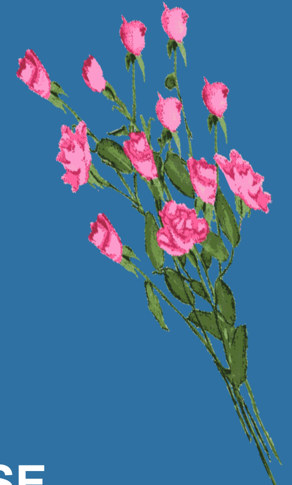
OBJECT ORIENTED PROGRAMMING THROUGH PYHTON



CSE

II SEMESTER

Prepared by:

Dr. P Govardhan, Associate Professor,CSE
Ms.N Jayanthi, Assistant Professor,CSE

Course Objectives

The course should enable the students to:	
1	Understand the fundamentals of Python programming concepts and its applications.
2	Understand the object-oriented concepts using Python in problem solving.
3	Apply string handling and function basics to solve real-time problems.
4	Illustrate the method of solving errors using exception handling.
5	Design and implement programs using graphical user interface

Course Outcomes

The course should enable the students to:	
CO 1	Describe Features of Python, Features of Object oriented programming system. Classes and Objects, Encapsulation, Abstraction, Inheritance, Polymorphism.
CO 2	Determine Creating a class, Inheritance and Polymorphism, Types of inheritance, Polymorphism, Abstract classes and Interfaces
CO 3	Understand Creating strings and Defining a function, Calling a function, Recursive functions.
CO 4	Explore the concept of Errors in a Python program, Exceptions, Exception handling,
CO 5	Knowledge The Root window, Fonts and colors, Working with containers, Canvas, Frames, Widgets ,Button widget, Label Widget, Message widget, Text widget, Radio button Widget, Entry widget.

Course Learning Outcomes

The course will enable the students to:	
CLO 1	Describe the Features of Python, Data types.
CLO 2	Summarize the concept of Operators, Input and output, Control Statements.
CLO 3	Identify the features of Object Oriented Programming System (OOPS).
CLO 4	Use the concept of Classes and Objects, Encapsulation.
CLO 5	Describe Abstraction, Inheritance, and Polymorphism.
CLO 6	Determine Creating a class, The Self variable.

Course Learning Outcomes cont.

The course will enable the students to:	
CLO 7	Understand types of variable, Namespaces.
CLO 8	Determine types of Methods, Inheritance and Polymorphism.
CLO 9	Use Constructors in inheritance, the super() method.
CLO 10	Illustrate types of inheritance, Polymorphism, Abstract classes and Interfaces.
CLO 11	Understand Creating strings and basic operations on strings.
CLO 12	Analyze the concept of String testing methods, Defining a function.

Course Learning Outcomes cont.

The course will enable the students to:	
CLO 13	Illustrate Calling a function, Returning multiple values from a function.
CLO 14	Contrast the Usage of Functions are first class objects, Formal and actual arguments,
CLO 15	Define Positional arguments, Recursive functions.
CLO 16	Discuss the concept of Errors in a Python program.
CLO 17	Understand Exceptions, Exception handling.
CLO 18	Summarize the concept of types of exceptions.

Course Learning Outcomes cont.

The course will enable the students to:	
CLO 19	Discuss the Except block, the assert statement.
CLO 20	Understand the concept of user-defined exceptions.

UNIT -I

Running Course Learning Outcomes

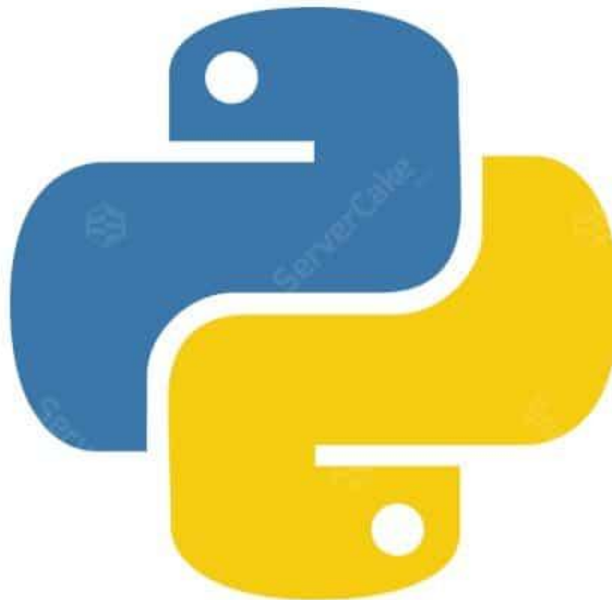
The course will enable the students to:	
CLO 1	Describe the Features of Python, Data types.
CLO 2	Summarize the concept of Operators, Input and output, Control Statements.
CLO 3	Identify the features of Object Oriented Programming System (OOPS),
CLO 4	Use the concept of Classes and Objects, Encapsulation.
CLO 5	Describe Abstraction, Inheritance, and Polymorphism.

Contents

- Features of Python
- Data types
- Operators in python
- Input and output
- Control Statements
- Features of object oriented programming system
- Classes and Objects
- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

Guido Van Rossum

DID YOU KNOW?

