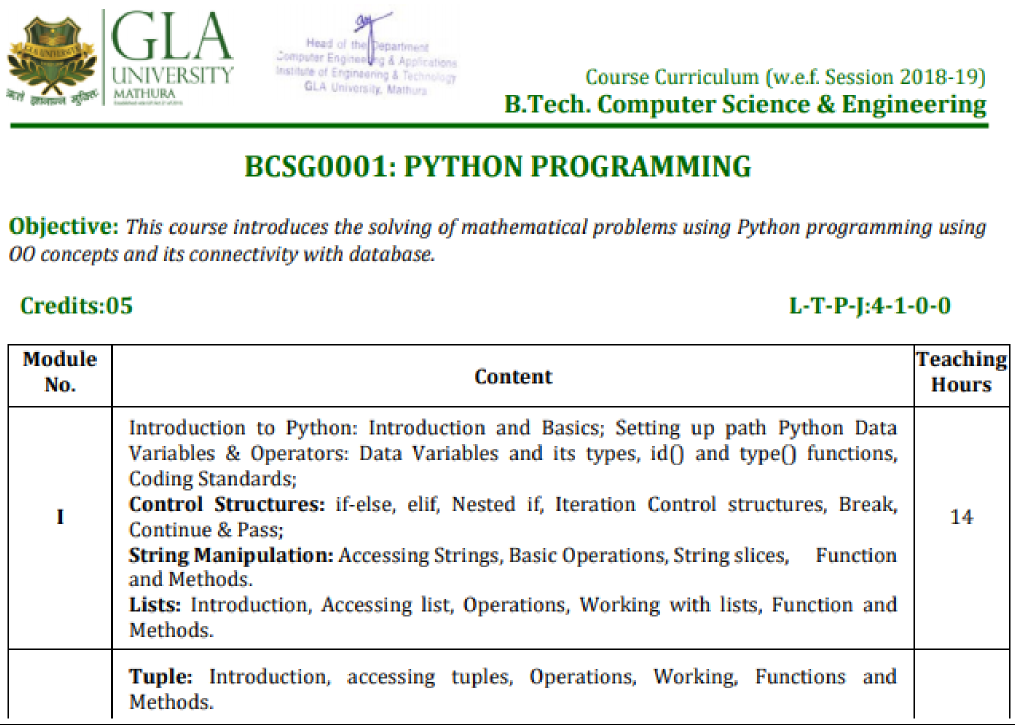
Python Programming-BCSG0001

Priya Agrawal (Technical Trainer)
Department of Training & Development (TND)

1st Mid-Term Syllabus



GLA UNIVERSITY MATHURA

Head of the Department
Computer Engineering & Applications
Institute of Engineering & Technology
GLA University, Mathura

Course Curriculum (w.e.f. Session 2018-19)
B.Tech. Computer Science & Engineering

BCSG0001: PYTHON PROGRAMMING

Objective: *This course introduces the solving of mathematical problems using Python programming using OO concepts and its connectivity with database.*

Credits:05 **L-T-P-J:4-1-0-0**

Module No.	Content	Teaching Hours
I	Introduction to Python: Introduction and Basics; Setting up path Python Data Variables & Operators: Data Variables and its types, id() and type() functions, Coding Standards; **Control Structures:** if-else, elif, Nested if, Iteration Control structures, Break, Continue & Pass; **String Manipulation:** Accessing Strings, Basic Operations, String slices, Function and Methods. **Lists:** Introduction, Accessing list, Operations, Working with lists, Function and Methods.	14
	Tuple: Introduction, accessing tuples, Operations, Working, Functions and Methods.	

Topic_1:

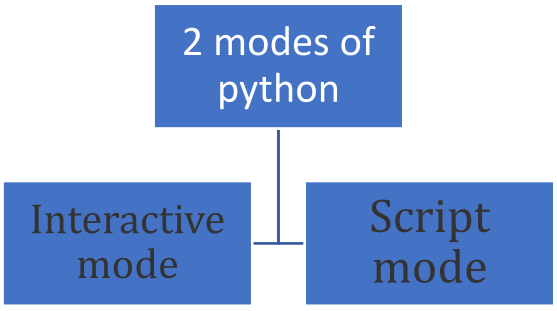
Introduction to Python: Introduction and Basics

1.1 What is Python?

- Python is a widely used general-purpose, high level programming language.
- Created by- **Guido van Rossum** **-** in 1990
- Beauty or Advantages of Python:
    1. Readability
    2. Express in fewer lines of code.
    3. Interpreted.
- Disadvantages:
    1. Speed limit.(**Reason :=** python is interpreted → cause slow execution.)
    2. Weak in Mobile Computing. (Best at server side)
    3. Underdeveloped Database Access Layers. (Python's database access layers are a bit underdeveloped. Consequently, it is less often applied in huge enterprises.)

Check the Python version in the script: import sys → sys.version

1.2 Modes of Python



Interactive mode:

❖ Interactive Mode (interact with OS).

❖ When we type Python statement, **interpreter displays the result(s) immediately.**

Advantages:

❖ testing small pieces of code.

