

This Material prepared by Nireekshan under Ramesh sir guidelines @ DVS

# **DVS**

# **Technologies**

## **PYTHON MATERIAL**

**Address : DVS Technologies, Opp to Home Town, Beside Biryani Zone, Marathahalli, Bangalore.** Web : [www.dvstechnologies.in](http://www.dvstechnologies.in)  
**Mob: 8892499499, 9632558585**  
**Phone : 080-42091111**

## Python Chapters Index

### 1. Introduction to Python

- ✓ What is Python
- ✓ Where are all python is using?
- ✓ History of Python?
- ✓ Why the name was python?
- ✓ Python versions
- ✓ Python supports Functional and OOPs as well
- ✓ Solving few requirements by using C, Java and python
- ✓ Machine language
- ✓ Translator
- ✓ Interpreter
- ✓ Compiler
- ✓ Python keywords (33)
- ✓ Features of Python
- ✓ Flavours of Python
- ✓ Python versions
- ✓ Programs

### 2. Coding introduction

- ✓ Installing Python on Windows OS
- ✓ Ways to write python program
- ✓ Python program execution steps
- ✓ Understanding first python program
- ✓ First python program executing by using python IDLE
- ✓ Python program internal flow: Short answer
- ✓ Python program internal flow: Long answer
- ✓ Python Virtual Machine
- ✓ JIT Compiler
- ✓ Programs

### 3. Naming conventions in python

- ✓ What is an identifier?
- ✓ Why should we follow naming conventions?
- ✓ Rules to define identifiers in Python:
- ✓ Summary points about rules
- ✓ Validating identifier names by taking examples
- ✓ Python program total identifiers in simple table
- ✓ Smart suggestions while writing identifiers
- ✓ Indentation and Comments
- ✓ Programs

### 4. Variables in python

- ✓ What is variable?
- ✓ Properties of variable
- ✓ Creating variable
- ✓ Invalid cases for variables
- ✓ Multiple variables in single line
- ✓ Single value for multiple variables
- ✓ Variable re-initialization
- ✓ Programs

## 5. Data types in python

- ✓ What is data type?
- ✓ type() function
- ✓ Different types of data types
- ✓ Built-in data types
- ✓ Numeric types
- ✓ int data type
- ✓ float data type
- ✓ complex data type
- ✓ bool data type (boolean data type)
- ✓ None data type
- ✓ Sequences in Python
- ✓ str data type
- ✓ bytes data type
- ✓ bytearray data type
- ✓ list data structure
- ✓ tuple data structure
- ✓ range data type
- ✓ Accessing range values
- ✓ set data structure
- ✓ dictionary data structure
- ✓ User defined data types
- ✓ Converting from one data type into another data type.
- ✓ Programs

## 6. Operators

- ✓ What is an operator?
- ✓ Different type of operators
- ✓ Arithmetic Operators
- ✓ Assignment operator
- ✓ Unary minus operator
- ✓ Relational operators
- ✓ Logical operators
- ✓ Membership operators
- ✓ Identity operators
- ✓ Programs

## 7. Input and Output

- ✓ Why should we learn about input and output?
- ✓ Input and output
- ✓ Convert from string type into other type
- ✓ eval() function
- ✓ Command line arguments
- ✓ IndexError
- ✓ len() function
- ✓ Programs

## 8. Flow control

- ✓ Why should we learn about flow control?
- ✓ Flow control
- ✓ Sequential flow
- ✓ Conditional flow
- ✓ Looping flow
- ✓ Sequential statements
- ✓ Conditional or Decision-making statements
- ✓ if statement
- ✓ if else statement
- ✓ if elif else statement
- ✓ Looping
- ✓ while loop
- ✓ for loop
- ✓ Infinite loops
- ✓ Nested loops
- ✓ Loops with else block (or) else suit
- ✓ Where loops with else block flow is helpful?
- ✓ break statement
- ✓ continue statement
- ✓ pass statement
- ✓ return statement
- ✓ Programs

## 9. String in python

- ✓ Gentle reminder
- ✓ What is a string?
  - i. Definition 1
  - ii. Definition 2
- ✓ String is more popular
- ✓ Creating string
- ✓ When should we go for triple single and triple double quotes?
- ✓ Multi-line string objects
- ✓ Empty string
- ✓ Accessing string characters
- ✓ Indexing in string
- ✓ Python support two types of indexes

- ✓ Positive index
- ✓ Negative index
- ✓ Internal representation
- ✓ Slicing in string
- ✓ Internal representation
- ✓ Default values
- ✓ String several cases in slicing
- ✓ What is Immutable?
- ✓ Strings are immutable
- ✓ Mathematical operators on string objects
- ✓ Interesting about + and \* operators
- ✓ Addition (+) operator with string
- ✓ Multiplication (\*) operator with string
- ✓ Length of the string
- ✓ Membership operators
- ✓ Comparing strings
- ✓ Removing spaces from String
- ✓ Predefined methods to removes spaces
- ✓ Finding substrings
- ✓ index() method
- ✓ Counting substring in the given String
- ✓ Replacing a string with another string
- ✓ String objects are immutable, Is replace () method will modify the string objects?
- ✓ Splitting of Strings
- ✓ Joining of Strings
- ✓ Changing case of a String
- ✓ Formatting the Strings
- ✓ Character data type

## 10. Function

- ✓ General example why function required
- ✓ When should we go for function?
- ✓ What is Function?
- ✓ Advantage of function
- ✓ Types of function
- ✓ Pre-defined or built-in functions
- ✓ User Defined Functions:
- ✓ Function related terminology
- ✓ Function explanation
- ✓ Defining a function
- ✓ Calling a function
- ✓ Function without parameters
- ✓ Function with parameters
- ✓ return keyword in python
- ✓ return vs None
- ✓ Returning multiple values from function
- ✓ Function can call other function

- ✓ Understandings
- ✓ Functions are first class objects
- ✓ Assigning a function to variable
- ✓ Pass function as a parameter to another function.
- ✓ Define one function inside another function
- ✓ function can return another function
- ✓ Formal and actual arguments
- ✓ Types of arguments
- ✓ Positional arguments
- ✓ Keyword arguments
- ✓ Default arguments
- ✓ non-default argument follows default argument
- ✓ Variable length arguments
- ✓ keyword variable length argument (\*\*variable)
- ✓ Function vs Module vs Library:
- ✓ Types of variables
- ✓ Local variables
- ✓ Global variables
- ✓ The global keyword
- ✓ When we can choose global keyword?
- ✓ Recursive function
- ✓ Anonymous functions or Lambdas
- ✓ Advantage
- ✓ A simple difference between normal and lambda functions
- ✓ Where lambda function fits exactly?
- ✓ filter() function
- ✓ map() function
- ✓ reduce() function
- ✓ Function Aliasing
- ✓ Function decorators
- ✓ Decorator explanation
- ✓ Steps to create decorator
- ✓ @ symbol
- ✓ Function generators
- ✓ next(generator) function
- ✓ Programs

## 11. Modules

- ✓ What is module?
- ✓ import module
- ✓ Renaming or aliasing a module
- ✓ from and import keywords
- ✓ import \* (star symbol)
- ✓ member aliasing
- ✓ Reloading a Module:
- ✓ dir() function
- ✓ The Special variable \_\_name\_\_
- ✓ Programs

## 12. Package

- ✓ What is package?
- ✓ Advantage
- ✓ Programs

## 13. List Data structure

- ✓ Why should we learn about data structures?
- ✓ Python Data structures
- ✓ Sequence of elements
- ✓ Introduction about list data structure
- ✓ Creating list
- ✓ creating empty list
- ✓ Creating list with elements
- ✓ Creating list by using list() function
- ✓ Difference between list() function and list class
- ✓ list having mutable nature
- ✓ Accessing elements from list
- ✓ Accessing list by using index
- ✓ What is Indexing
- ✓ Accessing list elements by Slicing
- ✓ Accessing list by using loops
- ✓ Important functions and methods in list
- ✓ len() function
- ✓ count() method
- ✓ append method
- ✓ insert() method
- ✓ Difference between append and insert
- ✓ 5. extend() method:
- ✓ 6. remove() method
- ✓ 7. pop() method:
- ✓ Difference between remove() and pop() methods
- ✓ Ordering elements of List:
- ✓ reverse():
- ✓ sort() method:
- ✓ Aliasing and Cloning of list objects
- ✓ Aliasing list visualization
- ✓ Cloning or Copying
- ✓ By using slice operator
- ✓ By using copy method
- ✓ Mathematical + and \* operators
- ✓ Concatenation operator +
- ✓ Repetition operator \*
- ✓ Comparison of lists
- ✓ Membership operators
- ✓ Nested Lists
- ✓ list comprehensions

- ✓ Programs

## 14. Tuple data structure

- ✓ What is tuple data structure?
- ✓ When should we go for tuple data structure?
- ✓ Single value tuple
- ✓ Different ways to create a tuple.
- ✓ empty tuple
- ✓ Single value tuple
- ✓ Tuple with group of values
- ✓ By using tuple() function
- ✓ Accessing elements of tuple:
- ✓ Accessing tuple elements by using Index
- ✓ Accessing tuple elements by using slice operator:
- ✓ Tuple vs immutability:
- ✓ Mathematical operators on tuple:
- ✓ Concatenation operator (+):
- ✓ Multiplication operator (\*)
- ✓ Important functions and methods of Tuple:
- ✓ len() function
- ✓ count() method
- ✓ index() method
- ✓ sorted() function
- ✓ min() and max() functions:
- ✓ Tuple packing
- ✓ Tuple unpacking:
- ✓ Tuple comprehension
- ✓ Differences between List and Tuple:
- ✓ Programs

## 15. Set Data structure

- ✓ What is set data structure?
- ✓ What symbol is required to create set?
- ✓ Set elements separated by what?
- ✓ Creating set by using same type of elements
- ✓ Creating set by using different type of elements
- ✓ Creating set by range() type of elements
- ✓ Difference between set() function and set class
- ✓ Empty set
- ✓ Important function and methods of set:
- ✓ add(x) method
- ✓ update(x, y) method
- ✓ Difference between add() and update() methods in set?
- ✓ Which of the following are valid for set s?
- ✓ copy() method
- ✓ pop() method
- ✓ remove(x) method

- ✓ discard(x) method
- ✓ clear() method
- ✓ Mathematical operations on the Set
- ✓ union() method
- ✓ intersection() method
- ✓ difference() method
- ✓ symmetric\_difference():
- ✓ Membership operators: (in and not in)
- ✓ Set Comprehension
- ✓ Remove duplicates in list elements
- ✓ Frozen set
- ✓ Creating frozen set
- ✓ Programs

## 16. Dictionary data structure

- ✓ What is dictionary data structure?
- ✓ What symbol is required to create set?
- ✓ Create dictionary
- ✓ Empty dictionary
- ✓ Access values by using keys from dictionary
- ✓ How to handle this KeyError?
- ✓ Update dictionary
- ✓ Removing or deleting elements from dictionary
- ✓ By using del keyword
- ✓ We can delete total dictionary object
- ✓ clear() method
- ✓ Important functions and methods of dictionary
- ✓ dict() function
- ✓ dict({key1:value1, key2:value2}) function
- ✓ dict([tuple1, tuple2])
- ✓ len() function
- ✓ clear() method
- ✓ get() method
- ✓ pop() method
- ✓ popitem() method
- ✓ keys() method
- ✓ values() method
- ✓ items() method
- ✓ copy() method
- ✓ Dictionary Comprehension
- ✓ Programs

## 17. OOPS Part 1 – class, object, variables, methods etc

- ✓ Before OOPS
- ✓ Limitations
- ✓ After OOPS
- ✓ Advantages

- ✓ Is Python follows Functional approach or Object-oriented approach
- ✓ OOPS (Object Oriented Programming Principles)
- ✓ Features of Object-Oriented Programming System
- ✓ class
  - i. Definition 1
  - ii. Definition 2
- ✓ How to define a class
- ✓ Brief discussion about class
- ✓ self variable
- ✓ object
- ✓ Why should we create an object?
- ✓ What is an object?
  - i. Definition 1
  - ii. Definition 2
  - iii. Definition 3
  - iv. Definition 4
- ✓ Syntax to create an object
- ✓ Calling instance data
- ✓ Can we create more than one object?
- ✓ Constructor
  - ✓ What is the main purpose of constructor?
  - ✓ When constructor will be executed?
  - ✓ How many times constructor will be executed?
  - ✓ Is constructor mandatory to define?
  - ✓ Can I call constructor explicitly like a method?
  - ✓ How many times constructor will be executed?
  - ✓ Constructor can contain how many parameters?
  - ✓ If constructor having no parameters, then how to define?
  - ✓ If trying to print self variable
  - ✓ If constructor having more parameters, then how to define?
  - ✓ Difference between methods and constructor
- ✓ Types of Variables:
  - ✓ 1. Instance variables:
  - ✓ What is instance variable?
  - ✓ Separate copy for every object
  - ✓ Where should we declare instance variable?
  - ✓ By using constructor
  - ✓ The \_\_dict\_\_ attribute
  - ✓ By using instance method
  - ✓ By using object name
  - ✓ Accessing instance variable
  - ✓ By using self variable
  - ✓ By using object name
  - ✓ Every object has a separate copy of variable exists
  - ✓ static variable (class level variable)
  - ✓ What is static variable
  - ✓ Where can we declare static variable?

- ✓ Only one copy of static variable to all objects
- ✓ How can we access static variable?
- ✓ Instance Variable vs Static Variable
- ✓ Declaring static variable
- ✓ Inside class and outside of the method
- ✓ Inside constructor
- ✓ Inside instance method
- ✓ Inside classmethod
- ✓ class name
- ✓ cls variable
- ✓ Inside static method
- ✓ @staticmethod declarator
- ✓ Accessing static variable
- ✓ Inside constructor
- ✓ Inside instance method
- ✓ Inside classmethod
- ✓ Inside staticmethod
- ✓ Local variable
- ✓ What is local variable
- ✓ Why we need to use local variable
- ✓ Where can we access local variable
- ✓ Types of methods
- ✓ Instance methods
- ✓ What is the use of Setter and Getter Methods
- ✓ Setter method
- ✓ Getter Method
- ✓ Class Methods
- ✓ What is class method
- ✓ When we can go to class methods
- ✓ @classmethod decorator
- ✓ How to access class methods
- ✓ static Methods
- ✓ How to declare static method
- ✓ How to access static methods
- ✓ Passing members of one class to another class
- ✓ Inner classes
- ✓ Garbage Collection
- ✓ What is the main objective of Garbage Collector
- ✓ How to enable and disable Garbage Collector in our program:
- ✓ Importance functions in gc module
- ✓ gc.isenabled()
- ✓ gc.disable()
- ✓ gc.enable()
- ✓ Programs

## 18. OOPS Part 2 – Inheritance

- ✓ What is inheritance
- ✓ Conclusion of the story

- ✓ Advantage
- ✓ How to implement inheritance
- ✓ Types of Inheritance:
- ✓ Single Inheritance
- ✓ Multi-level Inheritance
- ✓ Multiple inheritance
- ✓ Parent classes can contain a method with same name
- ✓ If parent classes contain a method with same name, then which method will access by child class
- ✓ Scenarios
- ✓ Constructors in Inheritance
- ✓ If child class and super class both having constructors, then?
- ✓ Calling super class constructor from child class constructors
- ✓ Method Resolution Order (MRO)
- ✓ There are three principles followed by MRO
- ✓ Why should we understand MRO?
- ✓ Is there any predefined method to check sequence of execution of classes?
- ✓ Demo program 1 for method resolution order
- ✓ Demo Program-2 for Method Resolution Order
- ✓ super() function
- ✓ Which scenario we can use super() function in child class to call super class members?
- ✓ Calling method of a specific super class
- ✓ Different cases for super() function
- ✓ Programs

## 19. OOPS Part 3 – Polymorphism

- ✓ What is Polymorphism
- ✓ Duck Typing Philosophy of Python
- ✓ Overloading
- ✓ What is overloading
- ✓ Operator Overloading:
  - ✓ Is python supports operator overloading?
  - ✓ How operator overloading works in python?
- ✓ Magic methods
  - ✓ List of operators and corresponding magic methods.
  - ✓ Method Overloading
- ✓ How we can handle overloaded method requirements in Python
- ✓ Constructor Overloading
- ✓ Constructor with Default Arguments
- ✓ Constructor with Variable Number of Arguments:
- ✓ Method overriding:
  - ✓ Demo Program for Method overriding
  - ✓ Constructor overriding
- ✓ Calling parent class constructor from child class constructor
- ✓ Programs

## 20. OOPS Part 4 – abstract class, interface etc

- ✓ In python which things we can make as an abstract?
- ✓ There are two types of methods in-terms of implementation
- ✓ Implemented method
- ✓ Un-implemented method
- ✓ So, how to declare abstract method
- ✓ abstract method
- ✓ abstract class
- ✓ If child class missed to provide abstract methods implementation, then
- ✓ abstract class can contain concrete methods also
- ✓ Can abstract class contains more sub classes?
- ✓ On which scenario an abstract class contains more than one child classes?
- ✓ Abstract class object creation
- ✓ Different cases and scenarios for abstract class object creation
- ✓ Interface
- ✓ What is an interface?
- ✓ When should we go for ...?
  - i. Interface
  - ii. Abstract class
  - iii. Normal class or concrete class
- ✓ Double underscore (Name mangling)
- ✓ `__str__(self)` method
- ✓ Programs

## 21. Exceptional handling

- ✓ Syntax Errors
- ✓ Programmer responsible
- ✓ Runtime Errors
- ✓ Scenarios where runtime errors will occur?
- ✓ Normal flow of the execution
- ✓ Abnormal flow of the execution
- ✓ What we need to do if program terminates abnormally
- ✓ What is an Exception?
- ✓ Is it really required to handle the exceptions?
- ✓ What is the meaning of exception handling?
- ✓ Default Exception Handling in Python:
- ✓ Exception hierarchy diagram
- ✓ Programmer focus
- ✓ Handling exceptions by using try except
  - i. try block
  - ii. except block
- ✓ try-except flow
- ✓ Control flow in try and except
  - i. 7 cases and scenarios with examples

- ✓ Printing exception information
- ✓ try with multiple except blocks
- ✓ Single except block can handle multiple exceptions
- ✓ Default except block
- ✓ finally block
- ✓ Why separate block for clean-up activities, can't we write inside try and except blocks
- ✓ Coming to the try block
- ✓ Coming to the except block
- ✓ What is the speciality of final block?
- ✓ Control flow about finally block
  - i. few cases and scenarios with examples
- ✓ Any situation like, finally will not execute?
- ✓ Control flow in try-except-finally
  - i. 6 cases and scenarios with examples
- ✓ Nested try-except-finally blocks
  - i. 4 cases and scenarios with examples
- ✓ else block with try-except-finally
- ✓ At what time which block
- ✓ Various possible combinations of try-except-else-finally
- ✓ Types of Exceptions
- ✓ Predefined Exceptions
- ✓ User Defined Exceptions
- ✓ User defined exceptions or Custom Exceptions
- ✓ Steps to follow to Define and Raise Customized Exceptions
- ✓ Programs

## 22. File IO

- ✓ What is a file
- ✓ Types of Files
- ✓ Text files
- ✓ Binary Files
- ✓ file modes and their meanings
- ✓ Opening a File
- ✓ Various properties of File Object
- ✓ Writing data to text files
- ✓ Instead of overriding the content, how to append the content to the file
- ✓ writelines(argument)
- ✓ Reading Character Data from text files
- ✓ readline() method
- ✓ readlines() method
- ✓ with keyword
- ✓ Advantage
- ✓ The seek() and tell() methods
- ✓ How to check a specific file exists or not?

- ✓ sys.exit(0)
- ✓ Program to print the number of lines, words and characters present in the given file?
- ✓ Working with Binary data
- ✓ Working with csv files
- ✓ Zipping and Unzipping Files
- ✓ To create Zip file
- ✓ To perform unzip operation
- ✓ Working with Directories
- ✓ To know contents of directory
- ✓ Pickling and Unpickling of Objects
- ✓ Programs

## 23. Database connectivity

- ✓ Data Storage Areas
- ✓ Temporary Storage Areas
- ✓ Permanent Storage Areas
- ✓ File Systems
- ✓ Databases
- ✓ Python Database Programming
- ✓ Standard Steps for Python database Programming
- ✓ import database specific module
- ✓ Establish Connection.
- ✓ Create Cursor object
- ✓ In-built methods to execute the sql queries
- ✓ commit or rollback
- ✓ Fetch the result from the Cursor
- ✓ close the resources
- ✓ Important methods while working with Database
  - i. connect()
  - ii. cursor()
  - iii. execute()
  - iv. executescript()
  - v. executemany()
  - vi. commit()
  - vii. rollback()
  - viii. fetchone()
  - ix. fetchall()
  - x. fetchmany(n)
  - xi. close()
- ✓ Working with Oracle Database
- ✓ Installing cx\_Oracle:
- ✓ How to Test Installation
- ✓ Expected common errors while executing programs
- ✓ Programs

## 24. Regular expressions

- ✓ What is Regular expression?
- ✓ When should we chose?
- ✓ The main important application areas of Regular Expressions are
  - ✓ compile()
  - ✓ finditer()
- ✓ On Match object we can call start(), end() and group() methods methods
- ✓ Character classes
- ✓ Pre-defined Character classes:
- ✓ Quantifiers
- ✓ Important functions of re module
  - i. match()
  - ii. fullmatch()
  - iii. search()
  - iv..findall()
  - v. finditer()
  - vi. sub()
  - vii. subn()
  - viii. split()
  - ix. compile()
- ✓ ^ symbol
- ✓ \$ symbol
- ✓ Programs

## 25. Multithreading

- ✓ Multi-Tasking
- ✓ Process based Multi-Tasking
- ✓ Where multi-tasking fits?
- ✓ Thread based Multi-tasking
- ✓ Where multi-tasking fits
- ✓ Areas of multi-threading
- ✓ The ways of Creating Thread in Python
- ✓ Creating a Thread without using any class
- ✓ Creating a Thread by inheriting Thread class
- ✓ Creating a Thread without inheriting Thread class
- ✓ Without multi-threading
- ✓ With multithreading
- ✓ Setting and Getting Name of a Thread:
- ✓ How to get and set name of the thread
- ✓ Thread Identification Number (ident):
  - ✓ active\_count()
  - ✓ enumerate() function
  - ✓ isAlive()
  - ✓ join() method
  - ✓ join(seconds) method
  - ✓ Daemon Threads
  - ✓ Example: Garbage Collector

- ✓ setDaemon() method
- ✓ Default Nature
- ✓ Main Thread is always Non-Daemon
- ✓ Scenarios from previous program
- ✓ Synchronization
  - ✓ How to overcome data inconsistency problems?
  - ✓ How to implement synchronization
    - i. Lock
    - ii. RLock
    - iii. Semaphore
- ✓ Synchronization By using Lock concept:
  - ✓ acquire() method
  - ✓ release() method
  - ✓ Lock()
  - ✓ Problem with Simple Lock
  - ✓ RLock()
  - ✓ Demo Program for synchronization by using RLock:
  - ✓ Difference between Lock and RLock
  - ✓ Synchronization by using Semaphore
  - ✓ How to create Semaphore object?
  - ✓ BoundedSemaphore:
    - ✓ Difference between Lock and Semaphore
    - ✓ Inter Thread Communication
    - ✓ Interthread communication by using Event Objects
    - ✓ Methods of Event class
    - ✓ set()
    - ✓ clear()
    - ✓ isSet()
    - ✓ wait() | wait(seconds)
  - ✓ Interthread communication by using Condition Object
  - ✓ Methods of Condition
  - ✓ Interthread communication by using Queue:
    - ✓ Important Methods of Queue
    - ✓ Types of Queues
      - i. FIFO Queue
      - ii. LIFO Queue
      - iii. Proprietary Queue
  - ✓ Programs

## 26. Logging module

- ✓ Why Logging is required?
- ✓ What is Logging
- ✓ The main advantages of logging
- ✓ Can we use print statement for debugging?
- ✓ logging module
- ✓ How to implement Logging

- ✓ How to configure log file in over writing mode:
- ✓ How to Format log messages:
- ✓ To display only level name:
- ✓ To displaylevelname and message
- ✓ Add timestamp in the log messages:
- ✓ change date and time format
- ✓ Writing Python program exceptions to the log file
- ✓ Problems with root logger
- ✓ Need of Our own customized logger:
- ✓ Advanced logging Module Features
- ✓ Logger
- ✓ Steps for Advanced Logging
- ✓ Demo Program for File Handler:
- ✓ Programs

## 27. Python + Spark (Advance to Hadoop) = PySpark

- ✓ What is spark?
- ✓ Why spark?
- ✓ Spark set up
- ✓ Spark programs by using python
- ✓ Spark core explanation
- ✓ Spark SQL explanation
- ✓ Spark core module with Programs
- ✓ Spark SQL modules with Programs

## 28. Data Science

- ✓ What is Data science?
- ✓ Where python is using in Data science?
- ✓ Data science programs by using python

## 29. IDEs explanation

- ✓ IntelliJ
- ✓ PyCharm

## 1. Python Introduction

### 1. What is Python?

- ✓ Python is a **general purpose** and **high-level programming** language.
  - All companies are using python programming language to develop the applications, testing and maintenance etc.
  - There are mainly two types of programming languages in this world,
    - High level
      - Human readable language.
      - Easy to understand
    - Low level
      - Machine readable language like bits (1's and 0's form)

### 2. Where are all python is using in application level?

To develop,

- ✓ Standalone applications
  - An application which needs to install on every machine to work with that application.
- ✓ Web applications
  - An application which follows **client-server architecture**.
  - Client is a program, which sends request to the server.
  - Server is a program, mainly it can do three things,
    - Captures the request from client
    - Process the request
    - Sends the response to the client
- ✓ Database applications.
- ✓ To process huge amount of data.
  - Hadoop
  - Spark.
- ✓ Machine learning.
- ✓ Artificial Intelligence.
- ✓ Data science.
- ✓ Network servers.
- ✓ IOT
- ✓ Application scripting etc.

### 3. History of Python?

- ✓ Python was created by Guido Van Rossum in the year of 1989.
- ✓ It is open source software means we can download freely and customise the code as well.

### 4. Why the name was python?

- ✓ A TV show Monty Python's Flying Circus was very much popular fun show in 1970's.
- ✓ So, Guido like this show and given this name to his programming language.

### 5. Python versions

- ✓ First version was released in Feb 20<sup>th</sup>, 1991.
- ✓ Current version 3.6.x release Dec 23<sup>rd</sup>, 2016.

### 6. Python supports

- ✓ Functional programming.
- ✓ Object oriented programming approach
- ✓ Now days to fulfil the requirement both are required.

Python = Functional programming + Object oriented programming

### 7. Which companies are using Python?

- ✓ Currently all companies are using the python.

### 8. To solve any requirement,

- ✓ Initial languages like C, Pascal or FORTRAIN follows functional approach.
- ✓ C++, Java and dot net follows object-oriented approach.
- ✓ Python follows both **functional** and **object-oriented approaches**

## DVS Technologies

---

Requirement 1 : Java programs to print, Welcome to Java programming  
Java program : HelloWorld.java

Requirement	Program prints Welcome to Java programming
output	Welcome to Java programming

```
class HelloWorld
{
    public static void main (String args [])
    {
        System.out.println("Welcome to Java programming");
    }
}
```

Output	Welcome to Java programming
--------	-----------------------------

- ✓ To write java code at least you should write a single class.

C program : HelloWorld.c

Requirement	Program to print, Welcome to C programming
output	Welcome to C programming

```
#include<stdio.h>
{
    void main ()
    {
        printf("Welcome to C programming");
    }
}
```

Output	Welcome to C programming
--------	--------------------------

- ✓ To write C code at least you should write a function

Python program : HelloWorld.py

Requirement	Program to print, Welcome to Python programming
output	Welcome to Python programming

```
print("Welcome to Python programming")
```

Output	Welcome to Python programming
--------	-------------------------------

- ✓ Above python program is simple approach and more understandable even for nursery student also.

**Requirement 2:** Take two numbers and apply addition logic.

- ✓ So, above requirement we are going to see by using Java, C and python

**Java program :** Addition.java

Requirement	Program prints to add two number in Java
output	30

```
class Addition
{
    public static void main (String args[])
    {
        int a = 10;
        int b = 20;

        System.out.println(a+b);
    }
}
```

Output	30
--------	----

- ✓ To write java code at least you should write a single class to express logic.
- ✓ Sometimes this makes program lengthy and consumes more time.

**C program :** Addition.c

Requirement	Program prints to add two number in Java
output	30

```
#include<stdio.h>
{
    void main()
    {
        int a = 10;
        int b = 20;
```

```
        println(a+b);  
    }  
}
```

Output	30
--------	----

- ✓ To write C code at least you should write a function to express logic.
- ✓ Sometimes this also makes program lengthy and consumes more time.

Python program : addition.py

Requirement	Program prints to add two number in Java
output	30

a = 10 b = 20  print(a+b)	Internal representation  30  Frames Global frame a   10 b   20
------------------------------------	---

Output	30
--------	----

- ✓ Above python code is simple approach and more understandable even for non-techy also.

Make a note:

- ✓ Semi colons are mandatory in C and Java programming languages.
- ✓ In python semi colons are not required.

Points to make a note

- ✓ In many programming languages to write a program,
  - We need to create the structure with symbols in a certain order.
  - Next step is, we need to write logic.
  - In python we can see very informative code by writing simple way.
- ✓ In Python programming a single line, which prints a short message.
- ✓ It's a very informative example of Python's syntax.

## 9. Machine language

- ✓ Representing the instructions and data in the form of *bits* (1's and 0's) is called machine code or machine language.
- ✓ Example to add two numbers then machine will convert these number into bits by division with 2.

Ex :  $12 + 14 = 26$

2		12		
		6	-	0      Reminder
		3	-	0      Reminder
		1	-	1      Reminder

- ✓  $12 == 1100$  Take the digits from bottom to top digits

2		14		
		7	-	0      Reminder
		3	-	1      Reminder
		1	-	1      Reminder

- ✓  $14 == 1110$  Take the digits from bottom to top digits

- ✓ Internally these bit values will be adding and generate the sum result as 26

## 1.9 Translator

- ✓ A Translator is a program that converts any computer programs into machine code.
- ✓ There are 'n' number of translators are existing but for us we need to understand 2 types of translators.

### 1. Interpreter

- ✓ Interpreter is a program, it can convert the program by **line by line**.

### 2. Compiler

- ✓ Compiler is a program converts the entire program in a **single step**.

## 10. Python Reserved keywords words (33)

- ✓ The words which are reserved to do specific functionality is called reserved keywords words.

	1 Flow control	2 Exception handling	3 class	4 function and method	5 variable	6 object	7 module	8 check	9 Build-in constants	10 logical
1	if	try	class	def	global	del	import	is	True	and
2	else	except		with	nonlocal		from	in	False	or
3	elif	raise		lambda			as		None	not
4	for	finally								
5	while	assert								
6	break									
7	continue									
8	return									
9	yield									
10	pass									
	10	5	1	3	2	1	3	2	3	3

Make a note:

- ✓ All keywords in Python contain only alphabet symbols.
- ✓ All keywords are in lowercase except 3 those are,
  - True
  - False
  - None

### 10.1 To see all these reserved words from python shell

```
>>> import keyword
>>> keyword.kwlist

['False', 'None', 'True', 'and', 'as', 'assert', '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']
```

Make a note:

- ✓ Program implementing by using notepad and executing command prompt is best practice

<b>Program Name</b>	To see all python keywords demo1.py
	<code>import keyword print(keyword.kwlist)</code>
<b>Output</b>	<code>['False', 'None', 'True', 'and', 'as', 'assert', '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']</code>

Note:

- ✓ Keyword is a module in python

## 11. Features of python

### 1. Simple

- ✓ Python syntax is very easy.
- ✓ Developing and understanding python is very easy than other.

### 2. Open source

- ✓ We can download freely and customise the code as well

### 3. High level language

- ✓ There are two types of languages,
  - Low level
    - Machine code instructions, difficult to learn programmer.
  - High level
    - English words with syntax, programmer can learn easily.

### 4. Dynamically typed

- ✓ Dynamically type will be assigned to data.

### 5. Platform independent

- ✓ Python programs are not dependent on any specific operating systems.
- ✓ We can run on all operating systems happily.

### 6. Portable

- ✓ If a program gives same result on any platform then it is a portable program.
- ✓ Python used to give same result on any platform.

### 7. Procedure and object oriented

- ✓ Python supports both procedural and object-oriented features.

### 8. Huge library

- ✓ Python has a big library to fulfil the requirements.

## 9. Database connectivity

- ✓ Python provides interfaces to connect with all major databases like, oracle, MySQL

## 10. Batteries included

- ✓ Python provides in built libraries called batteries,
- ✓ Some of them are below,
  - Boto
    - amazon web services library
  - MySQL -connector-python
    - To connect with MySQL
  - NumPy
    - To process arrays
  - Pandas
    - powerful data structures for data analysis, time series and statistics
  - so many others...

## 12. Different flavours of python

- ✓ Flavours of python refers to the different types of python compilers.
- ✓ These are useful to integrate various programming languages.

### 1. CPython

- ✓ Standard python compiler implemented in C language.
- ✓ This is the python software being downloaded and used by the programmers directly.

### 2. Jython

- ✓ Initially called as JPython, later renamed to Jython.
- ✓ Designed to run on Java program.

### 3. IronPython

- ✓ Designed for .NET framework.

### 4. PyPy

- ✓ The main advantage of PyPy is performance will be improved because JIT compiler is available inside PVM.

### 5. RubyPython

- ✓ This is a bridge between Ruby and python interpreters.
- ✓ It encloses a python interpreter inside Ruby application.

### 6. AnacondaPython

- ✓ Anaconda is a free and open source Python programming languages.
- ✓ This is mainly for data science and machine learning related applications (large-scale data processing, predictive analytics, scientific computing).
- ✓ This aims to simplify package management and deployment.

## 13. Python Versions

- ✓ Python 1.0V introduced in Jan 1994
- ✓ Python 2.0V introduced in October 2000
- ✓ Python 3.0V introduced in December 2008.
- ✓ Current version is 3.6

### Make a note

- ✓ Python 3 won't provide backward compatibility to Python2 i.e. there is no guarantee that Python2 programs will run in Python3.

## 2. Python coding introduction

### 2.1 Ways to write python program

- ✓ We can write the python programs in two ways.
  - ✓ By using any text editor like Notepad++, Edit plus.
  - ✓ We can also write python programs by using python IDLE(Integrated Development Environment)

### 2.2 Python program execution steps

- ✓ We need to **write** python programs in notepad (good approach for practice).
- ✓ We can **save** the program with **.py** or **.python** extension.
- ✓ **Run** or execute the program.
- ✓ Finally, we will get **output**.

<b>Program Name</b>	Print Welcome to python world message demo.py
	print("Welcome to python world")
<b>Run</b>	python demo.py
or	
<b>Run</b>	py demo.py
<b>output</b>	Welcome to python world

### 2.7 Understanding first python program

- ✓ `print("Welcome to python world")`,
  - `print()` is a predefined function.
  - `print()` function takes string values as parameters.
- ✓ This function prints the output on the console.

### 2.6 Execute first Python program by python IDLE or command prompt

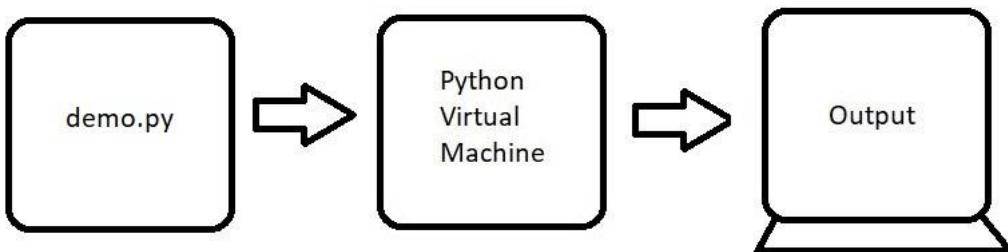
- ✓ Open python IDLE
- ✓ Then write directly like, `print("Welcome to python world")`

## Example

```
>>>print ("Welcome to python world")
Welcome to python world
```

## 2.3 Python program flow

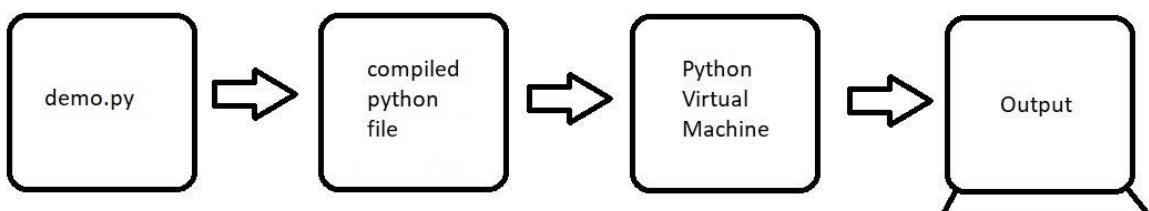
### Short answer



- ✓ Python program (source code), we need to save with .py (dot py) or .python (dot python) extension.
  - Example : **demo.py**
- ✓ We can run this program by using **python** command.
  - While running the program, internally automatic compilation will be done.
  - If everything perfect as per compiler checking, then goes to next level to execute the program.
  - Example : **python demo.py**
- ✓ We can see the output on the console.

## 2.3 Python program flow

- ✓ Long answer but more helpful for internal understandings...
- ✓ This answer mainly contains three parts,
  1. Write and Run the program
  2. Compiled python file
  3. Python Virtual machine



## 1. Write and run the program

- ✓ Python program (source code), we need to save with .py (dot py) or .python (dot python) extension.

○ Example : `demo.py`

- ✓ Next step is, we need to run this program (`demo.py`).

## 2. Compiled python file

- ✓ While running the program, the first step is, internally python compiler takes this source code and creates corresponding **compiled python file** to source code.

- This compiled python file is not visible.
- It is a hidden file, stored in cache memory.
- If we want to see this file then run this below command,

○ `python -m py_compile demo.py`

- ✓ -m means module here `py_compile` is module name.
- ✓ This module creates compiled python file.
- ✓ So, python creates separate folder in the current directory by the name `_pycache_` to store compiled python file.