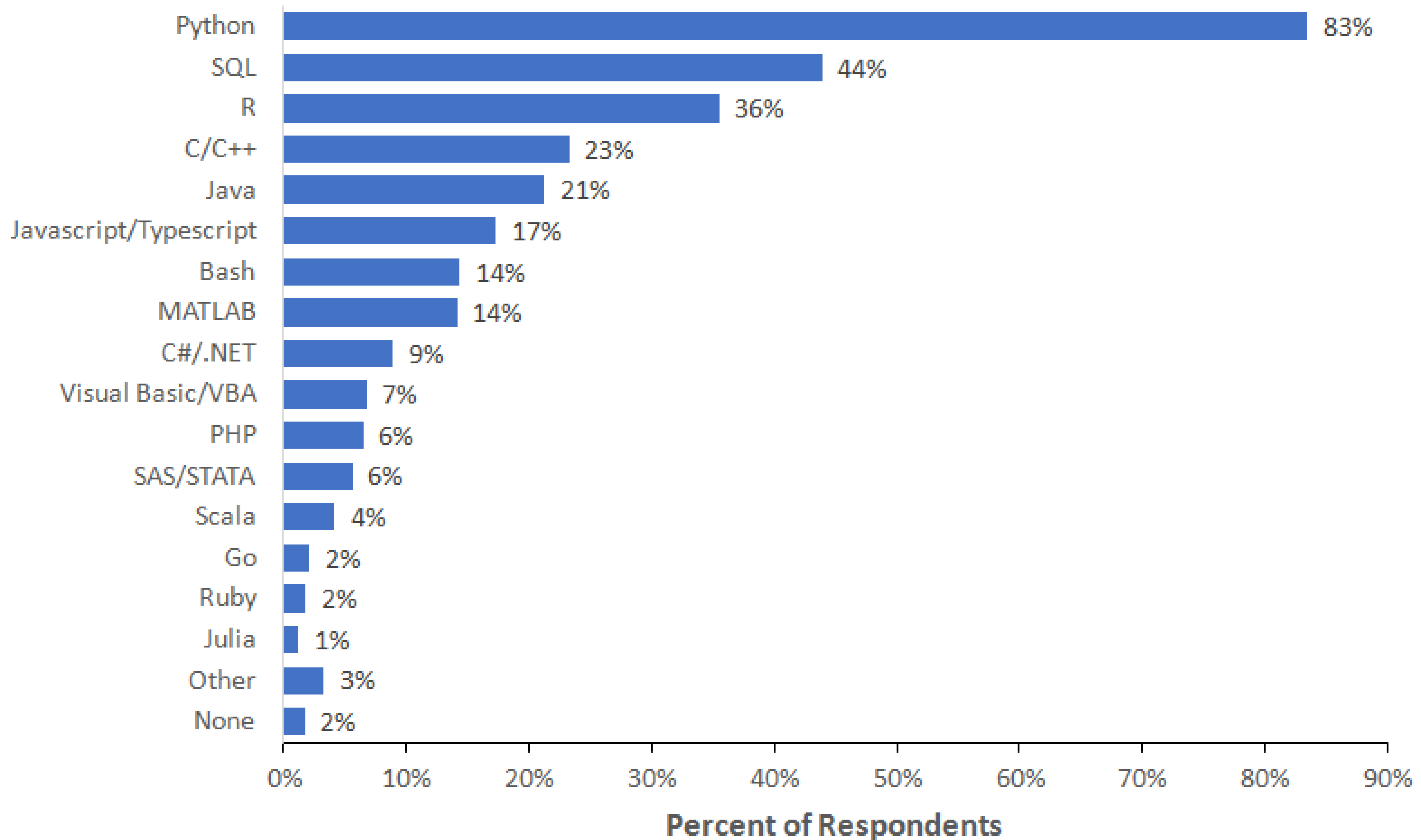


Guido Van Rossum is a Dutch programmer who is best known as the author of the Python Programming Language

Brief History of Python Language

- Python is a general-purpose, dynamic, interpreted high-level programming language.
- Conceptualized in the late 1980's.
- Created by Guido van Rossum (Netherlands) and first released in 1991.
- A descendant of ABC language.
- Open sourced from the beginning, managed by Python Software Foundation.
- Scalable, Object oriented and functional from the beginning.
- Python versions
  - First version 0.9.0 in February 1991
  - Version 1.0 in January 1994
  - Version 2.0 in October 2000
  - Version 3.0 in 2008

Best Programming Language



Features of Python Language

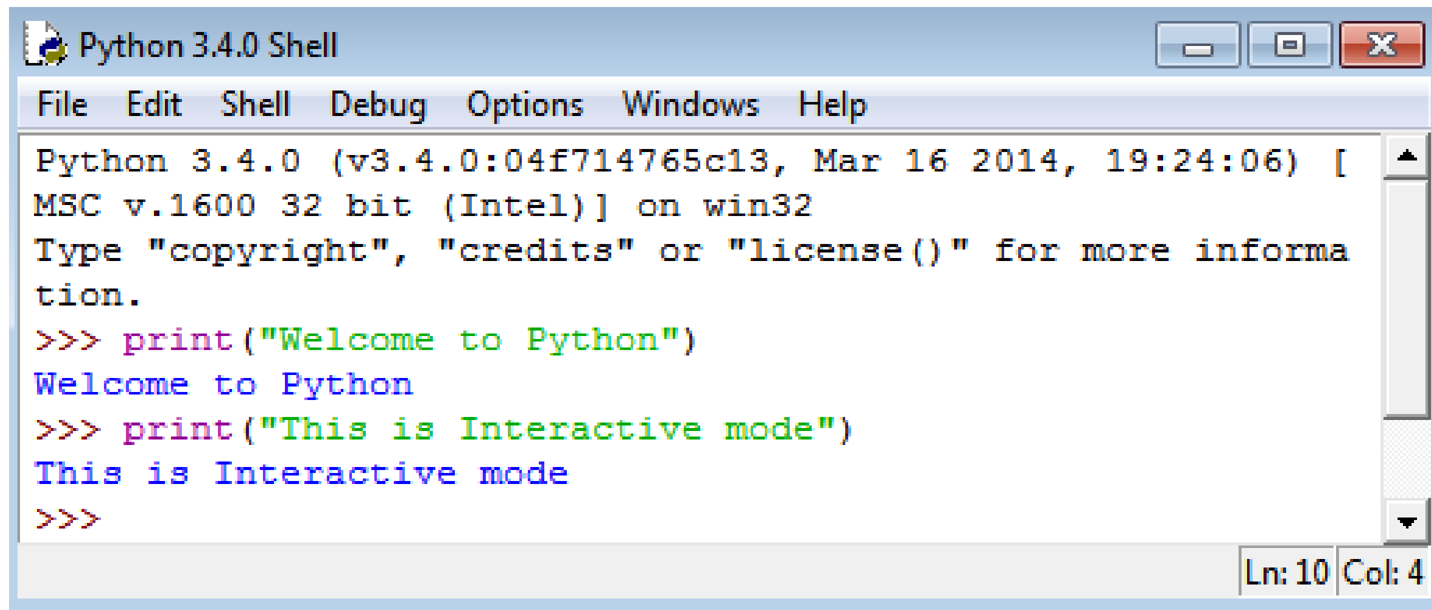
- Simple
- Easy to learn
- Open source
- High level language
- Dynamically typed
- Platform independent
- Portable
- Procedure and object oriented

Python Interactive Shell

- Python provides an interactive shell, which is used in between the user and operating system
- In other words, Python provides a command line interface with the Python shell known as Python interactive shell.
- Python commands are run using the Python interactive shell.
- User can work with Python shell in two modes: interactive mode and script mode.
- Interactive mode allows the user to interact with the operating system. When the user types any Python statement / expression, the interpreter displays the results instantly.
- In script mode, user types a Python program in a file and then uses the interpreter to execute the file. In interactive mode, user can't save the statements / expressions and need to retype once again to re-run them.

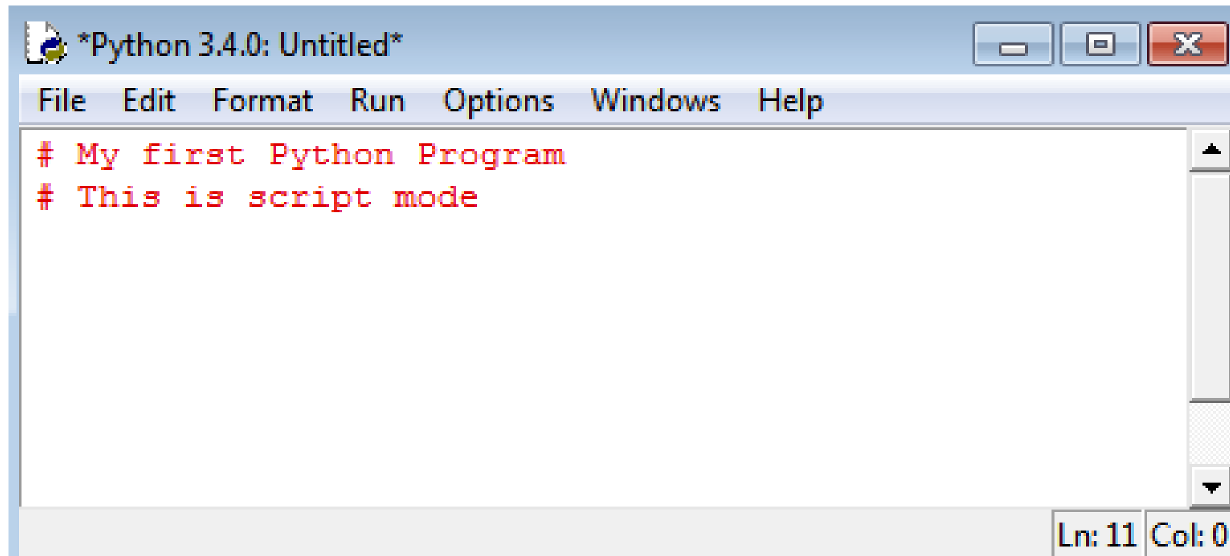
Interactive Mode

- When the user starts the Python IDLE the following window will appear and it shows the interactive shell. This window shows the primary prompt '>>>' where the user types commands to run by the interpreter.