Drawback:

❖ We cannot save the statements and have to retype all the statements once again to re-run them.

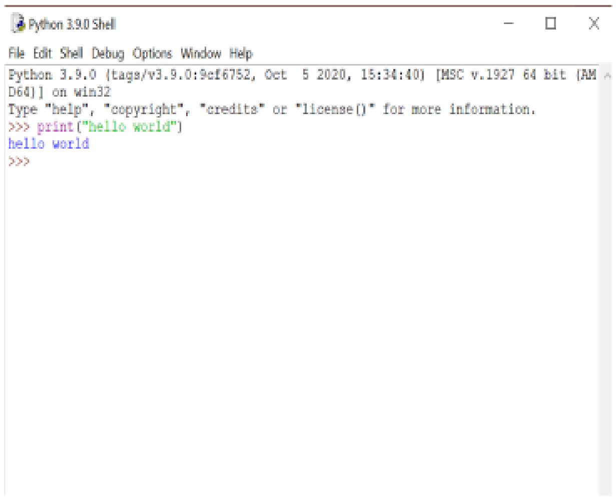
Script mode:

❖ we type python program in a file and then use interpreter to execute the content of the file.

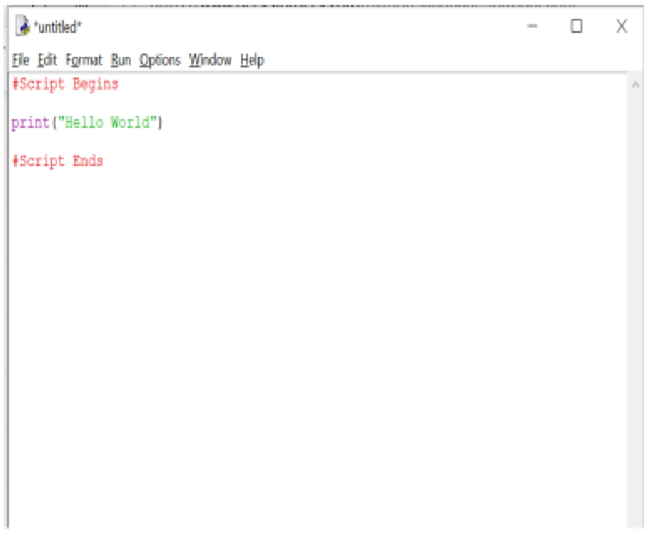
❖ Scripts can be saved to disk for future use. **Python scripts have the extension .py**, meaning that the filename ends with **.py**

❖ Save the code with **filename.py** and run the interpreter in script mode to execute the script.

First Program in Python



```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("hello world")
hello world
>>>
```



```
#Script Begins

print("Hello World")

#Script Ends
```

The " >>> " represents the python shell and its ready to take python commands and code.

1.3 Python Character Set

- Any Python program contains words or statements follow a sequence of characters.

Python uses the following Character Set:

- **Letters:** Upper case and Lower case
- **Digits:** 0,1,2,3,4,5,6,7,8,9
- **Special Symbols:** Underscore (_), (,), [,], {}, +, -, *, &, ^, %, $, #, !, Single quote('), Double quotes("), Back slash(\), Colon(:), and Semi Colon(;)
- **White Spaces:** (\t \n), Space, Tab.

Example:

```
num1 = 10

num2 = 20

print(num1 + num2)
```

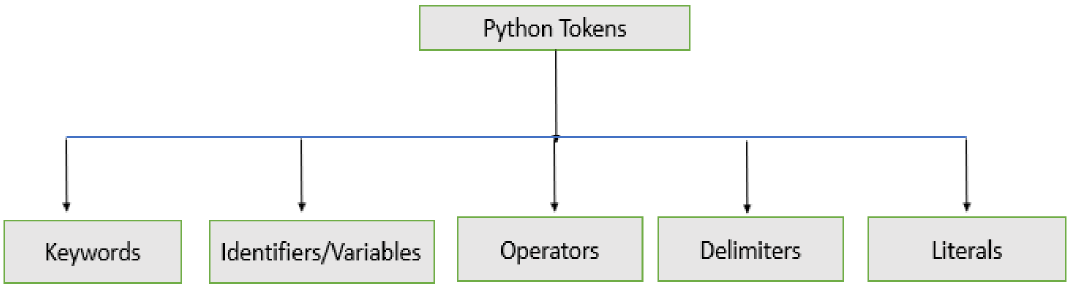
Here: In this python program (addition of two numbers) consist Letters, Digits, and special symbols.

Topic_2:

Python Data variables & Operators: Data variables and its types, id() and type() functions, Coding Standards;

2.1 Python Tokens

- Python breaks each statement into a sequence of lexical component known as Tokens.
- Python breaks each statement into a sequence of lexical component known as Tokens.