## 3. Python Virtual Machine

- ✓ This compiled python file contains **byte code** instructions.
- ✓ These byte code instructions are **not understandable** by the microprocessor to generate output.
- ✓ While running the program, **Python Virtual Machine** takes responsible to convert these byte code instructions into machine understandable format

○ Example : `python demo.py`

- ✓ Finally, we will see the output

## JIT compiler (Just In Time)

- ✓ Python Virtual Machine internally uses interpreter which converts line by line, so it is very slow.
- ✓ To resolve this problem few python flavours (PyPy) uses compiler which converts very faster than interpreter.
- ✓ This compiler is called JIT (Just In Time) compiler.
- ✓ It improves the speed and execution of a python program

## 3. Naming conventions

What is identifier?

- ✓ A name in a python program is called **identifier**.
- ✓ This name can be,
  - class name
  - package name
  - variable name
  - module name
  - function name
  - method name
- ✓ Python developers made some suggestions to the programmers regarding how to write identifiers in program.

Why should we follow naming conventions?

- ✓ If we follow the naming conventions, then the written code is,
  - Easy to understand.
  - Easy to read.
  - Easy to debug.

Rules to define identifiers in Python:

1. The only allowed characters to write identifier in python are,
  - **Alphabets**, these can be either lower case or upper case.
  - Digits (0 to 9)
  - Underscore symbol (\_)
- ✓ If we are using any other symbol like ( \$, !, -, etc) then we will get syntax error.

<b>Program Name</b>	printing variable name demo1.py
	student_id=101 print(student_id)
<b>Output</b>	101

**Program Name** printing variable name which having symbols in name  
demo2.py

```
$student_id=101  
print($student_id)
```

**Error** **SyntaxError:** invalid syntax

- Identifier allowed digits, but identifier should not start with digit.

**Program Name** printing variable name which having digits in end of the name  
demo3.py

```
student_id123=101  
print(student_id123)
```

**Output**  
101

**Program Name** printing variable name which starts with digits, it is invalid  
demo4.py

```
123student_id=101  
print(123student_id)
```

**Error** **SyntaxError:** invalid syntax

- Identifiers are case sensitive.

**Program Name** To prove python is case sensitive  
demo5.py

```
a = 10  
b = 20  
print(A + B)
```

**Error** **NameError:** name 'A' is not defined

4. We cannot use keywords as identifiers.

<b>Program Name</b>	printing variable name given as keyword name, it is invalid demo6.py
<b>Error</b>	<b>SyntaxError:</b> invalid syntax

5. Spaces are not allowed between identifier.

<b>Program Name</b>	spaces not allowed between identifier demo7.py
<b>Error</b>	<b>SyntaxError:</b> invalid syntax

## Summary points

1. The only allowed characters to write identifier in python are,
  - **Alphabets**, these can be either lower case or upper case.
  - Digits (0 to 9)
  - Underscore symbol (\_)
- ✓ If we are using any other symbol like ( \$, !, -, etc) then we will get syntax error.
2. Identifier allowed digits, but identifier should not start with digit.
3. Identifiers are case sensitive.
4. We cannot use keywords as identifiers.
5. Spaces are not allowed between identifier.

## Validate the below identifiers

- ✓ 435student # invalid
- ✓ student564 # valid
- ✓ student565info # valid
- ✓ \$student # invalid
- ✓ \_student\_info # valid
- ✓ class # invalid
- ✓ def # invalid

## Python program identifiers

1. class	<ul style="list-style-type: none"> <li>✓ class names should start with upper case and remaining letters are in lower case.</li> <li>✓ If name having multiple words, then every inner word should start with upper case letter.</li> <li>✓ <b>Example:</b> StudentInfo</li> <li>✓ <b>Info:</b> This rule is applicable for classes created by users only; the <b>in-built</b> class names used all are in lower-case.</li> </ul>
2. package 3. module 4. variable 5. function 6. method	<ul style="list-style-type: none"> <li>✓ Names should be in lower case.</li> <li>✓ If name having multiple words, then separating words with underscore (_) is good practice.</li> <li>✓ <b>Example:</b> one_two</li> </ul>
7. Non-public instance variables	<ul style="list-style-type: none"> <li>✓ Non-public instance variables should begin with underscore (_), we can say private data</li> <li>✓ <b>Example:</b> _one</li> </ul>
8. constants	<ul style="list-style-type: none"> <li>✓ Constants names should be written in all capital letters.</li> <li>✓ If name having multiple words, then separating words with underscore (_) is good practice.</li> <li>✓ <b>Example:</b> ONE_TWO</li> </ul>
9. Non-accessible entities	<ul style="list-style-type: none"> <li>✓ Some variables and functions not accessible outside.</li> <li>✓ Those variables and function names started with two underscores symbols.</li> <li>✓ <b>Example:</b> __init__(self)</li> </ul>

## Smart suggestions while writing identifiers

- ✓ Be descriptive for identifiers,
  - A variable name should describe exactly what it contains.

### Example

a = 10	#	valid but not recommended
student_id = 10	#	valid and highly recommended

- A function name should describe what exactly what kind of operation is performing.

### Example

abc()	#	valid but not recommended
sum()	#	valid and highly recommended

- ✓ Don't use abbreviations unnecessarily.
  - Abbreviations may be ambiguous and more difficult to read.

### Example

fourth_bentch_middle_student_id = 10	#valid but not recommended
student_id = 10	#valid and highly recommended

## Indentation and Comments

### Indentation

- ✓ In Python, we need to group the statements by using indentation.

### Why indentation

- ✓ Indentation keeps separate the group of statements.
- ✓ The recommended indentation is 4 spaces.
- ✓ We must follow the order of indentation otherwise we will get **IndentationError**

<b>Program Name</b>	valid indentation demo8.py
	print("statement one") print("statement two") print("statement three")

## output

```
statement one  
statement two  
statement three
```

## Program Name

Invalid indentation  
demo9.py

```
print("statement one")  
    print("statement two")  
print("statement three")
```

## Error

IndentationError: unexpected indent

## Program Name

Invalid indentation  
demo10.py

```
print("statement one")  
    print("statement two")  
        print("statement three")
```

## Error

IndentationError: unexpected indent

## Comments

- ✓ Comments are useful to describe about the code in an easy way.
- ✓ Python ignores comments while running the program.
- ✓ To comment python code, we can use hash symbol #

## Program Name

showing comments in program  
demo11.py

```
# This is Basic program in python  
print("Welcome to python programming")
```

## output

## 4. Variables

### 4. 1 variable

- ✓ A variable is a,
  - name.
  - refers to a value.
  - holds the data
  - name of the memory location.

Program Name	printing variable demo1.py
Output	emp_id=101 print(emp_id)  101

### 4.2 Properties of variable

- ✓ Every variable has a,
  - Type
  - Value
  - Scope
  - Location
  - Life time

### 4.3 Creating variable

- ✓ In python, to create a variable we need to specify,
  - The name of the variable
  - Assign a value to name of the variable.

Syntax	name_of_the_variable = value
--------	------------------------------

Program Name	creating a variable demo2.py
output	age=16 print(age)  16

**Program Name** creating a variable and displaying with meaningful text message  
demo3.py

```
age=16  
print("My age is sweet: ", age)
```

**output**

My age is sweet: 16

## Make a Note

- ✓ We can print meaningful text message along with variable for better understanding
  - Text message we should keep in within double quotes.
  - Text message and variable name should be separated by comma symbol.

Above example:

age	----->	name of the variable
50	----->	value or literal

## 4. 4 Invalid cases for variables

1. While defining a variable,
  - Variable name should be written in left hand side.
  - Value should be written in right hand side
  - Otherwise we will get syntax error.

**Program Name** Creating variable in wrong direction  
demo4.py

```
10 = emp_id  
print(emp_id)
```

**Error** **SyntaxError:** can't assign to literal

## 2. variables names,

- o should not give as keywords names, otherwise we will get error

**Program Name** printing variable name which having digits in name  
demo5.py

```
if = 10  
print(if)
```

**Error** **SyntaxError:** invalid syntax

## 4.5 Multiple variables in single line

1. We can assign multiple variables to multiple values in **single one line**.
2. **Rule**
  - a. While defining there should be same number on the left and right-hand sides. Otherwise we will get error.

**Program Name** assign multiple variables in single line  
demo6.py

```
a, b, c = 1, 2, 3  
print(a, b, c)
```

**output**

```
1 2 3
```

**Program Name** Invalid case: assign multiple variables which are not matched names and values  
demo7.py

```
a, b, c = 1, 2  
print(a, b, c)
```

**Error**

**ValueError:** not enough values to unpack (expected 3, got 2)

Program Name	Invalid case: assign multiple variables which are not matched names and values demo8.py
	a, b = 1, 2, 3 print(a, b, c)
Error	<b>ValueError:</b> too many values to unpack (expected 2)

## 4.6 Single value for multiple variables

- ✓ We can assign a single value to several variables simultaneously.

Program Name	Assign single value to multiple variables demo9.py
	a = b = c = 1 print(a, b, c)
Output	1 1 1

## 4.7 Variable re-initialization

- ✓ We can reinitialize variable values.
- ✓ **Re-initialize**
  - In variable re-initialization old values will be replaced or overridden with new values.

Program Name	Re-initialising variable demo10.py
	a = 10 print("first time value :", a)
	a = 20 print("variable is re-initialised, now value is : ", a)
	a = 30 print("again variable is re-initialised, now value is : ", a)

### output

```
first time value : 10
variable is re-initialised, now value is : 20
again variable is re-initialised, now value is : 30
```

## 5. Data types

What is a data type?

- ✓ A data type represents the type of the data stored into a variable or memory.

<b>Program Name</b>	print different kinds of variables demo1.py
	emp_id=1 name="Nireekshan" salary=100.50
	print("My employee id is: ", emp_id) print("My name is: ", name) print("My salary is: ", salary)
<b>Output</b>	My employee id is: 1 My name is: Nireekshan My salary is: 100.5

**type() function**

- ✓ type() is an in-built or predefined function in python.
- ✓ This is used to check type of the variables

<b>Program Name</b>	print different kinds of variables and checking their type demo2.py
	emp_id=1 name="Nireekshan" salary=100.50
	print("My employee id is: ", emp_id) print("My name is: ", name) print("My salary is: ", salary)
	print("emp_id type is: ", type(emp_id)) print("name type is: ", type(name)) print("salary type is: ", type(salary))
<b>Output</b>	My employee id is: 1 My name is: nireekshan My salary is: 100.5 emp_id type is: <class 'int'>

```
name type is: <class 'str'>
salary type is: <class 'float'>
```

## 2 Different types of data types

- ✓ There are two type of data types.
  1. Built-in data types:
    - ✓ The data types which are already available in python are called built-in data types.
  2. User defined data types:
    - ✓ Data types which are created by programmer.

### 1.Built-in data types

1. Numeric types
  - int
  - float
  - complex
2. bool (boolean type)
3. None
4. str
5. bytes
6. bytearray
7. list
8. set
9. tuple
10. dict
11. range

## 1. Numeric types

- ✓ The numeric types represent numbers, these are divided into three types,

1. int
2. float
3. complex

### 1.1 int data type

- ✓ The int data type represents values or numbers without decimal values.
- ✓ In python there is no limit for int data type.
- ✓ It can store very large values conveniently.

**Program Name** To print integer value  
demo3.py

```
a=20
print(a)
print(type(a))
```

**output**

```
20
<class 'int'>
```

**Program Name** To print integer value  
demo4.py

```
b=99999999999999999999
print(b)
print(type(b))
```

**output**

```
99999999999999999999
<class 'int'>
```

### Info:

- ✓ In python 2<sup>nd</sup> version long data type was existing but in python 3<sup>rd</sup> version long data type was removed

## 2. float data type

- ✓ The float data type represents a number with decimal values.

**Program Name** To print float value and data type  
demo5.py

```
salary = 50.5
print(salary)
print(type(salary))
```

**Output**

```
50.5
<class 'float'>
```

- ✓ floating point numbers can also be written in scientific notation.

**Program Name** float values in exponential notation  
demo6.py

```
a = 2e2
b = 2E2
c = 2e3

print(a)      # 200
print(b)      # 200
print(c)      # 2000
print(type(a)) # <class 'float'>
```

**output**

```
200
200
<class 'float'>
```

- ✓ **e** and **E** represent exponentiation.
- ✓ where **e** and **E** to represent the power of 10.
- ✓ For example
  - The number  $2 * 10^2$  is written as 2E2, such numbers are also treated as floating point numbers.

### 3. complex data type

- ✓ The complex data type represents the numbers which is written in the form of,
  - $a + bj$
  - or
  - $a + bJ$
- $a$  is representing a real part of number
- $b$  is representing an imaginary part of the number.
- The suffix small **j** or upper **J** after  $b$  indicates the square root of -1.
- The part ' $a$ ' and ' $b$ ' may contain integers or floats.

Program Name	print complex numbers demo7.py
	<pre>a = 3+5j b = 2-5.5j c = 3+10.5j  print(a) print(b) print(c)  print()  print(a+b) print(b+c) print(c+a)</pre>
output	<pre>(3+5j) (2-5.5j) (3+10.5j)  (5-0.5j) (5+5j) (6+15.5j)</pre>

### Make a note

- ✓ If we compare two complex values with `<`, `<=`, `>`, `>=` operators then we will get **TypeError**

Program Name	comparing two complex numbers by using <code>&gt;</code> symbol demo8.py
	<pre>a = 2 + 1j b = 100 + 10j</pre>

```
print(a>b)
```

Error

`TypeError: '>' not supported between instances of 'complex' and 'complex'`

## 2. bool data type (boolean data type)

- ✓ bool data type represents boolean values in python.
- ✓ bool data type having only two values those are,
  - `True`
  - `False`
- ✓ Python internally represents,
  - `True` as **1**(one)
  - `False` as **0**(zero)
- ✓ An empty string (" ") represented as False.

**Program Name** printing bool values  
demo9.py

```
a = True  
b = False
```

```
print(a)  
print(b)  
print(a+a)  
print(a+b)
```

**output**  
True  
False  
2  
1

## 3. None data type

- ✓ None data type represents an object that does not contain any value.
- ✓ If any object having no value, then we can assign that object with None type.

**Program Name** printing None data type  
demo10.py

```
a = None
print(a)
print(type(a))
```

**Output**

```
None
<class 'NoneType'>
```

- ✓ If any function and method is not returning anything then that method can return None data type.
- ✓ We will learn more about this function and method in upcoming chapters.

**Program Name** A function return int data type  
demo11.py

```
def balance():
    print("This function returns int data type")
    return 500
```

```
b = m1()
print(b)
```

**Output**

```
This function returns int data type
500
```

## Sequences in Python

- ✓ Sequence means an object can store a group of values,
  1. str
  2. bytes
  3. bytearray
  4. list
  5. tuple
  6. range

## 1. str data type

- ✓ A group of characters enclosed within single quotes or double quotes or triple quotes is called as string.
- ✓ We will discuss more about in string chapter.

Program Name	printing string data type demo12.py
	name1 = 'nireekshan' name2 = "nireekshan" name3 = ''' nireekshan '''
	print(name1) print(name2) print(name3)
Output	nireekshan nireekshan nireekshan

## 2. bytes data type

- ✓ bytes data type represents group of numbers just like an array
- ✓ bytes data type can store the values which are from 0 to 256.
- ✓ bytes data type cannot store negative numbers.

### create byte data type

- First, we need to create list.
  - The created list we need to pass to `bytes()` function as a parameter
- 
- ✓ bytes data type is **immutable** means we cannot modify or change the bytes object.

Program Name	Creating bytes data type demo14.py
	x = [10, 20, 100, 40, 15] y = bytes(x) print(type(y))
Output	<class 'bytes'>

**Program Name** accessing bytes data type elements by using index  
demo15.py

```
x = [10, 20, 30, 40, 15]
y = bytes(x)
print(y[0])
print(y[1])
print(y[2])
print(y[3])
print(y[4])
```

**Output**

```
10
20
30
40
15
```

- ✓ We can iterate bytes values by using **for** loop.

**Program Name** printing values from bytes data type  
demo16.py

```
x = [10, 20, 100, 40, 15]
y = bytes(x)
for a in y:
    print(a)
```

**output**

```
10
20
100
40
15
```

**Program Name** **ValueError:** bytes must be in range(0, 256)  
demo17.py

```
x = [10, 20, 300, 40, 15]
y = bytes(x)
```

**Error**

**ValueError:** bytes must be in range(0, 256)

**Program Name** bytes data type is immutable  
demo18.py

```
x = [10, 20, 30, 40, 15]
y = bytes(x)
y[0] = 30
```

**Error**

**TypeError:** 'bytes' object does not support item assignment

### 3. bytearray data type

- ✓ bytearray is same as bytes data type, but bytearray is **mutable** means we can modify the content of bytearray data type.

**creating bytearray type**

- First, we need to create list.
- The created list we need to pass as **bytearray()** parameter

**Program Name** creating bytearray data type  
demo19.py

```
x = [10, 20, 30, 40, 15]
y = bytearray(x)
print(type(y))
```

**Output**

<class 'bytearray'>

**Program Name** accessing bytes data type elements by using index  
demo20.py

```
x = [10, 20, 30, 40, 15]
y = bytearray(x)
print(y[0])
print(y[1])
print(y[2])
print(y[3])
print(y[4])
```

## Output

```
10  
20  
30  
40  
15
```

- ✓ We can iterate bytearray values by using for loop.

### Program Name

printing values from bytearray data type by using for loop

demo21.py

```
x = [10, 20, 30, 40, 15]  
y = bytearray(x)  
for a in y:  
    print(a)
```

### Output

```
10  
20  
30  
40  
15
```

### Program Name

bytearray data type is mutable

demo22.py

```
x = [10, 20, 30, 40, 15]  
y = bytearray(x)  
print("Before modifying y[0] value: ", y[0])  
y[0] = 30  
print("After modifying y[0] value: ", y[0])
```

### Output

```
before modifying y[0] value: 10  
after modifying y[0] value: 30
```

## 4. list data structure

- ✓ We can create list data structure by using square brackets **[ ]**
- ✓ list can store different data types
- ✓ list is mutable.
- ✓ We will discuss more about in list chapter.

## 5. tuple data structure

- ✓ We can create tuple data structure by using parenthesis **( )**
- ✓ tuple can store different data types.
- ✓ tuple is immutable.
- ✓ We will discuss more about in tuple chapter.

## 6. range data type

- ✓ We can create a range of values by using range() function
- ✓ The range datatype represents a sequence of numbers.
- ✓ range data type is immutable, means we cannot modify once it created.
- ✓ range data type values can be print by using for loop.

<b>Program Name</b>	creating a range of values 0 to 4 by using range() function demo23.py
---------------------	--

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

<b>Output</b>	
---------------	--

```
range(0, 5)
0
1
2
3
4
```

## Make a note

- ✓ We can create a list of values with range data type

**Program Name** Creating list of values  
demo24.py

```
a=list(range(1, 10))
print(a)
```

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

## Accessing range values

- ✓ By using index, we can access elements present in the range data type

**Program Name** Access elements from range data type  
demo25.py

```
a=range(1, 10)
print(a[0])
print(a[1])
```

**output**

```
1
2
```

- ✓ If we are trying to access range data type values in out of range then we will get error.

**Program Name** Access elements from range data type  
demo26.py

```
a=range(1, 10)
print(a[0])
print(a[20])
```

**Error**

**IndexError:** range object index out of range

## 7. set data structure

- ✓ We can create set data structure by using parenthesis symbols ()
- ✓ set can store same type and different type of elements
- ✓ We will learn more in set chapter

## 8. dictionary data structure

- ✓ We can create dictionary type by using curly braces {}
- ✓ dict represents group of elements in the form of **key value** pairs like map.
- ✓ We will discuss more in dict chapter

## User defined data types

- ✓ The datatype which are created by the programmers are called 'user-defined' datatypes, example is class, module, array etc
- ✓ We will discuss about in OOPS upcoming chapter.

## Make a note

- ✓ In Python the following data types are considered as Fundamental Data types,
  - int
  - float
  - complex
  - bool
  - str

## Converting from one data type into another data type.

- ✓ Based on requirement developer can convert from one data type into another data type explicitly, this is called type conversion.
- ✓ Python provides in build functions to convert from one type to another type.
  - int() : convert from other type into int type
  - float() : convert from other type into float type
  - complex() : convert from other type into complex type
  - bool() : convert from other type into bool type
  - str() : convert from other type into str type

## 6. Operators

What is an operator?

- ✓ An operator is a symbol that performs an operation.
- ✓ An operator acts on some variables, those variables are called as operands.
- ✓ If an operator acts on a single operand, then it is called as **unary** operator
- ✓ If an operator acts on 2 operands, then it is called as **binary** operator.
- ✓ If an operator acts on 3 operands, then it is called as **ternary** operator.

Program Name	operator and operands demo1.py
	a = 10 b = 20 c = a + b print(c)
output	
	30

- ✓ Here a, b and c are called as operands,
- ✓ + is called as operator.

Different type of operators

- ✓ Arithmetic Operators: (+, -, \*, /, %, \*\*, //)
- ✓ Assignment Operator (=)
- ✓ Unary minus Operator (-)
- ✓ Relational Operator (>, <, >=, <=, ==, !=)
- ✓ Logical Operators (and, or, not)
- ✓ Membership operators (in, not in)
- ✓ Identity operators (is, is not)

Make a note:

- ✓ Python does not have **increment** and **decrement** operators.

## 1. Arithmetic Operators: (+, -, \*, /, %, \*\*, //)

- ✓ These operators do their usual job, so please don't expect any surprises.

Assume that,

$a = 13$   
 $b = 5$

Operator	Meaning	Example	Result
+	Addition	$a+b$	18
-	Subtraction	$a-b$	8
*	Multiplication	$a*b$	65
/	Division	$a/b$	2.6
%	Modulus (Remainder of division)	$a \% b$	3
**	Exponent operator (exponential power value)	$a^{**}b$	371293
//	Integer division (gives only integer quotient)	$a//b$	2

### Make a note

- ✓ Division operator / always performs floating point arithmetic, so it returns float values.
- ✓ Floor division (//) can perform both floating point and integral as well,
  - If values are int type, then result is int type.
  - If at least one value is float type, then result is float type.

Program Name      Arithmetic Operators  
                      demo2.py

```
a = 13
b = 5
print(a+b)
print(a-b)
print(a*b)
print(a/b)
print(a%b)
print(a**b)
print(a//b)
```

Output

```
18
8
65
2.6
3
371293
2
```

## 2. Assignment operator: ( $=$ , $+=$ , $-=$ , $*=$ , $/=$ , $\%=$ , $**=$ , $//=$ )

- ✓ By using these operators, we can assign values to variables.

Assume that,

```
a = 13
b = 5
```

Operator	Example	Equals to	Result
$=$	$x=a+b$	$x = a + b$	18
$+=$	$a+=5$	$a = a+5$	18
$-=$	$a-=5$	$a = a-5$	8

Program Name	Assignment operator demo3.py
	<pre>a=13 print(a) a+=5 print(a)</pre>
Output	13 18

## 3. Unary minus operator: (-)

- ✓ Symbol of unary minus operator is  $-$
- ✓ When this operator before a single variable then it brings the corresponding results

Program Name	Unary minus operator demo4.py
	$a=10$

```
print(a)
print(-a)
```

## Output

```
10
-10
```

## 4.Relational operators (>, >=, <, <=, ==, !=)

- ✓ These operators are used to compare two values.
- ✓ These operators bring boolean result as **True** and **False** while comparing the values.
- ✓ By using these operators, we can construct simple conditions.

Assume that,

```
a = 13
b = 5
```

Operator	Example	Result
>	a>b	True
>=	a>=b	True
<	a<b	False
<=	a<=b	False
==	a==b	False
!=	a!=b	True

<b>Program Name</b>	Relational operators demo5.py
	<pre>a = 13 b = 5 print(a&gt;b)</pre>

```
print(a>=b)
print(a<b)
print(a<=b)
print(a==b)
print(a!=b)
```

## Output

```
True
True
False
False
False
True
```

## 5. Logical operators (and, or, not)

- ✓ In python there are three logical operators those are,
  - and
  - or
  - not
- ✓ Logical operators are useful to construct compound conditions.
- ✓ Compound condition is a combination of more than one simple conditions.
- ✓ Each simple condition brings the boolean result finally the total compound condition evaluates either True or False.

## Make a note

- ✓ In case of logical operators,
  - **False** indicates 0
  - **True** indicates any other number.

## For boolean types behaviour

- ✓ and
  - If both arguments are True, then only result is True
- ✓ or
  - If at least one argument is True, then result is True
- ✓ not
  - complement

**Program Name**      Logical operators on boolean data types  
demo6.py

```
a = True
b = False
print(a and a)
print(a or b)
print(not a)
```

## Output

True  
True  
False

## Logical operators (and, or, not) on non-boolean data types behaviour

Suggestion: Please understand these below statements carefully,

### and operator

- ✓ A **and** B returns A, if A is False.
- ✓ A **and** B returns B, if A is not False.

### Info

- ✓ 0 means **False**
- ✓ Any other number means **True**

### Example 1

- ✓ 0 **and** 4 returns 0
- ✓ 5 **and** 7 returns 7

### Example 2

- ✓ 21 **and** 0 returns 0
- ✓ 15 **and** 8 returns 8

Program Name	Logical operators on non-boolean types demo7.py
	print(0 <b>and</b> 4) print(5 <b>and</b> 7)
Output	0 7

**Program Name** Logical operators on non-boolean types  
demo8.py

```
print(21 and 0)
print(15 and 8)
```

**Output**

```
0
8
```

**Program Name** Logical operators on non-boolean types  
demo9.py

```
x = 0
y = 4

a = 4
b = 7

print(x and y)
print(a and b)
```

**Output**

```
0
7
```

**Program Name** Logical operators on non-boolean types  
demo10.py

```
x=21
y=0

a=15
b=8

print(x and y)
print(a and b)
```

**Output**

```
0
8
```

## Conclusion

Operator	Example	Meaning
and	x and y	✓ If x is False, it returns x, otherwise it returns y

## or operator

- ✓ A or B returns A, if A is True.
- ✓ A or B returns B, if A is not True.

## Info

- ✓ 0 means False
- ✓ Any other number means True

## Example 1

- ✓ 0 or 4 returns 0
- ✓ 5 or 7 returns 7

## Example 2

- ✓ 21 or 0 returns 21
- ✓ 15 or 8 returns 8

Program Name	Logical operators on non-boolean types demo11.py
Output	<pre>print(0 or 4) print(5 or 7)</pre> <p>4 5</p>

**Program Name** Logical operators on non-boolean types  
demo12.py

```
print(21 or 0)
print(15 or 8)
```

**Output**

```
21
15
```

**Program Name** Logical operators on non-boolean types  
demo13.py

```
x = 0
y = 4
```

```
a = 5
b = 7
```

```
print(x or y)
print(a or b)
```

**Output**

```
4
5
```

**Program Name** Logical operators on non-boolean types  
demo14.py

```
x=21
y=0
```

```
a=15
b=8
```

```
print(x or y)
print(a or b)
```

**Output**

```
21
15
```

## Conclusion

Operator	Example	Meaning
or	x or y	✓ If x is False, it returns y, otherwise it returns x

## not operator

- ✓ not A returns False, if A is True
- ✓ not A returns True, if A is False

## Info

- ✓ 0 means False
- ✓ Any other number means True

## Example 1

- ✓ not 5 returns False
- ✓ not 0 returns True

<b>Program Name</b>	not operator on non-boolean types demo15.py
<b>output</b>	print(not 5) print(not 0)

<b>Program Name</b>	False True
---------------------	---------------

<b>Program Name</b>	not operator on non-boolean types demo16.py
<b>output</b>	x = 5 y = 0  print(not x) print(not y)

<b>Program Name</b>	False True
---------------------	---------------

## Conclusion

Operator	Example	Meaning
not	not x	✓ If x is False, it returns True, otherwise False

## Logical operators (and, or, not)

Operator	Example	Meaning
and	x and y	✓ If x is False, it returns x, otherwise it returns y
or	x or y	✓ If x is False, it returns y, otherwise it returns x
not	not x	✓ If x is False, it returns True, otherwise False

## 7. Membership operators

- ✓ Membership operators are useful to check whether the given object is available in collection (sequence) or not. (It may be string, list, set, tuple and dict)
- ✓ There are two membership operators,
  - **in**
  - **not in**

### The **in** operator

- ✓ **in** operator returns True, if element is found in the collection or sequences.
- ✓ **in** operator returns False, if element is not found in the collection or sequences.

### The **not in** operator

- ✓ This work in reverse manner for 'in' operator.
- ✓ **not in** operator returns True, if an element is not found in the sequence.
- ✓ **not in** operator returns False, if an element is found in the sequence.

Program Name      example by using **in** and **not in** operator  
demo17.py

```
text = "Welcome to python programming"
```

```
print("Welcome" in text)
print("welcome" in text)
print("nireekshan" in text)
print("Hari" not in text)
```

**output**

```
True
False
False
True
```

**Program Name**

example by using **in** and **not in** operator  
demo18.py

```
names = ["Ramesh", "Nireekshan", "Arjun", "Prasad"]
print("Nireekshan" in names)
print("Hari" in names)
print("Hema" not in names)
```

**output**

```
True
False
True
```

## 8. Identity operators

- ✓ This operator compares the memory locations (address) of two objects.
- ✓ Hence, it is possible to know whether the two objects are pointing to same object or not.
- ✓ The memory locations of an object can be seen by using **id()** function.

**Program Name**

Checking two variables address by using **id()** function  
demo19.py

```
a=25
b=25

print(a)
print(b)

print(id(a))
print(id(b))
```

## output

```
25  
25  
  
1570989024  
1570989024
```

There are two identity operators

1. **is**
2. **is not**

- ✓ We can use identity operators for address comparison.
- ✓ We can use **id()** predefined function to check the address of every element.

**is** operator

- ✓ A **is** B returns **True**, if both A and B are pointing to the same object.
- ✓ A **is** B returns **False**, if both A and B are not pointing to the same object.

**is not** operator

- ✓ A **is not** B returns **True**, if both A and B are not pointing to the same object.
- ✓ A **is not** B returns **False**, if both A and B are pointing to the same object.

Make a note

- ✓ The '**is**' and '**is not**' operators are **not comparing the values** of the objects.
- ✓ They **compare the memory locations (address)** of the objects.
- ✓ If we want to compare the value of the objects, we should use equality operators (**==**).

## Program Name

example by using **is** operator  
demo20.py

```
a = 25  
b = 25  
  
print(a is b)  
print(id(a))  
print(id(b))
```

## output

```
True  
1416520672  
1416520672
```

**Program Name** example by using **is** operator  
demo21.py

```
a = 25  
b = 30  
  
print(a is b)  
print(id(a))  
print(id(b))
```

**output**  
False  
1420977120  
1420977280

**Program Name** example by using **==** operator  
demo22.py

```
a = 25  
b = 25  
  
print(a == b)
```

**output**  
True

**Program Name** example by using **==** operator  
demo23.py

```
a = 25  
b = 30  
  
print(a == b)
```

**output**  
False

**Program Name** example by using **is not** operator  
demo24.py

```
a = 25  
b = 25  
  
print(a is not b)
```

**output**  
False

## 7. Input and Output

Why should we learn about input and output chapter?

- ✓ Till now we have hard coded variable values and used to print those.

**Program Name** Hard coding the values  
demo1.py

```
age=16  
print(age)
```

**Output**  
16

- ✓ But in real time based on requirement few values we should take at **run time** or **dynamically**.

This way is good so, taking the values at run time is recommended than hard coding.

```
Please enter the age: 16
```

```
Entered value is: 16
```

### Input and output

- ✓ Input represents data given to the program.
- ✓ Output represents the result of the program.

### input()

- ✓ **input()** is a predefined function.
- ✓ This function accept input from the keyboard.
- ✓ This function takes a value from keyboard and returns it as a string type.
- ✓ Based on requirement we can convert from string to other types.

**Program Name** Printing name by taking value at run time  
demo2.py

```
name = input("Enter the name: ")  
print("You entered name as: ", name)
```

### Output

```
Enter the name: Nireekshan  
Your entered name as: Nireekshan
```

**Program Name** Printing name and age by taking value at run time  
demo3.py

```
name = input("Enter the name: ")
age = input("Enter the age: ")

print("You entered name as: ", name)
print("You entered age as: ", age)
```

**Output**

```
Enter the name: Anushka
Enter the age: 60

Your entered name as: Anushka
Your entered age as: 60
```

**Program Name** Checking return type value for `input()` function  
demo4.py

```
value = input("Enter the value ")
print("Your entered value as: ", value)
print("type is: ", type(value))
```

**Output:**

```
C:\Users\Nireekshan\Desktop\Programs>py demo4.py

Enter the value: Nireekshan
Your entered value as: Nireekshan
<class 'str'>

C:\Users\Nireekshan\Desktop\Programs>py demo4.py

Enter the value : 123
Your entered value as: 123
type is: <class 'str'>

C:\Users\Nireekshan\Desktop\Programs>py demo4.py

Enter the value : 123.456
Your entered value as: 123.456
type is: <class 'str'>
```

## Convert from string type into other type

- ✓ We can convert the string value into int, float etc by using corresponding functions.
  - From string to int - int() function
  - From string to float - float() function

### Program Name

Converting from string type to int type by using int() function  
demo5.py

```
age = input("Enter your age: ")
print("Your age is: ", age)
print("age type is: ", type(age))

x = int(age)
print("After converting from string to int your age is: ", x)
print("now age type is: ", type(x))
```

### Output

```
C:\Users\Nireekshan\Desktop\Programs>py demo5.py

Enter your age:16

age type is: <class 'str'>
Your age is: 16

After converting from string to int your age is: 16
now age type is: <class 'int'>
```

### Program Name

Converting from string to float type by using float() function  
demo6.py

```
salary = input("Enter your salary: ")
print("Your salary is: ", salary)
print("salary type is: ", type(salary))

x = float(salary)
print("After converting from string to float your salary is: ", x)
print("now salary type is: ", type(x))
```

### Output

```
C:\Users\Nireekshan\Desktop\Programs>py demo6.py
```

Enter your salary:12.34

salary type is: <class 'str'>  
Your salary is: 12.34

After converting from string to float your salary is: 12.34  
now salary type is: <class 'float'>

**Program Name**

Taking int value at run time.  
demo7.py

```
age = input("Enter your age: ")  
x=int(age)  
print("Your age is: ", x)
```

**Output**

Enter your age: 16  
Your age is: 16

**Program Name**

Taking int value at run time in a simple way  
demo8.py

```
age = int(input("Enter your age: "))  
print("Your age is: ", age)
```

**Output**

Enter your age: 16  
Your age is: 16

**Program Name**

Taking values at run time in simple way  
demo9.py

```
age = int(input("Enter your age: "))  
salary = float(input("Enter salary: "))  
  
print("Your age is: ", age)  
print("Your salary is: ", salary)
```

**Output**

Enter your age: 16  
Enter salary: 123.456  
  
Your age is: 16  
Your salary is: 123.456

**Program Name** Accept two numbers and find their sum  
demo10.py

```
x = int(input("Enter first number: "))
y = int(input("Enter second number: "))
print("Sum of two values: ", x+y)
```

**Output**

```
Enter first number: 10
Enter second number: 20
Sum of two values: 30
```

**Program Name** Accept two numbers and find their sum  
demo11.py

```
x = int(input("Enter first number: "))
y = int(input("Enter second number: "))
print("The sum of {} and {} is {}".format(x, y, (x+y)))
```

**Output**

```
Enter first number: 1
Enter second number: 2
The sum of 1 and 2 is: 3
```

## eval() function

- ✓ **input()** function returns string type.
- ✓ **eval()** is a predefined function which takes string as parameter.
- ✓ If you entered any expression in string, then **eval()** function evaluates the expression and returns the result.

**Program Name** eval() function evaluating expressions  
demo12.py

```
print("10+20")
sum = eval("10+20")
print(sum)
```

**output**

```
10+20
30
```

Program Name	eval() function evaluating expressions demo13.py
	sum = eval(input("Enter expression: ")) print(sum)
Output	Enter expression: 2*3+4 10

## Command line arguments

- ✓ While running the program we can provide the arguments.
- ✓ These provided arguments are called as Command line arguments.
- ✓ We need to use **argv** to work with command line arguments.
- ✓ **argv** is a list in python.
- ✓ **argv** is available **sys** module.
- ✓ So, we need to import **sys** module to access **argv**.

## Syntax:

```
C:\Users\Nireekshan\Desktop\Programs>py demo.py 10 20
```

- argv[0] => Name of the program (demo.py)
- argv[1] => 10
- argv[2] => 20

## Make a note:

- ✓ When providing arguments,
  - The first argument argv[0] is Name of the program.
  - Remaining arguments are argv values respectively.

Program Name	Printing run time arguments by using argv. demo14.py
	from sys import argv print(argv[0]) print(argv[1]) print(argv[2])
Run	python demo14.py 30 40
Output	demo14.py 30 40

## IndexError

- ✓ If we are trying to access command line arguments with out of range index, then we will get IndexError.

**Program Name** Accessing index which is out of range.  
demo15.py

```
from sys import argv  
print(argv[100])
```

**Run** python demo15.py 30 40

**Output**  
**IndexError:** list index out of range

**Program Name** Without command line arguments addition  
demo16.py

```
x = 10  
y = 20  
print(x+y)
```

**Output**  
30

## Make a note

- ✓ Command line arguments are string type, means argv returns string type,
- ✓ So, by default argv type is string.
- ✓ Based on requirement we can convert from string type to another type.

Program Name	Checking argv type demo17.py
	<pre>from sys import argv  first_name = argv[1] last_name=argv[2]  print("First name is: ", first_name) print("Last name is: ", last_name)  print("Type of first name is", type(first_name)) print("Type of last name is", type(last_name))</pre>
Run output	py demo17.py nireekshan subbamma  First name is: <b>nireekshan</b> Last name is: <b>subbamma</b> Type of first name is <class 'str'> Type of last name is <class 'str'>

Program Name	Adding items costs which is in sting type by using argv demo18.py
	<pre>from sys import argv  item1_cost = argv[1] item2_cost =argv[2]  print("First item cost is: ", item1_cost) print("Second item cost is: ", item2_cost)  print("Type of item1_cost is : ", type(item1_cost)) print("Type of item2_cost is : ", type(item2_cost))  total_items_cost= item1_cost + item2_cost  print("Total cost is: ", total_items_cost)</pre>

Run            py demo18.py 111 223

output

First item cost is: 111  
Second item cost is: 223

Type of item1\_cost is : <class 'str'>  
Type of item1\_cost is : <class 'str'>

Total cost is: 111223

Make a note:

- ✓ + operators join or concatenates two string values.
- ✓ If we are adding two string values the by using + operators then, joined string will be the output
- ✓ It's possible to convert from string type to others by using corresponding pre-defined methods.