Script Mode

- In this mode, user types a set of statements called a program in a file and then save the program with ‘filename.py’ as extension. Then the interpreter is used to execute the file contents. This mode is convenient when the user wants to write and save multiple lines of code, so that it can be easily modifiable and reusable.



Python Shell as a Simple Calculator

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

Flavors of Python

- Flavors of Python are nothing but different types of Python compilers available, which are useful to integrate various programming languages into Python. The following are some of the important and popularly used flavors of Python.
  - Cpython
  - Jython
  - IronPython
  - Pypy
  - Pythonxy
  - RubyPython
  - StacklessPython
  - ActivePython

Built-in Data Types in Python

- Every programming language has the ability to create and manipulate object / variable. In a program variables are used to store values so that it can be used later. Every object / variable has an identity, type and a value which it refers. Identity of an object is nothing but its address in memory when it is created. Type or data type indicates is a range of values and operations allowed on those values.

Keywords in Python

- Keywords are reserved words with predefined meaning in any programming languages and these words can’t be used as normal variables. One can check the number of keywords using help() command -> keywords in Python.

```
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get mo
re help.

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

Assigning values to variables

```
>>> a = 100                 # a is integer
>>> height = 50.5           #height is float
>>> player = "Sachin"       #player is string
>>> x = y = z = 10          # This statement assign 10 to x, y, z
>>> x = 5
>>> x                       #assigns 5 to x
>>> 5  = x                  #SyntaxError: can't assign to literal
```

Multiple Assignments

- Consider an example where multiple values are assigned to the same variable and when the program runs, it prints different results.

```
x = 5
print ('x = ' + x)
x = 10
print ('x = ' + x)
x = 15
print ('x = ' + x)
```

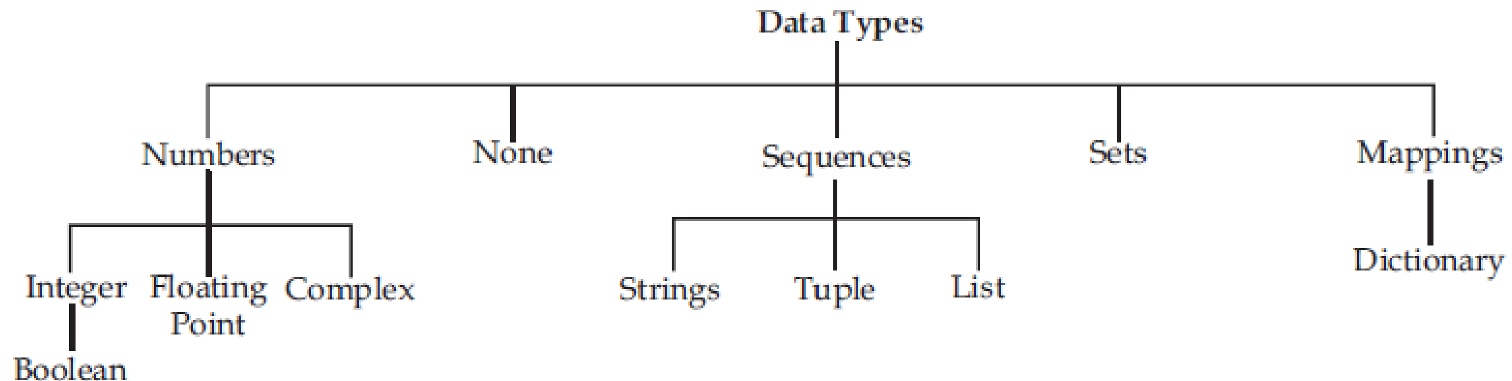
```
x =  5
x =  10
x =  15
```

```
x = 5
print ('x = ' + str(x))
x = 10
print ('x = ' + str(x))
x = 15
print ('x = ' + str(x))
```

```
x =  5
x =  10
x =  15
```

Standard Data Types in Python

- Python has five standard data types, named Numbers, None, Sequences, Sets and Mappings. Python sets the type of variable based on the type of value assigned to it and it will automatically change the variable type if the variable is set to some other value.



Numbers

Python supports the following numeric types.

- **int** - integers of unlimited length in Python 3.x .
- **long** - long integers of unlimited length, but exists only in Python 2.x.
- **float** - floating point numbers.
- **complex** - complex numbers.

```
>>> a = 10
>>> type(a)
<class 'int'>
>>> b = 125.50
>>> type(b)
<class 'float'>
>>> c = 5 + 6j
>>> type(c)
<class 'complex'>
>>> str1 = "Welcome to Python"
>>> type(str1)
<class 'str'>
```

Boolean

- True and False are Boolean literals used in Python and these are used to represent the truth / falsity of any condition / expression.

```
>>> x = True
>>> type(x)
<class 'bool'>
>>> y = (3 > 5)
>>> type(y)
<class 'bool'>
>>> y
False
>>> x
True
```

None

- In Python None keyword is an object which is equivalent to Null. A None can be assigned to a variable during declaration or while evaluating an expression.

```
>>> var = None
>>> type(var)
<class 'NoneType'>
```

Strings

- Strings are identified as group of characters represented in quotation marks. Python allows both a pair of single and double quotes for writing strings. Strings written in triple quotes can span multiple lines of text. Strings in Python are immutable data type i.e. each time a new string object is created when one makes any changes to a string.

```
>>> s1 = 'Hello Python'
>>> s1
'Hello Python'
>>> s2 = "Welcome"
>>> s2
'Welcome'
>>> s3 = s1[0]     #output will be first character
>>> s3
'H'
>>> s4 = s1[0:5]    #output will be first five characters
>>> s4
'Hello'
```

Strings

- Python can also manipulate strings. They can be enclosed in single quotes (‘abc’) or double quotes (“abc”) with the same result.

```
>>> "welcome to Python"
'welcome to Python'
>>> 'Enjoy learning'
'Enjoy learning'
>>> s = 'Beautiful Language'
>>> s
'Beautiful Language'
>>> #Strings can be concatenated with the + operator and repeated with *
>>> #print 3 times hello python
>>> 3 * 'hello' + 'Python'
'hellohellohelloPython'
>>> 'Hello' 'Python'   #another way of concatenation
'HelloPython'
>>> #break long strings
>>> line = ('This is the first line'
            'This is the second line'
            'This is the third line')
>>> line
'This is the first lineThis is the second lineThis is the third line'
>>> #concatenate a variable and a literal using + operator
>>> prefix = 'Py'
>>> prefix + 'thon'
'Python'
```

Tuple

- A tuple contains a list of items enclosed in parentheses and none of the items cannot be updated. Hence tuples are immutable.