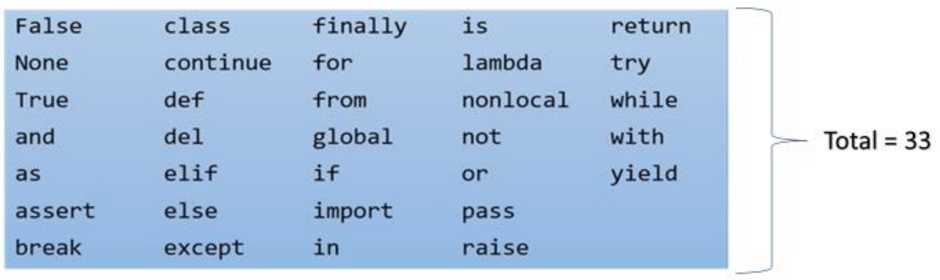


2.1.1 Python Keywords

Python Keywords

Python keywords = Reserved words

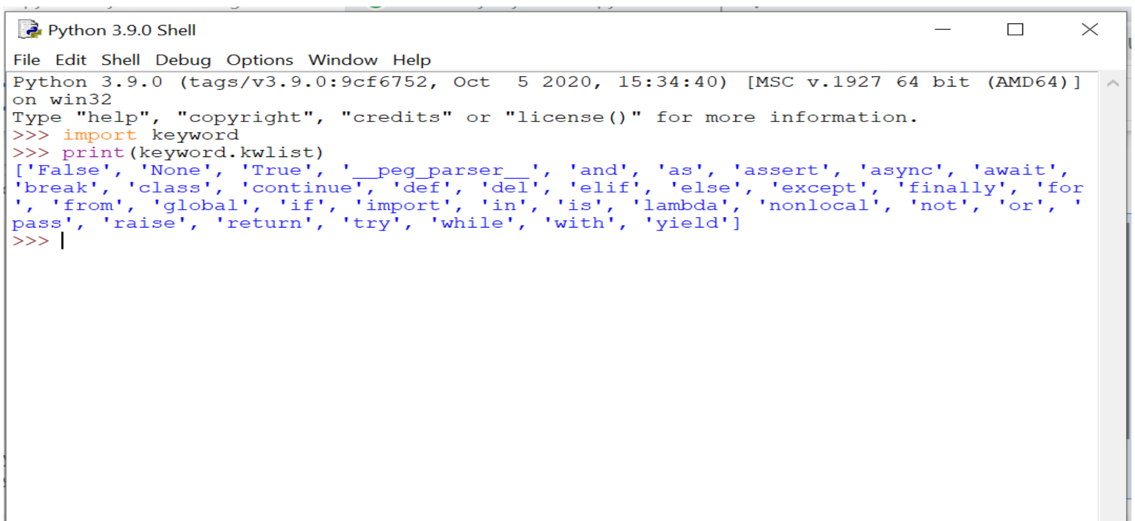
- Keywords are the words that convey a special meaning to the language compiler/interpreter.



False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

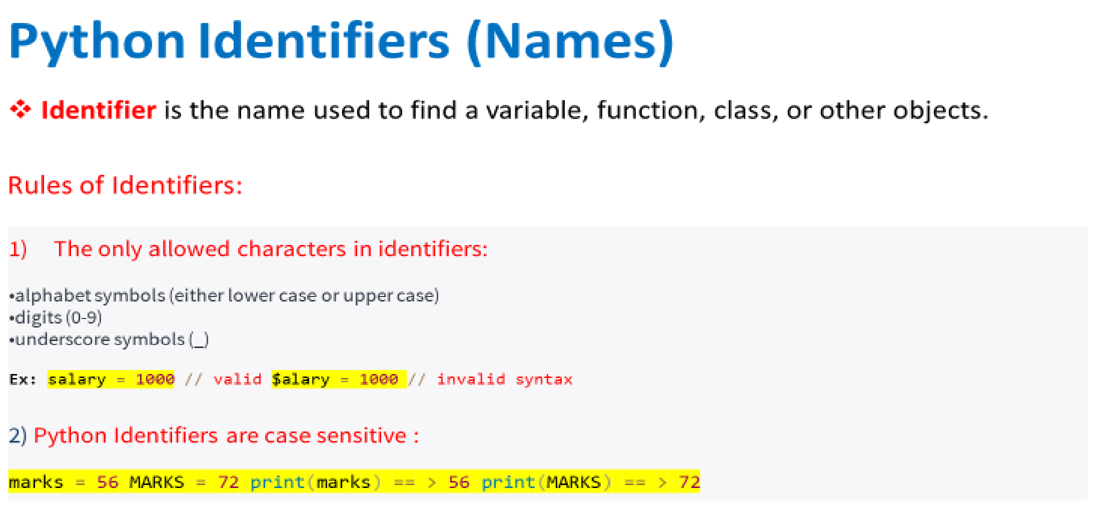
Total = 33

PTR: All the keywords except True , False and None are in lowercase and they must be written as they are.



```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'await',
'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for
', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', '
pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>> |
```

2.1.2 Python Identifiers



Python Identifiers (Names)

❖ **Identifier** is the name used to find a variable, function, class, or other objects.