Program Name     Adding items costs by using argv  
demo19.py

```
from sys import argv

item1_cost = argv[1]
item2_cost = argv[2]

x=int(item1_cost)
y=int(item2_cost)

print("First item cost is: ", x)
print("Second item cost is: ", y)

print("Type of item1_cost is : ", type(x))
print("Type of item2_cost is : ", type(y))

total_items_cost= x + y

print("Total cost is: ", total_items_cost)
```

Run            py demo19.py 111 223

output

First item cost is: 111  
Second item cost is: 223

Type of item1\_cost is : <class 'int'>

Type of item2\_cost is : <class 'int'>

Total cost is: 334

## len() function

- ✓ len() is a predefined function which returns the number of values

**Program Name** print command line arguments to find length  
demo20.py

```
from sys import argv
print("The length of values :", len(argv))
```

**Run** python demo20.py 10 20

**Output** The length of values :3

**Program Name** print command line arguments one by one  
demo21.py

```
from sys import argv
print("Total arguments: ", argv)
print("Command Line Arguments one by one:")

for x in argv:
    print(x)
```

**Run** python demo21.py nireekshan subbamma

**Output**

```
Total arguments: ['demo21.py', 'nireekshan', 'subbamma']
```

```
Command Line Arguments one by one:
demo21.py
nireekshan
subbamma
```

## Make a note

- ✓ By default, space is separator between command line arguments.
- ✓ While providing the values in command line arguments, if you want to provide space then we need to enclose **with double quotes (but not single quotes)**.

Program Name	passing text with spaces to command line arguments demo22.py
	<pre>from sys import argv print(argv[1])</pre>
Run	python demo22.py hello good morning
Output	hello

Program Name	Text with spaces by using double quotes in command line arguments demo23.py
	<pre>from sys import argv print(argv[1])</pre>
Run	python demo23.py "hello good morning"
Output	hello good morning

Program Name	Text with spaces by using single quotes in command line arguments demo24.py
	<pre>from sys import argv print(argv[1])</pre>
Run	python demo24.py 'hello good morning'
Output	'hello

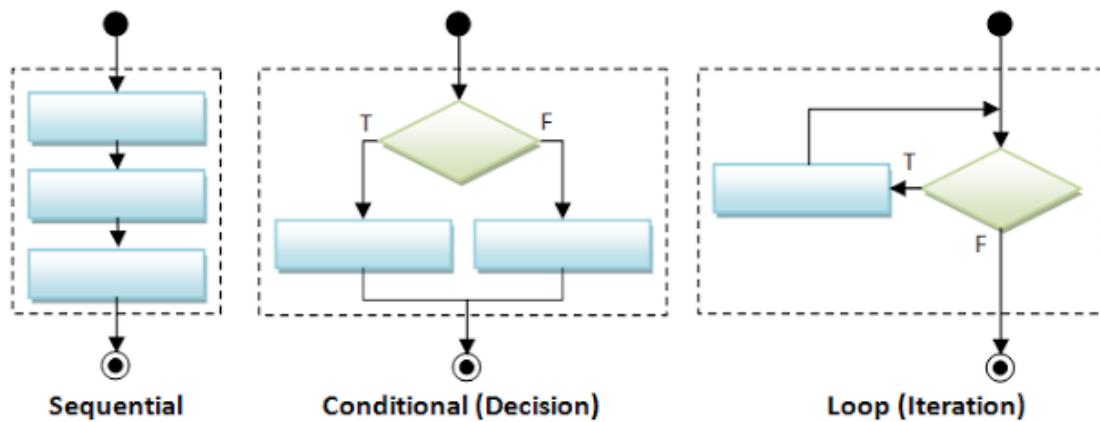
## 8. Flow control

Why should we learn about flow control?

- ✓ **Simple answer:** To understand the flow of statements execution in a program.
- ✓ In any programming language, statements will be executed mainly in three ways,
  - Sequential.
  - Conditional.
  - Looping.

### Flow control

- ✓ The order of statements execution is called as flow of control.
- ✓ Based on requirement the programs statements can execute in different ways like sequentially, conditionally and repeatedly etc.



### 1. Sequential

- ✓ Statements execute from top to bottom, means one by one sequentially.
- ✓ By using sequential statement, we can develop only simple programs.

### 2. Conditional

- ✓ Based on the conditions, statements used to execute.
- ✓ Conditional statements are useful to develop better and complex programs.

### 3. Looping

- ✓ Based on the conditions, statements used to execute randomly and repeatedly.
- ✓ Looping execution is useful to develop better and complex programs.

## 1. Sequential statements

- ✓ Sequential statements execute from top to bottom, means one by one sequentially.

<b>Program Name</b>	sequential execution demo1.py
	print("one") print("two") print("three")
<b>output</b>	one two three

## 2. Conditional or Decision-making statements

- if
- if else
- if elif else
- else suite

## 3. Looping

- while loop
- for loop

## 4. others

- break
- continue
- return
- pass

## 2. Conditional or Decision-making statements

### 2.1 if statement

#### syntax

```
if condition:  
    if block statements  
  
    out of if block statements
```

- ✓ **if** statement contains an expression/condition.
- ✓ As per the syntax colon (:) is mandatory otherwise it throws syntax error.
- ✓ Condition gives the result as a bool type, means either **True** or **False**.



- ✓ If the condition result is **True**, then **if** block statements will be executed
- ✓ If the condition result is **False**, then **if** block statements won't execute.

**Program Name** Executing if block statements by using **if** statement  
demo2.py

```
num = 1  
print("num==1 value is: ", (num==1))  
if num == 1:  
    print("if block statements executed")
```

**output**  
num==1 value is: True  
if block statements executed

**Program: Name** executing out of if block statements  
demo3.py

```
num = 1  
print("num==2 value is: ", (num==2))  
if num == 2:  
    print("if block statements")  
print("out of if block statements")
```

**output**  
num==2 value is: False  
out of if block statements

When should we use **if** statement?

- ✓ If you want to do either one thing or nothing at all, then you should go for **if** statement.

## 2.1 if else statement

### syntax

```
if condition:  
    if block statements1  
  
else:  
    else block statements2
```

- ✓ **if** statement contains an expression/condition.
- ✓ As per the syntax colon (:) is mandatory otherwise it throws syntax error
- ✓ Condition gives the result as bool type, means either **True** or **False**



- ✓ If the condition result is **True**, then **if** block statements will be executed
- ✓ If the condition result is **False**, then **else** block statements will be executed.

**Program Name** Executing if block statements by using **if else** statement  
demo4.py

```
num =1  
print("num==1 value is: ", (num==1))  
if num == 1:  
    print("if block statements executed")  
else:  
    print("else block statements executed")
```

### output

```
num==1 value is: True  
if block statements executed
```

**Program Name** printing else block statements  
demo5.py

```
num =1
print("num==2 value is: ", (num==2))

if num == 2:
    print("if block statements executed")
else:
    print("else block statements executed")
```

**output**

```
else block statements executed
```

**Program Name** User name validation checking by using **if else** statement  
demo6.py

```
user_name ="subbamma"
name = input("Please enter user name: ")

if user_name== name:
    print("Welcome to gmail : ", name)

else:
    print("Invalid user name, please try again")
```

**output**

```
Please enter user name: subbamma
Welcome to gmail: subbamma

Please enter user name: chiru
Invalid user name, please try again
```

**Program Name** User name validation checking by using **if else** statement  
demo7.py

```
user_name ="nireekshan"
user_password = "balayya"

name = input("Please enter user name: ")
password = input("Please enter your password: ")

if (user_name == name) and (user_password == password):
```

```
print("Welcome to gmail : ", name)  
else:  
    print("Invalid user name or password, please try again")  
output
```

```
Please enter your name: nireekshan  
Please enter your password: balayya  
Welcome to gmail : nireekshan
```

```
Please enter your name: nireekshan  
Please enter your password: sampurneshbabu  
Invalid user name or password, please try again
```

When should we use **if else** statement?

- ✓ If you want to do either one thing or other thing, then you should go for **if else** statement.

## 2.2 if elif else statement

### syntax

```
if condition1  
    if block statements  
  
elif condition2  
    elif block1 statements  
  
elif condition3  
    elif block2 statements  
  
else  
    else block statements
```

- ✓ **if** statement contains an expression/condition.
- ✓ As per the syntax colon (:) is mandatory otherwise it throws error
- ✓ Condition gives the result as bool type, means either **True** or **False**



- ✓ If the condition result is **True**, then any matched **if** or **elif** block statements will execute.
- ✓ If all **if** and **elif** conditions results are **False**, then **else** block statements will execute.

## Make a note

- ✓ Here, **else** part is an optional

**Program Name** printing corresponding value by using if, elif, else statements  
demo8.py

```
print("Please enter the values from 0 to 4")
x=int(input("Enter a number: "))

if x==0:
    print("You entered:", x)

elif x==1:
    print("You entered:", x)

elif x==2:
    print("You entered:", x)

elif x==3:
    print("You entered:", x)

elif x==4:
    print("You entered:", x)

else:
    print("Beyond the range than specified")
```

**output**

```
Enter a number: 1
You entered: 1

Enter a number: 100
Beyond the range
```

## When should we use **if elif else** statement

- ✓ If we want to choose one option from many options, then we should use **if elif else** statements.

## Programs by using conditional statements

**Program Name** Find biggest of given 2 numbers from the command prompt  
demo9.py

```
x=int(input("Enter First Number: "))
y=int(input("Enter Second Number: "))

if x>y :
    print("Biggest Number is: ", x)

else :
    print("Biggest Number is: ", y)
```

**Output**

```
Enter First Number: 10
Enter Second Number: 20
Biggest Number is: 20
```

**Program Name** Find biggest of given 3 numbers from the command prompt  
demo10.py

```
x=int(input("Enter First Number: "))
y=int(input("Enter Second Number: "))
z=int(input("Enter Third Number: "))

if (x>y) and (x>z):
    print("Biggest Number is: ",x)

elif y>z :
    print("Biggest Number is: ",y)

else :
    print("Biggest Number is: ",z)
```

**Output**

```
Enter First Number: 10
Enter Second Number: 20
Enter Second Number: 30

Biggest Number is: 30
```

Program Name	To find the number belongs to which group demo11.py
	<pre>x=int(input("Enter number:"))  if (x&gt;=1) and (x&lt;=100) :     print("Entered number", x , "is in between 1 to 100")  else :     print("Entered number", x , "is not in between 1 to 100")</pre>
Output	Enter number: 55 Entered number 55 is in between 1 to 100 Enter number: 201 Entered number 201 is not in between 1 to 100

### 3. Looping

- ✓ If we want to execute a group of statements in multiple times, then we should go for looping kind of execution.
  - ✓ while loop
  - ✓ for loop

#### 3.2 while loop

- ✓ If we want to execute a group of statements repeatedly until the condition reaches to **False**, then we should go for **while** loop.

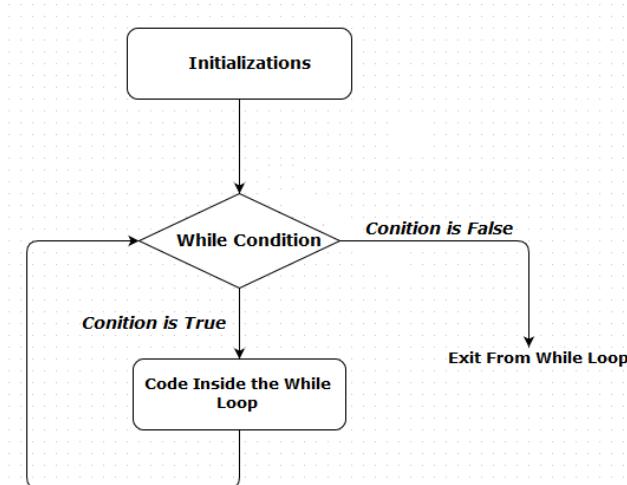
#### Syntax

```
while condition:  
    statements
```

- ✓ **while** loop contains an expression/condition.
- ✓ As per the syntax colon (:) is mandatory otherwise it throws syntax error
- ✓ Condition gives the result as bool type, means either **True** or **False**



- ✓ If the condition result is **True**, then while loop executes till the condition reaches to **False**.
- ✓ If the condition result is **False**, then **while** loop execution terminates.



## while loop main parts

- ✓ Initialization section.
- ✓ Condition section.
- ✓ Either increment or decrement by using arithmetic operator.

## while loop explanation

### Initialization section.

- ✓ Initialization section is the first part in while loop.
- ✓ Here, before entering the condition section, initialization part is required.

### Condition section

- ✓ With the initialized value, the next step is checking the condition.
- ✓ In python, **while** loop will check the condition at the beginning of the loop.
- ✓ In first iteration:
  - In first iteration, if the condition returns **True**, then it will execute the statements which having inside while loop.
  - In first iteration, if the condition returns **False**, then while loop terminates immediately.
- ✓ Then the loop execution goes to increment or decrement section.

## Increment or decrement

- ✓ Next step is, increment or decrement section.
- ✓ In python we need to use arithmetic operator inside **while** loop to increment or decrements the value.
- ✓ So, based on requirement, value will be either increment or decrement.
- ✓ In second iteration:
  - In second iteration, if the condition returns **True**, then it will execute the statements which having inside while loop.
  - In second iteration, if the condition returns **False**, then while loop terminates immediately.

## Conclusion

- ✓ Till condition is **True** the **while** loop statements will be executed.
- ✓ If the condition reaches to **False**, then **while** loop terminate the execution.

<b>Program Name</b>	Printing numbers from 1 to 5 by using while loop demo12.py
	<pre>x=1 while x&lt;=5:     print(x)     x+=1  print("End")</pre>
<b>output</b>	<pre>1 2 3 4 5 End</pre>

<b>Program Name</b>	Printing even numbers from 10 to 20 by using while loop demo13.py
	<pre>x=10 while (x&gt;=10) and (x&lt;=20):     print(x)     x+=2  print("End")</pre>

## output

```
10  
12  
14  
16  
18  
20  
End
```

## for loop

- ✓ Basically, **for** loop is used to get or iterate elements one by one from sequence like string, list, tuple, etc...
- ✓ While iterating elements from sequence we can perform operations on every element.

## Syntax

```
for variable in sequence:  
    statements
```

### Program Name

printing elements from list object  
demo14.py

```
l1 = [10, 20, 30, 'Nireekshan']  
for x in l1:  
    print(x)
```

### output

```
10  
20  
30  
Nireekshan
```

### Program Name

printing characters from string  
demo15.py

```
x= "python"  
for ch in x:  
    print(ch)
```

### output

p  
y  
t  
h  
o  
n

**Program Name** printing every item cost by adding gst from list  
demo16.py

```
items_costs = [10, 20, 30,]  
gst = 2  
  
for x in items_costs:  
    print(x+gst)
```

**output**

12  
22  
32

**Program Name** printing elements by using range() function and for loop  
demo17.py

```
for x in range(1, 5):  
    print(x)
```

**output**

1  
2  
3  
4

**Program Name** sum of the items cost in list using for loop  
demo18.py

```
item_costs = [10, 20, 30]  
sum=0  
for x in item_costs:  
    sum=sum + x  
print(sum)
```

**output**

60

## Infinite loops

- ✓ The loop which never ends is called infinite group.

**Program Name** infinite loop demo19.py

```
while True:  
    print("Hello")
```

**output**

```
Hello  
Hello  
Hello  
.  
.  
.
```

## Make a note

- ✓ To come out of infinite loop we need to press **ctrl + c**

## Nested loops

- ✓ It is possible to write one loop inside another loop, this is called nested loop

**Program Name** printing stars by using nested for loop demo20.py

```
rows = range(1, 5)  
for x in rows:  
    for star in range(1, x+1):  
        print('*', end=' ')  
    print()
```

**output**

```
*
```

```
* *
```

```
* * *
```

```
* * * *
```

## Loops with else block (or) else suit

- ✓ In python, it is possible to use 'else' statement along with **for** loop and **while** loop.

<b>for with else</b>	<b>while with else</b>
<pre>for variable in sequence:     statements  else:     statements</pre>	<pre>while condition:     statements  else:     statements</pre>

- ✓ The statements written after 'else' are called as **else suite**.
- ✓ The **else** suite will be always executed irrespective of the loop executed or not.

<b>Program Name</b> for with else demo21.py	<pre>values = range(5) for x in values:     print("item is available")  else:     print("sorry boss, item is not available")</pre>
<b>output</b> <pre>item is available item is available item is available item is available item is available sorry boss, item is not available</pre>	

<b>Program Name</b> for with else demo22.py	<pre>values = range(0) for x in values:     print("item is available")  else:</pre>
---	---

```
print("sorry boss, item is not available")  
  
output  
  
sorry boss, item is not available
```

## Make a note

- ✓ Seems to be for loop and else suit both got executed.

## Where this kind of execution will helpful?

- ✓ If you are searching element from list object, if unable to find element then **for** loop won't return anything, so at that time **else** suit help us to tell not found the element.

```
Program: Element searching in list  
Name demo23.py  
  
group = [1, 2, 3, 4]  
search = int(input("Enter the element in search: "))  
  
for element in group:  
    if search == element:  
        print("element found in group", element)  
        break  
    else:  
        print("Element not found")  
  
output  
Enter element to search: 4  
Element found in group : 4  
  
Enter element to search: 6  
Element not found in group
```

## break statement

- ✓ The **break** statement can be used inside the loops to break the execution based on some condition.
- ✓ Generally, **break** statement is used to terminate the **for** loop and **while** loop.
- ✓ **break** statement we can use inside loops only otherwise it gives **SyntaxError**.

**Program Name** while loop without break  
demo24.py

```
x=1
while x<=10:
    print('x= ', x)
    x+=1
print("out of loop")
```

**output**

```
x= 1
x= 2
x= 3
x= 4
x= 5
x= 6
x= 7
x= 8
x= 9
x= 10
out of loop
```

**Program Name** printing just 1 to 5 by using while loop and break  
demo25.py

```
x=1
while x<=10:
    print('x', x)
    x+=1
    if x == 5:
        break
print("out of the loop")
```

**output**

```
x= 1
x= 2
x= 3
x= 4
out of the loop
```

## continue statement

- ✓ We can use **continue** statement to skip current iteration and continue next iteration.

### Program Name

Applying discount based on condition and printing remaining  
demo26.py

```
cart=[10, 20, 500, 700, 50, 60]
for item in cart:
    if item>=500:
        continue
    print("No discount applicable: ", item)
```

### output

```
No discount applicable: 10
No discount applicable: 20
No discount applicable: 50
No discount applicable: 60
```

Please comment the above statement of continue then execute.

## pass statement

- ✓ We can use **pass** statement when we need a statement syntactically, but we do not want to do any operation.

### Program Name

pass statement  
demo27.py

```
num = [10, 20, 30, 400, 500, 600]
for i in num:
    if i<100:
        print("pass statement executed")
        pass
    else:
        print("Give festival coupon for these guys who bought: ",i)
```

### output

```
pass statement executed
pass statement executed
pass statement executed
Give festival coupon for these guys who bought : 400
Give festival coupon for these guys who bought : 500
Give festival coupon for these guys who bought: 600
```

## return statement

- ✓ Based on requirement a function or method can return the result.
- ✓ If any function or method is returning a value, then we need to write **return** statement along with the value.

**Program Name** Function displaying information  
demo28.py

```
def display():
    print("This is python programming")
display()
```

**output** This is python programming

**Program Name** Function displaying information and that function returns None  
demo29.py

```
def display():
    print("This is python programming")
x = display()
print(x)
```

**output** This is python programming  
None

## Make a note

- ✓ If any function is not returning anything then by default that function returns **None** data type.

## Make a note

- ✓ If any function or method is returning a value, then we need to write **return** statement along with the value.

**Program Name** Function returns a result  
demo30.py

```
def sum(a, b):
    return a+b
```

```
res=sum(10, 20)
print(res)
```

output

```
30
```

## 10. string in python

### Gentle reminder

- ✓ We already learnt first hello world program in python.
- ✓ In that program we just print a group of characters by using print() function.
- ✓ Those group of characters are called as a string.

Program Name	printing Welcome to python programming demo1.py
output	print("Welcome to python programming")  Welcome to python programming

### What is a string?

#### Definition 1

- ✓ A group of characters enclosed within single or double or triple quotes is called as string.

#### Definition 2

- ✓ We can say string is a sequential collection of characters.

### String is more popular

- ✓ In any kind of programming language, mostly usage data is string.

### Creating string

Syntax 1	With single quotes
	name1 = ' Nireekshan '

Syntax 2	With double quotes
	name2 = " Nireekshan "

Syntax 3	With triple single quotes
	name3 = ''' Nireekshan '''

## DVS Technologies

Syntax 4 With triple double quotes

```
name4 = " " " Nireekshan " " "
```

Program Name Creating string by using all possibilities  
demo2.py

```
name1= 'Nireekshan'  
name2 = "Subbamma"  
name3 = '''Ramesh'''  
name4 = """Arjun"""  
  
print(name1," name is created by using single quotes")  
print(name2," name is created by using double quotes")  
print(name3," name is created by using triple single quotes")  
print(name4," name is created by using triple double quotes")
```

output

```
nireekshan name is created by using single quotes  
Subbamma name is created by using double quotes  
Ramesh name is created by using triple single quotes  
Arjun name is created by using triple double quotes
```

Program Name printing Employee information  
demo3.py

```
name = 'Balayya'  
emp_id = 20  
print("Name of the employee: ", name)  
print("employee id is : ", emp_id)
```

output  
Name of the employee: Balayya  
id is: 20

Program Name Error: starts string with single quotes and closing with double quotes  
demo4.py

```
name = 'nireekshan'  
print(name)
```

Error

SyntaxError: EOL while scanning string literal

## Make a note

- ✓ Generally, to create a string mostly used syntax is double quotes syntax.

## When should we go for triple single and triple double quotes?

- ✓ If you want to create multiple lines of string, then triple single or triple double quotes are the best to use.

## Multi-line string objects

**Program Name** printing Employee information  
demo5.py

```
loc1 = "TCS company  
While Field  
Bangalore"
```

```
loc2 = """TCS company  
Banglore  
Manyatha tech park"""
```

```
print(loc1)  
print(loc2)
```

## output

```
TCS company  
White Field  
Bangalore
```

```
TCS company  
Banglore  
Manyatha tech park
```

## Make a note

- ✓ Inside string, we can use single or double quotes symbols.

**Program Name** Inside string, we can use single or double quotes symbols  
demo6.py

```
s1 = 'Welcome to "python" learning'
```

```
s2 = "Welcome to 'python' learning"  
s2 = """Welcome to 'python' learning"""
```

## output

```
Welcome to "python" learning  
Welcome to 'python' learning  
Welcome to 'python' learning
```

## Empty string

- ✓ If string contains no characters, then we can say it as empty string.

Program Name      Empty string  
demo7.py

```
s1 = ''  
print(s1)
```

## output

## Accessing string characters

- ✓ We can access string characters by using,
  - Indexing
  - Slicing

## Indexing in string

- ✓ Indexing means a position of string's characters where it stores.
- ✓ We need to use square brackets [] to access the string index.
- ✓ String indexing result is string type.
- ✓ String indices should be integer otherwise we will get error.
- ✓ We can access index within the index range otherwise we will get error.

## Python support two types of indexes

- ✓ Positive index
- ✓ Negative index

## Positive index

- ✓ The position of string characters can be positive index from left to right direction (we can say forward direction).
- ✓ In this way, the starting position is 0 (zero)

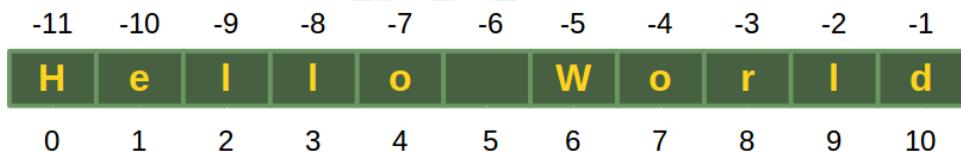
## Negative index

- ✓ The position of string characters can be negative index from **right to left** direction (we can say backward direction).
- ✓ In this way, the starting position is **-1** (minus one)
  - Please don't think too much like **-0** (minus zero), there is no minus zero boss.

## Internal representation

<b>Program Name</b>	Printing text message demo8.py
	wish = "Hello world" print(wish)
<b>output</b>	Hello world

## Diagram representation



## Make a note

- ✓ If we are trying to access characters of a string with **out of range index**, then we will get error as: **IndexError**

<b>Program Name</b>	Accessing string index by using integer values. demo9.py
	wish = 'Hello World'  print(wish[0]) print(wish[1])
<b>output</b>	H e

**Program Name** Error: Accessing string index by using float values  
demo10.py

```
wish = "Hello World"  
print(wish[1.3])
```

**Error** TypeError: string indices must be integers e

**Program Name** Error: Accessing string index which is out of bound index  
demo11.py

```
wish = "Hello World"  
print(wish[100])
```

**Error** IndexError: string index out of range

**Program Name** Accessing string index by using negative values.  
demo12.py

```
wish = 'Hello World'  
print(wish[-1])  
print(wish[-2])
```

**output**

```
d  
I
```

**Program Name** printing character by character from string by using for loop  
demo13.py

```
name = 'Python'  
for a in name:  
    print(a)
```

**output**

```
P  
y  
t
```

h  
o  
n

## Slicing in string

- ✓ A substring of a string is called as a slice.
- ✓ A slice can represent a part of string from string or a piece of string.
- ✓ String slicing result is string type.
- ✓ We need to use square brackets **[ ]** in slicing.
- ✓ In slicing we will not get any Index out of range exception.
- ✓ In slicing indices should be **integer** or **None** or **\_\_index\_\_** method otherwise we will get **error**.

### Syntax 1:

```
nameofthestring [start: stop]
```

- ✓ start: Represents from the starting index position
- ✓ stop: Represents the (ending index - 1) position

### Syntax 2:

```
nameofthestring [start : stop : step]
```

- ✓ start: Represents from the starting index position.
- ✓ stop: Represents the end **index - 1** position.
- ✓ step: Represents the increment the position of the index while accessing.

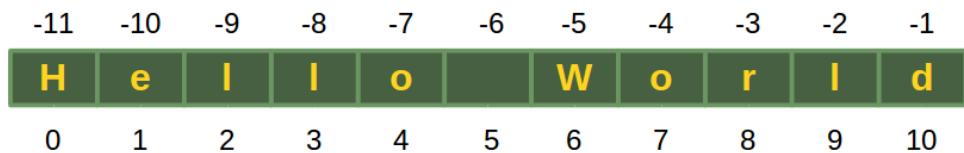
### Example1

- ✓ nameofthestring [n : m]
  - Accessing from **n<sup>th</sup>** character to **m<sup>th</sup> - 1** character
  - including the first character but excluding the last

### Example2

- ✓ nameofthestring [n : m : s]
  - Accessing from **n<sup>th</sup>** character to **m<sup>th</sup> - 1** character
  - including the first character but excluding the last
  - increment the **s** index position while accessing

## Internal representation



- ✓ If start and stop not specified, then slicing is done from 0<sup>th</sup> to n-1 element
- ✓ If step size not specified, then it takes default value as 1

## Default values

- ✓ start value is 0 (zero)
- ✓ stop value is n<sup>th</sup> - 1
- ✓ step value is 1

## String several cases in slicing

### Example

```
wish = "Hello World"
```

- ✓ wish [::] => accessing from 0<sup>th</sup> to last
- ✓ wish [:] => accessing from 0<sup>th</sup> to last
- ✓ wish [0:9:1] => accessing string from 0<sup>th</sup> to 8<sup>th</sup> means (9-1) element.
- ✓ wish [0:9:2] => accessing string from 0<sup>th</sup> to 8<sup>th</sup> means (9-1) element.
- ✓ wish [2:4:1] => accessing from 2<sup>nd</sup> to 3<sup>rd</sup> characters.
- ✓ wish [:: 2] => accessing entire in steps of 2
- ✓ wish [2 ::] => accessing from str[2] to ending
- ✓ wish [:4:] => accessing from 0<sup>th</sup> to 3 in steps of 1
- ✓ wish [-4: -1] => access -4 to -1

## Make a note

- ✓ If we are not specifying begin index, then it will consider from beginning of the string.
- ✓ If we are not specifying end index, then it will consider up to end of the string.
- ✓ The default value for step is 1

<b>Program Name</b>	slice operator several use cases demo14.py
	wish = "Hello World"

```
print(wish[:])
print(wish[:])
print(wish[0:9:1])
print(wish[0:9:2])
print(wish[2:4:1])
print(wish[::-2])
print(wish[2::])
print(wish[:4:])
print(wish[-4:-1])
```

## Output

```
Hello World
Hello World
Hello Wor
HloWr
Ii
HloWrd
Ilo World
Hell
orl
```

## Immutable

- ✓ Once if we create an object then the state of existing object cannot be change or modify.
- ✓ This behaviour is called as immutability

## Mutable

- ✓ Once if we create an object then the state of existing object can be change or modify.
- ✓ This behaviour is called as mutability.

## Strings are immutable

- ✓ String having immutable nature.
- ✓ Once we create a string object then we cannot change or modify the existing object.

<b>Program Name</b>	Printing name and first index in string demo15.py
<b>output</b>	name = "Nireekshan" print(name) print(name[0])  Nireekshan N

**Program Name** string having immutable nature  
demo16.py

```
name = "Nireekshan"  
print(name)  
print(name[0])  
name[0] = "X"
```

**output**

**TypeError:** 'str' object does not support item assignment

## Mathematical operators on string objects

- ✓ We can perform two mathematical operators on string.
- ✓ Those operators are,
  - Addition (+) operator.
  - Multiplication (\*) operator.

### Interesting about + and \* operators

#### Addition (+) operator with string

- ✓ The + operator works like concatenation or joins the strings.

**Program Name** + works as concatenation operator  
demo17.py

```
a = "Python"  
b = "Programming"  
print(a+b)
```

**output** PythonProgramming

**Program Name** + works as concatenation operator  
demo18.py

```
a = "Python"  
b = " Programming"  
print(a+b)
```

**output** Python Programming

## Multiplication (\*) operator with string

### Repeating the string

- ✓ This operator works with string to do repetition.

**Program Name** \* operator works as repetition in strings  
demo19.py

```
course = "Python"  
print(course * 3)
```

**output** PythonPythonPython

### Make a note:

- ✓ While using + operator on string then compulsory both arguments should be string type, otherwise we will get error.
- ✓ While using \* operator on string then compulsory one argument should be string and other arguments should be int type.

**Program Name** Error: applying + operator with string and integer values.  
demo20.py

```
a='nireekshan'  
b=6  
print(a+b)
```

**Error** TypeError: must be str, not int

**Program Name** Error: applying + operator with string and integer values.  
demo21.py

```
a="123"  
b=456  
print(a+b)
```

**Error** TypeError: must be str, not int

## Length of the string

- ✓ We can find length of string by using len() function.
- ✓ By using len() function we can find group of characters presents in string.
- ✓ len() function returns int as result.

<b>Program Name</b>	length of the string demo22.py
	course = 'Python' print(course, " length is : ",len(course))
<b>Output</b>	Length of string is: 6

## Membership operators

### Definition 1

- ✓ We can check, if a string or character is a member of string or not by using below operators,
  - in
  - not in

### Definition 2

- ✓ We can check, if string is a substring of main string or not by using **in** and **not in** operators.

### in operator

- ✓ **in** operator returns **True**, if the string or character found in the main string.

<b>Program Name</b>	in operator demo23.py
	print('p' in 'python') print('z' in 'python') print('on' in 'python') print('pa' in 'python')
<b>output</b>	True False True False

**Program Name** To find sub string in main string  
demo24.py

```
main=input("Enter main string:")
s=input("Enter sub string:")

if s in main:
    print(s, "is found in main string")

else:
    print(s, "is not found in main string")
```

**output**

```
Enter main string: python programming
Enter sub string: python
python is found in main string
```

```
Enter main string: python programming
Enter sub string: java
java is not found in main string
```

**not in operator**

- ✓ The **not in** operator returns opposite result of **in** operator.
- ✓ **not in** operator returns True, if the string or character not found in the main string.

**Program Name** not in operator  
demo25.py

```
print('b' not in 'apple')
```

**output**  
True

## Comparing strings

- ✓ We can use the relational operators like `>`, `>=`, `<`, `<=`, `==`, `!=` to compare two string objects.
- ✓ While comparing it returns boolean values, either **True** or **False**.
- ✓ Comparison will be done based on alphabetical order or dictionary order

<b>Program Name</b>	comparison operator on string demo26.py
	<pre>s1 = 'abcd' s2 = 'abcdefg'  print(s1 == s2)  if(s1 == s2):     print("Both are same") else:     print("not same")</pre>
<b>output</b>	False not same

<b>Program Name</b>	User name validation checking by using if else statements demo27.py
	<pre>user_name ="nireekshan" name = input("Please enter user name:")  if user_name == name:     print("Welcome to gmail: ", name)  else:     print("Invalid user name, please try again")</pre>
<b>output</b>	Please enter user name: nireekshan Welcome to gmail

## Removing spaces from String

- ✓ A space is also considered as character inside string.
- ✓ Sometimes unnecessary spaces in a string will lead to wrong results.

### Scenario 1

**Program Name** To check space importance in string at the end of the string  
demo28.py

```
s1 = "Nireekshan"
s2 = "Nireekshan"
print(s1 == s2)
if s1 == s2:
    print("same")
else:
    print("not same")
```

**output**

```
False
not same
```

## Predefined methods to removes spaces

- ✓ `rstrip()` -> remove spaces at right side of string
- ✓ `lstrip()` -> remove spaces at left side of string
- ✓ `strip()` -> remove spaces at left & right sides of string

### Make a note

- ✓ These methods will not remove the spaces which are available middle of string object.

**Program Name** removing spaces in starting and ending of the string  
demo29.py

```
course = "Python"
print("with spaces course length is: ",len(course))
x=course.strip()
print("after removing spaces, course length is: ",len(x))
```

**Output**

```
with spaces course length is: 18
```

after removing spaces, course length is: 6

## Finding substrings:

- ✓ We can find a sub string in two directions like forward and backward direction.
- ✓ We can use the following 4 methods

### For forward direction:

- ✓ `find()`
- ✓ `index()`

### For backward direction:

- ✓ `rfind()`
- ✓ `rindex()`

#### 1. `find():`

##### Syntax

```
nameofthestring.find(substring)
```

- ✓ This method returns index of first occurrence of the given substring, if it is not available, then we will get -1(minus one) value

**Program Name** finding sub string by using `find` method  
`demo30.py`

```
course="Python programming language"
print(course.find("Python"))
print(course.find("programming"))
print(course.find("Hello"))
print(course.find("z"))
print(course.rfind("ing"))
```

##### output

```
0
7
-1
-1
15
```

## Make a note:

- ✓ By default find() method can search total string of the object.
- ✓ We can specify the boundaries while searching.

### Syntax

```
nameofthestring.find(substring, begin, end)
```

- ✓ It will always search from begin index to **end-1** index

**Program Name** finding sub string by using find method  
demo31.py

```
course="Python programming language"
print(course.find("p"))
print(course.find("a", 1, 20))
```

**output**
7
12

### index() method:

- ✓ index() method is exactly same as find() method except that if the specified substring is not available then we will get ValueError.
- ✓ So, we can handle this error in application level

**Program Name** finding sub string by using find method  
demo32.py

```
s="Python programming language"
print(s.index("Python"))
```

**output**
0

Program Name	Error: finding sub string by using find method demo33.py
	s="Python programming language" print(s.index("Hello"))
Error	ValueError: substring not found

- ✓ So, we can handle this error in application level

Program Name	Handling error while finding sub string by using index method demo34.py
	s="Python programming language"  <b>try:</b> print(s.index("Hello"))  <b>except ValueError:</b> print("not found")
output	not found

## Counting substring in the given String:

- ✓ By using count() method we can find the number of occurrences of substring present in the string

Syntax	nameofthestring.count(substring)
--------	----------------------------------

Program Name	counting sub string by using count() method demo35.py
	s="Python programming language, Python is easy" print(s.count("Python")) print(s.count("Hello"))
output	2 0

## Replacing a string with another string:

- ✓ We can replace old string with new string by using replace() method.
- ✓ As we know string is immutable, so replace() method never perform changes on the existing string object.
- ✓ replace() method creates new string object, we can check this by using id() method

### Syntax

```
nameofthestring.replace(old string, new string)
```

**Program Name** replacing string by using replace method  
demo36.py

```
s1="Java programming language"
s2=s1.replace("Java", "Python")

print(s1)
print(s2)

print(id(s1))
print(id(s2))
```

**output**

```
Java programming language
Python programming language
49044256
48674600
```

## FAQ. String objects are immutable, Is replace () method will modify the string objects?

- ✓ Once we create a string object, we cannot change or modify the existing string object.
- ✓ This behaviour is called as immutability.
- ✓ If we are trying to change or modify the existing string object, then with those changes a new string object will be created.
- ✓ So, replace() method will create new string object with the modifications.

## Splitting of Strings:

- ✓ We can split the given string according to specified separator by using split() method.
- ✓ The default separator is space.
- ✓ split() method returns list.

## Syntax

```
nameofthestring=s.split(separator)
```

## Program Name

splitting string by using split() method

demo37.py

```
message="Python programming language"
n=message.split()
```

```
print("Before splitting: ",message)
print("After splitting: ",n)
print(type(n))
```

```
for x in n:
    print(x)
```

## output

```
Python programming language
['Python', 'programming', 'language']
<class 'list'>
Python
programming
language
```

- ✓ We can split based on separator as well

## Program Name

splitting string by using split() method

demo38.py

```
message="Python programming language,Python is easy"
n=message.split(",")
for x in n:
```

```
    print(x)
```

## output

```
['Python programming language', ' Python is easy']
```

## Joining of Strings:

- ✓ We can join a group of strings (list or tuple) with respect to the given separator.

## Syntax

```
s=separator.join(group of strings)
```

**Program Name** separator is - symbol joining string by using join() method  
demo39.py

```
profile = ['Nireekshan', 'MCA', 'Python']
candidate = '-'.join(profile)
print(profile)
print(candidate)
```

**output**

```
['Nireekshan', 'MCA', 'Python']
Nireekshan-MCA-Python
```

**Program Name** separator is : symbol joining string by using join() method  
demo40.py

```
profile = ['Nireekshan', 'MCA', 'Python']
candidate = ':'.join(profile)
print(profile)
print(candidate)
```

**output**

```
['Nireekshan', 'MCA', 'Python']
Nireekshan:MCA:Python
```

## Changing case of a String:

- ✓ We can change case of a string by using the following methods.