```
>>> tuple1 = (100, 200, 'hello', 456.789)
>>> tuple2 = ('Hello', 'World')
>>> tuple1
(100, 200, 'hello', 456.789)
>>> tuple2
('Hello', 'World')
>>> tuple1[0]      #gives first value in the tuple
100
>>> tuple1 + tuple2    #combines both the tuples
(100, 200, 'hello', 456.789, 'Hello', 'World')
>>> tuple1[1] = 300
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    tuple1[1] = 300
TypeError: 'tuple' object does not support item assignment
```

List

- A list contains items separated by commas and enclosed within square brackets. A list in Python can contain heterogeneous data types.

```
>>> list1 = [100, 'happy', 123.456, 'A']
>>> list1
[100, 'happy', 123.456, 'A']
>>> list2 = [10, 20, 30, 40 , 50]
>>> list2
[10, 20, 30, 40, 50]
>>> list3 = ['Hello', 'Python']
>>> list3
['Hello', 'Python']
>>> list1[0:2]     #outputs first two elements of list
[100, 'happy']
>>> list3 * 2      #outputs list3 two times
['Hello', 'Python', 'Hello', 'Python']
>>> list1 + list3  #combines both the lists
[100, 'happy', 123.456, 'A', 'Hello', 'Python']
```

Sets

- In Python sets are unordered collection of objects enclosed in parenthesis and there are basically two types of sets:
  - Sets - These are mutable and can be updated with new elements once sets are defined.
  - Frozen Sets - These are immutable and cannot be updated with new elements once frozen sets are created.

```
>>> flowers = {'Rose', 'Jasmine', 'Rose', 'Lily', 'Rose', 'Jasmine'}
>>> flowers       #eliminates duplicates
{'Lily', 'Jasmine', 'Rose'}
>>> set1 = set('Welcome')
>>> set1          #prints unique letters in set1
{'m', 'c', 'W', 'e', 'l', 'o'}
>>> set1.add('z')  #adds a new element to a set
>>> set1
{'m', 'z', 'c', 'W', 'e', 'l', 'o'}

>>> set2 = frozenset('Welcome')
>>> set2
frozenset({'m', 'c', 'W', 'e', 'l', 'o'})
>>> cities = frozenset(["Delhi", "Hyderabad", "Mumbai"])
>>> cities
frozenset({'Hyderabad', 'Delhi', 'Mumbai'})
>>> set2.add('z')
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    set2.add('z')
```

Dictionary

- In Python dictionary data type consists of key-value pairs and it is enclosed by curly braces. Values can be assigned and accessed using square brackets.

```
>>> dict1 = {'Name':'Happy', 'age': 25}
>>> dict1
{'Name': 'Happy', 'age': 25}
>>> dict1['Name']        #gives the value associated with the key
'Happy'
>>> dict1.values()
dict_values(['Happy', 25])
>>> dict1.keys()
dict_keys(['Name', 'age'])
```

Mutable and Immutable Data Types

- The following table gives examples of mutable and immutable data types in Python.

Mutable Data Types	Immutable Data Types
list	int, long
set	float, complex
dict	str
	tuple
	frozenset

Operators in Python

All the operators in Python are classified according to their nature and type and they are:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Boolean Operators
- Membership Operators
- Identity Operators

Arithmetic Operators

- These operators perform basic arithmetic operations like addition, subtraction, multiplication, division etc. and these operators are binary operators that means these operators acts on two operands. And there are 7 binary arithmetic operators available in Python.

Operator	Meaning	Example	Result
+	Addition	10 + 7	12
-	Subtraction	10.0 - 1.5	8.5
*	Multiplication	30 * 3	900
/	Float Division	5 / 2	2.5
//	Integer Division	5 // 2	2
**	Exponentiation	3 ** 2	9
%	Remainder	10 % 3	1

Operator	Priority
Parenthesis (( ), [ ])	First
Exponentiation (**)	Second
Multiplication (*), Division (/, //), Modulus (%)	Third
Addition (+), Subtraction (-)	Fourth
Assignment	Fifth

Relational Operators

- Relational operators are used for comparison and the output is either True or False depending on the values we compare. The following table shows the list of relational operators with example.

Operator	Meaning	Example	Result
<	Less than	5 < 7	True
>	Greater than	9 > 5	True
<=	Less than equal to	8 <= 8	True
>=	Greater than equal to	7 >= 9	False
==	Equal to	10 == 20	False
!=	Not equal to	9 != 6	True

Logical Operators

- Logical operators are used to form compound conditions which are a combination of more than one simple condition. Each of the simple conditions are evaluated first and based on the result compound condition is evaluated. The result of the expression is either True or False based on the result of simple conditions.

Operator	Meaning	Example	Result
and	Logical AND	(5 > 7) and (3 < 5)	False
or	Logical OR	(7 == 7) or (5 != 5)	True
not	Logical NOT	not(3 <= 2)	True

```
>>> (5 > 7) and (3 < 5)
False
>>> (7 == 7) or (5 != 5)
True
>>> not(3 <=2)
True
```

Assignment Operators

- These operators are used to store a value into a variable and also useful to perform simple arithmetic operations. Assignment operators are of two types: simple assignment operator and augmented assignment operator. Simple assignment operators are combined with arithmetic operators to form augmented assignment operators. The following table shows a list of assignment operators and its use.

Operator	Meaning	Example	Result
=	Simple assignment	a = 10	10
+=	Addition assignment	a = 5 a += 8	13
-=	Subtraction assignment	b = 5 b -= 8	-3
*=	Multiplication assignment	a =10 a *= 8	80
/=	Float Division assignment	a = 10 a /= 8	1.25
//=	Integer Division assignment	b = 10 b //= 10	1
**=	Exponentiation assignment	a = 10 a %= 5	0
%=	Remainder assignment	b = 10 b ** = 8	100000000

Bitwise Operators

- Bitwise Operators acts on individual bits of the operands. These operators directly act on binary numbers. If we want to use these operators on integers then first these numbers are converted into binary numbers and then bitwise operators act on those bits. The following table shows the list of bitwise operators available in Python.

Operator	Meaning	Example	Result
&	Bitwise AND	a = 10 = 0000 1010 b = 11 = 0000 1011 a & b = 0000 1010 = 10	a & b = 10
|	Bitwise OR	a = 10 = 0000 1010 b = 11 = 0000 1011 a | b = 0000 1011 = 11	a | b = 11
^	Bitwise XOR	a = 10 = 0000 1010 b = 11 = 0000 1011 a ^ b = 0000 0001 = 1	a ^ b = 1
~	Bitwise Complement	a = 10 = 0000 1010 ~a = 1111 0101 = -11	~a = -11
<<	Bitwise Left Shift	a = 10 a << 2 = 40	a << 2 = 40
>>	Bitwise Right Shift	a = 10 a >> 2 = 2	a >> 2 = 2

Boolean Operators

- There are three boolean operators that act on bool type literals and provide bool type output. The result of the boolean operators are either True or False.

Operator	Meaning	Example	Result
and	Boolean AND	a = True, b = False a and b = True and False	a and b = False
or	Boolean OR	a = True, b = False a or b = True or False	a or b = True
not	Boolean NOT	a = True not a = not True	not a = False

Membership Operators

There are two membership operators in Python that are useful to test for membership in a sequence.

- **in**: This operator returns True if an element is found in the specified sequence, otherwise it returns False.
- **not in: This** operator returns True if any element is not found in the sequence, otherwise it returns True.

Identity Operators

These operators are used to compare the memory locations of two objects. Therefore it is possible to verify whether the two objects are same or not. In Python id() function gives the memory location of an object. Example id(a) returns the identity number or memory location of object a. There are two identity operators available in Python. They are

- **is:** This operator is used to compare the memory location of two objects. If they are same then it returns True, otherwise returns False.
- **is not:** This operator returns True if the memory locations of two objects are not same.If they are same then it returns False.