Rules of Identifiers:

1) The only allowed characters in identifiers:

•alphabet symbols (either lower case or upper case)
•digits (0-9)
•underscore symbols (_)

```
Ex: salary = 1000 // valid $salary = 1000 // invalid syntax
```

2) Python Identifiers are case sensitive :

```
marks = 56 MARKS = 72 print(marks) == > 56 print(MARKS) == > 72
```

3) Identifiers should not start with digit :

5Ruppes is not allowd   ruppes5 is allowwd

4) Should not use Reserved words :

As per Python document, we have 33 reserved words in Python. We should not allowed to use these reserved words as identifiers.

Example : def is a reserved word in Python, we should not use def as identifier.

```
def = 10 // invalid if = 20 // invalid
```

5) $ symbol is not allowed :

In any cases $ symbol is not allowed as part of the identifiers.

```
ca$h = 200 // invalid identifier
```

Rules of Identifiers:

- You can't use reserved keywords as an identifier name.
- Python identifier can contain letters in a small case (a-z), upper case (A-Z), digits (0-9), and underscore (_).
- Identifier name can't begin with a digit.
- Python identifier can't contain only digits.
- Python identifier name can start with an underscore.
- There is no limit on the length of the identifier name.
- Python identifier names are case sensitive.

❖ **Identifier** is the name used to find a variable, function, class, or other objects.

Examples:

The following are some *valid* identifiers:

Num	Myfile
_ result	HJ3_JK
DATE_7_77	Z2T0Z9_

✓

The following are some *invalid* identifiers:

DATA-REC		My.file
29CLCT	for	
break	Z2T0Z9_	First name

✗

Practice:

1. Which of the following is correct?
   a) add1 = 5+2   b) 1add = 5+2
   c) _add. = 5+2   d) add-1 = 5+2

1. Which of the following is correct?
   a) for = “hello python”   b) print = “hello python”
   c) _ = “hello python”   d) _ _ = “hello python”

1. Which of the following is not correct?
   a) __ = 12   b) a = 12
   c) _2 = 12   d) “a” = 12

Answers: 1. a)
2. c)
3. d)

2.1.3 Python Delimiters

Delimiters are symbols that perform a special role in Python like grouping, punctuation and assignment.

Examples:

( ) [ ] { }

, : . ‘ = ;

+= -= *= /= //= %=

&= |= ^= >>= <<= **=

Num = 12

Res = num + 2

Print(“result = “,result)

2.1.4 python type() function

- returns the type of the specified object

Examples:

```
a = ('apple', 'banana', 'cherry')
b = "Hello World"
c = 33
x = type(a)
y = type(b)
z = type(c)
print(x)
print(y)
```

```
O/p → <class 'tuple’>
       <class 'str’>
       <class 'int'>
```

2.1.5 python id() function

- ❑ returns a unique id for the specified object.
- ❑ All objects in Python has its own unique id. *{ here object means Any object, String, Number, List, Class etc.}*
- ❑ The id is assigned to the object when it is created.

Examples:

```
x = ('apple', 'banana', 'cherry')

y = id(x)

print(y)

O/p → 64163752
```

2.1.5 python Literals

Python Literals can be defined as data that is given in a variable or constant.

Python supports the following literals:

1. String literals
2. Numeric literals
3. Boolean literals
4. Special literals
5. Literal Collections

Python Literals

- Single-line String Literals ➔ “python” & ‘python’
  - Multi-line String Literals ➔ “My\
    First\
    Program”

String literals

❖ String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes to create a string.

Example:
1.“Krishna" , ’1234’   2. text1='hello'

Numeric literals

❖ Numeric literals can belong to following four different numerical types.

Int(signed integers)	Long(long integers)	float(floating point)	Complex(complex)
Numbers(can be both positive and negative) with no fractional part.eg: 100	Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L	Real numbers with both integer and fractional part eg: -26.2	In the form of a+bj where a forms the real part and b forms the imaginary part of the complex number. eg: num = 3.14j (num.imag, num.real)

Boolean literals

❖ A Boolean literal can have any of the two values: True or False.

```
>>> #True ==> 1
>>> #False ==> 0
```

Special literals

❖ Python contains one special literal i.e., **None.**
❖ **‘None’** is used to define a null variable.
❖ Used for end of lists in Python.

Literal Collections

❖ Python provides the four types of literal collection:

✓ List
✓ Tuple
✓ Dictionary
✓ Set

FACTS about Python Literals…

❖ Integers in Python 3 are of unlimited size. Python 2 has two integer types - int and long. There is no '**long integer**' in Python 3 anymore.