### 1. upper()

- ✓ This method converts all characters into upper case

### 2. lower()

- ✓ This method converts all characters into lower case

### 3. swapcase()

- ✓ This method converts all lower-case characters to upper case and all upper-case characters to lower case

### 4. title()

- ✓ This method converts all character to title case means,
  - First character in every word will be in upper case and all remaining characters will be in lower case.

## 5. capitalize()

- ✓ Only first character will be converted to upper case and all remaining characters can be converted to lower case

**Program Name**

Changing string case  
demo41.py

```
message='Python programming language'  
print(message.upper())  
print(message.lower())  
print(message.swapcase())  
print(message.title())  
print(message.capitalize())
```

**output**

```
PYTHON PROGRAMMING LANGUAGE  
python programming language  
pYTHON PROGRAMMING LANGUAGE  
Python Programming Language  
Python programming language
```

## Formatting the Strings:

- ✓ We can format the strings with variable values by using replacement operator {} and format() method.

**Program Name**

formatting the string  
demo42.py

```
name='Nireekshan'  
salary=100  
age=16
```

```
print("{} 's salary is {} and his age is {}".format(name, salary, age))  
print("{} 's salary is {} and his age is {}".format(name, salary, age))  
print("{} 's salary is {} and his age is {}".format(z=age, y=salary, x=name))
```

**output**

```
Nireekshan 's salary is 100 and his age is 16  
Nireekshan 's salary is 100 and his age is 16  
Nireekshan 's salary is 100 and his age is 16
```

## Character data type

- ✓ If a single character stands alone then we can say that is single character.
- ✓ Example:
  - To specify gender
    - M -> Male
    - F -> Female
- ✓ In other programming languages char data type represents the character type, but in python there is no specific data type to represent characters.
- ✓ We can fulfil this requirement by taking single character in single quotes.
- ✓ A single character also represents as string type in python

**Program Name** character data type program  
demo43.py

```
gender1 = 'M'  
gender2 = 'F'  
  
print(gender1)  
print(gender2)  
  
print(type(gender1))  
print(type(gender2))
```

## Output

```
M  
F  
<class 'str'>  
<class 'str'>
```

## 10. Functions

### General example why function required?

- ✓ When you go for walk in the early morning,
  - 1) Get up from the bed,
  - 2) Do fresh up,
  - 3) Tie the shoe,
  - 4) Pick the smooth towel,
  - 5) Start the walk.
- ✓ Think of this sequence of steps to do morning walk.
- ✓ Now when my dad call for morning walk means, he doesn't want to explain all these steps each time.
- ✓ Whenever dad says, "Get ready for morning walk", means he is making a function call.
- ✓ 'Morning walk' is an abstraction for all the many steps involved.

### When should we go for function

- ✓ Reason 1:
  - While writing coding logics, instead of writing like a plain text, it's good to keep those coding statements in one separate block, because whenever required then we can call these.
- ✓ Reason 2:
  - If a group of statements is repeatedly required, then it is not recommended to write these statements in every time separately.
- ✓ So, it's good to define these statements in a separate block, after defining function we can call directly if required.
- ✓ This block of statements is called as function.
- ✓ Let us understand more by doing practically.

### Function

- ✓ A function can contain group of statement which performs the task.

### Advantages

- ✓ Maintaining the code is an easy way.
- ✓ Code reusability.

### Make a note

- ✓ `print()` is predefined function in python which prints output on the console.

## Types of function

- ✓ There are two types of functions,
  - Pre-defined or built-in functions
  - User-defined functions

### Pre-defined or built-in functions

- ✓ The functions which are already coming along with installed python software

### Examples

- `id()`
- `type()`
- `input()`

### User Defined Functions:

- ✓ The functions which are defined by developer as per the requirement is called as user defined functions.

### Make a note

- ✓ Till previous chapter we discussed about pre-defined functions like,
  - `print()`
  - `id()` etc.

### Make a note

- ✓ Now we are going to discuss about user defined functions.

### Function related terminology

- **def** keyword
- name of the function
- parenthesis **()**
- parameters (if required)
- colon symbol :
- method body
- **return** type (optional)
- After defined a function we need to call the function
- If function is parameterized, then pass the values while calling.

## Function

- ✓ A function can contain mainly two parts,

- Defining function
- Calling function

### 1. Defining a function

- ✓ **def** keyword is used to define function.
- ✓ After **def** keyword we should write name of the function.
  - After function name, we should write parenthesis ()
  - After parenthesis we should write colon :
- ✓ This parenthesis may contain **parameters**.
  - Parameters are just like variables which receives the values.
  - If function having parameters, then we need to provide the values while calling.
- ✓ Function body.
  - Perform the operations.
- ✓ Before closing the function, function may contain return type.

#### Syntax

```
def function_name(parameters) :  
    """ doc string """  
    Body of the function to perform operation  
    return value (if required)
```

**Program Name** Define a function which have no parameters  
`demo1.py`

```
def display():  
    print("welcome to function")
```

#### output

#### Make a note

- ✓ When we execute above program, then it will not display any output.
- ✓ To see the output we need call the function.

## 2. Calling a function

- ✓ After defining a function, we need to call to execute the function.
- ✓ While calling the function, function name should be match otherwise we will get error.

**Program Name** Define and call the function  
demo2.py

```
def display():
    print("welcome to function concept")
```

```
display()
display()
display()
```

**output**

```
welcome to function concept
welcome to function concept
welcome to function concept
```

### Make a note

- ✓ If we call the function in **one time**, then **one-time** function will get execute.
- ✓ If we call the function in **two times**, then **two times** function will get execute.
- ✓ If we call the function in **n times**, then **n times** function will get execute.

**Program Name** **Error:** calling function name with different name  
demo3.py

```
def one():
    print("This is one function")
```

```
two()
```

**Error**

**NameError:** name 'two' is not defined

## Functions are two types

- ✓ Generally, we can divide the functions in two ways,
  - Function without parameters
  - Function with parameters

### 1. Function without parameters

- ✓ A function which have no parameters is called **function without parameters**.
- ✓ We can define a function which having no parameters.

#### Syntax

```
def nameofthefunction():
    body of the function to perform operations
```

<b>Program Name</b>	Function which having no parameters demo4.py
	# defining a function def display1(): print("Welcome to function which having no parameters")
	# calling function display1()

#### output

Welcome to function which having no parameters

### 2. Function with parameters

- ✓ A function which having parameters are called parameterised function.
- ✓ These parameters are required to process the function operations.
- ✓ When we pass parameters then,
  - Function can capture or receive the parameter's values
  - Perform the operations by using received values
  - Function can print the result.

#### Syntax

```
def NameOfTheFunction(parameter1, parameter2, ...):  
    body of the function
```

**Program Name** Function performing addition operation  
demo5.py

```
# defining the function  
  
def sum(a, b):  
    print("Sum of two values=", (a+b))  
  
#calling the function  
  
sum(10, 20)
```

**output**

```
Sum of two values =30
```

**Program Name** Check a number as it is, even or odd by using function  
demo6.py

```
def checking(num):  
  
    if num % 2 == 0:  
        print(num," is even")  
  
    else:  
        print(num," is odd")  
  
checking(12)  
checking(31)
```

**output**

```
12 is even  
31 is odd
```

## return keyword in python

- ✓ **return** is a keyword in python programming language.
- ✓ By using **return**, we can return the result.
- ✓ Function can,
  - Take input values
  - Process the logic
  - returns output to the caller with **return** statement.

### Syntax

```
def NameOfTheFunction(parameter1, parameter2, ...):  
    body of the function  
    return result
```

Program Name	function returning the value demo7.py
output	<pre>def sum(a, b):     c = a + b     return c  x=sum(1, 2) print("Sum of two numbers is: ",x)</pre> 3

## return vs None

- ✓ If any function is not return anything, then by default it returns **None** data type.
- ✓ We can also say as, if we are not writing return statement, then default return value is None

Program Name	function returning the value demo8.py
	<pre>def m1():     print("This function is returning nothing")  # function calling m1()  # function calling</pre>

```
x=m1()  
print(x)
```

## output

```
This function is returning nothing  
This function is returning nothing  
None
```

## Returning multiple values from function

- ✓ In python, a function can return multiple values.
- ✓ If a function is returning multiple values then we can write return statement as,

### Syntax

```
def nameofthefunction():  
    body of the function  
    return multiple values
```

### Program Name

Define a function which can return multiple values  
demo9.py

```
def m1(a, b):  
    c = a+b  
    d = a-b  
    return c, d  
  
#calling function  
  
x, y = m1(10, 5)  
  
print("sum of a and b: ", x)  
print("subtraction of a and b: ", y)
```

### output

```
sum of a and b: 15  
subtraction of a and b: 5
```

## Function can call other function

- ✓ We can call a function inside another function.

### Syntax

```
def firstfunction :  
    body of the first function  
  
def secondfunction :  
    body of the second function  
    we can call the first function based on requirement
```

### Program Name

One function can call another function  
demo10.py

```
def m1():  
    print("first function information")  
  
def m2():  
    print("second function information")  
  
m2()
```

### output

second function information

### Program Name

Define a function which one function can call another function  
demo11.py

```
def m1():  
    print("first function information")  
  
def m2():  
    print("second function information")  
    m1()  
  
m2()
```

### output

first function information  
second function information

## Understandings

- ✓ We defined two functions m1 and m2
- ✓ In function m2, we are calling m1 function.
- ✓ So finally, we are calling only m2 function which calls internally m1 function.

## Functions are first class objects

- ✓ Functions are considered as first-class objects.
- ✓ When we create a function, the python interpreter internally creates an object.
- ✓ In python, below things are possible to,

- Assign function to variables (demo12.py)
- Pass function as a parameter to another function (demo13.py)
- Define one function inside another function (demo15.py)
- Function can return another function (demo17.py)

### 1. Assigning a function to variable

<b>Program Name</b>	Assign a function to variable demo12.py
	<pre>def add():     print("We assigned function to variable")</pre>
	<pre>#Assign function to variable sum=add</pre>
	<pre>#calling function sum()</pre>
<b>output</b>	We assigned function to variable

### 2. Pass function as a parameter to another function.

<b>Program Name</b>	Pass function as a parameter to another function demo13.py
	<pre>def display(x):     print("This is display function")</pre>
	<pre>def message():     print("This is message function")</pre>

```
# calling function  
message()  
output  
This is message function
```

**Program Name** Pass function as a parameter to another function  
demo14.py

```
def display(x):  
    print("This is display function")  
  
def message():  
    print("This is message function")  
  
# calling function  
  
message()  
display(10)  
  
output  
This is message function  
This is display function
```

**Program Name** Pass function as a parameter to another function  
demo15.py

```
def display(x):  
    print("This is display function")  
  
def message():  
    print("This is message function")  
  
# calling function  
  
display(message())  
  
output  
This is message function  
This is display function
```

### 3. Define one function inside another function

<p><b>Program Name</b></p> <p>function inside another function demo16.py</p> <pre>def first():     print("This is outer function")      def second():         print("this is inner function")     second()  #calling outer function first()</pre> <p><b>Output</b></p> <p>This is outer function this is inner function</p>
---

#### Make a note

- ✓ If we defined inner function, then we need to call that inner function in outer function.

### 4. function can return another function

<p><b>Program Name</b></p> <p>function can return another function demo17.py</p> <pre>def first():     def second():         print("This function is return type to outer function")     return second  x=first() x()</pre> <p><b>Output</b></p> <p>This function is return type to outer function</p>
--

#### Formal and actual arguments

- ✓ When a function is defined it may have some parameters.
- ✓ These parameters receive the values.
  - Parameters are called as a '**formal arguments**'
  - When we call the function, we should pass values or data to the function.
    - These values are called as '**actual arguments**'

Program Name      Formal and actual arguments  
demo18.py

```
def sum(a, b):
    c = a + b
    print(c)

# call the function

x = 10
y = 15
sum(x, y)
```

output  
25

- ✓ **a** and **b** called as formal arguments
- ✓ **x** and **y** called actual arguments

## Types of arguments

In python there are 4 types of actual arguments are existing,

1. positional arguments
2. keyword arguments
3. default arguments
4. variable length arguments

### 1. Positional arguments

- ✓ These are the arguments passed to a function in correct positional order.
- ✓ The number of arguments and position of arguments should be matched, otherwise we will get error.

Program Name      Positional arguments  
demo19.py

```
def sub(x, y):
    print(x-y)

# calling function

sub(10, 20)
```

output  
-10

## Make a note

- ✓ If we change the number of arguments, then we will get error.
- ✓ This function accepts only two arguments then if we are trying to provide three values then we will get error.

<b>Program Name</b>	<b>Error:</b> Positional arguments demo20.py
	<pre>def sub(x, y):     print(x-y)  # calling function  sub(10, 20, 30)</pre>
<b>output</b>	<b>TypeError:</b> sub() takes 2 positional arguments but 3 were given

## 2. Keyword arguments

- ✓ Keyword arguments are arguments that recognize the parameters by the name of the parameters.
- ✓ Example: Name of the function is **cart()** and parameters are **item** and **price** can be written as:
  - **def cart(item, price)**
- ✓ At the time of calling this function, we must pass two values and we can write which value is for what by using name of the parameter, for example
  - **cart(item='bangles', price=20000)**
  - **cart(item='handbag', price=100000)**
  - **item** and **price** are called as **keywords** in this scenario.
  - Here we can change the order of arguments.

<b>Program Name</b>	keyword arguments demo21.py
	<pre>def cart(item, price):     print(item, "cost is :" ,price)  # calling function</pre>

```
cart(item='bangles', price=20000)
cart(item='handbag', price=100000)
cart(price=1200, item='shirt')
```

**output**

```
bangles cost is: 20000
handbag cost is: 100000
shirt cost is: 1200
```

**Program Name**

keyword arguments  
demo22.py

```
def details(id, name):
    print("Emp id is: ",id)
    print("Emp name is: ",name)

# calling function
details(id=1, name='Balayya Babu')
details(id=2, name='Chiru')
```

**output**

```
Emp id is: 1
Emp name is: Balayya Babu
Emp id is: 2
Emp name is: Chiru
```

**Make a note**

- ✓ We can use both positional and keyword arguments simultaneously.
- ✓ But first we must take positional arguments and then keyword arguments, otherwise we will get syntax error.

**Program Name**

positional and keyword arguments  
demo23.py

```
def details(id, name):
    print("Emp id is: ",id)
    print("Emp name is: ",name)

details(1, name='Anushka')
```

**output**

```
Emp id is: 1
Emp name is: Anushka
```

Program Name	Error: positional and keyword arguments demo24.py
Error	<pre>def details(id, name):     print("Emp id is: ",id)     print("Emp name is: ",name)  details(name='Anushka', 1)</pre> <p>SyntaxError: positional argument follows keyword argument</p>

### 3. Default arguments

- ✓ We can provide some default values for the function parameters in the definition.
- ✓ Let's take the function name as `cart(item, price=40.0)` (Example: In few shops any item is 10 rupees)
- ✓ cases
  - In above example `item` have no default value, so we should provide.
  - `price` have default value as `40`
    - Still if we provide the value at time of calling then default values will be override with passing value.

Program Name	Default arguments demo25.py
	<pre>def cart(item, price = 40.0):     print(item, "cost is: ",price)  # calling function  cart(item='pen') cart(item='handbag', price=10000) cart(price=500, item='shirt')</pre>

#### output

pen cost is: 40.0  
handbag cost is: 10000  
shirt cost is: 500

#### Make a note

- ✓ If we are not passing any value, then only default value will be considered.

## non-default argument follows default argument

- ✓ While defining a function, after default arguments we should not take non-default arguments

Program Name

SyntaxError: non-default argument follows default argument  
demo26.py

```
def m1(a=10, b):  
    print(a)  
    print(b)  
  
# calling function  
  
m1(b=20)
```

Error

SyntaxError: non-default argument follows default argument

## 4. Variable length arguments

- ✓ Sometimes, the programmer does not know how many values need to pass to function.
- ✓ In that case, the programmer cannot decide how many arguments to be given in the function definition.
  - totalcost(item1\_cost, item2\_cost) => totalcost(1, 2) # valid
  - totalcost(item1\_cost, item2\_cost) => totalcost(1, 2, 3) # Invalid
- ✓ To accept 'n' number of arguments, we need to use **variable length argument**.
- ✓ The variable length argument is an argument that can accept any number of values.
- ✓ The variable length argument is written with a '\*' (one star) before variable in function definition.

Syntax

```
def nameofthefunction(x, *y):  
    body of the function
```

- **x** is formal argument
  - **\*y** is variable length argument
- ✓ Now we can pass any number of values to this **\*y**.
  - ✓ Internally the provided values will be represented in **tuple**.

**Program Name** Variable length argument  
demo27.py

```
def totalcost(x, *y):
    sum=0
    for i in y:
        sum+=i
    print(x + sum)

#calling function

totalcost(100, 200)      # valid
totalcost(110, 226, 311) # valid
totalcost(11,)            # valid
```

**Output**

```
300
647
11
```

## keyword variable length argument (\*\*variable)

**Syntax**

```
def m1(**x):
    body of the function
```

- ✓ **\*\*x** represents as keyword variable argument
- ✓ Internally it represents like a dictionary object
- ✓ dictionary stores the data in the form of key value pairs.

**Program Name** keyword variable length argument (\*\*variable)  
demo.py

```
def print_kw_args(**kw_args):
    print(kw_args)

print_kw_args(id=1, name="Nireekshan", qualification="MCA")
```

**Output**

```
{'id': 1, 'name': 'Nireekshan', 'qualification': 'MCA'}
```

Program Name keyword variable length argument (\*\*variable)  
demo28.py

```
def m1(**x):
    for k, v in x.items():
        print(k,"=",v)
```

```
m1(a=10, b=20, c=30)
m1(id=100, name="Subbalaxmi")
```

output

```
a = 10
b = 20
c = 30
id = 100
name = Subbalaxmi
```

## Function vs Module vs Library:

- ✓ A group of lines with some name is called a function
- ✓ A group of functions saved to a file, is called Module
- ✓ A group of Modules is nothing but Library

## Types of variables

- ✓ Python supports two types of variables.
  1. Local variables
  2. Global variables

### 1. Local variables

- ✓ The variables which are declared inside of the function are called as local variable.
- ✓ We can access local variables within the function only.
- ✓ **Error:** If we are trying to access local variables in outside of the function, then we will get error.

Program Name local variables  
demo29.py

```
def m():
    a=11
    print(a)
```

```
m()
```

output

```
11
```

Program Name **NameError**: name 'a' is not defined  
demo30.py

```
def m():
    a=11
    print(a)

def n():
    print(a)

m()
n()
```

output

**NameError**: name 'a' is not defined

## 2. Global variables

- ✓ The variables which are **declared** outside of the function are called as global variable.
- ✓ Global variables can be accessed in all functions of that module.

Program Name global variables  
demo31.py

```
a=11
b=12
```

```
def m():
    print("a from function m(): ",a)
    print("b from function m(): ",b)
```

```
def n():
    print("a from function n(): ",a)
    print("b from function n(): ",b)
```

```
m()
n()
```

## output

```
a from function m(): 11  
b from function m(): 12  
a from function n(): 11  
b from function n(): 12
```

## The global keyword

We can use global keyword for the following 2 purposes:

- ✓ To declare global variable inside function
- ✓ To make global variable available to the function.

## When we can choose global keyword?

- ✓ Sometimes the global variable names and local variable names can contain same name.
- ✓ In that case, function by default refers local variable and ignores the global variable.
- ✓ So, the global variable is not accessible inside the function but still outside is accessible.

## Program Name

```
global and local variables having same name  
demo32.py  
  
a=1  
  
def m1():  
    a=2  
    print("a value from m1() function: ", a)  
  
def m2():  
    print("a value from m2() function:", a)
```

## output

```
a value from m1() function: 2  
a value from m2() function: 1
```

- ✓ If programmer wants to use the global variable inside a function, then need to use **global** keyword before the variable in the beginning of the function body.

<b>Program Name</b>	global and local variables having same name demo32.py
	a=1
	def m1(): global a a=2 print("a value from m1() function: ", a)
	def m2(): print("a value from m2() function:", a)
	m1() m2()
<b>output</b>	a value from m1() function: 2 a value from m2() function: 2

### Make a note

- ✓ If we use global keyword inside function, then function can able to read only global variable.

### Limitation

- ✓ Now, local variable is no more available now.

### Solution

- ✓ The `globals()` function will solve this problem.
- ✓ `globals()` is a built in function which returns a table of current global variables in the form of a dictionary.
- ✓ Using this function, we can refer to the global variable 'a' as: `global()['a']`.

<b>Program Name</b>	global and local variable names same demo34.py
	a=1
	def m(): a=2 print(a) print(globals()['a'])
	m()

output

2  
1

## Recursive function

- ✓ If a function calls that function itself, then it is called as recursive function

Example

$$3! = 3 * 2 * 1$$

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

```
factorial(n) = n * factorial(n-1)
```

## Advantage

- ✓ We can reduce length of the code and improves readability
- ✓ We can solve complex problems in very easy way.

Program Name factorial using recursion  
demo35.py

```
def factorial(n):
    if n ==0:
        result=1
    else:
        result = n * factorial(n-1)
    return result

x=factorial(4)
print("Factorial of 4 is: ",x)
```

output

24

Program Name Factorial without using recursive function  
demo.py

```
def f1():
    n=4
    fact=1
    while(n>0):
        fact=fact*n
        n=n-1
    print("Factorial of the number is: ", fact)

f1()
Output
Factorial of the number is: 24
```

## Anonymous functions or Lambdas

- ✓ A function without a name is called as anonymous function.
- ✓ Generally, to define normal function we need to use `def` keyword.
- ✓ To define anonymous function, we need to use `lambda` keyword.

## Lambda function

### Syntax

```
lambda argument_list: expression
```

### Advantage

- ✓ By using Lambda Functions, we can write very concise code so that readability of the program will be improved.

<b>Program Name</b>	anonymous function demo36.py
	<pre>s = lambda a: a*a</pre>
	<pre>x=s(4) print(x)</pre>
<b>output</b>	16

- ✓ Here because of lambda keyword it creates anonymous function.

## A simple difference between normal and lambda functions

**Program Name** To find square by using a normal function  
demo37.py

```
def square(t):
    return t*t

s=square(2)
print(s)
```

**output** 4

**Program Name** To find sum of two values by using normal function  
demo39.py

```
def add(x, y):
    return x + y

b=add(2, 3)
print(b)
```

**output** 5

**Program Name** To find sum of two values by using anonymous function  
demo40.py

```
add=lambda x, y: x+y

result = add(1, 2)
print('The sum of value is: ', result)
```

**output** 3

### Make notes

- ✓ Lambda Function internally returns expression value and we **no need to write return statement** explicitly.

### Where lambda function fits exactly?

- ✓ Sometimes we can pass function as argument to another function. In such cases lambda functions are best choice.

- ✓ We can use lambda functions very commonly with filter(), map() and reduce() functions, these functions expect function as argument.

## filter() function

- ✓ Based on some condition we can filter values from sequence of values.
- ✓ Where function argument is responsible to perform conditional check, sequence can be list or tuple or string.

### Syntax

```
filter(function, sequence)
```

### Program Name

example by using filter function  
demo41.py

```
items_cost = [999, 888, 1100, 1200, 1300, 777]
gt_thousand = filter(lambda x : x>1000, items_cost)

x=list(gt_thousand)
print("Eligible for discount: ",x)
```

### output

```
Eligible for discount : [1100, 1200, 1300]
```

## map() function

- ✓ The map() function can apply the condition on every element which is available in the sequence of elements.
- ✓ After applying map function can generate new group of values

### Syntax

```
map(function, sequence)
```

### Program Name

Find square by using map function  
demo42.py

```
without_gst_cost = [100, 200, 300, 400]
with_gst_cost = map(lambda x: x+10, without_gst_cost)

x=list(with_gst_cost)
```

```
print("Without GST items costs: ",without_gst_cost)
print("With GST items costs: ",x)
```

## output

```
Without GST items costs: [100, 200, 300, 400]
With GST items costs: [110, 210, 310, 410]
```

## reduce() function

- ✓ reduce() function reduces sequence of elements into a single element by applying the specific condition or logic.
- ✓ reduce() function present in **functools** module.
- ✓ So, to work with reduce we need to import **functools** module.

## Syntax

```
reduce(function, sequence)
```

## Program Name

```
reduce function
demo43.py
```

```
from functools import reduce
each_items_costs = [111, 222, 333, 444]
total_cost = reduce(lambda x, y: x+y, each_items_costs)
print(total_cost)
```

## output

```
1110
```

## Function Aliasing

- ✓ We can give another name for existing function, this is called as function aliasing.

## Program Name

```
function aliasing
demo44.py
```

```
def a():
    print("function - a")
```

```
b=a
```

```
b()
```

## output

function - a

## Function decorators

- ✓ Before understanding about function decorator, we need to understand about,
  - Nested function
    - We already discussed previously.
  - Functions are first class objects.
    - We already discussed previously

## Decorator explanation

- ✓ A decorator is a function that accepts a function as parameter and returns a function.
- ✓ Decorators are useful to perform some additional processing required by a function.

## Steps to create decorator

- ✓ **Step 1:**
  - decorator function takes parameter as another function

### Syntax

```
def decor(number):  
    body of decorator function
```

- ✓ **Step 2:**
  - We should define inner function inside the decorator function.
  - This function modifies or decorates the value of the function passed to the decorator function.

### Basic code

```
def decor(number):  
    def inner():  
        value = number()  
        return value + 2  
    return inner
```

- ✓ **Step 3:**
  - Return the inner function that has processed the value.

- ✓ Step 4:
  - After decorator created then we can call where are all required.
- ✓ While calling decorator function we need to pass parameter as a function.

## Basic code

```
def num():
    return 10

#calling decorator

result = decor(num)
print(result)
```

Program Name      Creating decorator  
demo45.py

```
def m1(number):
    def m2():
        x = number()
        return x + 2
    return m2

    def m3():
        return 10

    result = m1(m3)
    print(result())
```

## output

12

## @ symbol

- ✓ Syntactic Sugar, Python allows you to simplify the calling of decorators using the @ symbol (this is called "pie" syntax).
- ✓ To apply decorator to any function we can use the @ symbol and decorator name on top of the function name.

```
@decor  
def m3():  
    return 10
```

- ✓ When we are using @ symbol then we no need to call decorator explicitly and pass the function name.
- ✓ We can call directly like,
  - print(m3())

**Program Name** @symbol using on decorator demo46.py

```
def m1(n):  
    def m2():  
        value = n()  
        return value + 2  
    return m2  
  
@m1  
def m3():  
    return 10  
  
print(m3())
```

**output**

12

## Function generators

- ✓ Generators are just like functions that return a sequence of values.
- ✓ A generator function is written like an ordinary function, but it uses **yield** keyword.

**Program Name** Generator example dem47.py

```
def m():  
    yield 'Prasad'  
    yield 'Nireekshan'  
  
g = m()  
print(g)  
print(type(g))
```

```
for y in g:  
    print(y)  
  
output  
<generator object m at 0x000002AA28E97938>  
<class 'generator'>  
Prasad  
Nireekshan
```

Program Name Generator example  
demo48.py

```
def m(x, y):  
    while x<=y:  
        yield x  
        x+=1  
  
g = m(5, 10)  
for y in g:  
    print(y)  
  
output  
5  
6  
7  
8  
9  
10
```

### next(generator) function

- ✓ If we want to retrieve elements from generator we can use next(generator) function.

Program Name Generator example  
dem49.py

```
def m():  
    yield 'Prasad'  
    yield 'Nireekshan'  
  
g = m()  
  
print(type(g))
```

```
print(next(g))
print(next(g))
```

output

```
<class 'generator'>
Prasad
Nireekshan
```

## 11. Module

### Modules

- ✓ In python a module means, a saved python file.
- ✓ This file can contain a group of classes, methods, functions and variables.
- ✓ Every Python file mean **.py** or **.python** extension file is called as a module.

<b>Program Name</b>	module program <b>addmultiplication.py</b>
	<pre>x = 10</pre>
	<pre>def sum(a, b):</pre>
	<pre>    print("Sum of two values: ', (a+b))</pre>
	<pre>def multiplication(a, b):</pre>
	<pre>    print("Multiplication of two values: ', (a*b))</pre>

- ✓ Now **addmultiplication.py** file is a module.
- ✓ **Addmultiplication.py** module contains one variable and two functions.

### import module

- ✓ If we want to use other members(variable, function, etc) of module in our program, then we should import that module by using **import** keyword.
- ✓ After importing we can access members by using name of the module.

<b>Program Name</b>	importing <b>addmultiplication</b> module and calling members <b>demo1.py</b>
	<pre>import addmultiplication print(addmultiplication.x) addmultiplication.sum(1, 2) addmultiplication.multiplication(2, 3)</pre>
<b>output</b>	<pre>10 Sum of two values: 3 Multiplication of two values: 6</pre>

### Make a note:

- ✓ Whenever we are using a module in our program, for that module compiled file will be generated and stored in the hard disk permanently.

Renaming or aliasing a module.

## Syntax

```
import addmultiplication as dvs
```

- ✓ Here **addmultiplication** is module name and alias name is **dvs**
- ✓ We can access members by using alias name **dvs**

## Program Name

```
importing module  
demo2.py  
  
import addmultiplication as dvs  
print(dvs.x)  
dvs.sum(1, 2)  
dvs.multiplication(3, 4)
```

## output

```
10  
Sum of two values: 3  
Multiplication of two values: 12
```

**from** and **import** keywords

- ✓ We can import some specific members of module by using **from** keyword.
- ✓ The main advantage of **from** keyword is we can access members directly without using module name.

## Program Name

```
program by using from and import keywords  
demo3.py  
  
from addmultiplication import x, sum  
print(x)  
sum(10,20)
```

## output

```
10  
Sum of two values: 30
```

## Program Name

```
Error: Didn't imported multiplication member but trying to access.  
demo4.py  
  
from addmultiplication import x, sum  
print(x)  
multiplication(10,20)
```

## Error

```
10  
NameError: name 'multiplication' is not defined
```

```
import * (star symbol)
```

- ✓ We can import all members of a module as by using import \* (symbol)

**Program Name** importing by using \*  
demo5.py

```
from addmultiplication import *  
print(x)  
sum(10,20)  
multiplication(10,20)
```

**output**  
10  
Sum of two values: 30  
Multiplication of two values: 200

## member aliasing:

- ✓ We can give alias name to the members of a module

**Program Name** member aliasing  
demo6.py

```
from addmultiplication import x as y, sum as add  
print(y)  
add(10,20)
```

**output**  
10  
Sum of two values: 30

- ✓ Once we defined alias name, we should use alias name only and we should not use original name.

**Program Name** Error: should use alias name only instead of original name  
demo7.py

```
from addmultiplication import x as y  
print(x)
```

## Error

NameError: name 'x' is not defined

## Reloading a Module:

- ✓ By default, module will be loaded only once even though we are importing multiple times.

<b>Program Name</b>	reloading module module1.py
	print("This is from module1")
<b>Output</b>	This is from module1

<b>Program Name</b>	reloading module test.py
	import module1 import module1 import module1 import module1 print("This is test module")
<b>Output</b>	This is from module1 This is test module

- ✓ The problem in this approach is after loading a module if it is updated outside then updated version of **module1** is not available to our program.
- ✓ We can solve this problem by reloading module explicitly based on our requirement. We can reload by using **reload()** function of **imp** module.

<b>Syntax</b>	import imp imp.reload(module1)
---------------	-----------------------------------

**Program Name** reloading module  
reloadaddemo1.py

```
import module1
import module1
from imp import reload
reload(module1)
reload(module1)
reload(module1)
print("This is test module")
```

## output

```
This is from module1
This is from module1
This is from module1
This is from module1
This is test module
```

- ✓ In the above program module1 will be loaded 4 times in that 1 time by default and 3 times explicitly.
- ✓ The main advantage of explicit module reloading is we can ensure that updated version is always available to our program.

## dir() function

- ✓ We can find members of module by using dir() function
- ✓ For every module at the time of execution, Python interpreter will add some special properties automatically for internal use, example
  - `__name__`
  - `__package__`
- ✓ Python provides inbuilt function dir() to list out all members of current module or a specified module.
  - `dir()` =====> To list out all members of current module
  - `dir(moduleName)` =====> To list out all members of specified module

**Program Name** To list out all members of current module  
demo8.py

```
x=10
y=20
def f1():
    print("Hello")
print(dir())
```

**Output:**

```
['__annotations__',  
 '__builtins__',  
 '__cached__',  
 '__doc__',  
 '__file__',  
 '__loader__',  
 '__name__',  
 '__package__',  
 '__spec__',  
 'f1',  
 'x',  
 'y']
```

**Program Name** To display members of specific module  
**demo9.py**

```
x=10  
  
def sum(x, y):  
    print("The Sum: of", (x+y))  
  
def multiple(a, b):  
    print("The multiplication of: ", (x*y))
```

**Program Name** To display members of specific module  
**demo10.py**

```
import demo9  
print(dir(demo9))
```

**Output:**

```
['__builtins__',  
 '__cached__',  
 '__doc__',  
 '__file__',  
 '__loader__',  
 '__name__',  
 '__package__',  
 '__spec__',  
 'sum',  
 'multiple',  
 'x']
```

## The Special variable `__name__`

- ✓ For every Python program, a special variable `__name__` will be added internally.
- ✓ This variable store information regarding whether the program is executed as an individual program or as a module.

### Case 1:

- ✓ If the program executed as an individual program, then the value of this variable is `__main__`

### Case 2:

- ✓ If the program executed as a module from some other program, then the value of this variable is the name of module where it is defined.

### Conclusion

- ✓ Hence by using this `__name__` variable we can identify whether the program executed directly or as a module.

<b>Program Name</b>	By using <code>__name__</code> demo11.py
<b>output</b>	<pre>def f1():     if __name__=='__main__':         print("The code executed as a program")     else:         print("Executed as a module from some other program") f1()</pre> <p>The code executed as a program</p>

<b>Program Name</b>	importing reloaded module demo12.py
<b>output</b>	<pre>import demo11</pre>

## DVS Technologies

---

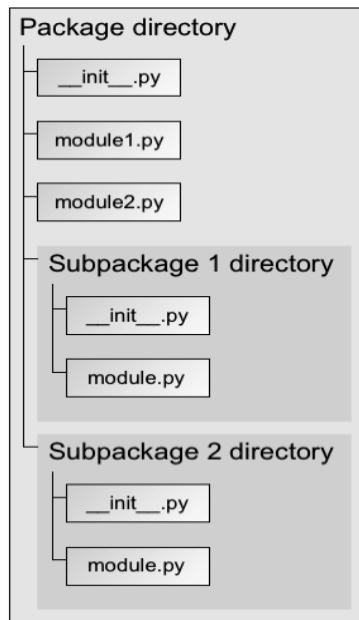
Executed as a module from some other program

DVS Technologies

## 12. Package

### What is a package?

- ✓ A package is nothing but folder or directory which represents collection of python modules(programs)
- ✓ Any folder or directory contains `__init__.py` file, is considered as a Python package. `__init__.py` can be empty file.
- ✓ A package can contain sub packages also.



### Advantage

- ✓ We can resolve naming conflicts.
- ✓ We can identify our components uniquely.
- ✓ It improves the modularity of the application.

### Example1

```
|---demo1.py  
|---demo2.py  
|---pack1  
|   |--- module1.py  
|   |--- __init__.py
```

**Program Name** Creating \_\_init\_\_.py file  
\_\_init\_\_.py

Empty file

**Program Name** executing a package  
module1.py

```
def m1():
    print("Hello this is from module1 present in pack1")
```

**Program Name** executing a package  
demo1.py

```
import pack1.module1
pack1.module1.m1()
```

**output**

Hello this is from module1 present in pack1

**Program Name** executing a package  
demo2.py

```
from pack1.module1 import m1
m1()
```

**output**

Hello this is from module1 present in pack1

## Example 2

```
|---demo3.py  
|---maindir  
|   |-----module1.py  
|   |  
|   |-----__init__.py  
|  
|   |----- subdir  
|   |       |-----module2.py  
|   |       |-----__init__.py
```

**Program Name** package \_\_init\_\_.py

Empty file

**Program Name** executing a package module1.py

```
def m1():  
    print("This is from module1 present in maindir")
```

**Program Name** executing a package module2.py

```
def m2():  
    print("This is from module2 present in maindir.subdir")
```

**Program Name** executing a package demo3.py

```
from maindir.module1 import m1
from maindir.subdir.module2 import m2
m1()
m2()
```

### output

```
This is from module1 present in maindir
This is from module2 present in maindir.subdir
```

### Make a note

- ✓ Summary diagram of library, packages, modules which contains functions, classes and variables.
  - Library - A group of packages
  - Package - A group of modules
  - Modules - A group of functions or
  - Modules - A group of classes and methods

## 13. list data structure - []

Why should we learn about data structures?

- ✓ In any real-time projects, the common requirement is as below operations,
  - Storing
  - Searching
  - Retrieving
  - Deleting
  - Processing
  - Duplicate
  - Ordered
  - Unordered
  - Size
  - Capacity
  - Sorting
  - Unsorting
  - Random access
  - Keys
  - Values
  - Key – value pairs
- ✓ To learn about those operations, you should learn python data structures.

### Python Data structures

- ✓ If you want to store a group of individual objects as a single entity, then you should go for data structures.

### Sequence of elements

- ✓ A sequence is a datatype that can contains a group of elements.
- ✓ The purpose of any sequence is to store and process a group of elements.
- ✓ In python, strings, lists, tuples and dictionaries are very important sequence datatype.

### list data structure

- ✓ A list can store group of objects or elements.
- ✓ list can store same type as well as different types of elements.
- ✓ list having dynamic nature means size will increase dynamically.
- ✓ **Insertion** order is preserved.

- The order in which elements are added to the list is the order in which the output displays.
  - Means
    - Input => [10, 20, 30]
    - Output => [10, 20, 30]
  - Elements printed in order way of how the elements inserted or added.
- 
- ✓ List can store duplicates elements.
  - ✓ In list index plays main role.
  - ✓ List objects are mutable means we can modify or change the content of the list.
  - ✓ Python supports both positive and negative indexes.
    - Positive index means from left to right
    - Negative index means right to left

## Creating list

- ✓ We can create list by using square brackets []
- ✓ Inside list, elements will be separated by comma separator

### 1. creating empty list

Program Name	creating empty list demo1.py
Output	I = [] print(I) print(type(I))  [] <class 'list'>

### 2. Creating list with elements

- ✓ We can create list directly with elements.