```
# Identity Operators
a = 100
print("Identity Number of a =  ", id(a))
b = 200
print("Identity Number of b =  ", id(b))
if a is b:
    print(" a and b have same identity")
else:
    print("a and b have different identity")


"""
Identity Number of a =   1528939480
Identity Number of b =   1528941080
a and b have different identity

"""
```

Operator Precedence and Associativity

- An expression may contain several operators and the order in which these operators are executed in sequence is called operator precedence. The following table summarizes the operators in descending order of their precedence.

Operator	Name	Precedence
( )	Parenthesis	1st
**	Exponentiation	2nd
-, ~	Unary minus, bitwise complement	3rd
*, /, //, %	Multiplication, Division, Floor Division, Modulus	4th
+, -	Addition, Subtraction	5th
<<, >>	Bitwise left shift, bitwise right shift	6th
&	Bitwise AND	7th
^	Bitwise XOR	8th
|	Bitwise OR	9th
>, >=, <, <=, = =, !=	Relational Operators	10th
=, %=, /=, //=, -=, +=, *=, **=	Assignment Operators	11th
is, is not	Identity Operators	12th
in, not in	Membership Operators	13th
not	Logical NOT	14th
or	Logical OR	15th
and	Logical AND	16th

Single Line and Multiline Comments

- There are two types of comments used in Python:
- **Single Line Comments**: These are created simply by starting a line with the hash character (#), and they are automatically terminated by the end of line. If a line using the hash character (#) is written after the Python statement, then it is known as inline comment.

- **Multiline Comments**: When multiple lines are used as comment lines, then writing hash character (#) in the beginning of every line is a tedious task. So instead of writing # character in the beginning of every line, we can enclose multiple comment lines within ''' (triple single quotes) or """ (triple double quotes). Multi line comments are also known as block comments.

INPUT AND OUTPUT

- The purpose of a computer is to process data and return results.The data given to the computer is called input. The results returned by the computer are called output. So, we can say that a computer takes input, processes that input and produces the output.

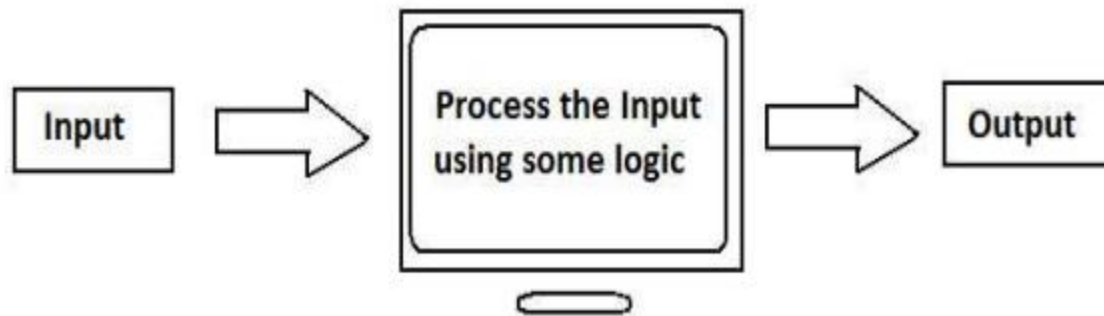


Figure 5.1: Processing Input by the Computer

UNIT -II

Control Structures

A control structure is a block of programming that analyzes variables and decides which statement to execute next, based on the given parameters. The term 'control' denotes the direction in which the program flows. Usually, loops are used to execute a control statement, a certain number of times.

Basically, control structures determine the flow of events in the program.

If statement: This is used to check a condition and executes the operations/statements within the if block only when the given condition is true.

Syntax:

```
if condition:
    True Statements
```

If…else statements

If…else statements: These statements are used to check a condition and executes the operations/statements within the if block only when the given condition is true. If the given condition is false, the statements in the else block will be executed.

Syntax:

```
if condition:
    True Statements
else:
    False Statements
```

If …elif… else statements

If …elif… else statements: If we want to check more than one condition we can use the elif statements. If a condition is true then the statements within the if block will be executed. If the condition is false, we can provide an elif statement with a second condition and the statements within the elif block will be executed only when the condition is true. We can provide multiple elif statements and an else statement at the end if all the above conditions are false.

Syntax:

```
if condition:
    True Statements
elif condition2:
    True Statements
 elif condition3:
    True Statements
…….
else:
    False Statements
```

Loops in Python

- Loops are used to repeat a set of statements/single statement, a certain number of times. In Python, there are two loops, for loop and while loop. The Python for loop also works as an iterator to iterate over items in list/dictionary or characters in strings.

for Loop: It can be used to iterate over a list/string/dictionary or iterate over a range of numbers.

Syntax:

```
for variable in range(starting number , ending number + 1 , step size):
    statements
 (or)
 for element in sequence:
    statements
```

While Loop

While Loop: This is used, whenever a set of statements should be repeated based on a condition. The control comes out of the loop when the condition is false. In while loop we must explicitly increment/decrement the loop variable (if any) whereas in for, the range function would automatically increment the loop variable.

Syntax:

```
while condition:
    statement(s)

    increment/decrement
```

Break and Continue Statement

break statement: This statement is used to terminate the loop it is present in. Control goes outside the loop it is present in. If a break statement is present in a nested loop, it only comes out of the innermost loop.

Syntax:

```
while condition:
    statments
    if condition:
        break
    statements
```

Continue statement: This statement is used to skip the current iteration. The loop will not be terminated, it just won't execute the statements below the continue statement. The incrementing will be done in for loop. If the increment statement is written below continue, it won't be executed in while loop.

Syntax:

```
while condition:
    statement(s)
    if condition:
        continue
     statements
```

Pass Statement

Pass statement: This statement is used as placeholder. For example, we want to create a function but are not sure of its content. If we create a function and leave it, an error will occur. To counter this error, we use pass statement.

Syntax:

```
def function(parameters):
   pass
(or)
for elements in sequence:
   pass
(or)
while condition:
   pass
(or)
if condition:
   pass
```

UNIT -III

LIST

List

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

Create a List:

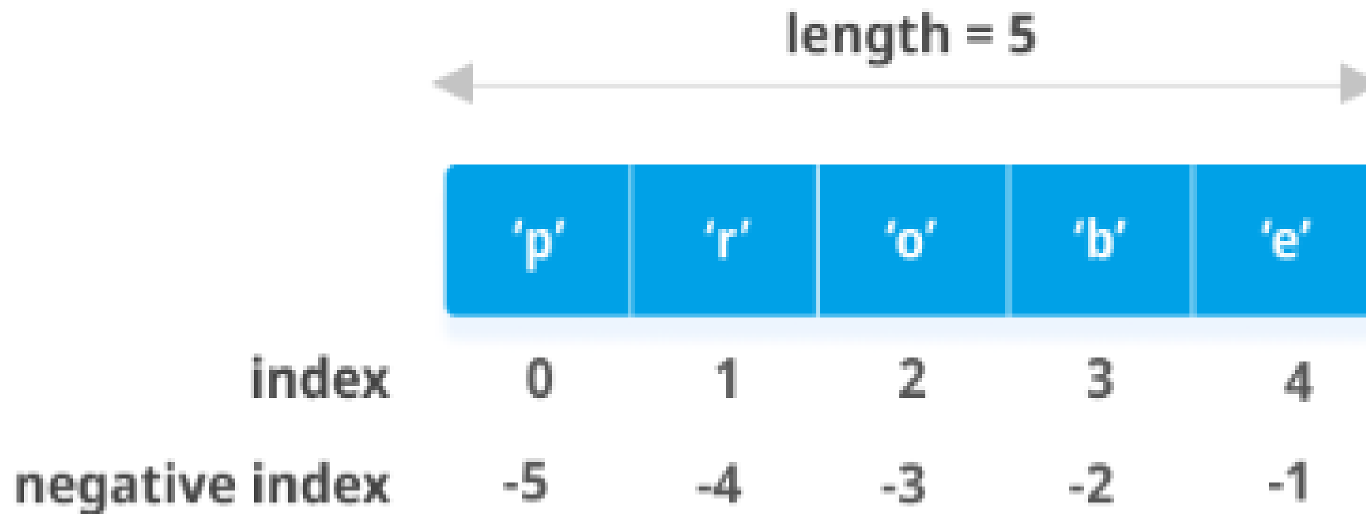
list = ["aaa", "bbb", "ccc"]

Access Items

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index.

LIST

How to slice lists in Python?



LIST

```
my_list = ['p','r','o','g','r','a','m','i','z']
print(my_list[2:5])  # elements 3rd to 5th
print(my_list[:-5])  # elements beginning to 4th
print(my_list[5:])   # elements 6th to end
print(my_list[:])              # elements beginning to
end
```

LIST

FUNCTION	DESCRIPTION
Append()	Add an element to the end of the list
Extend()	Add all elements of a list to the another list
Insert()	Insert an item at the defined index
Remove()	Removes an item from the list
Pop()	Removes and returns an element at the given index
Clear()	Removes all items from the list
Index()	Returns the index of the first matched item
Count()	Returns the count of number of items passed as an argument
Sort()	Sort items in a list in ascending order
Reverse()	Reverse the order of items in the list
copy()	Returns a copy of the list

Tuples

2000 IARE EDUCATION FOR LIBERATION

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Create a Tuple:

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
```

Tuples

Basic Tuples Operations

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Built-in Tuple Functions

SN	Function	Description
1	cmp(tuple1, tuple2)	It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false.
2	len(tuple)	It calculates the length of the tuple.
3	max(tuple)	It returns the maximum element of the tuple.
4	min(tuple)	It returns the minimum element of the tuple.
5	tuple(seq)	It converts the specified sequence to the tuple.

Dictionary

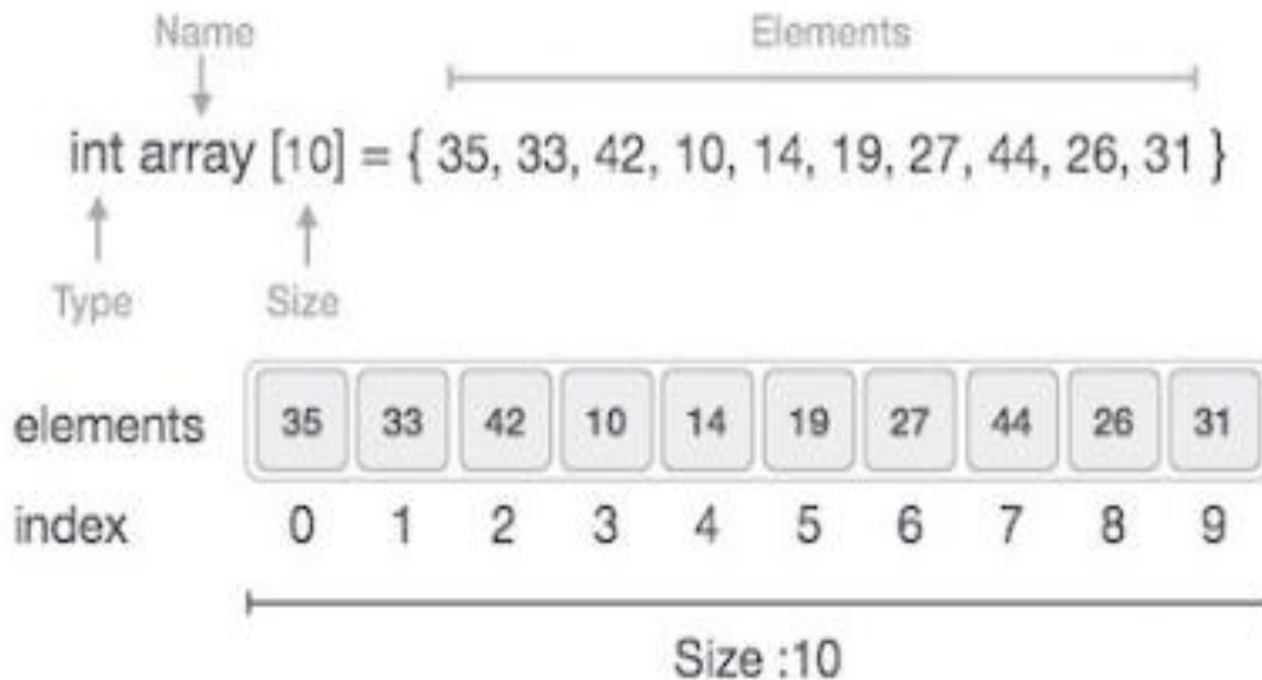
A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Dictionary Methods

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Arrays

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.



Basic Operations

Following are the basic operations supported by an array.

Traverse – print all the array elements one by one.

Insertion – Adds an element at the given index.

Deletion – Deletes an element at the given index.

Search – Searches an element using the given index or by the value.

Update – Updates an element at the given index.

NumPy

NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays. It is the fundamental package for scientific computing with Python.

NumPy

NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays. It is the fundamental package for scientific computing with Python.

UNIT -IV

IARE
2000
EDUCATION FOR LIBERATION

Creating strings

We can create a string in Python by assigning a group of characters to a variable. The group of characters should be enclosed inside single quotes or double quotes as:

```
s1 = 'Welcome to Core Python learning'
s2 = "Welcome to Core Python learning"
```

Strings and Functions

Escape Character	Meaning
\a	Bell or alert
\b	Backspace
\n	New line
\t	Horizontal tab space
\v	Vertical tab space
\r	Enter button
\x	Character x
\\	Displays single\

Strings and Functions

Sr.No.	Methods with Description
1	capitalize() Capitalizes first letter of string
2	center(width, fillchar) Returns a space-padded string with the original string centered to a total of width columns.
3	count(str, beg= 0,end=len(string)) Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	decode(encoding='UTF-8',errors='strict') Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	encode(encoding='UTF-8',errors='strict') Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.

Strings and Functions

6	endswith(suffix, beg=0, end=len(string)) Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	expandtabs(tabsize=8) Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	find(str, beg=0 end=len(string)) Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	index(str, beg=0, end=len(string)) Same as find(), but raises an exception if str not found.
10	isalnum() Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	isalpha() Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

Strings and Functions

12	isdigit() Returns true if string contains only digits and false otherwise.
13	islower() Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	isnumeric() Returns true if a unicode string contains only numeric characters and false otherwise.
15	isspace() Returns true if string contains only whitespace characters and false otherwise.
16	istitle() Returns true if string is properly "titlecased" and false otherwise.
17	isupper() Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	join(seq) Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.