❖ Number data types store numeric values. They are immutable data types. This means, changing the value of a number data type results in a newly allocated object

For example −

```
var1 = 1      # Number objects are created when you assign a value to them.
var2 = 10
```

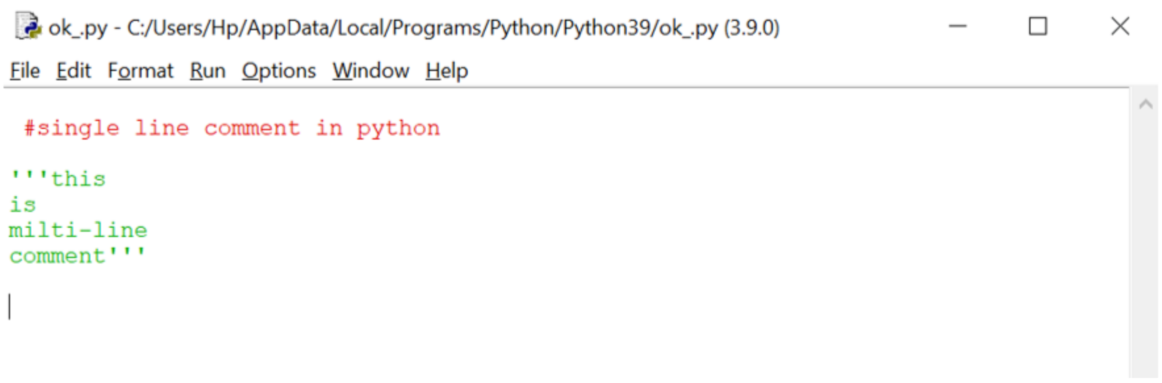
❖ You can also delete the reference to a number object by using the **del** statement.

Syntax ➔ `del var1[,var2[,var3[....,varN]]]]`

For example −

```
del var del var1, var2  # You can delete a single object or multiple objects by using the del statement..
```

Python Comments


ok_.py - C:/Users/Hp/AppData/Local/Programs/Python/Python39/ok_.py (3.9.0)
File Edit Format Run Options Window Help

```
 #single line comment in python

'''this
is
milti-line
comment'''
```

2.2 Python Operators

- Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

1. Arithmetic operators
2. Assignment operators
3. Comparison operators
4. Logical operators
5. Identity operators
6. Membership operators
7. Bitwise operators

Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Program:

WAPP to check the following; where input (two integer variables) is entered by the user.

1. ADD
2. SUBTRACT
3. MULTIPLY
4. DIVIDE (floor)

```
Ex: input ➔     6    3

     output➔    addition = 9
                subtraction = 3
                multiplication = 18
                division = 3
```

Comparison Operators

Comparison operators are used to compare two values.

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Program:

Your program will take input as 2 integer variables from user. WAPP to check and print the following;

1. If val1 ==0 print "please enter valid input"
2. If val1<val2 print val1
3. If val1>val2 print val1

```
Ex1: input ➔  1   3
     output➔   3
```

```
Ex2: input ➔  0   3
     output➔   0
```

Identity Operators

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

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Difference between == and **is** operator

Equality operator (==) compares the values of both the operands and checks for value equality.

Whereas the **'is'** operator checks whether both the operands refer to the same object or not.

Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Topic_3:

Control Structures: if-else, elif, Nested if, Iteration Control structures, Break, Continue, & Pass;

3.1 if-else, elif, nested if

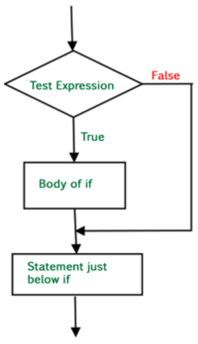
Decision making statements

Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in python are:

- If statement
- If else statement
- If elif ladder
- Nested if statement
- Short hand if statement
- Short hand if else statement

If statement

It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.



Syntax:

```
if condition:
       # Statements to execute if
       # condition is true
```

As we know, python uses indentation to identify a block. So the block under an if statement will be identified as shown in the below example:

```
if condition:
       statement1
statement2

# Here if the condition is true, if block
# will consider only statement1 to be inside
# its block.
```

```
# python program to illustrate If statement

i = 10
if (i > 20):
  print ("10 is less than 15")
print ("I am Not in if")
```

Output:

```
I am Not in if
```

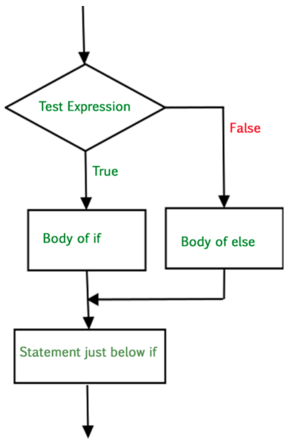
If else statement

We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

```
if (condition):
 # Executes this block if
 # condition is true

else:
# Executes this block if
 # condition is false
```



```
# python program to illustrate If else statement

i = 20;
if (i < 15):
		print ("i is smaller than 15")
		print ("i'm in if Block")
else:
		print ("i is greater than 15")
		print ("i'm in else Block")
print ("i'm not in if and not in else Block")
```

Output:

```
i is greater than 15
i'm in else Block
i'm not in if and not in else Block
```

Nested If statement

Nested if statements means an if statement inside another if statement.