Program Name	creating list with same type of elements with duplicates demo2.py
Output	names = ['Mohan', 'Prasad', 'Ramesh', 'Mohan'] print(names)  ['Mohan', 'Prasad', 'Ramesh', 'Mohan']

**Program Name** creating list with different type of elements  
demo3.py

```
student_info = ['Mohan', 10, 35]
print(student_info)
```

**Output** ['Mohan', 10, 35]

## Make a note

- ✓ Observe the above programs output,
  - Order is preserved
  - Duplicates are allowed.

### 3. Creating list by using list() function

- ✓ We can create list by using list() function.

**Program Name** creating list by using list() function  
demo4.py

```
r=range(0, 10)
l=list(r)
print(l)
```

**output**

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## Difference between **list()** function and **list class**

- ✓ list() is predefined function in python.
- ✓ list() function is used to create a list of elements or objects, it takes sequence as parameter.
- ✓ list is predefined class in python.
- ✓ Once if we created a list then internally it represents as list class type.
- ✓ We can check this by using type function.

**Program Name** creating list by using list() function  
demo5.py

```
r=range(0, 10)
l=list(r)
```

```
print(l)
print(type(l))
output
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
<class 'list'>
```

## list having mutable nature

- Once we created a list object then we can change or modify the elements in the existing list object.
- So, list having mutable nature.

**Program Name** list having mutable nature  
demo6.py

```
I = [1, 2, 3, 4, 5]
print(I)
print("Before modifying I[0] : ",I[0])
I[0]=20
print("After modifying I[0] : ",I[0])
print(I)
```

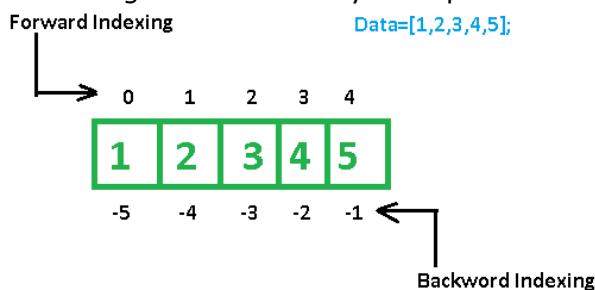
**output**
[1, 2, 3, 4, 5]
Before modifying I[0] : 1
After modifying I[0] : 20
[20, 2, 3, 4, 5]

## Accessing elements from list

- We can access elements from list by using,
  - Index.
  - slice operator.
  - loops

### 1. By using index

- index** represents accessing the elements by their position numbers in the list.



- ✓ **Indexing** represents accessing the elements by their position numbers in the list.
- ✓ Index starts from **0** onwards.
- ✓ List supports both positive and negative indexes.
  - Positive index represents from left to right direction
  - Negative index represents from right to left.
- ✓ If we are trying to access beyond the range of list index, then we will get error like **IndexError**.

**Program****Name**

```
list indexing  
demo7.py  
  
names = ['Nireekshan', 'Prasad', 'Ramesh']  
  
print(names)  
  
print(names[0])  
print(names[1])  
print(names[2])  
  
print(type(names))  
  
print(type(names[0]))  
print(type(names[1]))  
print(type(names[2]))
```

**output**

```
['Nireekshan', 'Prasad', 'Ramesh']  
Nireekshan  
Prasad  
Ramesh  
<class 'list'>  
<class 'str'>  
<class 'str'>  
<class 'str'>
```

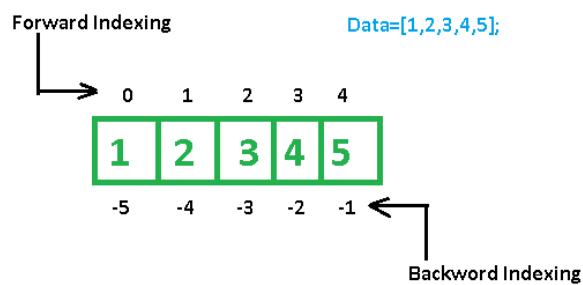
**Program  
Name****IndexError: list index out of range**  
demo8.py

```
names = ['Nireekshan', 'Prasad', 'Ramesh']  
  
print(names)  
  
print(names[0])  
print(names[1])  
print(names[2])
```

```
print(names[30])  
  
output  
  
['Nireekshan', 'Prasad', 'Ramesh']  
Nireekshan  
Prasad  
Ramesh  
IndexError: list index out of range
```

## 2. Slicing

- ✓ Slicing represents extracting a piece of the list from already created list



### Syntax

```
[start: stop: stepsize]
```

- ✓ start
  - It indicates the index where slice can start, default value is 0
- ✓ stop
  - It indicates the index where slice can end, default value is max allowed index of list i.e. length of the list
- ✓ stepsize
  - increment value, default value is 1

**Program Name** Slice example  
demo9.py

```
n = [1, 2, 3, 4, 5, 6]
print(n)

print(n[2:5:2])
print(n[4::2])
print(n[3:5])
```

**output**

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

### 3. Accessing list by using loops

- ✓ We can access elements from list by using **for** and **while** loops.

**Program Name** accessing elements from list by using for loop  
demo10.py

```
a = [100, 200, 300, 400]

for x in a:
    print(x)
```

**output**

```
100
200
300
400
```

**Program Name** accessing elements from list by using while loop  
demo11.py

```
a = [100, 200, 300, 400]

x = 0
while x<len(a):
    print(a[x])
    x = x+1
```

## output

```
100  
200  
300  
400
```

## Important functions and methods in list

1. len() function
2. count() method
3. append() method
4. insert() method
5. extend() method
6. remove() method
7. pop() method

### 1. len() function

- ✓ This function returns the number of elements present in the list.

**Program Name** To find length of list  
`demo12.py`

```
n = [1, 2, 3, 4, 5]  
print(len(n))
```

**Output**

```
5
```

### 2. count() method

- ✓ This method returns the number of occurrences of specified item in the list

**Program Name** To find count of specific element in list  
`demo13.py`

```
n = [1, 2, 3, 4, 5, 5, 3]  
print(n.count(5))  
print(n.count(3))  
print(n.count(2))
```

**output**

```
3  
2  
1
```

### 3. append method

- ✓ We can add elements to the list object by using append() method.
- ✓ append() method will add the elements to list at the end of the list.

**Program Name** appending elements into list  
demo14.py

```
l=[]
l.append("Nireekshan")
l.append("Ramesh")
l.append("Prasad")
print(l)
```

**output**

```
['Nireekshan', 'Ramesh', 'Prasad']
```

### 4. insert() method:

- ✓ We can add elements to the list object by using insert() method.
- ✓ insert() method will add the elements to list at the specified index.

**Program Name** inserting elements into list  
demo15.py

```
n=[10, 20, 30, 40, 50]
n.insert(0, 76)
print(n)
```

**output**

```
[76, 10, 20, 30, 40, 50]
```

append()	insert()
In List when we add any element it will be add into last element.	In List we can insert any element in specified index number

## Make a note

- ✓ While you are inserting,
  - If the specified index is greater than max index, then element will be inserted at last position.
  - If the specified index is smaller than min index, then element will be inserted at first position.

### Program Name

inserting elements into list  
demo16.py

```
l=[10, 20, 30, 40]
print(l)          # [10, 20, 30, 40]
l.insert(1, 111)
print(l)          # [10, 111, 20, 30, 40]
l.insert(-1, 222)
print(l)          # [10, 111, 20, 30, 222, 40]
l.insert(10, 333)
print(l)          # [10, 111, 20, 30, 222, 40, 333]
l.insert(-10, 444)
print(l)          # [444, 10, 111, 20, 30, 222, 40, 333]
```

### Output

```
[10, 20, 30, 40]
[10, 111, 20, 30, 40]
[10, 111, 20, 30, 222, 40]
[10, 111, 20, 30, 222, 40, 333]
[444, 10, 111, 20, 30, 222, 40, 333]
```

## 5. extend() method:

- ✓ We can add all items of one list to another list

### Syntax

l2.extend(l1)

## Make a note

- ✓ All items present in l1 will be added to l2, finally l2 holding total l1 and l2 items

**Program Name** adding one list elements into another list  
demo17.py

```
I1 = [1, 2, 3]
I2 = ['Nireekshan', 'Ramesh', 'Arjun']
print("Before extend I1", I1)
print("Before extend I2", I2)

I2.extend(I1)

print("After extend I1", I1)
print("After extend I2", I2)
```

**output**

```
Before extend I1 is: [1, 2, 3]
Before extend I2 is: ['Nireekshan', 'Ramesh', 'Arjun']
After extend I1 is: [1, 2, 3]
After extend I2 is: ['Nireekshan', 'Ramesh', 'Arjun', 1, 2, 3]
```

**Program Name** adding one list elements into another list  
demo17a.py

```
august_txns = [100, 200, 500, 600, 400, 500, 900]
sept_txns = [111, 222, 333, 600, 790, 100, 200]
print("August month transactions are : ",august_txns)
print("September month transactions are : ",sept_txns)
sept_txns.extend(august_txns)
print("August and Sept total transactions amount: ",sum(sept_txns))
```

**output**

```
August month transactions are : [100, 200, 500, 600, 400, 500, 900]
September month transactions are : [111, 222, 333, 600, 790, 100, 200]
August and Sept total transactions amount: 5556
```

## 6. remove() method:

- ✓ We can use this method to remove specified item from the list.

**Program Name** Removing element from list  
demo18.py

```
n=[1, 2, 3]
n.remove(1)
print(n)
```

output

```
[2, 3]
```

- ✓ If the item exists in multiple times, then only first occurrence will be removed.

Program Name Removing element from list  
demo19.py

```
n=[1, 2, 3, 1]
n.remove(1)
print(n)
```

output

```
[2, 3, 1]
```

- ✓ If the specified item not present in list, then we will get **ValueError**

Program Name **ValueError**: list.remove(x): x not in list  
demo20.py

```
n=[1, 2, 3, 1]
n.remove(10)
print(n)
```

Error

**ValueError**: list.remove(x): x not in list

Make a note

- ✓ Before removing elements its good approach to check the element is exists or not to avoid error if element not exists.

## 7. pop() method:

- ✓ This method removes and returns the last element of the list.

Program Name remove the last element from list by using pop method  
demo21.py

```
n=[1, 2, 3]
print(n.pop())
print(n)
```

## output

```
3  
[1, 2]
```

- ✓ While removing if the list is empty then pop() method raises IndexError

**Program Name**      **IndexError:** pop from empty list  
demo22.py

```
n=[]  
print(n.pop())  
print(n)
```

**Error**      **IndexError:** pop from empty list

## Difference between remove() and pop() methods

remove()	pop()
✓ To remove based on index.	✓ To remove last element from list.
✓ It can't return any value.	✓ It returned removed element.
✓ While removing if element not found then we get value error.	✓ While applying pop() method if list is empty, then we get Error.

## Ordering elements of List:

### 1. reverse():

- ✓ This method reverse the order of list elements.

**Program Name**      reverse of the list  
demo23.py

```
n=[1, 2, 3, 4]  
print(n)  
n.reverse()  
print(n)
```

**output**

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

**2. sort() method:**

- ✓ In list by default insertion order is preserved.
- ✓ If we want to sort the elements of list according to default natural sorting order then we should go for sort() method.
- ✓ For numbers the default natural sorting order is ascending order.
- ✓ For strings the default natural sorting order is alphabetical order

**Program Name** sorting the numbers and names  
demo24.py

```
n=[1, 4, 5, 2, 3]  
n.sort()  
print(n)  
  
s=['Nireekshan', 'Ramesh', 'Arjun']  
s.sort()  
print(s)
```

**output**

```
[1, 2, 3, 4, 5]  
['Arjun', 'Nireekshan', 'Ramesh']
```

**Make a note**

- ✓ To use sort() method, compulsory list should contain only homogeneous elements. otherwise we will get TypeError

**Program Name** **TypeError**: '<' not supported between instances of 'int' and 'str'  
demo25.py

```
s=['Nireekshan', 'Ramesh', 'Arjun', 3, 6]  
s.sort()  
print(s)
```

**Error**

**TypeError**: '<' not supported between instances of 'int' and 'str'

## Aliasing and Cloning of list objects

- ✓ The process of giving a new name to an existing list is called aliasing.
- ✓ The new name is called as alias name.
- ✓ Both names refer to single object.
- ✓ Any modifications done on existing object then that updated object will be referred with both original and alias names.

Program Name aliasing list demo26.py

```
x=[10, 20, 30]
y=x

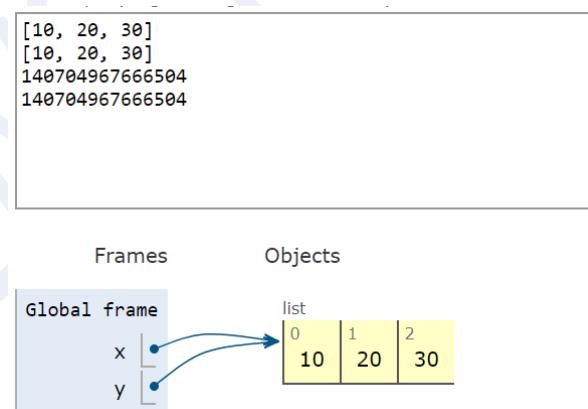
print(x)
print(y)

print(id(x))
print(id(y))
```

output

```
[10, 20, 30]
[10, 20, 30]
140704967666504
140704967666504
```

## Aliasing list visualization



Program Name after modification  
demo27.py

```
x=[10, 20, 30]
y=x

print(x)
print(y)

print(id(x))
print(id(y))

x[1] = 99

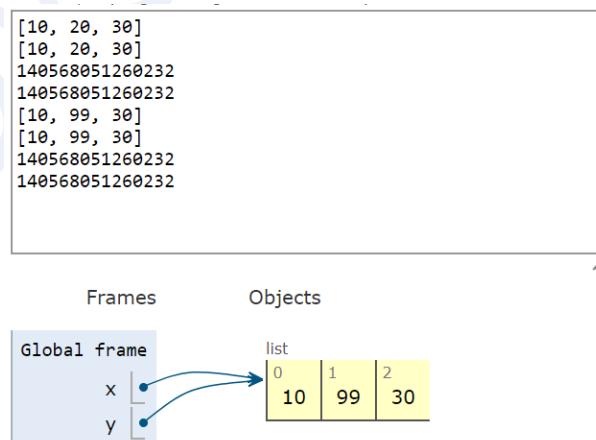
print(x)
print(y)

print(id(x))
print(id(y))
```

output

```
[10, 20, 30]
[10, 20, 30]
140631732596552
140631732596552
[10, 99, 30]
[10, 99, 30]
140631732596552
140631732596552
```

after modification visualization



## Make a note

- o x is original name
- o y is alias name
- o Any modifications done in x will applicable for y, vice versa.

## Cloning or Copying

- ✓ The process of creating exact duplicate independent object is called cloning.
- ✓ We can implement cloning by using `slice` operator or by using `copy()` method.
- ✓ Now these two are independent, applying any modifications on one will not impact the other.

## By using slice operator

Program Name before modification  
demo28.py

```
x=[10, 20, 30]
y=x[:]
```

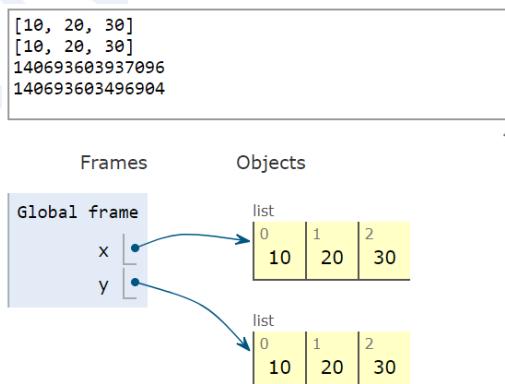
```
print(x)
print(y)
```

```
print(id(x))
print(id(y))
```

## output

```
[10, 20, 30]
[10, 20, 30]
1421938150664
1421938150728
```

## Visualization



Program Name after modification  
demo29.py

```
x=[10, 20, 30]  
y=x[:]
```

```
print(x)  
print(y)
```

```
print(id(x))  
print(id(y))
```

```
x[1] = 99
```

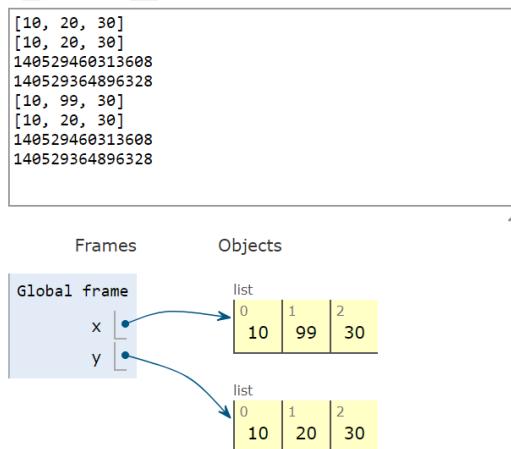
```
print(x)  
print(y)
```

```
print(id(x))  
print(id(y))
```

## output

```
[10, 20, 30]  
[10, 20, 30]  
2342978209544  
2342978209736  
[10, 99, 30]  
[10, 20, 30]  
2342978209544  
2342978209736
```

## visualization



## By using copy method

<b>Program Name</b>	copy method demo30.py
	x=[10, 20, 30] y=x.copy()
	print(x) print(y)
	print(id(x)) print(id(y))
<b>output</b>	[10, 20, 30] [10, 20, 30] 2535062736648 2535062736712

## Mathematical + and \* operators

### Concatenation operator +

- ✓ '+' operator concatenate two list objects to join them and returns single list.

<b>Program Name</b>	+ operator concatenates the lists demo31.py
	a= [1, 2, 3] b= [4, 5, 6] c = a + b print(c)
<b>output</b>	[1, 2, 3, 4, 5, 6]

## Make a note

- ✓ If two objects are list type than only + operators concatenate them otherwise we will get error **TypeError**

**Program Name**      **Error** while concatenates list and normal object  
demo32.py

```
a= [1, 2, 3]
b= 'nireekshan'
c = a + b
print(c)
```

**Error**

**TypeError:** can only concatenate list (not "str") to list

**Program Name**      + operator concatenates the lists  
demo33.py

```
a= [1, 2, 3]
b= ['nireekshan']
c = a + b
print(c)
```

**output**

[1, 2, 3, 'nireekshan']

## Repetition operator \*

- ✓ '\*' operator works to repetition of elements in the list.

**Program Name**      \* operator repetition the lists  
demo34.py

```
a = [1, 2, 3]
print(a)
print(2*a)
```

**output**

[1, 2, 3]
[1, 2, 3, 1, 2, 3]

## Comparison of lists

- ✓ We can use comparison operators for List objects.
- ✓ Whenever we are using relational operators (<, <=,>,>=) between List objects then,
  - Initially the first two items are compared, and if they differ this determines the outcome of the comparison;
  - if they are equal, the next two items are compared, and so on
- ✓ Reference url:<https://docs.python.org/3/tutorial/datastructures.html#comparing-sequences-and-other-types>

**Program Name** comparing the two lists  
demo35.py

```
print([1, 2, 3] < [2, 2, 3])      # True
print([1, 2, 3] < [1, 2, 3])      # False
print([1, 2, 3] <= [1, 2, 3])     # True
print([1, 2, 3] < [1, 2, 4])      # True
print([1, 2, 3] < [0, 2, 3])      # False
print([1, 2, 3] == [1, 2, 3])      # True
```

**Output**

```
True
False
True
True
False
True
```

While comparing lists which are loaded with strings then

- ✓ The number of elements
- ✓ The order of elements
- ✓ The content of elements (case sensitive)

**Program Name** list comparison  
demo35a.py

```
x =["abc", "def", "ghi"]
y =[ "abc", "def", " ghi"]
z =[ "ABC", "DEF", "GHI"]
a =[ "abc", "def", "ghi", "jkl"]

print(x==y)      #      True
print(x==z)      #      False
print(x==a)      #      False
```

**Output**

True  
False  
False

## Membership operators

- ✓ We can check if the element is a member of a list or not by using membership operators those are,
  - **in** operator
  - **not in** operator
- ✓ If the element is member of list, then **in** operator returns True otherwise False.
- ✓ If the element is not in the list, then **not in** operator returns True otherwise False

Program Name	Membership operators demo36.py
	<pre>x=[10, 20, 30, 40, 50]  print(20 in x)      # True print(20 not in x) # False  print(90 in x)      # False print(90 not in x) # True</pre>
output	<pre>True False False True</pre>

## Nested Lists

- ✓ A list within another list is called nested list.
- ✓ We can take a list as an element in another list.

Program Name	Nested lists demo37.py
	<pre>a = [80, 90] b = [10, 20, 30, a]  print(b[0]) print(b[1]) print(b[2])</pre>

```
print(b[3])
```

output

```
10  
20  
30  
[80, 90]
```

**Program Name** Accessing nested lists  
demo38.py

```
a = [80, 90]  
b = [10, 20, 30, a]
```

```
for x in b[3]:  
    print(x)
```

output

```
80  
90
```

**Program Name** Accessing nested lists  
demo39.py

```
l=[5,10, [25, 35]]  
print(l)  
print(l[0])  
print(l[2])  
print(l[2][0])  
print(l[2][1])
```

output

```
[5, 10, [25, 35]]  
5  
[25, 35]  
25  
35
```

## list comprehensions

- ✓ List comprehensions represents creating new lists from iterable object like a list, set, tuple, dictionary and range.
- ✓ List comprehensions code is very concise way.

## Syntax

```
list = [expression for item1 in iterable1 if statement]
```

- ✓ Here Iterable represents a list, set, tuple, dictionary or range object.
- ✓ The result of list comprehension is new list based on the applying conditions.

**Program Name** square numbers from 1 to 10 by using list comprehension  
demo40.py

```
squares = [x**2 for x in range(1, 11)]  
print(squares)
```

**output**

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

**Program Name** list comprehension  
demo41.py

```
s=range(1, 20, 3)  
for i in s:  
    print(i)  
  
m=[x for x in s if x%2==0]  
print(m)
```

**output**

```
1  
4  
7  
10  
13  
16  
19  
[4, 10, 16]
```

## 14. Tuple Data structure - ()

- ✓ Tuple is same as list but only the difference is, tuple is immutable, and list is mutable.
- ✓ **Immutable**
  - Once if we create a tuple object, then we cannot modify the content of tuple object.
- ✓ **Insertion order** is preserved
  - The order in which elements are added to the tuple is the order in which the output displays.
- ✓ Tuple can store duplicates.
- ✓ Tuple can store same type of objects.
- ✓ Tuples can store different type of objects.
- ✓ Index plays main role in tuple.
- ✓ Tuple support both positive and negative index.
  - Positive index means forward direction (from left to right)
  - Negative index means backward direction (from right to left)
- ✓ We can create tuple by using parenthesis **()** symbol.
  - Parenthesis is optional but its recommended to use.
- ✓ Inside tuple every object can be separated by comma separator.

When should we go for tuple data structure?

- ✓ If we are going to define a data which never change over all the period, then we should go for tuple data structure.

Example:

1. Week days names
2. Month names
3. Year names

<b>Program Name</b>	Tuple having same type of objects demo1.py
	employee_ids = (1, 2, 3, 4, 5) print("same type of objects:", employee_ids) print(type(employee_ids))
<b>Output</b>	tuple having same type of objects: (1, 2, 3, 4, 5) <class 'tuple'>

Make a note

- ✓ When you are creating tuple then parenthesis is optional

**Program Name** Parenthesis is optional for tuple  
demo2.py

```
employee_ids = 1, 2, 3, 4, 5
print(employee_ids)
print(type(employee_ids))
```

**Output**

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

**Program Name** Different elements in tuple  
demo3.py

```
employee_details = (1, 'Nireekshan', 1000.123)
print("different type of objects:", employee_details)
```

**Output**

```
different type of objects: (1, 'Nireekshan', 1000.123)
```

## Single value tuple

- ✓ If tuple having only one object, then that object should end with comma separator otherwise python internally not considered as it is tuple.

**Program Name** Tuple single value ends with comma separator otherwise it's not tuple  
demo4.py

```
name = ('nireekshan')
print(name)
print(type(name))
```

**Output**

```
nireekshan
<class 'str'>
```

**Program Name** Tuple single value ends with comma separator otherwise it's not tuple  
demo5.py

```
id = (11)
print(id)
```

```
print(type(id))
```

**Output**

```
(11)  
<class 'int'>
```

**Program Name**

Tuple single value ends with comma separator it's tuple  
demo6.py

```
name = ('nireekshan',)  
print(name)  
print(type(name))
```

**Output**

```
('nireekshan')  
<class 'tuple'>
```

Different ways to create a tuple.

**1. empty tuple**

- ✓ We can create an empty tuple by using empty parenthesis.

**Program Name**

empty tuple  
demo7.py

```
emp_id = ()  
print(emp_id)  
print(type(emp_id))
```

**output**

```
()  
<class 'tuple'>
```

**2. Single value tuple**

- ✓ We can create single value tuple.
- ✓ Parenthesis is optional.
- ✓ In tuple, single value should end with comma separator.

**Program Name**

Tuple example  
demo8.py

```
emp_id = (11,)  
std_id = 120,
```

```
print(emp_id)
print(std_id)
```

output

```
(11,
120
```

### 3. Tuple with group of values

- ✓ Tuple can contain group of objects; those objects can be same type or different type.
- ✓ While creating tuple with group of values parentheses are optional

Program Name Tuple example  
demo9.py

```
emp_id = (11, 12, 13)
std_id = 120, 130, 140
print(emp_id)
print(std_id)
```

output

```
(11, 12, 13)
(120, 130, 140)
```

### 4. By using tuple() function

- ✓ We can create tuple by using tuple() function.

Program Name Creating tuple by using tuple function  
demo10.py

```
l = [11, 22, 33]
t=tuple(l)
print(t)
```

output

```
(11, 22, 33)
```

Program Name Tuple example  
demo11.py

```
t=tuple(range(1, 10, 2))
print(t)
```

## output

(1, 3, 5, 7, 9)

## Accessing elements of tuple:

- ✓ We can access tuple elements by using,
  - Index
  - Slice operator

## Index

- ✓ Index means position where element stores

### Program Name

Accessing tuple by using index  
demo12.py

```
t=(10,20,30,40,50,60)
print(t[0])          #    10
print(t[-1])         #    60
#print(t[100])       #    IndexError: tuple index out of range
```

### Output

10  
60

## slice operator:

- ✓ A group of objects from starting point to ending point

### Program Name

Accessing tuple by using slice  
demo13.py

```
t=(10,20,30,40,50,60)
print(t[2:5])
print(t[2:100])
print(t[::-2])
```

### Output

30, 40, 50)  
(30, 40, 50, 60)  
(10, 30, 50)

## Tuple vs immutability:

- ✓ Tuple having immutable nature.
- ✓ If we create a tuple then we cannot modify the elements of existing tuple.

**Program Name** Prove tuple having immutable nature  
demo14.py

```
t=(10,20,30,40)
print(t[1])
t[1]=70
```

### Output

```
20
TypeError: 'tuple' object does not support item assignment
```

## Mathematical operators on tuple:

- ✓ We can apply plus (+) and Multiplication (\*) operators on tuple.
- ✓ + operator works as concatenation.
- ✓ \* operator works as multiplication.

### Concatenation operator (+):

- ✓ + operator concatenates two tuples and returns single tuple

**Program Name** Concatenation operator on tuple  
demo15.py

```
t1=(10,20,30)
t2=(40,50,60)
t3=t1+t2
print(t3)
```

### Output

```
(10,20,30,40,50,60)
```

### Multiplication operator (\*)

- ✓ Multiplication operator works as repetition operator

<b>Program Name</b>	Repetition operator on tuple demo16.py
	<pre>t1=(10,20,30) t2=t1*3 print(t2)</pre>
<b>Output</b>	(10, 20, 30, 10, 20, 30, 10, 20, 30)

## Important functions and methods of Tuple:

### 1. len() function

- ✓ To return number of elements present in the tuple

<b>Program Name</b>	len() function demo17.py
	<pre>t=(10,20,30,40) print(len(t))</pre>
<b>Output</b>	4

### 2. count() method

- ✓ To return number of occurrences of given element in the tuple

<b>Program Name</b>	count() method demo18.py
	<pre>t=(10,20,10,10,20) print(t.count(10))</pre>
<b>Output</b>	3

### 3. index() method

- ✓ returns index of first occurrence of the given element.
- ✓ If the specified element is not available, then we will get **ValueError**.

Program Name	index() method demo19.py
	t=(10,20,10,10,20) print(t.index(10)) # print(t.index(30)) # ValueError
Output	0

Program Name	index() method demo20.py
	t=(10,20,10,10,20) print(t.index(30))
Output	ValueError: tuple.index(x): x not in tuple

#### 4. sorted() function

- ✓ This function can sort the elements which is default natural sorting order.

Program Name	sorted() function demo21.py
	t=(40,10,30,20) t1=sorted(t) print(t) print(t1)
Output	(40, 10, 30, 20) [10, 20, 30, 40]

- ✓ We can sort according to reverse of default natural sorting order as follows

Program Name	sorted() function demo22.py
	t=(40,10,30,20) t1=sorted(t, reverse=True) print(t1)

## Output

```
[40, 30, 20, 10]
```

## 5. min() and max() functions:

- ✓ These functions return min and max values according to default natural sorting order.

<b>Program Name</b>	min() and max() functions demo23.py
	<pre>t=(40,10,30,20) print(min(t))      #    10 print(max(t))      #    40</pre>
<b>output</b>	<pre>10 40</pre>

## Tuple packing

- ✓ We can create a tuple by packing a group of variables.

<b>Program Name</b>	Tuple packing demo24.py
	<pre>a=10 b=20 c=30 d=40 t=a, b, c, d print(t)</pre>
<b>output</b>	<pre>(10, 20, 30, 40)</pre>

- ✓ Here a, b, c, d are packed into a tuple t.
- ✓ This is nothing but tuple packing.

## Tuple unpacking:

- ✓ Tuple unpacking is the reverse process of tuple packing, we can unpack a tuple and assign its values to different variables

<b>Program Name</b>	Tuple unpacking demo25.py
	<pre>t=(10, 20, 30, 40) a, b, c, d = t print("a=",a , "b=" , b," c=", c , "d=",d)</pre>
<b>output</b>	a= 10 b= 20 c= 30 d= 40

## Make a note

- ✓ At the time of tuple unpacking the number of variables and number of values should be same, otherwise we will get ValueError.

<b>Program Name</b>	<b>ValueError</b> : too many values to unpack demo25a.py
	<pre>t=(10, 20, 30, 40) a, b, c=t</pre>
<b>Error</b>	<b>ValueError</b> : too many values to unpack (expected 3)

## Tuple comprehension

- ✓ Tuple comprehension is not supported by Python.

## Example

```
t= (x**2 for x in range (1,6))
```

- ✓ Here we are not getting tuple object and we are getting generator object.

<b>Program Name</b>	Tuple comprehension demo26.py
	<pre>t= ( x**2 for x in range(1,6))</pre>

```

print(type(t))
for x in t:
    print(x)

```

**Output**

```

<class 'generator'>
1
4
9
16
25

```

**Differences between List and Tuple:**

- ✓ List and Tuple are exactly same except small difference: List objects are mutable whereas Tuple objects are immutable.
- ✓ In both cases insertion order is preserved, duplicate objects are allowed, heterogenous objects are allowed, index and slicing are supported.

<b>List</b>	<b>Tuple</b>
<ul style="list-style-type: none"> <li>✓ List is a Group of Comma separated Values within Square Brackets and Square Brackets are mandatory.</li> <li>✓ Example: i = [10, 20, 30, 40]</li> </ul>	<ul style="list-style-type: none"> <li>✓ Tuple is a Group of Comma separated Values within Parenthesis and Parenthesis are optional.</li> <li>✓ Example: t = (10, 20, 30, 40)</li> <li>✓ Example: t = 10, 20, 30, 40</li> </ul>
<ul style="list-style-type: none"> <li>✓ List Objects are Mutable i.e. once we create List Object we can perform any changes in that Object.</li> <li>✓ Example: i[1] = 70</li> </ul>	<ul style="list-style-type: none"> <li>✓ Tuple Objects are Immutable i.e. once we create Tuple Object we cannot change its content.</li> <li>✓ Example: t [1] = 70</li> <li>✓ <b>ValueError</b>: tuple object does not support item assignment.</li> </ul>
<ul style="list-style-type: none"> <li>✓ If the Content is not fixed and keep on changing, then we should go for List.</li> </ul>	<ul style="list-style-type: none"> <li>✓ If the content is fixed and never changes then we should go for Tuple.</li> </ul>
<ul style="list-style-type: none"> <li>✓ List Objects cannot be used as Keys for Dictionaries because Keys should be Hashable and Immutable.</li> </ul>	<ul style="list-style-type: none"> <li>✓ Tuple Objects can be used as Keys for Dictionaries because Keys should be Hashable and Immutable.</li> </ul>

## 15. Set Data Structure

### set data structure

- ✓ If we want to represent a group of **unique** values as a single entity, then we should go for set.
- ✓ Set cannot store duplicate elements.
- ✓ Insertion order is not preserved.
- ✓ Indexing and slicing are not allowed for the set.
- ✓ Set can store same and different type of elements or objects.
- ✓ Set objects are mutable means once we create set object we can perform any changes in existing object.

### What symbol is required to create set?

- ✓ To create list, we need to use square bracket symbols : []
- ✓ To create tuple, we need to use parenthesis : ()
- ✓ So, to create **set** we need to use **curly braces** : {}

### Set elements separated by what?

- ✓ We can create set by using curly braces {} and all elements separated by comma separator in set.

### Creating set by using same type of elements

**Program Name** creating same type of elements by using set  
demo1.py

```
s = {10,20,30,40}  
print(s)  
print(type(s))
```

**Output**  
{40, 10, 20, 30}  
<class 'set'>

### Creating set by using different type of elements

**Program Name** creating different type of elements by using set  
demo2.py

```
s = {10, 'Ramesh', 30.9, 'Mohan' ,40}  
print(s)  
print(type(s))
```

**Output**  
{40, 10, 'Ramesh', 'Mohan', 30.9}

```
<class 'set'>
```

## Creating set by range() type of elements

**Program Name** creating set by using range()  
demo3.py

```
s=set(range(5))  
print(s)
```

**Output**

```
{0, 1, 2, 3, 4}
```

## Make a note

- ✓ Observe the above programs output,
  - Order is not preserved
  - Duplicates are not allowed.

**Program Name** set not allowed duplicates  
demo4.py

```
s = {10, 20, 30, 40, 10, 10}  
print(s)  
print(type(s))
```

**Output**

```
{40, 10, 20, 30}  
<class 'set'>
```

## Difference between **set()** function and **set class**

- ✓ **set()** is predefined function in python.
- ✓ **set()** function is used to create a set of elements or objects, it takes sequence as parameter.
- ✓ **set** is predefined class in python.
- ✓ Once if we created a set then internally it represents as set class type.
- ✓ We can check this by using type function.