19	len(string) Returns the length of the string
20	ljust(width[, fillchar]) Returns a space-padded string with the original string left-justified to a total of width columns.
21	lower() Converts all uppercase letters in string to lowercase.
22	lstrip() Removes all leading whitespace in string.
23	maketrans() Returns a translation table to be used in translate function.
24	max(str) Returns the max alphabetical character from the string str.
25	min(str) Returns the min alphabetical character from the string str.

Strings and Functions

26	replace(old, new [, max]) Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	rfind(str, beg=0,end=len(string)) Same as find(), but search backwards in string.
28	rindex(str, beg=0, end=len(string)) Same as index(), but search backwards in string.
29	rjust(width,[, fillchar]) Returns a space-padded string with the original string right-justified to a total of width columns.
30	rstrip() Removes all trailing whitespace of string.
31	split(str="", num=string.count(str)) Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	splitlines(num=string.count('\n')) Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.

33	startswith(str, beg=0,end=len(string)) Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	strip([chars]) Performs both lstrip() and rstrip() on string.
35	swapcase() Inverts case for all letters in string.
36	title() Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	translate(table, deletechars="") Translates string according to translation table str(256 chars), removing those in the del string.
38	upper() Converts lowercase letters in string to uppercase.
39	zfill (width) Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).

40	isdecimal() Returns true if a unicode string contains only decimal characters and false otherwise.

A function is similar to a program that consists of a group of statements that are intended to perform a specific task. The main purpose of a function is to perform a specific task or work. Thus when there are several tasks to be performed, the programmer will write several functions. There are several ‘built-in’ functions in Python to perform various tasks. For example, to display output, Python has print() function. Similarly, to calculate square root value, there is sqrt() function and to calculate power value, there is power() function. Similar to these functions, a programmer can also create his own functions which are called ‘user-defined’ functions.

A function returns a single value in the programming languages like C or Java. But in Python, a function can return multiple values. When a function calculates multiple results and wants to return the results, we can use the return statement as:

```
return a, b, c
```

IARE
EDUCATION FOR LIBERATION
2000

Functions are First Class Objects

The following possibilities are noteworthy:

- It is possible to assign a function to a variable.
- It is possible to define one function inside another function.
- It is possible to pass a function as parameter to another function.
- It is possible that a function can return another function.

Functions are First Class Objects

The following possibilities are noteworthy:

- It is possible to assign a function to a variable.
- It is possible to define one function inside another function.
- It is possible to pass a function as parameter to another function.
- It is possible that a function can return another function.

Arguments

The actual arguments used in a function call are of 4 types:

1. Positional arguments
2. Keyword arguments
3. Default arguments
4. Variable length arguments

Recursive Functions

A function that calls itself is known as ‘recursive function’.

factorial(3) = 3 * factorial(2)
= 3 * 2 * factorial(1)
= 3 * 2 * 1 * factorial(0)
= 3 * 2 * 1 * 1 = 6

UNIT - V

Object Oriented Concepts

- Object oriented programming concept is associated with the concept of class, objects and various other concepts like abstraction, inheritance, polymorphism, encapsulation etc.
- **Class**: - Class is a user defined data type. It is a set of attributes (variables) and methods (functions). It is created using the keyword 'class'.
- **Object:** - Object is a unique instance of a class. We can use the same class as blueprint for creating number of different objects. The class describes what the object will be.
- **Attributes**: - Attributes are the member variables defined inside a class and can be accessed by the objects by using dot operator.
- **Method**: - Methods are functions defined inside a class. They can be accessed by the objects by using dot operator. All the methods in class have self as first parameter.

Example

```
class Car:
    #constructor method where attributes are defined
    def __init__(self):
        self.speed=50
        self.color="white"
        self.modelno=1795
    def accelerate(self):
        self.speed+=5
    def paint(self,newcolor):
        self.color=newcolor
    def speed_down(self):
        self.speed-=1
    def brake(self):
        self.speed=0
    def get_speed(self):
        return self.speed

bmw=Car()
print('Initial speed',end='=')
print(bmw.speed)
bmw.accelerate()
print('Speed after accelerating',end='=')
print(bmw.speed)
print('Initial color',end='=')
print(bmw.color)
bmw.paint('red')
print('Color after painting',end='=')
print(bmw.color)
```

```
Initial speed=50
Speed after accelerating=55
Initial color=white
Color after painting=red
```

__init__ method

- **__init__:** The method init is the most important method in the class. This is called when an instance (object) of the class is created, using the class name as a function. The init___ method is called as constructor.
- **self:** In class, all methods have self as their first parameter (python adds self as argument which is well known to us) , although it isn’t explicitly passed(passed by users). We can’t use self while we call the method in a class. Within a method definition, self refers to the instance calling the method.
- In an ___init___method, attributes can be used to set the initial value of instance’s attributes in a class.

Features of Object Oriented Programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

- **Encapsulation:** Encapsulation refers to binding data and methods together inside a class. It keeps the data and methods safe from outside interference and misuse. Encapsulation prevents accessing data accidentally.

Inheritance

- **Inheritance:** It refers to creating a child class such that the child class would inherit all the properties (variables and methods) of the parent class. The parent class is called super class while the child class is called subclass.
- We have 3 types of inheritance mainly:
- **Single inheritance:** Only one sub class from super class.(superclass->subclass)
- **Hierarchical inheritance:** Inheriting from super class to as many subclasses.
- **Multilevel inheritance:** Inheriting properties from super class to sub class and then other sub classes.

Abstraction and Polymorphism

IARE
EDUCATION FOR LIBERATION
2000

- **Abstraction:** It refers to creating structure classes that are not implemented. Abstract classes are like a base class and many other classes inherit the properties of abstract class but the abstract class itself is not implemented.

- **Polymorphism:** It is derived from two Greek words, poly (many) and morph (form). Polymorphism allows us to define methods with the same name in two different classes. If the two different classes are parent class and child class then the parent class's method will be overwritten by the child class's method. This is known as Method Overriding.

Creating A Class

CLASS

- we write a class with the attributes and actions of objects. Attributes are represented by variables and actions are performed by methods. So, a class contains variable and methods.
- A function written inside a class is called a method. Generally, a method is called using one of the following two ways:
- class name.methodname()
- instancename.methodname()
- The general format of a class is given as follows:

```
Class Classname(object):
""" docstring describing the class """
attributes  def __init__(self):
def method1():
def method2():
```

Creating A CLASS(Contd..)

- A class is created with the keyword class and then writing the Classname. After the Classname, ‘object’ is written inside the Classname.
- This ‘object’ represents the base class name from where all classes in Python are derived.
- Even our own classes are also derived from ‘object’ class. Hence, we should mention ‘object’ in the parentheses.

```
class Student:
#another way is:
class Student(object):
#the below block defines attributes
def __init__(self):
self.name = ‘Vishnu’
self.age = 20
self.marks = 900
#the below block defines a method
```

Creating A CLASS(Contd..)

```
def talk(self):
    print('Hi, I am ', self.name)
    print('My age is', self.age)
    print('My marks are', self.marks)
```

- To create an instance, the following syntax is used:

```
instancename = Classname()
```

So, to create an instance (or object) to the Student class, we can write as:

```
s1 = Student()
```

When we create an instance like this, the following steps will take place internally:

1. First of all, a block of memory is allocated on heap. How much memory is to be allocated is decided from the attributes and methods available in the Student class.
2. After allocating the memory block, the special method by the name '__init__(self)' is called internally. This method stores the initial data into the variables. Since this method is useful to construct the instance, it is called 'constructor'.