Syntax:

```
if (condition1):
     # Executes when condition1 is true
     if (condition2):
         # Executes when condition2 is true
     # if Block is end here
# if Block is end here
```

```
# python program to illustrate nested If statement
i = 10
if (i == 10):
            # First if statement
            if (i < 15):
                        print ("i is smaller than 15")
            # Nested - if statement
            # Will only be executed if statement above
            # it is true
            if (i < 12):
                        print ("i is smaller than 12 too")
            else:
                        print ("i is greater than 15")
```

Output:

```
i is smaller than 15 i is smaller than 12 too
```

If elif else ladder

Syntax:-
```
if (condition):
    statement
elif (condition):
    Statement
.
.
else:
    statement
```

```
# Python program to illustrate if-elif-else ladder

i = 20
if (i == 10):
            print ("i is 10")
elif (i == 15):
            print ("i is 15")
elif (i == 20):
            print ("i is 20")
else:
            print ("i is not present")
```

Output:
```
i is 20
```

Short Hand if statement

Whenever there is only a single statement to be executed inside the if block then shorthand if can be used. The statement can be put on the same line as the if statement.

Syntax:
```
if condition: statement
```

```
# Python program to illustrate short hand if

i = 10
if i < 15: print("i is less than 15")
```

Output:
```
i is less than 15
```

Short Hand if-else statement

This can be used to write the if-else statements in a single line where there is only one statement to be executed in both if and else block.

Syntax:
```
statement_when_True if condition else statement_when_False
```

```
# Python program to illustrate short hand if-else

i = 10
print(True) if i < 15 else print(False)
```

Output:
```
True
```

3.2 Iterations ~ Loops, Iteration Control Structures, break, continue, & pass;

Python- Loops

- Allows to execute a statement or group of statements multiple times.

Python programming language provides following types of loops to handle looping requirements.

- While loop
- For loop
- Nested loops

For Loop

works more like an iterator method as found in other object-orientated programming languages.

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

```
for x in "banana":
  print(x)
```

```
O/p:
b
a
n
a
n
a
```

```
for x in range(6):
  print(x)
```

```
O/p:
0
1
2
3
4
5
```

The range() Function

-To loop through a set of code a specified number of times

PTR: The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(6):
  print(x)
```

```
O/p:
0
1
2
3
4
5
```

Note that `range(6)` is not the values of 0 to 6, but the values 0 to 5.

```
for x in range(2, 6):
  print(x)
```

```
O/p:
2
3
4
5
```

```
for x in range(2, 30, 3):
  print(x)
```

```
O/p:
2
5
8
11
14
17
20
23
26
29
```

Else in For Loop

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

```
for x in range(5):
  print(x)
else:
  print(“Done!")
```

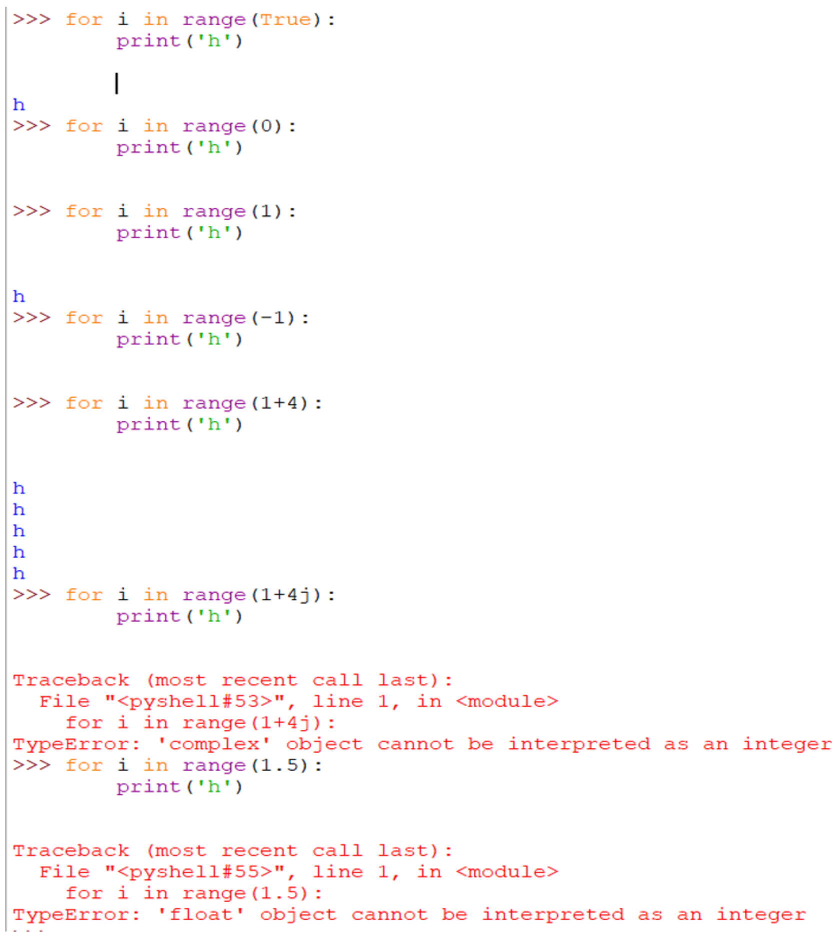
```
0
1
2
3
4
Done!
```

Pass in For Loop

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

```
for x in [1, 2, 3]:
  pass
```

Some experiments:



```
>>> for i in range(True):
        print('h')


h
>>> for i in range(0):
        print('h')