**Program Name** creating set by using set() function  
demo5.py

```
r=range(0, 10)
s=set(r)
print(s)
print(type(s))
```

## Output

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
<class 'set'>
```

## Empty set

- ✓ We can create empty set.
- ✓ While creating empty set compulsory we should use **set()** function.
- ✓ If we didn't use set function, then it treats as dictionary instead of set.

**Program Name** Trying to create empty set with empty curly braces, but its not set.  
demo6.py

```
s={}
print(s)
print(type(s))
```

## Output

```
{}
<class 'dict'>
```

**Program Name** Creating empty set by using set() function  
demo7.py

```
s=set()
print(s)
print(type(s))
```

## Output

```
set()
<class 'set'>
```

## Important function and methods of set:

### 1. add(x) method:

- ✓ add() method can add element to the set.

**Program Name** add() function can add element to the set  
demo8.py

```
s={10,20,30}  
s.add(40)  
print(s)
```

**Output**

```
{40, 10, 20, 30}
```

### 2. update(x, y) method:

- ✓ To add multiple items to the set.
- ✓ Arguments are not individual elements, and these are Iterable objects like list, range etc.
- ✓ All elements present in the given Iterable objects will be added to the set.

**Program Name** update(single argument) method in set  
demo9.py

```
s = {10,20,30}  
l = [40,50,60,10]  
s.update(l)  
print(s)
```

**Output**

```
{40, 10, 50, 20, 60, 30}
```

**Program Name** update(two arguments) method in set  
demo10.py

```
s = {10,20,30}  
l = [40,50,60,10]  
s.update(l, range(5))  
print(s)
```

**Output**

```
{0, 1, 2, 3, 4, 40, 10, 50, 20, 60, 30}
```

## Difference between add() and update() methods in set?

- ✓ We can use add() to add individual item to the set, whereas we can use update() method to add multiple items to Set.
- ✓ add() method can take only one argument whereas update() method can take any number of arguments but all arguments should be iterable objects.

## Q. Which of the following are valid for set s?

1. s.add(10)
  2. s.add(10,20,30) TypeError: add() takes exactly one argument (3 given)
  3. s.update(10) TypeError: 'int' object is not iterable
  4. s.update(range(1,10,2),range(0,10,2))
3. copy() method

- ✓ This method returns copy of the set. It is cloned object.

Program Name      copy() method in set  
demo11.py

```
s={10,20,30}  
s1=s.copy()  
print(s1)
```

Output

```
{10, 20, 30}
```

## 4. pop() method

- ✓ This method removes and returns some random element from the set.

Program Name      pop() method in set  
demo12.py

```
s = {40,10,30,20}  
print(s)  
print(s.pop())  
print(s)
```

Output

```
{40, 10, 20, 30}  
40  
{10, 20, 30}
```

## 5. remove(x) method

- ✓ This method removes specified element from the set.
- ✓ If the specified element not present in the set then we will get **KeyError**

**Program Name** remove() method in set  
demo13.py

```
s={40,10,30,20}  
s.remove(30)  
print(s)
```

**Output**  
{40, 10, 20}

**Program Name** remove() method in set  
demo14.py

```
s={40,10,30,20}  
s.remove(50)
```

**Error**  
KeyError: 50

## 6. discard(x) method

- ✓ This method removes the specified element from the set.
- ✓ If the specified element not present in the set, then we won't get any error.

**Program Name** discard() method in set  
demo15.py

```
s={10,20,30}  
s.discard(10)  
print(s)
```

**output**  
{20, 30}

**Program Name** discard() method in set  
demo16.py

```
s={10,20,30}
```

```
s.discard(10)
print(s)
s.discard(50)
print(s)
```

## Output

```
{20, 30}
{20, 30}
```

## 7.clear() method

- ✓ This method remove all elements from the set.

**Program Name** clear() method in set  
demo17.py

```
s={10,20,30}
print(s)
s.clear()
print(s)
```

## Output

```
{10, 20, 30}
set()
```

## Mathematical operations on the Set:

### 1. union() method:

- ✓ This method return all elements present in both sets

## Syntax

```
x.union(y)
```

or

```
x|y
```

**Program Name** union() method in set  
demo18.py

```
x={10,20,30,40}
y={30,40,50,60}
print(x.union(y))
```

```
print(x|y)

output

{40, 10, 50, 20, 60, 30}
{40, 10, 50, 20, 60, 30}
```

## 2. intersection() method:

- ✓ This method returns common elements present in both x and y

### Syntax

```
x.intersection(y)

or

x&y
```

**Program Name** intersection() method in set  
demo19.py

```
x = {10,20,30,40}
y = {30,40,50,60}
print(x.intersection(y))
print(x&y)
```

### output

```
{40, 30}
{40, 30}
```

## 3. difference() method:

- ✓ This method returns the elements present in x but not in y

### Syntax

```
x.difference(y)

or

x-y
```

**Program Name** difference() method in set  
demo20.py

```
x={10,20,30,40}  
y={30,40,50,60}  
print(x.difference(y))  
print(x-y)  
print(y-x)
```

**output**

```
{10, 20}  
{10, 20}  
{50, 60}
```

#### 4.symmetric\_difference():

- ✓ This method returns elements present in either x or y but not in both

**Syntax**

```
x.symmetric_difference(y)  
or  
x^y
```

**Program Name** symmetric\_difference() method in set  
demo21.py

```
x={10,20,30,40}  
y={30,40,50,60}  
print(x.symmetric_difference(y))  
print(x^y)
```

**output**

```
{10, 50, 20, 60}  
{10, 50, 20, 60}
```

## Membership operators: (in and not in)

- ✓ By using these operators, we can find the specified element is exists in set or not

Program Name

```
in operator  
demo22.py  
  
s= {1, 2, 3, 'nireekshan'}  
print(s)  
print(1 in s)  
print('z' in s)
```

Output

```
{1, 2, 3, 'nireekshan'}  
True  
False
```

## Set Comprehension:

- ✓ Set comprehension is possible.

Program Name

```
set comprehension  
demo23.py  
  
s = {x*x for x in range(5)}  
print(s)
```

output

```
{0, 1, 4, 9, 16}
```

## Make a note

- ✓ set objects won't support indexing and slicing:

Program Name

```
set not support indexing  
demo24.py  
  
s = {10,20,30,40}  
print(s[0])
```

output

```
TypeError: 'set' object does not support indexing
```

<b>Program Name</b>	set not support slicing demo25.py
	<pre>s = {10,20,30,40} print(s[1:3])</pre>
<b>Output</b>	<b>TypeError:</b> 'set' object is not subscriptable

## Remove duplicates in list elements

- ✓ We can remove duplicates elements which are exists in list by passing list as parameter to set function

<b>Program Name</b>	removing duplicates from list demo26.py
	<pre>l=[10,20,30,10,20,40] s=set(l) print(s)</pre>
<b>Output</b>	{40, 10, 20, 30}

## Frozen set

- ✓ As we know set is mutable means we can modify the elements in set
- ✓ Frozen set is same as set but it is immutable we cannot modify the elements in frozen set.

## Creating frozen set

- ✓ `fromzensest()` method returns an immutable frozenset object.

<b>Program Name</b>	creating frozen set object demo27.py
	<pre>vowels = ('a', 'e', 'i', 'o', 'u') fSet = frozenset(vowels) print(fSet) print(type(fSet))</pre>
<b>Output</b>	<pre>frozenset({'u', 'a', 'e', 'i', 'o'}) &lt;class 'frozenset'&gt;</pre>

## 16. Dictionaries – {}

### Dictionary

- ✓ If we want to represent group of **individual objects** as a single entity then we should go for,
  - list
  - set
  - tuple
- ✓ If we want to represent a group of objects as **key-value** pairs then we should go for,
  - dict or dictionary
- ✓ Dictionary can contain key, value pairs.
- ✓ key, value pairs are separated by colon **:** symbol
- ✓ one key-value pair represents as item.
- ✓ Duplicate keys are not allowed.
- ✓ Duplicate values can be allowed.
- ✓ Heterogeneous objects are allowed for both key and values.
- ✓ Insertion order is not preserved.
- ✓ Dictionary object having mutable nature.
- ✓ Dictionary objects are dynamic.
- ✓ Indexing and slicing concepts are not applicable

### What symbol is required to create set?

- ✓ To create list, we need to use square bracket symbols : []
- ✓ To create tuple, we need to use parenthesis : ()
- ✓ To create **set** we need to use **curly braces** : {}
- ✓ So, to create **dict** we need to use **curly braces** : {}

### Create dictionary

#### Syntax

```
d = {key1:value1, key2:value2, ..., keyN:valueN}
```

### Creating dictionary

- ✓ We can create dictionary with key, value pairs.

Program Name	creating dictionary demo1.py
--------------	---------------------------------

```
d = {1:'Ramesh', 2:'Arjun', 3:'Nireekshan'}  
print(d)
```

output

```
{1:'Ramesh', 2:'Arjun', 3:'Nireekshan'}
```

## Empty dictionary

- ✓ We can create empty dictionary.
- ✓ For that empty dictionary we can add key, value pairs.

Program Name creating empty dictionary and adding elements  
demo2.py

```
d = {}  
d[1] = "Ramesh"  
d[2] = "Arjun"  
d[3] = "Nireekshan"  
print(d)
```

output

```
{1:'Ramesh', 2:'Arjun', 3:'Nireekshan'}
```

## Access values by using keys from dictionary

- ✓ We can access dictionary values by using keys
- ✓ Keys plays main role to access the data.

Program Name Accessing dictionary values by using keys  
demo3.py

```
d = {1:'Ramesh', 2:'Arjun', 3:'Nireekshan'}  
  
print(d[1])  
print(d[2])  
print(d[3])
```

output

```
Ramesh  
Arjun  
Nireekshan
```

## Make a note

- ✓ While accessing, if specified key is not available then we will get KeyError

Program Name	Trying to access key which is not exists in dictionary demo4.py
	d = {1:'Ramesh', 2:'Arjun', 3:'Nireekshan'} print(d[40])
Error	KeyError: 40

## How to handle this KeyError?

- ✓ We can handle this error by checking whether key is already available or not by using **in** operator.

Program Name	Checking key is exist or not in dictionary demo5.py
	d = {1:'Ramesh', 2:'Arjun', 3:'Nireekshan'}
	if 400 in d: print(d[400]) else: print('key not found')

Program Name	Employee info program by using dictionary demo6.py
	d={} n=int(input("Enter number of employees: ")) i=1 while i <=n: name=input("Enter Employee Name: ") salary=input("Enter Employee salary: ") d[name]=salary i=i+1 print("Name of Employee", "\t", "salary") for x in d: print("\t", x ,"\t\t", d[x])
output	

```
Enter number of employees: 2
```

```
Enter Employee Name: Mohan  
Enter Employee salary: 20000
```

```
Enter Employee Name: Prasad  
Enter Employee salary: 10000
```

Name of Employee	salary
Mohan	20000
Prasad	10000

## Update dictionary

- ✓ We can update the key in dictionary.

### Syntax

```
d[key] = value
```

#### case 1

- ✓ While updating the key in dictionary, if key is not available then a new key will be added at the end of the dictionary with specified value.

#### case 2

- ✓ If the key is already existing in dictionary, then old value will be replaced with new value.

### Program Name

Adding key-value pair to dictionary  
demo7.py

```
d = {1:'Ramesh', 2:'Arjun', 3:'Nireekshan'}  
print("Old dictionary:", d)  
d[10] = 'Hari'  
print("Added key-value(10:Hari) pair to dictionary: ", d)
```

### output

```
Old dictionary: {1: 'ramesh', 2: 'arjun', 3: 'nireekshan'}  
Added key-value 10:Hari pair to dictionary:  
{1: 'ramesh', 2: 'arjun', 3: 'nireekshan', 10: 'Hari'}
```

<b>Program Name</b>	Updating key-value pair in dictionary demo8.py
	<pre>d = {1:'Ramesh', 2:'Arjun', 3:'Nireekshan'} print("Old dictionary:", d) d[3] = 'Chandhu' print("Updated dictionary 3:Chandhu :", d)</pre>
<b>output</b>	Old dictionary: {1: 'Ramesh', 2: 'Arjun', 3: 'Nireekshan'} Updated dictionary 3:Chandhu : {1: 'Ramesh', 2: 'Arjun', 3: 'Chandhu'}

## Removing or deleting elements from dictionary

- ✓ By using **del** keyword, we can remove the keys
- ✓ By using **clear()** we can clear the objects in dictionary

### 1. By using **del** keyword

#### Syntax

```
del d[key]
```

- ✓ As per the syntax, it deletes entry associated with the specified key.
- ✓ If the key is not available, then we will get **KeyError**

<b>Program Name</b>	Deleting key in dictionary demo9.py
	<pre>d = {1:'Ramesh', 2:"Arjun", 3:"Nireekshan"} print("Before deleting key from dictionary: ", d) del d[1] print("After deleting key from dictionary:", d)</pre>
<b>output</b>	Before deleting key from dictionary: {1: 'Ramesh', 2: 'Arjun', 3: 'Nireekshan'}
	After deleting key from dictionary: {2: 'Arjun', 3: 'Nireekshan'}

**Program Name** Deleting the key which is not exists in dictionary  
demo10.py

```
d = {1:'Ramesh', 2:"Arjun", 3:"Nireekshan"}  
print(d)  
del d[10]
```

**Error**

**KeyError: 10**

We can delete total dictionary object

**Syntax**

```
del nameofthedictionary
```

- ✓ It can delete the total dictionary object.
- ✓ Once it deletes then we cannot access the dictionary.

**Program Name** Delete key in dictionary  
demo11.py

```
d = {1:'Ramesh', 2:"Arjun", 3:"Nireekshan"}  
print("Before deleting dictionary: ", d)  
del d  
print(d)
```

**output**

```
Before deleting dictionary: {1: 'Ramesh', 2: 'Arjun', 3: 'Nireekshan'}  
NameError: name 'd' is not defined
```

**clear() method**

- ✓ Clear() method removes the all entries in dictionary.
- ✓ After deleting all entries, it just keeps empty dictionary

**Program Name** removing dictionary object by using clear() method  
demo12.py

```
d = {1:'Ramesh', 2:"Arjun", 3:"Nireekshan"}  
print("Before clearing dictionary: ", d)
```

```
d.clear()
print("After cleared entries in dictionary: ", d)

output
```

```
Before clearing dictionary: {1: 'Ramesh', 2: 'Arjun', 3: 'Nireekshan'}
After cleared entries in dictionary: {}
```

## Important functions and methods of dictionary

### 1. dict() function

- ✓ To create an empty dictionary.
- ✓ We can add elements to that created dictionary

<b>Program Name</b>	creating empty dictionary by using dict() function demo13.py
	<pre>d=dict() print(d) print(type(d))</pre>

**output**

```
{}
<class 'dict'>
```

### 2. dict({key1:value1, key2:value2}) function

- ✓ It creates dictionary with specified elements

<b>Program Name</b>	creating dictionary by using dict() function demo14.py
	<pre>d=dict({100:"Ramesh",200:"Arjun"}) print(d)</pre>

**output**

```
{100: 'Ramesh', 200: 'Arjun'}
```

### 3. dict([tuple1, tuple2])

- ✓ This function creates dictionary with the given list of tuple elements

<b>Program Name</b>	creating dictionary with tuples demo15.py
<b>output</b>	<pre>d = dict([(1, "Ramesh"), (2, "Arjun")]) print(d)</pre> {1: 'Ramesh', 2: 'Arjun'}

### 4. len() function

- ✓ This function returns the number of items in the dictionary

<b>Program Name</b>	Finding length of dictionary demo16.py
<b>output</b>	<pre>d = {100:"Ramesh",200:"Arjun"} print("length of dictionary is: ",len(d))</pre> length of dictionary is:

### 5. clear() method

- ✓ This method can remove all elements from the dictionary

### 6. get() method

- ✓ This method used to get the value associated with the key

#### Case 1

- ✓ If the key is available, then returns the corresponding value otherwise returns **None**. It won't raise any error.

#### Syntax

```
d.get(key)
```

#### Case 2

- ✓ If the key is available, then returns the corresponding value otherwise returns default value.

## Syntax

```
d.get(key, defaultvalue)
```

**Program Name**      get() method  
demo17.py

```
d={100:"Ramesh",200:"Arjun",300:"Prasad"}  
print(d[100])  
print(d.get(100))
```

**output**  
Ramesh  
Ramesh

**Program Name**      get() function  
demo18.py

```
d={100:"Ramesh",200:"Arjun",300:"Prasad"}  
print(d[100])  
print(d.get(400))
```

**output**  
Ramesh  
None

## 7. pop() method

- ✓ This method removes the entry associated with the specified key and returns the corresponding value.
- ✓ If the specified key is not available, then we will get KeyError

## Syntax

```
d.pop(key)
```

**Program Name**

pop() method  
demo19.py

```
d={10:"Ramesh", 20:"Arjun", 30:"Prasad"}  
print("Before pop:", d)  
d.pop(10)  
print("After pop:", d)
```

**output**

```
Before pop: {10: 'Ramesh', 20: 'Arjun', 30: 'Prasad'}  
After pop: {20: 'Arjun', 30: 'Prasad'}
```

**Program Name**

pop() method  
demo20.py

```
d={10:"Ramesh", 20:"Arjun", 30:"Prasad"}  
d.pop(400)
```

**output**

```
KeyError: 100
```

## 8. `popitem()` method

- ✓ This method removes an arbitrary item(key-value) from the dictionary and returns it.

**Program Name**

`popitem()` function  
demo21.py

```
d={100:"Ramesh",200:"Arjun",300:"Prasad"}  
print(d)  
d.popitem()  
print(d)
```

**output**

```
{100: 'Ramesh', 200: 'Arjun', 300: 'Prasad'}  
{100: 'Ramesh', 200: 'Arjun'}
```

## Make a note

- ✓ If the dictionary is empty, then we will get `KeyError`

**Program Name** popitem() function  
demo22.py

```
d={}
print(d.popitem())
```

**Output**

```
KeyError: 'popitem(): dictionary is empty'
```

## 9. keys() method

- ✓ This method returns all keys associated with dictionary

**Program Name** keys() method  
demo23.py

```
d={100:"Ramesh",200:"Nireekshan",300:"Mohan"}
print(d)
```

```
for k in d.keys():
    print(k)
```

**Output**

```
{100: 'Ramesh', 200: 'Nireekshan', 300: 'Mohan'}
100
200
300
```

## 10. values()

- ✓ This method returns all values associated with the dictionary

**Program Name** values() method  
demo24.py

```
d={100:"Ramesh",200:"Nireekshan",300:"Mohan"}
print(d)
for k in d.values():
    print(k)
```

**Output**

```
{100: 'Ramesh', 200: 'Nireekshan', 300: 'Mohan'}
Ramesh
```

Nireekshan  
Mohan

## 11. items() method

- ✓ A key value pair in dictionary is called as item.
- ✓ items() method returns list of tuples representing key-value pairs.

Example

```
[(k,v),(k,v),(k,v)]
```

<b>Program Name</b>	items() method demo25.py
	<pre>d={100:"Ramesh",200:"Nireekshan",300:"Mohan"}  for k, v in d.items():     print(k, "---", v)</pre>
<b>Output</b>	<pre>100 --- Ramesh 200 --- Nireekshan 300 --- Mohan</pre>

## 12. copy() method

- ✓ To create exactly duplicate dictionary (cloned copy)

<b>Program Name</b>	copy() method demo26.py
	<pre>d1 = {1:"Ramesh", 2:"Arjun"} d2=d1.copy() print(d1) print(d2)</pre>
<b>output</b>	<pre>{1: 'Ramesh', 2: 'Arjun'} {1: 'Ramesh', 2: 'Arjun'}</pre>

## Dictionary Comprehension:

- ✓ Comprehension concept applicable for dictionaries also.

Program Name      Dictionary comprehension  
demo27.py

```
squares={a:a*a for a in range(1,6)}  
print(squares)
```

Output

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

## 17. Object-Oriented Programming System (OOPS) (class, object, variable, method etc)

### Before OOPS

- ✓ Before oops, pop (procedure-oriented programming) exists.
- ✓ POP languages are C, Pascal, Fortran etc.
- ✓ The above languages used procedures and functions to perform the task.
- ✓ While developing any software is divided into sub tasks composed of several procedures and functions.
- ✓ This approach is called Procedure Oriented Approach.
  
- ✓ Main Task
  - Sub task1 - function1
  - Sub task2 - function2
  - Sub task3 - function3

### Limitations

- ✓ While developing task by using POP, if project is coding is growing then handling code is too difficult.
- ✓ Debugging the code is very difficult.
- ✓ Enhancements are difficult

### After OOPS

- ✓ OOPS is paradigm or methodology to implement software.
- ✓ OOPS languages are C++, Java, Dot Net etc...
- ✓ The above languages used OOPS to perform the task.
- ✓ While developing any software, by using OOPS we need to create classes and objects to implement the software.
- ✓ This approach is called as Object-Oriented Programming.
  
- ✓ Main Task
  - Sub task1 - Class1 with data and method
  - Sub task2 - Class2 with data and method
  - Sub task3 - Class3 with data and method

### Advantages

- ✓ While developing task by using OOPS, if project is code is growing then handling the code is very easy.
- ✓ Debugging the code is easy.
- ✓ Enhancements are easy.

Is Python follows Functional approach or Object-oriented approach

- ✓ Python supports both functional and object-oriented programming.

OOPs (Object Oriented Programming Principles)

- ✓ Object-Oriented Programming is a methodology to design a software by using classes and objects.
- ✓ It simplifies the software development and maintenance by providing the below features,

Features of Object Oriented Programming System

- ✓ class
- ✓ object
- ✓ Encapsulation
- ✓ Abstraction
- ✓ Inheritance
- ✓ Polymorphism
- ✓ Abstract class
- ✓ Interface etc...

1. class:

Def 1:

- ✓ A class is a model for creating an object and it does not exist physically.

Def 2:

- ✓ A class is a specification (idea/plan/theory) of properties and actions of objects.

## Syntax

```
class name_of_the_class:  
    constructor  
    properties (attributes)  
    actions (behaviour)
```

- ✓ We can create class by using **class** keyword.
- ✓ class can contain,
  - constructor
  - properties
  - actions
- ✓ Properties represented by variables.
- ✓ Actions represented by methods.

## How to define a class?

- ✓ A python class may contain the below things,

### Syntax

```
class name_of_the_class:  
    "" documentation string ""  
  
    constructors  
  
    variables  
        1. instance variables,  
        2. static variables,  
        3. local variables  
  
    methods  
        1. instance methods,  
        2. static methods,  
        3. class methods,
```

### Brief discussion about class

- ✓ Documentation string represents description of the class. Within the class doc string is always optional.
- ✓ We can create class by using **class** keyword.
- ✓ **class** keyword follows the name of the class.
- ✓ After name of the class we should give colon **:** symbol.
- ✓ **class** can contain,
  - **Constructors** are used for initialization purpose
  - **Variables** are used to represent the data.
  - Variables are three types,
    - instance variables
    - static variables
    - local variables
  - **Methods** are used to represent actions.
  - Methods are three types,
    - instance methods
    - static methods
    - class methods

Program Name Define a class demo1.py

```
class Employee:  
  
    def display(self):  
        print("Hello My name is Nireekshan")
```

output

## Make a note

- ✓ In the above program Employee represents a **class** which is defined by developer.
- ✓ Developer defined only one method as display

## Kind info:

- ✓ Writing a class is not enough, we should know how to use the variables and methods.

So,

- ✓ We need to create an object to access instance data of a class.

## self variable

- ✓ **self** is a default variable that refers to current class object.
- ✓ By using self, we can initialize the instance variables in inside constructor **\_\_init\_\_**
- ✓ By using **self**, we can access,
  - instance variables.
  - instance methods.
- ✓ self should be first parameter in constructor,
  - **def \_\_init\_\_(self):**
- ✓ self should be first parameter for instance methods,
  - **def display(self):**

## 2. object

### Why should we create an object?

- ✓ As per requirement we used to define variables and methods in a class.
- ✓ These variables and methods hold the data or values.
- ✓ When we create an object for a class, then only data will be stored for the data members of a class.

### What is an object?

#### Def 1:

- ✓ Instance of a class is known as an object.
  - Instance is a mechanism to allocates sufficient amount of memory space for data members of a class.

#### Def 2:

- ✓ Grouped item is known as an object.
  - Grouped item is a variable which stores more than one value.

#### Def 3:

- ✓ Real world entities are called as objects.

#### Def 4:

- ✓ Logical runtime entities are called as objects.

### Make some notes

- ✓ An object exists physically in this world, but class does not exist.
- ✓ An object does not exist without class, but class can exist without an object.

### Make a note

- ✓ If you didn't use the object, then the created object is useless.

### Syntax to create an object

#### Syntax

```
nameoftheobject = nameoftheclass ()
```

## Example

```
emp = Employee()
```

**Program Name** Define an Employee class and create an object for Employee class  
demo2.py

```
class Employee:  
  
    def display(self):  
        print("Hello my name is Nireekshan")  
  
    emp=Employee()  
    emp.display()
```

## output

```
Hello my name is Nireekshan
```

## Make a note

- ✓ We can create object for class.
- ✓ In the above example **emp** is object name and this name can be anything.

## Calling instance data

- ✓ After object is created then we can use that object to call instance data with the name of object by using dot notation.
- ✓ Instance data,
  - Instance variables
  - Instance methods

## Syntax

```
name_of_the_object.instancedata
```

## Example 1

```
name_of_the_object.variable
```

## Example 2

```
name_of_the_object.method
```

Can we create more than one object?

- ✓ Yes, we can create any number of objects for a class.
- ✓ Before creating object, class should exist otherwise we will get error.

**Program Name** Create an object for Employee class and access method demo3.py

```
class Employee:  
  
    def display(self):  
        print("Hello my name is Nireekshan")  
  
emp1 = Employee()  
emp1.display()  
  
emp2 = Employee()  
emp2.display()
```

**output**

```
Hello my name is Nireekshan  
Hello my name is Nireekshan
```

## Constructor

- ✓ Constructor is a special kind of method in python.
- ✓ So, we can define constructor with `def` keyword
- ✓ The name of the constructor should be `__init__(self)`
- ✓ `self` should be first parameter in constructor,

### Syntax

```
def __init__(self):  
    body of the constructor
```

What is the main purpose of constructor?

- ✓ The main purpose of constructor is to initialize instance variables.

## When constructor will be executed?

- ✓ Constructor will be executed automatically at the time of object creation.

## How many times constructor will be executed?

- ✓ If we create one object, then one-time constructor will be executed.
- ✓ If we create two objects, then two times constructor will be executed.
- ✓ If we create 'N' number of objects, then N number of times constructor will be executed.

## Is constructor mandatory to define?

- ✓ Based on requirement we need to define constructor.
- ✓ At the time of object creation if any initialization is required then we should go for constructor.
- ✓ Without constructor also, python program is valid.

**Program Name** Defining a constructor and executed automatically while object creation  
demo4.py

```
class Employee:  
    def __init__(self):  
        print("constructor is executed automatically at the time of  
object creation")  
  
emp = Employee()
```

**output**

constructor is executed automatically at the time of object creation

## Make a note

- ✓ Observe the above program code, we didn't call the constructor, but constructor got executed automatically at the time of object creation.
- ✓ Methods we need to call explicitly.

## Can I call constructor explicitly like a method?

- ✓ Yes, we can call constructor explicitly with object name.
- ✓ But constructor is executing automatically at the time of object creation, so it is not recommended to call explicitly.

**Program Name** Defining a constructor and executed automatically while object creation  
demo5.py

```
class Employee:  
    def __init__(self):  
        print("constructor")  
  
emp = Employee()  
emp.__init__()
```

**output**

```
constructor  
constructor
```

**Make a note**

- ✓ Methods we need to call explicitly.

**Program Name** Defining a method and calling explicitly to execute.  
demo6.py

```
class Employee:  
    def m1(self):  
        print("m1 is instance method called with object name")  
  
e = Employee()  
e.m1()
```

**output**

```
m1 is instance method called with object name
```

**Make a note**

- ✓ If developer is not defining any constructor, then internally python will create constructor.
- ✓ This we can check by using `dir`(Name of the class)

**Program Name** default constructor in python.  
demo7.py

```
class Sample:  
    def m1(self):  
        print("m1 is instance method called with object name")  
  
s = Sample()  
s.m1()
```

```
print(dir(Sample))  
  
output  
m1 is instance method called with object name  
[..., __init__, ...]
```

How many times constructor will be executed?

- ✓ If we create one object, then one-time constructor will be executed.
- ✓ If we create two objects, then two times constructor will be executed.
- ✓ If we create 'N' number of objects, then N number of times constructor will be executed.

Program Name	Creating 3 objects, so 3 times constructor will execute demo8.py
	<pre>class Employee:     def __init__(self):         print("Constructor executed automatically")  e1 = Employee() e2 = Employee() e3 = Employee()</pre>
output	Constructor executed automatically Constructor executed automatically Constructor executed automatically

Constructor can contain how many parameters?

- ✓ Based on requirement constructor can contain any number of parameters.

If constructor having no parameters, then how to define?

- ✓ If constructor having no parameters, then at least it should contain **self** as one parameter.

Syntax

```
def __init__(self):
```

<b>Program Name</b>	Define a constructor in python program. demo9.py
	<pre>class Employee:     def __init__(self):         print("constructor is executed automatically at the time of object creation")  emp = Employee()</pre>
<b>output</b>	constructor is executed automatically at the time of object creation

## Make a note

- ✓ No parameterised constructor used to initialize, all objects with same data.

<b>Program Name</b>	No parameterised constructor used to initialize, all objects with same data.
<b>Name</b>	demo10.py
	<pre>class Employee:     def __init__(self):         self.id = 1         print("Employee id is: ", self.id)  e1 = Employee() e2 = Employee() e3 = Employee()</pre>
<b>output</b>	Employee id is: 1 Employee id is: 1 Employee id is: 1

## Make a note

- ✓ So, if we want to initialise different objects with different data then we should go for parameterised constructor.

## If trying to print self variable

- ✓ Happily, we can print self variable inside constructor, internally it represents object representation.

<p><b>Program Name</b></p> <p>demo11.py</p> <p><b>output</b></p>	<p>Trying to print self variable and checking what it contains</p> <pre>class Employee:     def __init__(self):         print("only self is parameter, no other parameters", self)         print(self)  e1 = Employee()</pre> <p>only self is parameters, no other parameters &lt;__main__.Employee object at 0x000001F3E258B0F0&gt;</p>
--	--

## If constructor having more parameters, then how to define?

- ✓ Constructor can contain more parameters along with **self**
- ✓ If constructor having more parameters, then the first parameter should be **self** and remaining parameters will be next.

### Syntax

```
def __init__(self, parameter1, parameter2,...parametern):
```

<p><b>Program Name</b></p> <p>demo12.py</p> <p><b>Output</b></p>	<p>One parameterised constructor</p> <pre>class Employee:      def __init__(self, id):         self.id = id         print("Employee id is: ", self.id)  e1 = Employee(1)</pre>
--	--

```
e2 = Employee(2)
e3 = Employee(3)
```

## output

```
Employee id is: 1
Employee id is: 2
Employee id is: 3
```

## Conclusion

- ✓ If constructor having more parameters, then first parameter should be self and follows remaining.

**Program Name** One parameterised constructor and instance method  
demo13.py

```
class Employee:

    def __init__(self, id):
        self.id = id

    def display(self):
        print("Employee id is: ", self.id)

e1 = Employee(1)
e2 = Employee(2)
e3 = Employee(3)

e1.display()
e2.display()
e3.display()
```

## output

```
Employee id is: 1
Employee id is: 2
Employee id is: 3
```

**Program Name** Two parameterised constructor and instance method  
demo14.py

```
class Employee:
```

```
def __init__(self, id, name):
    self.id=id
    self.name=name

def display(self):
    print("Hello my id is :", self.id)
    print("My name is :", self.name)

e1=Employee(1, 'Nireekshan')
e1.display()

e2=Employee(2, 'Arjun')
e2.display()
```

## Output

```
Hello my id is : 1
My name is : Nireekshan
```

```
Hello my id is : 2
My name is : Arjun
```

## Program Name

Three parameterised constructor and instance method  
demo15.py

```
class Employee:

    def __init__(self, id, name, age):
        self.id=id
        self.name=name
        self.age=age

    def display(self):
        print("Hello my id is :", self.id)
        print("My name is :", self.name)
        print("My age is sweet :", self.age)

e1=Employee(1, 'Nireekshan', 16)
e1.display()

e2=Employee(2, 'Arjun', 17)
e2.display()

e3=Employee(3, 'Prasad', 18)
e3.display()
```

## Output

```
Hello my id is : 1  
My name is : nireekshan  
My age is sweet : 16
```

```
Hello my id is : 2  
My name is : Arjun  
My age is sweet : 17
```

```
Hello my id is : 3  
My name is : Prasad  
My age is sweet : 18
```

## Difference between methods and constructor

Method	Constructor
✓ Methods are used to do operations or actions	✓ Constructors are used to initialize the instance variables.
✓ Method name can be any name.	✓ Constructor name should be <code>__init__(self)</code>
✓ Methods we should call explicitly to execute	✓ Constructor automatically executed at the time of object creation.

## Types of Variables:

- Once if we defined a python class then we can define three types of variables in python, those are as follows
  - 1. Instance variables (object level variables)
  - 2. Static variables (class level variables)
  - 3. Local variables

### 1. Instance variables:

#### What is instance variable?

- If the value of a variable is changing from object to object such type of variables is called as instance variables.

## Separate copy for every object

- ✓ For every object a separate copy of instance variables will be created.

Program Name      Instance variables  
demo16.py

```
class Student:  
    def __init__(self, name, id):  
        self.name=name  
        self.id=id  
  
    s1=Student('Nireekshan', 1)  
    s2=Student('Anushka', 2)  
  
    print("Studen1 info:")  
    print("Name: ", s1.name)  
    print("Id : ", s1.id)  
  
    print("Studen2 info:")  
    print("Name: ",s2.name)  
    print("Id : ",s2.id)
```

### Output

```
Studen1 info:  
  
Name: Nireekshan  
Id : 1  
  
Studen2 info:  
Name: Anushka  
Id : 2
```

### Make a note

- ✓ In the case of instance variables for every object a separate copy will be created.
- ✓ In the case of static variable for all objects will be shared a single copy of static variable.

### Where should we declare instance variable?

- ✓ We can declare and initialize the instance variables in following ways,
  1. By using constructor.
  2. By using instance method.
  3. By using object name

## 1. By using constructor

- ✓ We can declare and initialize instance variables inside a constructor by using **self** variable.

**Program Name** Initializing instance variables inside Constructor by using self variable  
demo17.py

```
class Employee:  
    def __init__(self):  
        self.eno=1  
        self.ename='Nireekshan'  
        self.esal=100  
  
    e=Employee()  
    print("Employee number: ",e.eno)  
    print("Employee name: ", e.ename)  
    print("Employee salary: ", e.esal)  
    print(e.__dict__)
```

### output

```
Employee number: 1  
Employee name : Nireekshan  
Employee salary : 100  
  
{'eno': 1, 'ename': 'Nireekshan', 'esal': 100}
```

### The **\_\_dict\_\_** attribute

- ✓ Every object in Python has an attribute denoted as **\_\_dict\_\_**.
- ✓ This displays the object information in dictionary form.
- ✓ It maps the attribute name to its value.

## 2. By using instance method

- ✓ We can declare and initialize instance variables inside instance method by using **self** variable.

**Program Name** Initializing instance variables inside instance method by using self  
demo18.py

```
class Student:  
  
    def m1(self):  
        self.a=11
```

```
self.b=21  
self.c=34
```

```
print(self.a)  
print(self.b)  
print(self.c)
```

```
s= Student()  
s.m1()  
print(s.__dict__)
```

## output

```
11  
21  
34  
{'a': 11, 'b': 21, 'c': 34}
```

### 3. By using object name

- ✓ We can declare and initialize instance variables by using object name as well.

**Program Name** Initializing instance variables by using object  
`demo19.py`

```
class Test:  
    def __init__(self):  
        print("This is constructor")  
  
    def m1(self):  
        print("This is instance method")  
  
t=Test()  
t.m1()  
  
t.a=10  
t.b=20  
t.c=55  
  
print(t.a)  
print(t.b)  
print(t.c)  
  
print(t.__dict__)
```

## output

```
This is constructor  
This is instance method
```

```
10  
20  
55  
{'a': 10, 'b': 20, 'c': 55}
```

## Accessing instance variable

- ✓ By using self variable
- ✓ By using object name

### 1. By using self variable

- ✓ We can access instance variables within the class by using self variable.

**Program Name** Accessing instance variables by using self  
demo20.py

```
class Student:  
    def __init__(self):  
        self.a=10  
        self.b=20  
  
    def display(self):  
        print(self.a)  
        print(self.b)  
  
s= Student ()  
s.display()
```

**output**

```
10  
20
```

### 2. By using object name

- ✓ We can access instance variables by using object name

**Program Name** Accessing instance variables by using object  
dem21.py

```
class Test:  
    def __init__(self):  
        self.a=10  
        self.b=20
```

```
t=Test()  
print(t.a)  
print(t.b)
```

output

```
10  
20
```

Every object has a separate copy of variable exists

- ✓ If we change the values of instance variables of one object, then those changes won't be reflected to the remaining objects, because for every object separate copy of instance variable are available.

Program Name

Every object separate copy of variable exists  
demo22.py

```
class Test:  
    def __init__(self):  
        self.a=10  
  
t1=Test()  
t2=Test()  
  
print('a value in t1 object: ',t1.a)  
print('a value in t2 object: ',t2.a)
```

output

```
10  
10
```

Program Name

Every object separate copy of variable exists  
demo23.py

```
class Test:  
    def __init__(self):  
        self.a=10  
  
t1=Test()  
t1.a=888  
  
t2=Test()  
  
print('a value in t1 object: ',t1.a)
```

```
print('a value in t2 object: ',t2.a)
```

output

```
888  
10
```

## 2. static variable (class level variable)

### What is static variable?

- ✓ If the value of a variable is not changing from object to object, such type of variables is called as static variable or class level variable.

### Where can we declare static variable?

- ✓ We can declare static variables with in the class directly but outside of methods.

### Only one copy of static variable to all objects

- ✓ Once if we create a static variable in class level then that variable value will be shared with all over objects.

### How can we access static variable?

- ✓ We can access static variables either by **class name** or by **object name**.
- ✓ Accessing static variable with class name is highly recommended than object name.
- ✓ Accessing static variable,
  - If we are trying to access static variable with class name, then we can access directly with name of the class.
  - If we are trying to access with object name, then we need to create object and we can access with object name

Program Name static variables accessing with class name  
demo24.py

```
class Student:  
    college_name='ANR'  
    def __init__(self, name, id):  
        self.name=name  
        self.id=id
```

```
s1=Student('Nireekshan', 1)  
s2=Student('Anushka', 2)
```

```
print("Studen1 info:")
print("Name: ", s1.name)
print("Id : ", s1.id)
print("College name : ", Student.college_name)

print("\n")

print("Studen2 info:")
print("Name: ",s2.name)
print("Id : ",s2.id)
print("College name : ", Student.college_name)
```

## Output

Studen1 info:  
Name: Nireekshan  
Id : 1  
College name: ANR

Studen2 info:  
Name: Anushka  
Id : 2  
College name: ANR

**Program Name** static variables accessing with object name which is not recommended  
**demo24a.py**

```
class Student:
    college_name='ANR'
    def __init__(self, name, id):
        self.name=name
        self.id=id

s1=Student('Nireekshan', 1)
s2=Student('Anushka', 2)

print("Studen1 info:")
print("Name: ", s1.name)
print("Id : ", s1.id)
print("College name : ", s1.college_name)

print("\n")

print("Studen2 info:")
print("Name: ",s2.name)
print("Id : ",s2.id)
print("College name : ", s2.college_name)
```

## Output

```
Studen1 info:  
Name: Nireekshan  
Id : 1  
College name: ANR
```

```
Studen2 info:  
Name: Anushka  
Id : 2  
College name: ANR
```

## Instance Variable vs Static Variable:

- ✓ In the case of instance variables for every object a separate copy will be created.
- ✓ In the case of static variable for all objects will be shared a single copy of static variable.

## Declaring static variable

- ✓ We can declare static variable in several ways like below,

1. Inside class and outside of the method
2. Inside constructor
3. Inside instance method
4. Inside classmethod
5. Inside static method

### 1. Inside class and outside of the method

- ✓ Generally, we can declare and initialize static variable within the class and outside of the method.

#### Program Name

Declaring static variable: inside of the class and outside of method  
demo25.py

```
class Demo:  
    a=20  
    def m(self):  
        print("this is method")  
  
    print(Demo.__dict__)
```

#### Output

```
{..., 'a': 20 ,...}
```

## 2. Inside constructor

- ✓ We can declare and initialize static variable within constructor **by using class name**

**Program Name** Declaring static variable: Inside constructor by using class name  
demo26.py

```
class Demo:  
    def __init__(self):  
        Demo.b=20  
  
    d=Demo()  
    print(Demo.__dict__)
```

**Output**  
{..., 'b': 20 ,...}

## 3. Inside instance method

- ✓ We can declare and initialize static variable inside instance method **by using class name.**

**Program Name** Declaring static variable: inside instance method by using class name  
demo27.py

```
class Demo:  
    def m1(self):  
        Demo.c=20  
  
    obj=Demo()  
    obj.m1()  
    print(Demo.__dict__)
```

**Output**  
{..., 'c': 20 ,...}

## 4. Inside classmethod

- ✓ We can declare and initialise static variable inside classmethod in two ways,
  - class name
  - **cls** pre-defined variable

### 1. class name

- ✓ Before method if we write **@classmethod** decorator then that method will become a classmethod.

- ✓ Inside class method we can declare and initialize static variable by using class name.
- ✓ **Info:** We will learn more in classmethod which is upcoming chapter

## 2. **cls** variable

- ✓ **cls** is predefined variable in python
- ✓ We should pass **cls** parameter to the class methods.
- ✓ So, with that **cls** we can access static variable in side class method
- ✓ **Info:** We will learn more in classmethod which is upcoming chapter

## 1. **class name**

- ✓ Before method if we write **@classmethod** decorator then that method will become a classmethod.
- ✓ Inside class method we can declare and initialize static variable by using class name.

<b>Program Name</b>	static variable: inside classmethod by using class name and <b>cls</b> variable demo28.py
	<pre>class Demo:     @classmethod     def m2(cls):         Demo.x=10      Demo.m2()     print(Demo.__dict__)</pre>
<b>Output</b>	{..., 'x': 10, ...}

## 2. **cls** variable

- ✓ **cls** is predefined variable in python
- ✓ We should pass **cls** parameter to the class methods.
- ✓ So, with that **cls** we can access static variable in side class method
- ✓ **Info:** We will learn more in classmethod which is upcoming chapter

<b>Program Name</b>	static variable: inside classmethod by using <b>cls</b> variable demo29.py
	<pre>class Demo:     @classmethod     def m2(cls):         cls.y=20      Demo.m2()     print(Demo.__dict__)</pre>

## Output