3. Finally, the allocated memory location address of the instance is returned into ‘s1’ variable. To see this memory location in decimal number format, we can use id() function as id(s1).

Program

Program 1: A Python program to define Student class and create an object to it. Also, we will call the method and display the student’s details.

```
#instance variables and instance method
class Student:
#this is a special method called constructor.
    def __init__(self):
        self.name = 'Vishnu'
        self.age = 20
        self.marks = 900
#this is an instance method.
    def talk(self):
        print('Hi, I am', self.name)
```

```
print('My age is', self.age)
print('My marks are', self.marks)
#create an instance to Student class.
s1 = Student()
#call the method using the instance.
s1.talk()
```

Output:

```
C:\>python cl.py
Hi, I am Vishnu
My age is 20
My marks are 900
```

The Self Variable

- ‘self’ is a default variable that contains the memory address of the instance of the current class.
- For example, we create an instance to Student class as:

s1 = Student()

We use ‘self’ in two ways:

1. The ‘self’ variable is used as first parameter in the constructor as:

def __init__(self):

In this case, ‘self’ can be used to refer to the instance variables inside the constructor.

2. ‘self’ can be used as first parameter in the instance methods as:

def talk(self):

Here, talk() is instance method as it acts on the instance variables.

Constructor

- A constructor is a special method that is used to initialize the instance variables of a class. In the constructor, we create the instance variables and initialize them with some starting values. The first parameter of the constructor will be 'self' variable that contains the memory address of the instance. For example,

```
def __init__(self):
    self.name = 'Vishnu'
    self.marks = 900
```

Program 2: A Python program to create Student class with a constructor having more than one parameter.

```
#instance vars and instance method - v.20
class Student:                    #this is constructor.
def __init__(self, n ='', m=0):
    self.name = n
    self.marks = m                #this is an instance method.
    def display(self):
```

Constructor(Contd..)

```
print('Hi', self.name)
print('Your marks', self.marks)    #constructor is called without any arguments
s = Student()
s.display()
print('-----------------')        #constructor is called with 2 arguments
s1 = Student('Lakshmi Roy', 880)
s1.display()
print('-----------------')
```

Output: C:\>python cl.py

```
Hi
Your marks 0
-----------------
Hi Lakshmi Roy
Your marks 880
-----------------
```

Types of Variables

- The variables which are written inside a class are of 2 types:

1. Instance variables
2. Class variables or Static variables

Program 3: A Python program to understand instance variables.

```
#instance vars example
class Sample:              #this is a constructor.
def __init__(self):
self.x = 10                #this is an instance method.
def modify(self):
self.x+=1                  #create 2 instances
s1 = Sample()
s2 = Sample()
print('x in s1= ', s1.x)
print('x in s2= ', s2.x)       #modify x in s1
s1.modify()
```

Types of Variables (Contd..)

```
print('x in s1= ', s1.x)
print('x in s2= ', s2.x)
```

Output: C:\>python cl.py

x in s1= 10

x in s2= 10

x in s1= 11

x in s2= 10

Program 4: A Python program to understand class variables or static variables.

```
#class vars or static vars example
class Sample:                    #this is a class var
x = 10                   #this is a class method.
@classmethod
def modify(cls):
cls.x+=1                 #create 2 instances
s1 = Sample()
s2 = Sample()
```

Types of Variables (Contd..)

```
print('x in s1= ', s1.x)
print('x in s2= ', s2.x)
#modify x in s1
s1.modify()
print('x in s1= ', s1.x)
print('x in s2= ', s2.x)
```

Output: C:\>python cl.py

```
x in s1= 10
x in s2= 10
x in s1= 11
x in s2= 11
```

Namespaces (Contd..)

Namespaces

A namespace represents a memory block where names are mapped (or linked) to objects. Suppose we write:

```
n = 10
#understanding class namespace
class Student:          #this is a class var
    n=10                #access class var in the class namespace
    print(Student.n)    #displays 10
    Student.n+=1        #modify it in class namespace
    print(Student.n)    #displays 11
```

Types of Methods

2000 IARE EDUCATION FOR LIBERATION

- The purpose of a method is to process the variables provided in the class or in the method.
- We can classify the methods in the following 3 types:

1. Instance methods (a) Accessor methods (b) Mutator methods
2. Class methods
3. Static methods

Instance Methods

- Instance methods are the methods which act upon the instance variables of the class. Instance methods are bound to instances (or objects) and hence called as: instancename.method().
- Program: A Python program to store data into instances using mutator methods and to retrieve data from the instances using accessor methods.

```
#accessor and mutator methods  class
Student:                    #mutator method
def setName(self, name):
self.name = name          #accessor method
```

Types of Methods (Contd...)

```
def getName(self):
return self.name           #mutator method
def setMarks(self, marks):
self.marks = marks         #accessor method
def getMarks(self):
 return self.marks         #create instances with some data from keyboard
n = int(input('How many students? '))
i=0
while(i<n):                #create Student class instance
s = Student()
 name = input('Enter name: ')
s.setName(name)
marks = int(input('Enter marks: '))
s.setMarks(marks)          #retrieve data from Student class instance
print('Hi', s.getName())
```

Types of Methods (Contd...)

```
    print('Your marks', s.getMarks())
    i+=1  print('------------------')
```

Output: C:\>python cl.py

```
How many students? 2
Enter name: Vinay Krishna
Enter marks: 890
Hi Vinay Krishna
Your marks 890
 ------------------
Enter name: Vimala Rao
Enter marks: 750
Hi Vimala Rao
Your marks 750
------------------
```

Types of Methods (Contd...)

Class Methods

- These methods act on class level. Class methods are the methods which act on the class variables or static variables. These methods are written using @classmethod decorator above them. By default, the first parameter for class methods is 'cls' which refers to the class itself.
- Program 7: A Python program to use class method to handle the common feature of all the instances of Bird class.

```
#understanding class methods class Bird:                #this is a class var
        wings = 2                                       #this is a class method
    @classmethod
     def fly(cls, name):
print('{} flies with {} wings'.format(name, cls.wings))  #display information for 2 birds
    Bird.fly('Sparrow')
        Bird.fly('Pigeon')
```

Output: C:\>python cl.py

Sparrow flies with 2 wings   Pigeon flies with 2 wings

Types of Methods (Contd...)

IARE
EDUCATION FOR LIBERATION
2000

Static Methods

- We need static methods when the processing is at the class level but we need not involve the class or instances. Static methods are used when some processing is related to the class but does not need the class or its instances to perform any work.

Program : A Python program to create a static method that counts the number of instances created for a class.

```
#understanding static methods            class Myclass:
#this is class var or static var             n=0
#constructor that increments n when an instance is created
def __init__(self):                          Myclass.n = Myclass.n+1
#this is a static method to display the no. of instances
@staticmethod  def noObjects():
print('No. of instances created: ', Myclass.n)
#create 3 instances   obj1 = Myclass()    obj2 = Myclass()   obj3 = Myclass()
Myclass.noObjects()
```

Output: C:\>python cl.py   No. of instances created: 3