>>> for i in range(1):
        print('h')


h
>>> for i in range(-1):
        print('h')


>>> for i in range(1+4):
        print('h')


h
h
h
h
h
>>> for i in range(1+4j):
        print('h')


Traceback (most recent call last):
  File "<pyshell#53>", line 1, in <module>
    for i in range(1+4j):
TypeError: 'complex' object cannot be interpreted as an integer
>>> for i in range(1.5):
        print('h')


Traceback (most recent call last):
  File "<pyshell#55>", line 1, in <module>
    for i in range(1.5):
TypeError: 'float' object cannot be interpreted as an integer
```

While Loop

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

```
i = 1
while i < 6:
  print(i)
  i += 1
```

```
O/p:
1
2
3
4
5
```

Difference b/w for loop & while loop

*The main difference is that we use **while loop** when we are **not** certain of the number of times the loop requires execution, on the other hand when we exactly know how many times we need to run the loop, we use for loop.*

---

break

- Used to bring the program control out of the loop.
- The break statement breaks the loops one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops.

```
str = "python"
for i in str:
    if i == 'o':
        break
    print(i);
```

o/p:

```
p
Y
t
h
```

continue

- Used to bring the program control to the beginning of the loop.
- The continue statement skips the remaining lines of code inside the loop and start with the next iteration.

```
str = “python"
for i in str:
    if(i == ’o'):
        continue
    print(i)
```

O/p:

```
P
Y
T
H
n
```

Topic_4:

Lists: Introduction, Accessing list, Operations, Working with Lists, Function amd Methods.

Python Lists

Collection of elements.

- Just like Arrays ~ Dynamic size
- Can be Homogeneous (similar data type) or Heterogeneous (data type different types)
- Lists are *Mutable* ~ *can be altered after creation.*
- *Allows duplicates.*

Note- Lists are a useful tool for preserving a sequence of data and further iterating over it.

Creating a List

```
Ls = []  //creating a list

Print(Ls)
```

```
Ls = list()  //creating a list

Print(Ls)
```

Can we have duplicates in a list..?

```
Ls = [1,2,3,4,2,2,1,0,-1]
//creating a list

Print(Ls)

[1, 2, 3, 4, 2, 2, 1, 0, -1]
```

Size of a List

```
Ls = []  //creating a list

Print(len(Ls))


0
```

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

Print(len(Ls))


6
```

Accessing elements

```
Ls = [1,2,3,4]     //creating a list

Print(Ls[0])
1
```

```
Ls = [2, 4.65, ‘a’,  “python” ]     //creating a list

Print(Ls[0])
Print(Ls[1])
Print(Ls[2])
Print(Ls[3])
Print(Ls[10])
Print(Ls[-1])
```

Accessing elements

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

Print(lst[0])               Print(lst[1][0])
print(lst[1])               Print(lst[1][2])
Print(lst[2]                Print(lst[1][2])
Print(lst[3])
1                           1
[1,2,3]                     2
3                           3
[4,3,1]
```

Accessing elements using loops

```
lst = [1,2,3,4,5]
for i in range(len(lst)):
    print(i, lst[i])

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

```
lst = [1,2,3,4,5]
for i in lst:
    print(i)

1
2
3
4
5
```

Accessing elements using negative indexing

```
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']


print(List[-1])         # print the last element of list

print(List[-3])         # print the third last element of list
```

```
Geeks
For
```

Adding elements in a list *Using append() method*

```
List = []          # Creating a List
print(List)

# Addition of Elements  in the List
List.append(1)
List.append(2)
List.append(4)

print(List)
```

```
[]

[1, 2, 4]
```

Adding elements in a list *Using insert() method*

```
List = [1,2,3,4]
print(List)

List.insert(3, 12)
List.insert(0, 'Geeks')
print(List)
```

```
[1, 2, 3, 4]
['Geeks', 1, 2, 3, 12, 4]
```

`append()` method only works for addition of elements at the end of the List, for addition of element at the desired position, `insert()` method is used. Unlike `append()` which takes only one argument, `insert()` method requires two arguments(position, value).

Adding elements in a list *Using extend() method*

`extend()`, this method is used to add multiple elements at the same time at the end of the list.

```
List = [1,2,3,4]
print(List)

List.extend([5,6])
print(List)
```

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

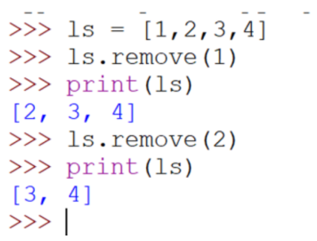
Other than `append()` and `insert()` methods, there's one more method for Addition of elements, `extend()`, this method is used to **add multiple elements at the same time at the end of the list**.

Note – append() and extend() methods can only add elements at the end.