```
{..., 'y': 20 ,...}
```

## 5. Inside static method

- ✓ We can declare and initialize static variable inside static method by using class name.

### @staticmethod declarator

- ✓ Before method if we write @staticmethod decorator then that method will become a staticmethod
- ✓ **Info:** We will learn more in staticmethod which is upcoming chapter

#### Program Name

static variable: inside static method by using class name  
demo30.py

```
class Demo:  
    @staticmethod  
    def m3():  
        Demo.z=10  
  
    Demo.m3()  
    print(Demo.__dict__)
```

#### Output

```
{..., 'z': 10 ,...}
```

## Accessing static variable

- ✓ We can access static variable in following ways,

1. Inside constructor
2. Inside instance method
3. Inside classmethod
4. Inside static method

### 1. Inside constructor

- ✓ We can access static variable inside constructor by using either **self** and **class name**

#### Program Name

Accessing static variable inside constructor by using class name  
demo31.py

```
class Demo:
```

```
a=10
def __init__(self):
    print(self.a)
    print(Demo.a)

d=Demo()
```

## Output

```
10
10
```

## 2. Inside instance method

- ✓ We can access static variable inside instance method by using either `self` and `class name`

**Program Name** Accessing static variable inside instance method by using class name  
demo32.py

```
class Demo:
    a=10
    def m1(self):
        print(self.a)
        print(Demo.a)

obj=Demo()
obj.m1()
```

## Output

```
10
10
```

## 3. Inside classmethod

- ✓ We can access static variable inside classmethod by using `cls` variable and `class name`

**Program Name** Accessing static variable inside class method by using class name  
demo33.py

```
class Demo:
    a=10
    @classmethod
    def m1(cls):
        print(cls.a)
        print(Demo.a)
```

```
obj=Demo()
```

```
obj.m1()
```

Output

```
10  
10
```

#### 4. Inside staticmethod

- ✓ We can access static variable inside staticmethod by using **class name**.

Program Name Accessing static variable inside static method by using class name  
`demo34.py`

```
class Demo:  
    a=10  
    @staticmethod  
    def m1():  
        print(Demo.a)  
  
obj=Demo()  
obj.m1()
```

Output

```
10
```

#### 3. Local variable

What is local variable?

- ✓ The variable which we declare inside of the method is called as local variable.

Why we need to use local variable?

- ✓ Just for temporary usage we can use local variable.

Where can we access local variable?

- ✓ Local variable can access within the method only, we can not access in outside of method.

Program Name local variable  
`demo35.py`

```
class Demo:  
    def m(self):  
        a=10  
        print(a)
```

```
d=Demo()  
d.m()
```

## Output

```
10
```

**Program Name** Trying to access local variable in out of scope  
`demo36.py`

```
class Demo:  
    def m(self):  
        a=10  
        print(a)  
  
    def n(self):  
        print(a)
```

```
d=Demo()  
d.m()  
d.n()
```

## Output

```
10  
NameError: name 'a' is not defined
```

## Types of methods

In python 3 types of method are available,

1. instance Methods
  - a. Getter or Accessor methods
  - b. Setter or Mutator methods
2. class Methods
3. static Methods

### 1. Instance methods

- ✓ Instance methods are methods which act upon the instance variables of the class.
- ✓ Instance methods are bound with instances or objects, that's why called as instance methods.
- ✓ The first parameter for instance methods is `self` variable.
- ✓ Along with `self` variable it can contains other variables as well.

**Program Name** Instance methods  
demo37.py

```
class Demo:  
    def __init__(self, a):  
        self.a=a  
  
    def m(self):  
        print(self.a)  
  
d=Demo(10)  
d.m()
```

**Output**  
10

## What is the use of Setter and Getter Methods?

- ✓ We can set and get the values of instance variables by using setter and getter methods.

### Setter method

- ✓ setter methods can be used to set values to the instance variables.
- ✓ setter methods also known as mutator methods.

**Syntax**  
`def set_variable(self, variable):  
 self.variable=variable`

**Example**  
`def set_name(self, name):  
 self.name=name`

**Program Name** Setter methods  
demo38.py

```
class Customer:  
  
    def set_name(self, name):  
        self.name=name  
  
    def set_id(self, id):
```

```
self.id=id  
  
c=Customer()  
  
c.set_name("Balayya")  
c.set_id(1)
```

## Output

### Getter Method

- ✓ We can get the values of instance variable by using getter methods.
- ✓ Getter methods having **return** statement.
- ✓ Getter methods also known as accessor methods.

### Syntax

```
def get_variable(self):  
    return self.variable
```

### Example

```
def get_name(self):  
    return self.name
```

### Program Name

Setter and Getter methods  
demo39.py

```
class Customer:  
  
    def set_name(self, name):  
        self.name=name  
  
    def get_name(self):  
        return self.name  
  
    def set_id(self, id):  
        self.id=id  
  
    def get_id(self):  
        return self.id  
  
c=Customer()  
  
c.set_name("Nireekshan")  
c.set_id(1)
```

```
print("My name is: ", c.get_name())
print("My id is: ", c.get_id())
```

## Output

```
My name is: Nireekshan
My id is: 1
```

## 2. Class Methods

### What is class method?

- ✓ Class methods are methods which act upon the class or static variables of the class.

### When we can go to class methods?

- ✓ Inside method implementation if we are using only class variables (static variables), then such type of methods we should declare as class method.

### @classmethod decorator

- ✓ We can declare classmethods by using **@classmethod** decorator
- ✓ The first parameter for static methods is **cls** variable.
- ✓ Along with **cls** variable it can contains other variables as well.

### How to access class methods?

- ✓ We can access class method by using class name or object name.

### Make a note

- ✓ class methods are rarely used methods in python.

<b>Program Name</b>	Declaring classmethod in class demo40.py
	<pre>class Pizza:     radius=10     @classmethod     def get_radius(cls):         return cls.radius  print(Pizza.get_radius())</pre>

## Output

10

### 3. static Methods:

- ✓ In general, these methods are general utility methods.
- ✓ Inside these methods we won't use any instance or class variables.
- ✓ Here we won't provide `self` or `cls` arguments at the time of declaration.

#### How to declare static method?

- ✓ We can declare static method explicitly by using `@staticmethod` decorator.

#### How to access static methods?

- ✓ We can access static methods by using class name or object reference

**Program Name** Declaring staticmethod methods  
demo41.py

```
class Demo:  
    @staticmethod  
    def sum(x, y):  
        print(x+y)  
  
    @staticmethod  
    def multiply(x, y):  
        print(x*y)
```

Demo.sum(2, 3)  
Demo.multiply(2,4)

**Output**

```
5  
8
```

#### Passing members of one class to another class

- ✓ We can pass members of one class into another class

**Program Name** Passing one member of one class to another class  
demo42.py

```
class Demo:  
    @staticmethod  
    def m(x):  
        print("static method")  
  
class Test:  
    def n(self):
```

```
print("instance method")
```

```
t=Test()  
d=Demo()
```

```
d.m(t)
```

## Output

```
static method
```

**Program Name** demo43.py

```
class Employee:  
    def __init__(self, eno, ename, esal):  
        self.eno=eno  
        self.ename=ename  
        self.esal=esal  
  
    def display(self):  
        print('Employee Number:', self.eno)  
        print('Employee Name:', self.ename)  
        print('Employee Salary:', self.esal)
```

```
class Test:  
    @staticmethod  
    def modify(emp):  
        emp.esal=emp.esal+10000  
        emp.display()
```

```
e=Employee(101,'Nireekshan',1000)  
Test.modify(e)
```

## Output

```
Employee Number: 101  
Employee Name: Nireekshan  
Employee Salary: 11000
```

## Inner classes

- ✓ Writing a class within another class is called as inner class or nested class

Program Name

Inner class

demo44.py

```
class A:  
    def __init__(self):  
        print("outer class object creation")
```

```
class B:  
    def __init__(self):  
        print("inner class object creation")
```

```
def m1(self):  
    print("inner class method")
```

```
a=A()  
b=a.B()  
b.m1()
```

output

```
outer class object creation  
inner class object creation  
inner class method
```

## Make a note

- ✓ So, to create inner class object, first we need to create outer class object
- ✓ With outer class object we can create inner class object.
- ✓ After created inner class object, we can access the members of inner class.

## Garbage Collection:

- ✓ In old languages like C++, programmer is responsible for both creation and destruction of objects. Usually programmer takes very much care to create object, chances are exists he may neglect destruction of useless objects.
- ✓ At certain point of time because of his negligence there may be a chance, total memory can be filled with useless objects which creates memory problems and total application will be down with Out of memory error.
- ✓ But in Python, internally a program will run in background always to destroy useless objects.
- ✓ So, because of this program the chance of failing Python program with memory problems is very less.
- ✓ This program is nothing but Garbage Collector.

## What is the main objective of Garbage Collector?

- ✓ Hence the main objective of Garbage Collector is to destroy useless objects.
- ✓ If an object does not have any reference variable, then that object eligible for Garbage Collection.

## How to enable and disable Garbage Collector in our program:

- ✓ By default, Garbage collector is enabled, but we can disable based on our requirement. In this context we can use the following functions of `gc` module.

### Importance functions in gc module

#### 1. `gc.isenabled()`

- ✓ This function returns True if garbage collector is enabled

#### 2. `gc.disable()`

- ✓ This function is used to is disable the garbage collector explicitly

#### 3. `gc.enable()`

- ✓ This function is used to is enable the garbage collector explicitly

<b>Program Name</b>	Garbage collector methods demo45.py
	<pre>import gc print(gc.isenabled())</pre>
<b>Output</b>	<pre>gc.disable() print(gc.isenabled())</pre> <pre>gc.enable() print(gc.isenabled())</pre>  <pre>True False True</pre>

## 18. Inheritance

What is inheritance?

- ✓ Creating new classes from already existing classes is called as inheritance.
- ✓ The existing class is called a **super** class or **base** class or **parent** class.
- ✓ The new class is called a called a **sub** class or **derived** class or **child** class.
- ✓ Inheritance allows sub classes to inherit the variables, methods and constructors of their super class.

Conclusion of the story:

- ✓ In inheritance all variables and methods by default available to child classes.

Advantage

- ✓ The main advantage of inheritance is code reusability.
- ✓ Application development time is very less.
- ✓ Redundancy (repetition) of the code is reducing.

How to implement inheritance?

- ✓ While declaring sub class, we need to pass super class name into sub class's parenthesis

**Program Name** Inheriting methods super class methods in sub class  
demo1.py

```
class One:  
    def m1(self):  
        print("Parent class m1 method")  
  
class Two(One):  
    def m2(self):  
        print("Child class m2 method")  
  
c = Two()  
  
c.m1()  
c.m2()
```

**output**

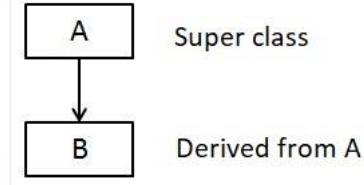
Parent class m1 method  
Child class m2 method

## Types of Inheritance:

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance

### 1. Single Inheritance

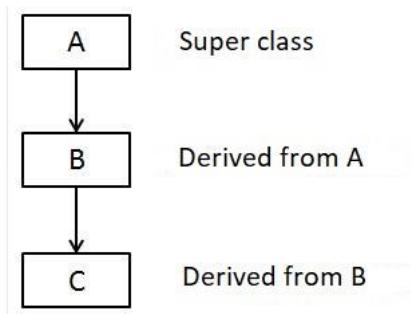
- ✓ Creating a sub class from a single super class is called single inheritance.



Program Name	Single inheritance demo2.py
	<pre>class A:     def m1(self):         print("A class m1 Method")  class B(A):     def m2(self):         print("Child B is derived from A class: m2 Method")  obj=B() obj.m1() obj.m2()</pre>
output	A class m1 Method Child B is derived from A class: m2 Method

## 2. Multi level Inheritance:

- ✓ A class is derived from another derived class is called multi level inheritance.



Program Name

Multilevel inheritance  
demo3.py

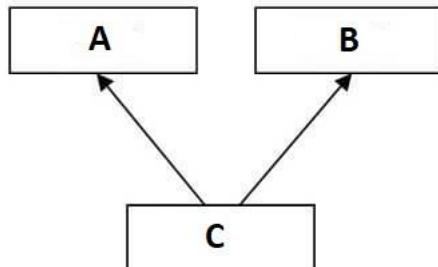
```
class A:  
    def m1(self):  
        print("Parent class A: m1 Method")  
  
class B(A):  
    def m2(self):  
        print("Child class B derived from A: m2 Method")  
  
class C(B):  
    def m3(self):  
        print("Child class C derived from B: m3 Method")  
  
obj=C()  
  
obj.m1()  
obj.m2()  
obj.m3()
```

output

```
Parent class A: m1 Method  
Child class B derived from A: m2 Method  
Child class C derived from B: m3 Method
```

### 3. Multiple inheritance

- ✓ Child class is derived from multiple super classes is called as multiple inheritance.



<b>Program Name</b>	Multiple inheritance demo4.py
<pre>class P1:     def m1(self):         print("Parent1 Method")  class P2:     def m2(self):         print("Parent2 Method")  class C(P1, P2):     def m3(self):         print("Child Method")  c=C() c.m1() c.m2() c.m3()</pre>	
<b>output</b>	Parent1 Method Parent2 Method Child Method

Parent classes can contain a method with same name

- ✓ There may be a chance of two parent classes can contains method which having same method name in both classes.
- ✓ Syntactically this is valid.

**Program Name** Parent classes having a method which having same name  
demo5.py

```
class P1:  
    def m1(self):  
        print("Parent1 Method")  
  
class P2:  
    def m1(self):  
        print("Parent2 Method")  
  
class C(P1, P2):  
    def m2(self):  
        print("Child Method")  
c=C()  
c.m2()
```

**Output**

Child Method

If parent classes contain a method with same name, then which method will access by child class

- ✓ If parent classes contain a method with same name, then while accessing child class will consider the order of parent classes in the declaration of child class.

**Scenarios**

class C(P1, P2): ==>P1's class method will be considered  
class C(P2, P1): ==>P2's class method will be considered

**Program Name** Parent classes having a method which having same name  
demo6.py

```
class P1:  
    def m1(self):  
        print("Parent1 Method")  
  
class P2:  
    def m1(self):  
        print("Parent2 Method")
```

```
class C(P1, P2):
    def m2(self):
        print("Child Method")

c=C()
c.m1()
c.m2()
```

## Output

Parent1 Method  
Child Method

**Program Name** Parent classes having a method which having same name  
demo7.py

```
class P1:
    def m1(self):
        print("Parent1 Method")

class P2:
    def m1(self):
        print("Parent2 Method")

class C(P2, P1):
    def m2(self):
        print("Child Method")

c=C()
c.m1()
c.m2()
```

## Output

Parent2 Method  
Child Method

## Constructors in Inheritance

- ✓ By default, the super class's constructor will be available to the sub class.

**Program Name** Super class constructor by default available to child class  
demo8.py

```
class A:
    def __init__(self):
```

```
print("super class A constructor")
```

```
class B(A):
    def m1():
        print("Child class B: m1 method from B")

b=B()
```

## Output

```
super class A constructor
```

If child class and super class both having constructors, then?

- ✓ If child class and super class both having constructors, if you create object to child class then child class constructor will be executed.
- ✓ While creating object the main priority is current class constructor.

Program Name

If child class and super class both having constructors, then  
demo9.py

```
class A:
    def __init__(self):
        print("super class A constructor")

class B(A):
    def __init__(self):
        print("child class B constructor")

b=B()
```

## Output

```
child class B constructor
```

Calling super class constructor from child class constructors

- ✓ We can call super class constructor from child class constructor by using super() function.
- ✓ super() is a pre-defined function which is useful to call the super class constructors, variables and methods from the child class.

Program Name

Calling super class constructor from child class constructor  
demo10.py

```
class A:
    def __init__(self):
        print("Super class A constructor ")
```

```
class B(A):
    def __init__(self):
        print("Child class B constructor")
        super().__init__()
b=B()
```

output

```
Child class B constructor
Super class A constructor
```

## Method Resolution Order (MRO)

- ✓ In multiple inheritance scenario, initially any specific attribute or method will be searched in the current class.
- ✓ In current class if not found, then next search continues into parent classes in depth-first left to right fashion.
- ✓ Searching in this order is called Method Resolution Order (MRO).

There are three principles followed by MRO

- ✓ **The first principle** is to search for the sub class before going for its base classes. Thus, if class B is inherited from A, it will search B first and then goes to A
- ✓ **The second principle** is, if any class is inherited from several classes, it searches in the order from left to right in the base classes.
- ✓ For example, if class C is inherited from A and B, syntactically class C(A, B), then first it will search in A and then in B.
- ✓ **The third principle** is that it will not visit any class more than once. That means a class in the inheritance hierarchy is traversed only once exactly.

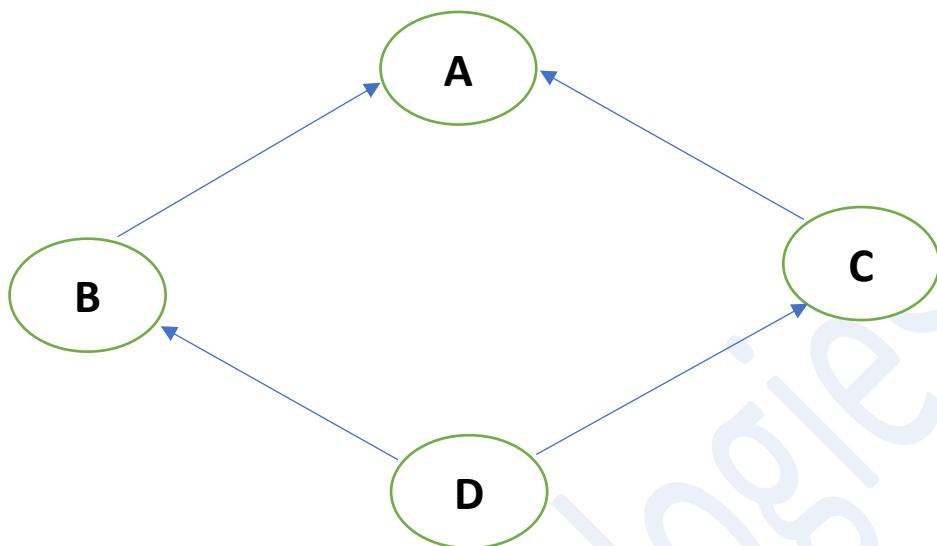
Why should we understand MRO?

- ✓ Understanding MRO gives us clear idea regarding which classes are executed and in which sequence.

Is there any predefined method to sequence of execution of classes?

- ✓ `classname.mro()`

## Demo program 1 for method resolution order



- ✓ mro(A)=A, object
- ✓ mro(B)=B, A, object
- ✓ mro(C)=C, A, object
- ✓ mro(D)=D, B, C, A, object

### Make a note

- ✓ object is the by default super class in python

Program Name	mro program demo11.py
<pre>class A:     def m1(self):         print("m1 from A")  class B(A):     def m1(self):         print("m1 from B")  class C(A):     def m1(self):         print("m1 from C")  class D(B, C):     def m1(self):         print("m1 from D")</pre>	

```
print(A.mro())
print(B.mro())
print(C.mro())
print(D.mro())
```

## Output

```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
 <class '__main__.A'>, <class 'object'>]
```

Program Name mro program  
demo12.py

```
class A:
    def m1(self):
        print("m1 from A")

class B(A):
    def m1(self):
        print("m1 from B")

class C(A):
    def m1(self):
        print("m1 from C")

class D(B, C):
    def m1(self):
        print("m1 from D")

c=C()
c.m1()
print(C.mro())
```

## Output

```
m1 from C
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

## Make a note

- ✓ In above program, with c object we are calling m1 method.
- ✓ So, m1() is already available in C class, so output is m1 from C

Program Name      mro program  
demo13.py

```
class A:  
    def m1(self):  
        print("m1 from A")  
  
class B(A):  
    def m1(self):  
        print("m1 from B")  
  
class C(A):  
    def m2(self):  
        print("m2 from C")  
  
class D(B, C):  
    def m1(self):  
        print("m1 from D")
```

```
c=C()  
c.m1()  
print(C.mro())
```

## Output

```
m1 from A  
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

## Make a note

- ✓ In above program, with c object we are calling m1 method.
- ✓ Now, m1() method is not available C class.
- ✓ So, next it will search in A class because A is super class to C.
- ✓ In A class m1() method is available, so output is **m1 from A**

Program Name      mro program  
demo14.py

```
class A:  
    def m1(self):  
        print("m1 from A")  
  
class B(A):  
    def m1(self):  
        print("m1 from B")  
  
class C(A):
```

```
def m1(self):
    print("m1 from C")

class D(B, C):
    def m1(self):
        print("m1 from D")

d=D()
d.m1()
print(D.mro())
```

## Output

```
m1 from D
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
 <class '__main__.A'>, <class 'object'>]
```

## Make a note

- ✓ In above program, with d object we are calling m1 method.
- ✓ So, m1() is already available in D class, so output is m1 from D

Program Name mro program  
demo15.py

```
class A:
    def m1(self):
        print("m1 from A")

class B(A):
    def m1(self):
        print("m1 from B")

class C(A):
    def m1(self):
        print("m1 from C")

class D(B, C):
    def m3(self):
        print("m1 from D")
```

```
d=D()
d.m1()
```

## Output

```
m1 from B
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
```

```
<class '__main__.A'>, <class 'object'>]
```

## Make a note

- ✓ In above program, with d object we are calling m1 method.
- ✓ Now, m1() method is not available D class.
- ✓ So, next it will search in B class because B is super class to D.
- ✓ In B class m1() method is available, so output is **m1 from B**

Program Name      mro program  
demo16.py

```
class A:  
    def m1(self):  
        print("m1 from A")  
  
class B(A):  
    def m4(self):  
        print("m4 from B")  
  
class C(A):  
    def m1(self):  
        print("m1 from C")  
  
class D(B, C):  
    def m3(self):  
        print("m3 from D")
```

```
d=D()  
d.m1()  
print(D.mro())
```

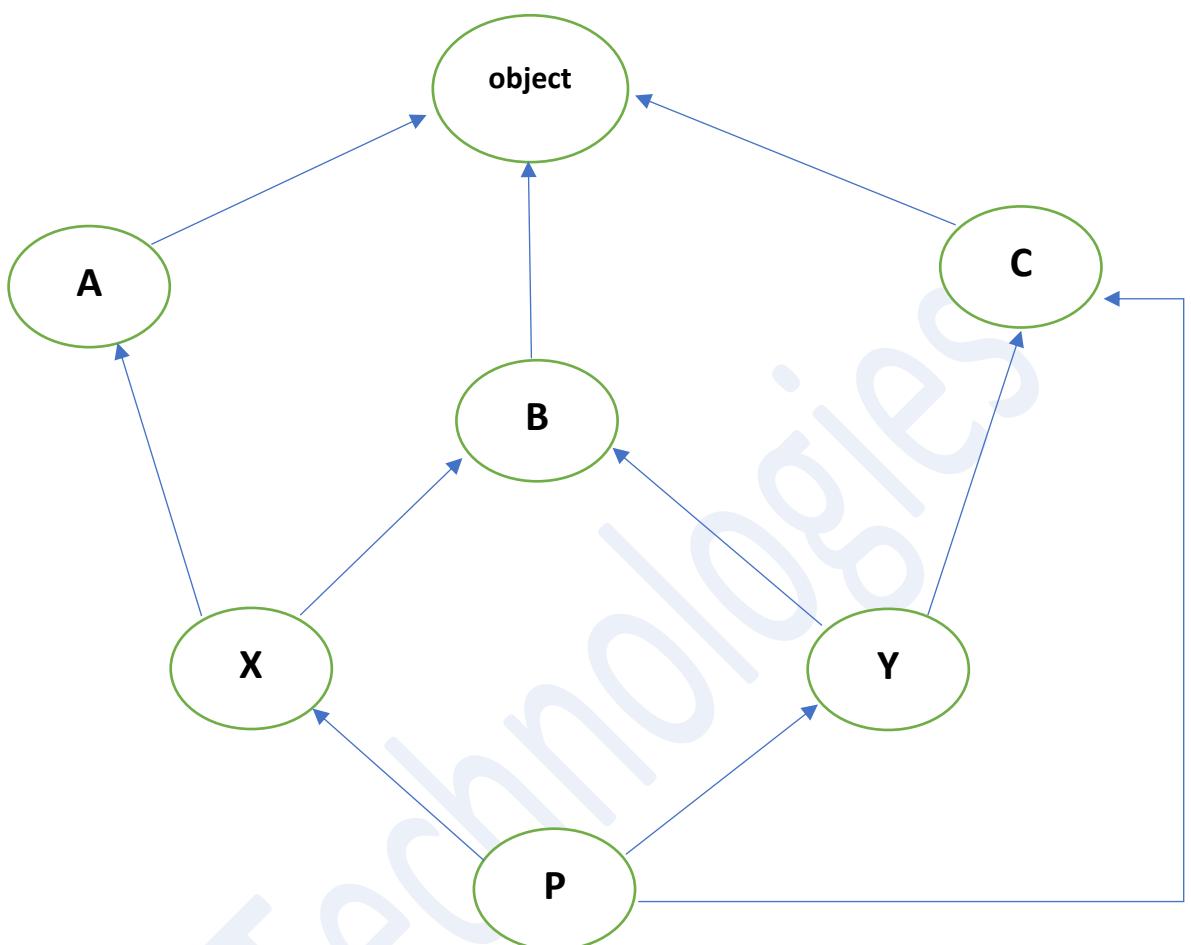
## Output

```
m1 from C  
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,  
<class '__main__.A'>, <class 'object'>]
```

## Make a note

- ✓ In above program, with d object we are calling m1 method.
- ✓ Now, m1() method is not available D class.
- ✓ So, next it will search in B class because B is super class to D
- ✓ m1() method is not available B class also.
- ✓ So, next it will search in C class because C is super class to D
- ✓ In C class m1() method is available, so output is **m1 from C**

Demo Program-2 for Method Resolution Order:



- ✓ mro(A)=A, object
- ✓ mro(B)=B, object
- ✓ mro(C)=C, object
- ✓ mro(X)=X, A, B, object
- ✓ mro(Y)=Y, B, C, object
- ✓ mro(P)=P, X, A, Y, B, C, object

<b>Program Name</b>	mro program demo17.py
---------------------	--------------------------

```

class A:
    def m1(self):
        print("m1 from A")

class B:
    def m1(self):
        print("m1 from B")
    
```

```
class C:  
    def m1(self):  
        print("m1 from C")  
  
class X(A, B):  
    def m1(self):  
        print("m1 from C")  
  
class Y(B, C):  
    def m1(self):  
        print("m1 from A")  
  
class P(X, Y, C):  
    def m1(self):  
        print("m1 from P")  
  
print(A.mro())#AO  
print(X.mro())#XABO  
print(Y.mro())#YBCO  
print(P.mro())#PXAYBCO
```

## Output

```
[<class '__main__.A'>, <class 'object'>]  
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]  
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]  
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,  
<class '__main__.A'>, <class 'object'>]
```

## super() function

- ✓ super() is a predefined function in python
- ✓ By using super() function we can call,
  - Super class constructor.
  - Super class variable.
  - Super class method.

### Program Name

Calling super class constructor from child class constructor  
demo18.py

```
class A:  
    def __init__(self):  
        print("Super class A constructor ")  
  
class B(A):
```

```
def __init__(self):
    print("Child class B constructor")
    super().__init__()
b=B()
```

## Output

```
Child class B constructor
Super class A constructor
```

Which scenario we can use super() function in child class to call super class members?

- ✓ Sometimes in super class and child class having same method names, same variable names.

```
class A:
    x=10
    def m(self):
        print("Super class A: m method")

class B(A):
    x=20
    def m(self):
        print("Child class B: m method")
```

- ✓ So, if we create object for child class then child class object can access only child class method and variables.
- ✓ In child class if we want to call super class methods and variables then we can use super() function to call

## Program

Super and sub classes having same method name, by using super() function child class is accessing super class method

Name demo19.py

```
class A:
    def m1(self):
        print("Super class A: m1 method")

class B(A):
    def m1(self):
        print("Child class B: m1 method")
        super().m1()
b=B()
b.m1()
```

**output**

Child class B: m1 method  
Super class A: m1 method

**Program**

Super and sub classes having same variable name, by using super()  
method child class is accessing super class variable

**Name**

demo20.py

```
class A:  
    x=10  
    def m(self):  
        print('Super class A: m method')  
  
class B(A):  
    x=20  
    def m(self):  
        print('Child class x variable', self.x)  
        print('Super class x variable', super().x)  
  
b=B()  
b.m()
```

**output**

Child class x variable 20  
Super class x variable 10

**Program  
Name**

Calling super class constructor and method from child class  
demo21.py

```
class Person:  
    def __init__(self, name, age):  
        self.name=name  
        self.age=age  
  
    def display(self):  
        print('Name:', self.name)  
        print('Age:', self.age)  
  
class Employee(Person):  
    def __init__(self, name, age, empno, address):  
        super().__init__(name, age)  
        self.empno=empno  
        self.address =address
```

```
def display(self):
    super().display()
    print('Emp No:', self.empno)
    print('Address:', self.address)

e1=Employee('Nireekshan',16,111,'Banglore')
e1.display()
```

## Output

Name: Nireekshan  
Age: 16  
Emp No: 111  
Address: Banglore

## Calling method of a specific super class

We can use the following approaches

- ✓ NameOfTheClass.methodname(self)
  - It will call specified class method
- ✓ super(NameOfTheClass, self).methodname
  - It will call method of specified class.
- ✓ classname.methodname(self)
  - It will call super class method

Program Name	classname.methodname(self) demo22.py
	<pre>class A:     def m1(self):         print('m1() method from A class')  class B(A):     def m1(self):         print('m1() method from B class')  class C(B):     def m1(self):         print('m1() method from C class')  class D(C):</pre>

```
def m1(self):
    print('m1() method from D class')

class E(D):
    def m1(self):
        A.m1(self)

e=E()
e.m1()
```

## Output

m1() method from A class

- ✓ super(NameOfTheClass, self).methodname
  - It will call method of super class of mentioned class name.

Program Name	super(NameOfTheClass, self).methodname demo23.py
	<pre>class A:     def m1(self):         print('m1() method from A class')  class B(A):     def m1(self):         print('m1() method from B class')  class C(B):     def m1(self):         print('m1() method from C class')  class D(C):     def m1(self):         print('m1() method from D class')  class E(D):     def m1(self):         super(D, self).m1()  e=E() e.m1()</pre>

## Output

m1() method from C class

## Different cases for super() function

### Case-1:

- ✓ By using super() function we cannot access parent class instance variable from child.
- ✓ From child if we want to access parent class instance variable then we should use self only.

#### Program

By using super() function we cannot access parent class instance variable from child.

#### Name

demo24.py

```
class P:  
    def __init__(self):  
        self.a=20  
  
class C(P):  
    def m1(self):  
        print(super().a)  
  
c=C()  
c.m1()
```

#### Error

AttributeError: 'super' object has no attribute 'a'

#### Program

From child if we want to access parent class instance variable then we should use self only.

#### Name

demo25.py

```
class P:  
    def __init__(self):  
        self.a=20  
  
class C(P):  
    def m1(self):  
        print(self.a)
```

```
c=C()  
c.m1()
```

#### Output

20

## Case-2:

- ✓ By using super() function we can access parent class static variables.

**Program Name** By using super() function we can access parent class static variables  
demo26.py

```
class P:  
    a=10  
    def m1(self):  
        print("m1 from Parent class")  
  
class C(P):  
    def m2(self):  
        print(super().a)  
  
c=C()  
c.m2()
```

**Output**

10

## Case-3:

By using super() function

- ✓ From child class constructor, we can access parent class
  - instance method,
  - static method
  - class method

By using super() function

- ✓ From child class instance method, we can access parent class
  - instance method,
  - static method
  - class method

By using super() function

- ✓ From child class constructor, we can access parent class
  - instance method,
  - static method
  - class method

## Program

From child class constructor, we can access parent class

- instance method,
- static method
- class method

Name demo27.py

```
class P:  
    def __init__(self):  
        print('Parent class Constructor')  
  
    def m1(self):  
        print('m1() instance method from Parent class')  
  
    @classmethod  
    def m2(cls):  
        print('m2() class method from Parent class')  
  
    @staticmethod  
    def m3():  
        print('m3() static method from Parent class')  
  
class C(P):  
    def __init__(self):  
        super().__init__()  
        super().m1()  
        super().m2()  
        super().m3()  
  
c=C()
```

## Output

```
Parent class Constructor  
m1() instance method from Parent class  
m2() class method from Parent class  
m3() static method from Parent class
```

By using super() function

- ✓ From child class instance method, we can access parent class
  - instance method,
  - static method
  - class method

## Program

From child class instance method, we can access parent class

- instance method,
- static method
- class method

## Name

demo28.py

```
class P:  
    def __init__(self):  
        print('Parent class Constructor')  
  
    def m1(self):  
        print('m1() instance method from Parent class')  
  
    @classmethod  
    def m2(cls):  
        print('m2() class method from Parent class')  
  
    @staticmethod  
    def m3():  
        print('m3() static method from Parent class')  
  
class C(P):  
    def __init__(self):  
        print("Child class constructor")  
  
    def m1(self):  
        super().__init__()  
        super().m1()  
        super().m2()  
        super().m3()  
  
c=C()  
c.m1()
```

## Output

```
Child class constructor  
Parent class Constructor  
m1() instance method from Parent class  
m2() class method from Parent class  
m3() static method from Parent class
```

## Case-4

- ✓ By using super() function we cannot access parent class instance method and constructor from child class classmethod.

### Program

By using super() function we cannot access parent class instance method and constructor from child class classmethod.

### Name

demo29.py

```
class P:  
    def __init__(self):  
        print('Parent Constructor')  
  
    def m1(self):  
        print('Parent instance method')  
  
    @classmethod  
    def m2(cls):  
        print('Parent class method')  
  
    @staticmethod  
    def m3():  
        print('Parent static method')  
  
class C(P):  
    @classmethod  
    def m1(cls):  
        super().__init__()  
        super().m1()
```

C.m1()

### Output

Error

## Case-5

- ✓ By using super() function we can access parent class static method and class method from child class classmethod

### Program

By using super() function we can access parent class static method and class method from child class classmethod

Name demo30.py

```
class P:  
    def __init__(self):  
        print('Parent Constructor')  
  
    def m1(self):  
        print('Parent instance method')  
  
    @classmethod  
    def m2(cls):  
        print('Parent class method')  
  
    @staticmethod  
    def m3():  
        print('Parent static method')  
  
class C(P):  
    @classmethod  
    def m1(cls):  
        super().m2()  
        super().m3()  
  
C.m1()
```

## Output

Parent class method  
Parent static method

## Case 6

- ✓ From Class Method of Child class, how to call parent class instance methods and constructors

Program From Class Method of Child class, how to call parent class instance methods and constructors

Name demo31.py

```
class A:  
    def __init__(self):  
        print('Parent constructor')  
  
    def m1(self):  
        print('Parent instance method')  
  
class B(A):  
    @classmethod
```

```
def m2(cls):
    super(B, cls).__init__(cls)
    super(B, cls).m1(cls)

B.m2()
```

**Output**

Parent constructor  
Parent instance method

## 19. Polymorphism

### What is Polymorphism?

- ✓ The process of representing "one form in many forms".
- ✓ Poly means many.
- ✓ Morphs means forms.
- ✓ Polymorphism means 'Many Forms'.

### Examples

Example 1: + operator acts as concatenation and arithmetic addition

Example 2: \* operator acts as multiplication and repetition operator

The following topics are examples to polymorphism

1. Duck Typing Philosophy of Python
2. Overloading
  - a. Operator Overloading
  - b. Method Overloading
  - c. Constructor Overloading
3. Overriding
  - a. Method overriding
  - b. constructor overriding

### 1. Duck Typing Philosophy of Python

- ✓ In python we cannot specify the type explicitly.
- ✓ At the runtime we can provide any type of value.
- ✓ Based on provided value at runtime the type will be considered automatically.
- ✓ So, Python is considered as Dynamically Typed Programming Language.
- ✓ Python follows this principle.
- ✓ This is called Duck Typing Philosophy of Python.

Program Name	Duck typing example demo1.py
	<pre>class Duck:     def talk(self):         print("Quack... Quack")</pre>
	<pre>class Cat:     def talk(self):</pre>

```
print("Moew... Moew ")

class Dog:
    def talk(self):
        print("Bow... Bow")

def m(obj):
    obj.talk()

duck = Duck()
m(duck)

cat = Cat()
m(cat)

dog = Dog()
m(dog)
```

## output

```
Quack... Quack
Moew... Moew
Bow... Bow
```

## Warning

- ✓ The problem in this approach is if obj does not contain talk() method then we will get **AttributeError**

<b>Program Name</b>	Duck typing example demo2.py
	<pre>class Duck:     def talk(self):         print("Quack.. Quack")  class Dog:     def bark(self):         print("Dog barking")  def m(obj):     obj.talk()  duck = Duck() dog=Dog()</pre>

```
m(duck)
m(dog)
```

## Error

```
AttributeError: 'Dog' object has no attribute 'talk'
```

## Solution

- ✓ We can solve this problem by using `hasattr()` function.

### Syntax

```
hasattr(obj, 'attributename')
```

- ✓ `attributename` can be method name or variable name

**Program Name** Duck typing example  
demo3.py

