if test_expr:
        break
    # code inside loop
# codes outside for loop
```

Example:

```
# Use of break
for letter in 'python':
    if letter=='o':
        break
    print(letter)
```



2) Continue statement

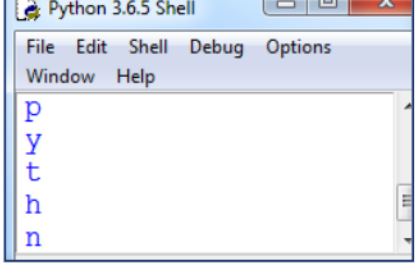
The continue statement is used to skip the current block and move ahead to the next iteration, without executing the statements inside the loop.

```
for u_var in sequence:
    # codes inside loop
    if test_expr:
        continue
    # code inside loop
# codes outside for loop
```

```
while test_expr:
    # codes inside loop
    if test_expr:
        continue
    # code inside loop
# codes outside for loop
```

Example:

```
# Use of continue
for letter in 'python':
    if letter=='o':
        continue
    print(letter)
```



Chapter-1 Revision Tour -I

In this program, we iterate through the "***python***" sequence. We check if the letter is '***o'***, upon which we break from the loop. Hence, we see in our output that all the letters up till '***o'*** gets printed. After that, the loop terminates.	This program is same as the break example except the break statement has been replaced with continue. We continue with the loop, if the string is '***o'***, not executing the rest of the block. Hence, we see in our output that all the letters except '***o'*** gets printed.

3) pass Statement:

- ***pass*** in Python basically does nothing, but unlike a comment it is not ignored by interpreter.
- It can be used when a statement is required syntactically but the program requires no action.

Can be use in loop and conditional statements:

```
if (something == true):          # used in conditional
    pass                         # Statement

while (some condition is true):  # user is not sure about
    pass                         # the body of the loop
```