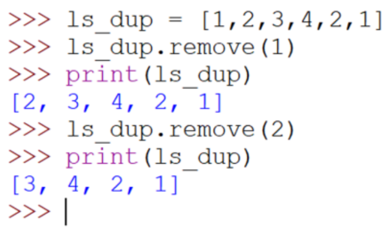
---

removing elements from the list *Using remove() method*

Remove() method only removes one element at a time, to remove out range of elements but an Error arises if element doesn't exist in the set.



```
>>> ls = [1,2,3,4]
>>> ls.remove(1)
>>> print(ls)
[2, 3, 4]
>>> ls.remove(2)
>>> print(ls)
[3, 4]
>>> 
```

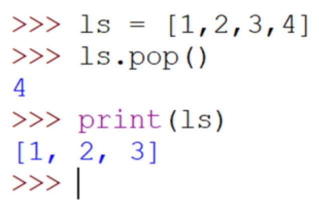


```
>>> ls_dup = [1,2,3,4,2,1]
>>> ls_dup.remove(1)
>>> print(ls_dup)
[2, 3, 4, 2, 1]
>>> ls_dup.remove(2)
>>> print(ls_dup)
[3, 4, 2, 1]
>>> 
```

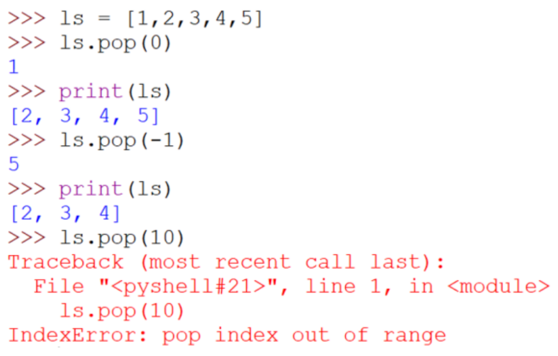
removing elements from the list *Using pop() method*

-By default it removes only the last element of the set.

-To remove element from a specific position of the List, index of the element is passed as an argument to the pop( ) method.



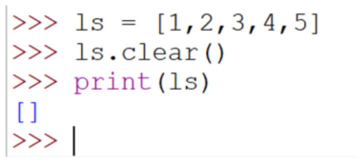
```
>>> ls = [1,2,3,4]
>>> ls.pop()
4
>>> print(ls)
[1, 2, 3]
>>> |
```



```
>>> ls = [1,2,3,4,5]
>>> ls.pop(0)
1
>>> print(ls)
[2, 3, 4, 5]
>>> ls.pop(-1)
5
>>> print(ls)
[2, 3, 4]
>>> ls.pop(10)
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    ls.pop(10)
IndexError: pop index out of range
```

removing elements from the list *Using clear() method*

ısed to **erase all the elements** of list. After this operation, list becomes empty.



```
>>> ls = [1,2,3,4,5]
>>> ls.clear()
>>> print(ls)
[]
>>> |
```

Some more functions

- sort vs sorted Difference between sort and sorted in python *~ Interview Question*
- Count() -- Returns the count of number of items passed as an argument
- Index() -- Searches for a given element from the start of the list and returns the lowest index where the element appears.

Python Slicing

Slice operator

---

*There are multiple ways to print the whole List with all the elements, but to print a specific range of elements from the list, we use **Slice operation**.*

Slice operation is performed on Lists with the use of a colon(:).

- To print elements from beginning to a range use [: Index],
- to print elements from end-use [:-Index],
- to print elements from specific Index till the end use [Index:],
- to print elements within a range, use [Start Index: End Index],
- to print the whole List with the use of slicing operation, use [:].
- Further, to print the whole List in reverse order, use [::-1].

```
#syntax of slice operator: list_var_name[start_index: stop_index: step]
```

Some experiments:

```
>>> ls[0: len(ls): 1] #to print the whole list
[1, 2, 3, 4, 5]
>>> ls[: : ] #to print the whole list
[1, 2, 3, 4, 5]
>>> ls[1: 4 : 1] #to print the elements [2,3,4]
[2, 3, 4]
>>>
```

```
>>>
>>> #negative indexing
>>> ls[-1: -6: -1] #to print the list in reverse
[5, 4, 3, 2, 1]
>>> ls[: : -1] #to print the list in reverse
[5, 4, 3, 2, 1]
>>> 
```

Some more methods:

min()
max()
len()
sum()
reverse()

in operator
not in operator

+ operator in list → **concatenate** two lists
* operator in list → **multiply the list “n” times** and return the single list.

```
ex: del ls[2 : 5]
```

How to Input a list from user with passed delimiter:

```
ls = list(map(int, input("entered number space seperated").split(',')))
```

```
>>>
>>> ls = list(map(int, input("entered number comma seperated").split(',')))
entered number comma seperated1,2,3,4,5
>>> print(ls)
[1, 2, 3, 4, 5]
>>>
>>> ls = list(map(int, input("entered number space seperated").split(' ')))
entered number space seperated1 2 3 4 5
>>> print(ls)
[1, 2, 3, 4, 5]
>>>
>>>
```

Practice Questions:

1. WAPP to check the number n is prime or not.
2. WAPP to check the entered number n is Armstrong or not.
3. WAPP to reverse the number(integer) entered by the user.
4. WAPP to check the entered number is even or not.
5. WAPP to find minimum and maximum number among three numbers (entered by the user).