```
class Duck:
    def talk(self):
        print("Quack.. Quack")

class Cat:
    def talk(self):
        print("Moew... Moew ")

class Dog:
    def bark(self):
        print("Bow... Bow")

def m(obj):

    if hasattr(obj, 'talk'):
        obj.talk()

    elif hasattr(obj, 'bark'):
        obj.bark()

duck = Duck()
m(duck)

cat = Cat()
m(cat)
```

```
dog=Dog()  
m(dog)
```

## Output

```
Quack.. Quack  
Moew... Moew  
Bow... Bow
```

## 2. Overloading

### What is overloading?

- ✓ We can use same operator or methods for different purposes.
- ✓ There are 3 types of overloading
  1. Operator Overloading
  2. Method Overloading
  3. Constructor Overloading

### 1. Operator Overloading:

- ✓ We can use the same operator for multiple purposes, which is nothing but operator overloading.

### Is python supports operator overloading?

- ✓ Yes, python supports operator overloading.

### How operator overloading works in python?

- ✓ **+** addition operator can be used for **Arithmetic** addition and **String concatenation**
- ✓ **\*** multiplication operator can be used for **multiplication** for numbers and **repetition** for string

Program Name	+ operator on numbers and strings demo4.py  print(10+20) print("nireekshan" + "Subbamma")
--------------	---

## Output

```
30  
nireekshanSubbamma
```

**Program Name** \* operator on numbers and strings  
demo5.py

```
print(10*20)  
print("nireekshan" * 3)
```

**Output**

```
200  
nireekshannireekshannireekshan
```

+ addition operator

**Program Name** Trying to check type of user defined objects.  
demo6a.py

```
class Book:  
    def __init__(self, pages):  
        self.pages=pages
```

```
b1=Book(100)  
b2=Book(200)
```

```
print(type(b1))  
print(type(b2))
```

**Output**

```
<class '__main__.Book'>  
<class '__main__.Book'>
```

**Program Name** Trying to check type of user defined objects.  
demo6b.py

```
class Book:  
    def __init__(self, pages):  
        self.pages=pages
```

```
b1=Book(100)  
b2=Book(200)
```

```
print(type(b1))
print(type(b2))

print(type(b1.pages))
print(type(b2.pages))
```

### Output

```
<class '__main__.Book'>
<class '__main__.Book'>
<class 'int'>
<class 'int'>
```

### Program Name

Pulling int values from objects and using + operator on int values  
demo6c.py

```
class Book:
    def __init__(self, pages):
        self.pages=pages

b1=Book(100)
b2=Book(200)

print(type(b1))
print(type(b2))

print(type(b1.pages))
print(type(b2.pages))

print(b1.pages + b2.pages)
```

### Output

```
<class '__main__.Book'>
<class '__main__.Book'>
<class 'int'>
<class 'int'>
300
```

**Program Name** Pulling int values from objects and using `__add__` method on int values  
demo6d.py

```
class Book:  
    def __init__(self, pages):  
        self.pages=pages  
  
b1=Book(100)  
b2=Book(200)  
  
print(type(b1))  
print(type(b2))  
  
print(type(b1.pages))  
print(type(b2.pages))  
  
print(b1.pages + b2.pages)  
  
print((b1.pages).__add__(b2.pages))
```

## Output

```
<class '__main__.Book'>  
<class '__main__.Book'>  
<class 'int'>  
<class 'int'>  
300  
300
```

**Program Name** Checking `__add__` method in int class.  
demo6e.py

```
class Book:  
    def __init__(self, pages):  
        self.pages=pages  
  
b1=Book(100)  
b2=Book(200)  
  
print(type(b1))  
print(type(b2))  
  
print(type(b1.pages))  
print(type(b2.pages))  
  
print(b1.pages + b2.pages)
```

```
print((b1.pages).__add__(b2.pages))  
print(dir(int))
```

**Output**

```
<class '__main__.Book'>  
<class '__main__.Book'>  
<class 'int'>  
<class 'int'>  
300  
300  
[...,'__add__',...]
```

**Program Name** Using + operator on int values.  
demo6f.py

```
print(10+20)
```

**Output**

```
30
```

**Program Name** Using \_\_add\_\_ method on int values.  
demo6g.py

```
print((10).__add__(20))
```

**Output**

```
30
```

- ✓ Trying to use + operator for user defined objects

Program Name

Error: Trying to use + operator for user defined objects  
demo6.py

```
class Book:  
    def __init__(self, pages):  
        self.pages=pages  
  
b1=Book(100)  
b2=Book(200)  
  
print(b1+b2)
```

Error

TypeError: unsupported operand type(s) for +: 'Book' and 'Book'

- ✓ We can overload **+** operator to work with Book objects also. i.e Python supports Operator Overloading.

## Magic methods

- ✓ For every operator **Magic** methods are available.
- ✓ To overload any operator, we should override that Method in our class.
- ✓ Internally + operator is implemented by using **\_\_add\_\_()** method.
- ✓ This method is called magic method for + operator.
- ✓ We can override this method in our class.

Program Name

Trying to using + operator on user defined objects  
demo7.py

```
class Book:  
    def __init__(self, pages):  
        self.pages=pages  
  
b1=Book(100)  
b2=Book(200)  
print("The total number of pages: ", b1+b2)
```

Error

TypeError: unsupported operand type(s) for +: 'Book' and 'Book'

**Program Name** Overloading magic method in our program demo8.py

```
class Book:  
    def __init__(self, pages):  
        self.pages=pages  
  
    def __add__(self, others):  
        return self.pages + others.pages  
  
b1=Book(100)  
b2=Book(200)  
  
print("The total number of pages: ", (b1+b2))
```

**Output**

The total number of pages: 300

List of operators and corresponding magic methods.

+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
/	object.__div__(self, other)
//	object.__floordiv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other)
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other)
<	object.__lt__(self, other)
<=	object.__le__(self, other)
>	object.__gt__(self, other)
>=	object.__ge__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)

**Program Name** Using less than and greater than symbols on user defined objects  
demo9.py

```
class Student:  
    def __init__(self, name, marks):  
        self.name=name  
        self.marks=marks  
  
    print("10>20 =",10>20)  
  
    s1=Student("Nireekshan", 100)  
    s2=Student("Suryakantham", 200)  
  
    print("s1>s2=", s1>s2)  
    print("s1<s2=", s1<s2)  
    print("s1<=s2=", s1<=s2)  
    print("s1>=s2=", s1>=s2)
```

**Error**

```
10>20 = False  
TypeError: '>' not supported between instances of 'Student' and  
'Student'
```

**Program Name** Overloading magic methods on our class.  
demo10.py

```
class Student:  
    def __init__(self, name, marks):  
        self.name=name  
        self.marks=marks  
  
    def __gt__(self, other):  
        return self.marks>other.marks  
  
    def __le__(self, other):  
        return self.marks<=other.marks  
  
    print("10>20 =",10>20)  
  
    s1=Student("Nireekshan", 100)  
    s2=Student("Suryakantham", 200)  
  
    print("s1>s2=", s1>s2)  
    print("s1<s2=", s1<s2)  
    print("s1<=s2=", s1<=s2)  
    print("s1>=s2=", s1>=s2)
```

## output

```
10>20 = False
```

```
s1>s2= False  
s1<s2= True  
s1<=s2= True  
s1>=s2= False
```

**Program Name** Using star symbol on user defined objects  
`demo11.py`

```
class Employee:  
  
    def __init__(self, name, salary):  
        self.name=name  
        self.salary=salary  
  
class TimeSheet:  
  
    def __init__(self, name, days):  
        self.name=name  
        self.days=days  
  
emp=Employee('Nireekshan', 500)  
ts=TimeSheet('Nireekshan', 25)  
print('This Month Salary:', emp*ts)
```

## Error

```
TypeError: unsupported operand type(s) for *: 'Employee' and  
'TimeSheet'
```

**Program Name** Overloading magic method on our class.  
`demo12.py`

```
class Employee:  
  
    def __init__(self, name, salary):  
        self.name=name  
        self.salary=salary  
  
    def __mul__(self, other):  
        return self.salary*other.days  
  
class TimeSheet:
```

```
def __init__(self, name, days):
    self.name=name
    self.days=days
```

```
emp=Employee('Nireekshan', 500)
ts=TimeSheet('Nireekshan', 25)
print('This Month Salary:', emp*ts)
```

## Output

This Month Salary: 12500

## 2. Method Overloading:

- ✓ If 2 methods having same name but different type of arguments, then those methods are said to be overloaded methods.

### Example

```
m1(a)
m1(p, q)
m1(x, y, z)
```

- ✓ But in Python Method overloading is not possible.
- ✓ If we are trying to declare multiple methods with same name and different number of arguments, then Python will always consider **only last method**.

**Program Name** Methods names same and number of arguments are different  
demo13.py

```
class Demo:
    def m1(self):
        print('no-arg method')

    def m1(self, a):
        print('one-arg method')

    def m1(self, a, b):
        print('two-arg method')
```

```
d= Demo()
d.m1()
```

### Error

**TypeError:** m1() missing 2 required positional arguments: 'a' and 'b'

**Program Name** Methods names same and number of arguments different  
demo14.py

```
class Demo:  
    def m1(self):  
        print('no-arg method')  
  
    def m1(self, a):  
        print('one-arg method')  
  
    def m1(self, a, b):  
        print('two-arg method')  
  
d=Demo()  
d.m1(10)
```

**Error**

`TypeError: m1() missing 2 required positional arguments: 'a' and 'b'`

**Program Name** Methods names same and number of arguments different  
demo15.py

```
class Demo:  
    def m1(self):  
        print('no-arg method')  
  
    def m1(self, a):  
        print('one-arg method')  
  
    def m1(self, a, b):  
        print('two-arg method')  
  
d=Demo()  
d.m1(10,20)
```

**Output**

two-arg method

## Conclusion

- ✓ If we are trying to declare multiple methods with same name and different number of arguments, then Python will always consider **only last method**.
- ✓ In the above program python will consider only last method.

## How we can handle overloaded method requirements in Python:

- ✓ Most of the times, if method with variable number of arguments required then we can handle with default arguments or with variable number of argument methods.

**Program Name** Demo program with default arguments  
demo16a.py

```
def sum(a=None, b=None):
    print(a+b)

sum(10, 20)
```

**Output** 30

**Program Name** Demo Program with Default Arguments:  
demo16.py

```
class Demo:
    def sum(self, a=None, b=None, c=None):

        if a!=None and b!= None and c!= None:
            print('The Sum of 3 Numbers:', a + b + c)

        elif a!=None and b!= None:
            print('The Sum of 2 Numbers:', a + b)

        else:
            print('Please provide 2 or 3 arguments')

d=Demo()
d.sum(10,20,30)
d.sum(10,20)
d.sum(10)
```

**output**

```
The Sum of 3 Numbers: 60
The Sum of 2 Numbers: 30
Please provide 2 or 3 arguments
```

**Program Name** Demo Program with Variable Number of Arguments:  
demo17.py

```
class Demo:  
    def sum(self, *a):  
        total=0  
        for x in a:  
            total=total+x  
        print('The Sum:', total)  
  
d=Demo()  
  
d.sum(10,20,30)  
d.sum(10,20)  
d.sum(10)  
d.sum()
```

**Output**

```
The Sum: 60  
The Sum: 30  
The Sum: 10  
The Sum: 0
```

### 3. Constructor Overloading:

- ✓ Constructor overloading is not possible in Python.
- ✓ If we define multiple constructors, only the **last constructor** will be considered.

**Program Name** Constructor overloading  
demo18.py

```
class Demo:  
    def __init__(self):  
        print('No-Arge Constructor')  
  
    def __init__(self, a):  
        print('One-Arge constructor')  
  
    def __init__(self, a, b):  
        print('Two-Arge constructor')
```

**Output** d1=Demo()

```
TypeError: __init__() missing 2 required positional arguments: 'a' and  
'b'
```

Program Name	Constructor overloading demo19.py
Output	<pre>class Demo:     def __init__(self):         print('No-Arg Constructor')      def __init__(self, a):         print('One-Arg constructor')      def __init__(self, a, b):         print('Two-Arg constructor')  d1=Demo(10)  TypeError: __init__() missing 1 required positional argument: 'b'</pre>

Program Name	Constructor overloading demo20.py
Output	<pre>class Demo:     def __init__(self):         print('No-Arg Constructor')      def __init__(self, a):         print('One-Arg constructor')      def __init__(self, a, b):         print('Two-Arg constructor')  d1=Demo(10,20)  Two-Arg constructor</pre>

In the above program only Two-Arg Constructor is available.

- ✓ But based on our requirement we can declare constructor with default arguments and variable number of arguments.

## Constructor with Default Arguments:

Program Name	Constructor with default arguments demo21a.py
	<pre>class Demo:     def __init__(self, a=None, b=None, c=None):         print(a)         print(b)         print(c)  d1=Demo()</pre>
Output	None None None

Program Name	Constructor with default arguments demo21b.py
	<pre>class Demo:     def __init__(self, a=None, b=None, c=None):         print(a)         print(b)         print(c)  d1=Demo(10)</pre>
Output	10 None None

**Program Name** Constructor with default arguments  
demo21c.py

```
class Demo:  
    def __init__(self, a=None, b=None, c=None):  
        print(a)  
        print(b)  
        print(c)  
  
d1=Demo(10, 20)
```

**Output**

```
10  
20  
None
```

**Program Name** Constructor with default arguments  
demo21.py

```
class Demo:  
    def __init__(self, a=None, b=None, c=None):  
        print(a)  
        print(b)  
        print(c)  
  
d1=Demo(10, 20, 30)
```

**Output**

```
10  
20  
30
```

**Constructor with Variable Number of Arguments:**

**Program Name** Constructor with variable number of arguments  
demo22.py

```
class Demo:  
    def __init__(self, *a):  
        print('Constructor with variable number of arguments')
```

```
d1=Demo()  
  
d2 = Demo(10)  
d3 = Demo(10,20)  
d4 = Demo(10,20,30)  
d5 = Demo(10,20,30,40,50,60)
```

## Output

```
Constructor with variable number of arguments  
Constructor with variable number of arguments
```

### 3. Method overriding:

- ✓ Whatever members available in the parent class, those are by-default available to the child class through inheritance.
- ✓ If the child class not satisfied with parent class implementation, then child class is allowed to redefine that method in the child class based on its requirement.
- ✓ This concept is called as overriding.
- ✓ Overriding concept applicable for both methods and constructors.

## Demo Program for Method overriding:

Program Name	Creating parent and child class methods demo23.py
<pre>class P:     def property(self):         print('Money, Land, Gold')      def marry(self):         print('Subbamma')  class C(P):     def study(self):         print('Studies done waiting for job')  c=C()  c.property() c.marry() c.study()</pre>	

## Output

```
Money, Land, Gold
```

Subbamma  
Studies done waiting for job

**Program Name** Child class is overriding parent class method  
demo24.py

```
class P:  
    def property(self):  
        print('Money, Land, Gold')  
  
    def marry(self):  
        print('Subbamma')  
  
class C(P):  
  
    def study(self):  
        print('Studies done waiting for job')  
  
    def marry(self):  
        print('Anushka')  
  
c=C()  
  
c.property()  
c.study()  
c.marry()
```

## Output

Money, Land, Gold  
Studies done waiting for job  
Anushka

## Constructor overriding

- ✓ If child class does not have constructor, then parent class constructor will be executed at the time of child class object creation

**Program Name** Constructor overriding  
demo25.py

```
class P:  
    def __init__(self):  
        print('Parent class Constructor')  
  
class C(P):  
    def m(self):  
        print('m() method from Child Constructor')
```

```
c=C()
```

Output

Parent class Constructor

- ✓ If child class having constructor, then child class constructor will be executed at the time of child class object creation

Program Name Parent and child classes having constructors  
demo26.py

```
class P:  
    def __init__(self):  
        print('Parent class Constructor')  
  
class C(P):  
    def __init__(self):  
        print('Child class Constructor')  
  
c=C()
```

Output

Child class Constructor

Calling parent class constructor from child class constructor

- ✓ From child class constructor we can call parent class constructor by using super() method.

Program Name Calling parent class constructor from child class constructor  
demo27.py

```
class P:  
    def __init__(self):  
        print('Parent class Constructor')  
  
class C(P):  
    def __init__(self):  
        print('Child class Constructor')  
        super().__init__()  
  
c=C()
```

## Output

```
Child class Constructor  
Parent class Constructor
```

## Program Name

Constructor overriding  
demo28.py

```
class Person:  
    def __init__(self, name, age):  
        self.name=name  
        self.age=age  
  
class Employee(Person):  
    def __init__(self, name, age, eno, esal):  
        super().__init__(name, age)  
        self.eno=eno  
        self.esal=esal  
  
    def display(self):  
        print('Employee Name:', self.name)  
        print('Employee Age:', self.age)  
        print('Employee Number:', self.eno)  
        print('Employee Salary:', self.esal)
```

```
e1=Employee('Nireekshan', 16, 872425,26000)  
e1.display()
```

```
e2=Employee('Mohan',20,872426,36000)  
e2.display()
```

## Output

```
Employee Name: Nireekshan  
Employee Age: 16  
Employee Number: 872425  
Employee Salary: 26000
```

```
Employee Name: Mohan  
Employee Age: 20  
Employee Number: 872426  
Employee Salary: 36000
```

## 20. OOPS - part 4 – abstract class and interface

In python which things we can make as an abstract?

- ✓ In Python,
  - ✓ A class can be abstract.
  - ✓ A method can be abstract.

There are two types of methods in-terms of implementation

1. Implemented methods.
2. Un-implemented method.

### 1. Implemented method

- ✓ A method which have a **method name** and **method body**, that method is called as implemented method.
- ✓ Also called as concrete method or non-abstract method

```
class Test:  
    def m1(self):  
        print("This is body of the method")
```

### 2. Un-implemented method

- ✓ A method which have **only method name** and **no method body**, that method is called as un-implemented method.
- ✓ Also called as non-concrete or abstract method.

So, how to declare abstract method

- ✓ In python we can declare abstract method by using **@abstractmethod** decorator as follows.

```
class Demo:  
    @abstractmethod  
    def one(self):  
        pass  
  
    def two(self):  
        print("implemented method")
```

## abstract method

- ✓ `@abstractmethod` decorator presents in `abc` module.
- ✓ We should `import` abc module to create abstract methods otherwise we will get error.

abc ==> abstract base class module.

- ✓ abstract class and interface can contain abstract methods.
- ✓ abstract method will not have method body.
- ✓ By using `@abstractmethod` **decorator** we can declare a method as abstract method.
- ✓ If any method having no method body, then we need to give `pass` keyword
- ✓ For abstract method we need to give implementation in sub class of abstract class.

## Syntax

```
@abstractmethod  
def methodName(self):  
    pass
```

**Program Name** class contains two abstract methods and one implemented method  
`demo1.py`

```
from abc import *  
class Demo1(ABC):  
  
    @abstractmethod  
    def m1(self):  
        pass  
  
    @abstractmethod  
    def m2(self):  
        pass  
  
    def m3(self):  
        print("Implemented method")
```

## output

## abstract class

- ✓ Every abstract class in Python should be derived from **ABC** class which is present in **abc** module.
- ✓ abstract class can contain,
  - constructors
  - variables
  - **abstract methods**
  - **non-abstract methods**
  - **sub class**
- ✓ abstract methods should be implemented in **sub class** of abstract class.  
(demo2.py)
- ✓ If **sub class** didn't provide implementation of abstract method, then that sub class automatically will become an abstract class. (demo3.py)
- ✓ **Make a note**
  - If any sub class is inheriting abstract class and not providing implementations for abstract methods, then automatically child class is also will become abstract class.
- ✓ If any class is inheriting this **sub class**, then that sub class should provide the implementation for abstract methods. (demo4.py)
- ✓ **object creation is not possible** for abstract class. (demo4.py or demo8.py)
- ✓ We can create object for child class of abstract class to access implemented methods.

## Syntax

```
from abc import ABC, abstractmethod

class Name_Of_The_Class(ABC):

    @abstractmethod
    def method1(self):
        pass

    def method2(self):
        print("Implemented method")
```

Program Name	abstract class and sub class demo2.py
	<pre>from abc import ABC, abstractmethod  class Bank(ABC):      def bank_info(self):         print("Welcome to bank")      @abstractmethod     def interest(self):         pass  class SBI(Bank):     def interest(self):         print("In sbi bank 5 rupees interest")  s= SBI() s.bank_info () s.interest()</pre>

## Output

```
Welcome to bank
In sbi bank 5 rupees interest
```

## Explanation

- ✓ Above example, Bank is an abstract class which having abstract method as `intertest()` method.
- ✓ SBI class is a child class to Bank class, so SBI class should provide implementation for abstract method which is available in Bank abstract class.
- ✓ Finally creating object to sub class which is SBI and calling implemented method which is interest.

If child class missed to provide abstract methods implementation then,

- ✓ If any class is inheriting abstract class, then that child class should provide the implementation for abstract methods.
- ✓ If child class is not providing implementation for abstract methods, then we should not create object to that child class.
- ✓ If this case if we create object to child class, then it throws error as:
  - `TypeError`: Can't instantiate abstract class SBI with abstract methods interest

**Program Name** If child class missed to provide abstract methods implementation then, demo3.py

```
from abc import ABC, abstractmethod

class Bank(ABC):

    def bank_info(self):
        print("Welcome to bank")

    @abstractmethod
    def interest(self):
        pass

class SBI(Bank):
    def balance(self):
        print("Balance is 100")

s= SBI()

s.bank_info()
s.balance()
```

**Output**

**TypeError:**  
Can't instantiate abstract class SBI with abstract methods interest

## Explanation

- ✓ Above example, Bank is an abstract class and having abstract method which is an `interest()` method.
- ✓ SBI is a child class to Bank class, so SBI should provide implementation for abstract method which is available in Bank abstract class, but not providing the implementation to `interest()` method.
- ✓ So, if we create object to child class then we will get **TypeError**.

## Solution

- ✓ So, now SBI will become an abstract class because it's not providing implementation for abstract method `interest()` which is from Bank.
- ✓ If any child class is inheriting SBI class, then that child class should provide the implementation of `interest` method.

Program Name abstract class and sub class of abstract class  
demo4.py

```
from abc import ABC, abstractmethod

class Bank(ABC):

    def bank_info(self):
        print("Welcome to bank")

    @abstractmethod
    def interest(self):
        pass

class SBI(Bank):
    def balance(self):
        print("Balance is 100")

class Sub1(SBI):
    def interest(self):
        print("In sbi bank interest is 5 rupees")

s= Sub1()
s.bank_info()
s.balance()
s.interest()
```

## Output

```
Welcome to bank
Balance is 100
In sbi bank interest is 5 rupees
```

abstract class can contain concrete methods also.

- ✓ As discussed abstract class can contains implemented methods also along with abstract methods.

Program Name abstract class can contain concrete methods also.  
demo5.py

```
from abc import ABC, abstractmethod

class Bank(ABC):
    @abstractmethod
```

```
def interest(self):
    pass

def offers(self):
    print("Providing offers")

class SBI(Bank):
    def interest(self):
        print("In SBI bank 5 rupees interest")

s= SBI()
s.interest()
s.offers()
```

## Output

```
In SBI bank 5 rupees interest
Providing offers
```

Can abstract class contains more sub classes?

- ✓ Yes, an abstract class can contain more than one sub classes.

On which scenario an abstract class contains more than one child classes?

- ✓ If different child classes require different kind of implementations, then at that time an abstract class can contain more than one sub class

<b>Program Name</b>	abstract class and two sub classes of abstract class demo6.py
	<pre>from abc import ABC, abstractmethod  class Bank(ABC):      def bank_info(self):         print("Welcome to bank")      @abstractmethod     def interest(self):         pass  class SBI(Bank):     def interest(self):         print("In sbi 5 rupees interest")    class HDFC(Bank):     def interest(self):         print("In HDFC 7 rupees interest")</pre>

```
s= SBI()
h=HDFC()

s.bank_info()
s.interest()

h.bank_info()
h.interest()
```

## Output

```
Welcome to bank
In sbi 5 rupees interest

Welcome to bank
In HDFC 7 rupees interest
```

**Program Name** Two child classes for abstract class and having different implementation  
demo7.py

```
from abc import ABC, abstractmethod

class One(ABC):

    @abstractmethod
    def calculate(self, a):
        pass

    def m1(self):
        print("implemented method")

class Square(One):
    def calculate(self, a):
        print("square: ", (a*a))

class Cube(One):
    def calculate(self, a):
        print("cube: ", (a*a*a))

s = Square()
c = Cube()

s.calculate(2)
c.calculate(2)
```

## output

```
square: 4  
cube: 8
```

## Abstract class object creation

- ✓ If any class is inheriting ABC class,
  - If that class contains abstract method, then that class will become an abstract class.
  - So, for abstract class, **object creation is not possible** for abstract class.

**Program Name** object creation is not possible to abstract class  
demo8.py

```
from abc import ABC, abstractmethod

class Bank(ABC):

    def bank_info(self):
        print("Welcome to bank")

    @abstractmethod
    def interest(self):
        pass

class SBI(Bank):
    def interest(self):
        print("SBI interest is 5 rupees")

obj = Bank()
```

## Output

```
TypeError:  
Can't instantiate abstract class Bank with abstract methods interest
```

Different cases and scenarios for abstract class object creation

## Case 1:

- ✓ We can create object for normal class.
- ✓ As per below example Demo is a normal class.

Program Name	We can create object for normal class demo9.py
	<pre>from abc import * class Demo:     def m(self):         print("calling")</pre>
Output	calling

## Case 2:

- ✓ If any class is inheriting ABC class,
  - If that class cannot contain abstract method, then happily we can create object to that class.

Program Name	Valid: creating object for abstract class which have not even single abstract method demo10.py
	<pre>from abc import *  class Demo(ABC):     def m(self):         print("calling")</pre>
Output	calling

## Case 3:

- ✓ If any class is inheriting ABC class,
  - If that class contains abstract method, then that class will become an abstract class.
  - So, for abstract class, **object creation is not possible** for abstract class.

### Program

Valid: object creation is not possible for abstract class which having abstract methods

### Name

demo11.py

```
from abc import *
class Demo(ABC):
    @abstractmethod
    def m(self):
        pass
    def n(self):
        print("implemented method")
t=Demo()
```

### Output

**TypeError:**

Can't instantiate abstract class Demo with abstract methods interest

## Case 4:

- ✓ If any class is not inheriting ABC class, then **we can create object** for that class even though it contains abstract method.
- ✓ As per below example, class not inheriting ABC class, but it contains abstract method

### Program

Valid: If any class is not inheriting ABC class, then **we can create object** for that class even though it contains abstract method

### Name

demo12.py

```
from abc import *
```

```
class Demo:  
    @abstractmethod  
    def m(self):  
        pass  
  
    def n(self):  
        print("implemented method")  
  
d=Demo()  
  
d.m()  
d.n()
```

Output

```
implemented method
```

## Interface

### What is an interface?

- ✓ An abstract class is a class which may contains some abstract methods as well as non-abstract methods also.
- ✓ Imagine there is an abstract class which contain **only abstract methods** and there are no concrete methods, then that class is called as an **interface**.
- ✓ This means, an interface is an abstract class, but it contains only abstract methods.

### Definition 1

- ✓ We can say, an interface is nothing but an abstract class which can contains only abstract methods.

### Make a note

- ✓ In python there is no separate keyword to create **interface**.
- ✓ We can create interface by using abstract class which having all are abstract methods.
- ✓ interface can contain,
  - constructors
  - variables
  - **abstract methods**
  - **sub class**
- ✓ abstract methods will be implemented in **sub class** of interface. (demo13.py)
- ✓ If **sub class** didn't provide implementation of abstract methods, then automatically that sub class will become an abstract class. (demo14.py)
- ✓ **Make a note**
  - If any sub class is inheriting interface and not providing implementations for abstract methods, then automatically that sub class will become abstract class.
- ✓ If any class is inheriting this sub class, then that sub class should provide the implementation for abstract methods. (demo15.py)
- ✓ **object creation is not possible for interface** (demo14.py)
- ✓ We can create object for child class of interface to access implemented methods.

**Program Name** Interface having two abstract methods and one sub class demo13.py

```
from abc import ABC, abstractmethod

class Bank(ABC):

    @abstractmethod
    def balance_check(self):
        pass

    @abstractmethod
    def interest(self):
        pass

class SBI(Bank):

    def balance_check(self):
        print("Balance is 100 rupees")

    def interest(self):
        print("SBI interest is 5 rupees")

s = SBI()

s.balance_check()
s.interest()
```

**Output**

```
Balance is 100 rupees
SBI interest is 5 rupees
```

**Program**

Sub class should provide implementation for all abstract methods of interface otherwise sub class considered as abstract class.

**Name**

demo14.py

```
from abc import ABC, abstractmethod

class Bank(ABC):

    @abstractmethod
    def balance_check(self):
        pass

    @abstractmethod
    def interest(self):
```

```
pass

class SBI(Bank):
    def balance_check(self):
        print("Balance is 100 rupees")

s = SBI()
s.balance_check()
```

## Output

```
TypeError:
Can't instantiate abstract class SBI with abstract methods interest
```

## Program

First sub class left out one abstract method implementation then second sub class implementing that abstract method

## Name

demo15.py

```
from abc import ABC, abstractmethod

class Bank(ABC):

    @abstractmethod
    def balance_check(self):
        pass

    @abstractmethod
    def interest(self):
        pass

class SBI(Bank):
    def balance_check(self):
        print("Balance is 100 rupees")

class Sub1(SBI):
    def interest(self):
        print("SBI interest is 5 rupees")

s = Sub1()

s.balance_check()
s.interest()
```

## Output

Balance is 100 rupees  
SBI interest is 5 rupees

**Program Name**

one interface and two sub classes to that interface  
demo16.py

```
from abc import *
class DBInterface(ABC):
    @abstractmethod
    def connect(self):
        pass
    @abstractmethod
    def disconnect(self):
        pass
class Oracle(DBInterface):
    def connect(self):
        print('Connecting to Oracle Database...')
    def disconnect(self):
        print('Disconnecting to Oracle Database...')
class Sybase(DBInterface):
    def connect(self):
        print('Connecting to Sybase Database...')
    def disconnect(self):
        print('Disconnecting to Sybase Database...')
```

**output**

```
C:\Users\Nireekshana\Desktop>py demo13.py
Enter Database Name either Oracle or Sybase: Oracle
Connecting to Oracle Database...
Disconnecting to Oracle Database...

C:\Users\Nireekshana\Desktop>py demo13.py
Enter Database Name either Oracle or Sybase: Sybase
Connecting to Sybase Database...
Disconnecting to Sybase Database...
```

## Make a note

- ✓ `globals()[str]` is a predefined function, it can converts the string 'str' into a class name and returns the classname.

## When should we go for ...?

### When should we go for **interface**?

- ✓ Interface can contain only abstract methods means only requirement specification.
- ✓ If we don't know anything about implementation of requirement then we should go for interface.
- ✓ Interface cannot contain implemented methods.

### When should we go for **abstract class**?

- ✓ An abstract class is a class which can contains few implemented methods and few unimplemented methods as well.
- ✓ We know implementation about requirement but not completely, means we know partial implementation then we should go for abstract class.
- ✓ We can say as; abstract class is partially implemented class.

### When should we go for **concrete class**?

- ✓ Concrete class is a class which can contain all are implemented methods.
- ✓ If we know complete implementation about requirements, then we should go for concrete class.

## Double underscore (Name mangling)

- ✓ We can give two underscores symbols before an attribute name

### Example

```
__name = 'Nireekshan'
```

- ✓ If any attribute name having two underscores symbols before the name then internally it is replaced with **\_classname\_\_variable name**, where classname is the current class name.

<b>Program Name</b>	Double underscores demo17.py
	<pre>class Demo:     def __init__(self):         self.name = 'Nireekshan'      d=Demo()     print(d.name)     print(d.__dict__)</pre>
<b>Output</b>	Nireekshan {'name': 'Nireekshan'}

<b>Program Name</b>	Double underscores demo18.py
	<pre>class Demo:     def __init__(self):         self.__name = 'Nireekshan'      d=Demo()     print(d.__dict__)</pre>
<b>Output</b>	{'_Demo__name': 'Nireekshan'}

Program Name

Double underscores  
demo19.py

```
class Demo:  
    def __init__(self):  
        self.__name = 'Nireekshan'  
  
d=Demo()  
print(d.__name)
```

Error

AttributeError: 'Demo' object has no attribute '\_\_name'

Program Name

Double underscores  
demo20.py

```
class Demo:  
    def __init__(self):  
        self.__name = 'Nireekshan'  
  
d=Demo()  
print(d.__name)
```

Output

Nireekshan

## **\_\_str\_\_(self) method:**

- ✓ Whenever we are printing any object reference internally **\_\_str\_\_(self)** method will be called which returns string in the following format

```
<__main__.classname object at 0x022144B0>
```

- ✓ To return meaningful string representation we have to override **\_\_str\_\_()** method.

**Program Name** without overriding **\_\_str\_\_(self)** method  
demo21.py

```
class Demo:  
    def m(self):  
        print("m() method")  
  
d=Demo()  
print(d)  
print(d.__str__())
```

**Output**

```
<__main__.Demo object at 0x000001C813370898>  
<__main__.Demo object at 0x000001C813370898>
```

**Program Name** overriding **\_\_str\_\_(self)** method  
demo22.py

```
class Demo:  
    def m(self):  
        print("m() method")  
  
    def __str__(self):  
        return "My own str method called"
```

```
d=Demo()  
print(d)
```

**Output**

## DVS Technologies

My own str method called

**Program Name** without overriding `__str__(self)` method  
demo23.py

```
class Student:  
    def __init__(self, name, rollno):  
        self.name=name  
        self.rollno=rollno
```

```
s1=Student('Nireekshan',10)  
s2=Student('Mohan',12)  
print(s1)  
print(s2)
```

**Output**

```
<__main__.Student object at 0x000001EED2B90978>  
<__main__.Student object at 0x000001EED2B90C88>
```

**Program Name** overriding `__str__(self)` method  
demo24.py

```
class Student:  
    def __init__(self, name, rollno):  
        self.name=name  
        self.rollno=rollno  
    def __str__(self):  
        return 'This is a Student with Name: {} and Rollno: {}'.format(self.name, self.rollno)  
  
s1=Student('Nireekshan',101)  
s2=Student('Mohan',102)  
  
print(s1)  
print(s2)
```

**Output**

```
This is Student with Name: Nireekshan and Rollno: 101  
This is Student with Name: Mohan and Rollno: 102
```

## 21. Exceptional Handling

In any programming language there are 2 types of errors are possible.

1. Syntax Errors
2. Runtime Errors

### 1. Syntax Errors

- ✓ The errors which occurs because of invalid syntax are called syntax errors.

Program Name	Syntax error demo1.py
	<pre>x=123 if x==123     print("Hello")</pre>
Output	SyntaxError: invalid syntax

- ✓ In above program SyntaxError because missed out colon symbol

Program Name	Syntax error demo2.py
	<pre>print("Hello"</pre>
Output	SyntaxError: unexpected EOF while parsing

### Programmer responsible

- ✓ It's a programmer is responsible to correct these syntax errors.
- ✓ Once all syntax errors are corrected then only program execution will be started.

### 2. Runtime Errors

#### Scenarios where runtime errors will occur?

- ✓ While executing the program if something goes wrong then we will get Runtime errors, those are might be cause of,
  - End user input or
  - Programming logic or
  - Memory problems etc.

- ✓ Also known as exceptions.

Program Name	Runtime error demo3.py
	print(10/0)
Error	<b>ZeroDivisionError</b> : division by zero

Program Name	Runtime error demo4.py
	print(10/"two")
Error	<b>TypeError</b> : unsupported operand type(s) for /: 'int' and 'str'

Program Name	Runtime error demo5.py
	x=int(input("Enter Number:")) print(x)
Error	D:\Nireekshan\Programs>py demo5.py Enter Number: hiiii <b>ValueError</b> : invalid literal for int() with base 10: "

## Make a note

- ✓ Exception Handling concept applicable for Runtime Errors but not for syntax errors.

## Normal flow of the execution

- ✓ In a program if all statements are executed as per the conditions and successfully got output then that flow is called as **normal flow** of the execution.
- ✓ Below program executed successfully from starting to ending.

**Program Name** Normal flow of execution  
demo6.py

```
print("One")
print("Two")
print("Three")
print("Four")
print("Five")
```

**Output**

```
One
Two
Three
Four
Five
```

**Abnormal flow of the execution**

- ✓ While executing statements in a program, if any error occurred at runtime then immediately program flow will get terminates abnormally
- ✓ This kind termination is called as abnormal flow of the execution.



**Program Name** Program execution terminated abnormally  
demo7.py

```
print("One")
print("Two")
print(10/0)
print("Four")
print("Five")
```

**Output**

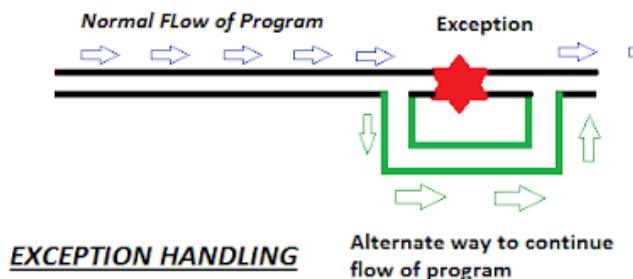
```
One
Two
Traceback (most recent call last):
  File "demo7.py", line 3, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

- ✓ Above program terminated in middle where run time error got occurred.

- ✓ As discussed, if run time error means it won't execute remaining statements from error onwards.

What we need to do if program terminates abnormally?

- ✓ We need to find an alternative way to finish the work successfully.



What is an Exception?

- ✓ An unwanted, unexpected event which disturbs the normal flow of the program is called exception.
- ✓ When an exception occurred then immediately program will terminate abnormally.
- ✓ We need to handle those exceptions on high priority for normal flow of execution.

Program Name	Execution terminated abnormally demo8.py
Output	One Two Traceback (most recent call last): File " demo8.py", line 3, in <module> print(10/0) ZeroDivisionError: division by zero

## Examples

ZeroDivisionError  
TypeError  
ValueError  
FileNotFoundException  
EOFError

## Is it really required to handle the exceptions?

- ✓ It is highly recommended to handle exceptions.
- ✓ The main objective of exception handling is for graceful termination of the program (i.e we should not block our resources and we should not miss anything)

## What is the meaning of exception handling?

- ✓ Exception handling does not mean repairing exception.
- ✓ We have to define an alternative way to continue rest of the program normally.
- ✓ Defining an alternative way is nothing but exception handling.

## Default Exception Handing in Python:

- ✓ For every exception type the corresponding classes are available.
- ✓ In python every exception is an object.
- ✓ Whenever an exception occurs Python Virtual Machine (PVM) will create the corresponding exception object and will check for handling code.
- ✓ If handling code is not available, then Python interpreter terminates the program abnormally and prints corresponding exception information to the console.
- ✓ The rest of the program won't be executed.

**Program Name** Normal flow error  
demo9.py

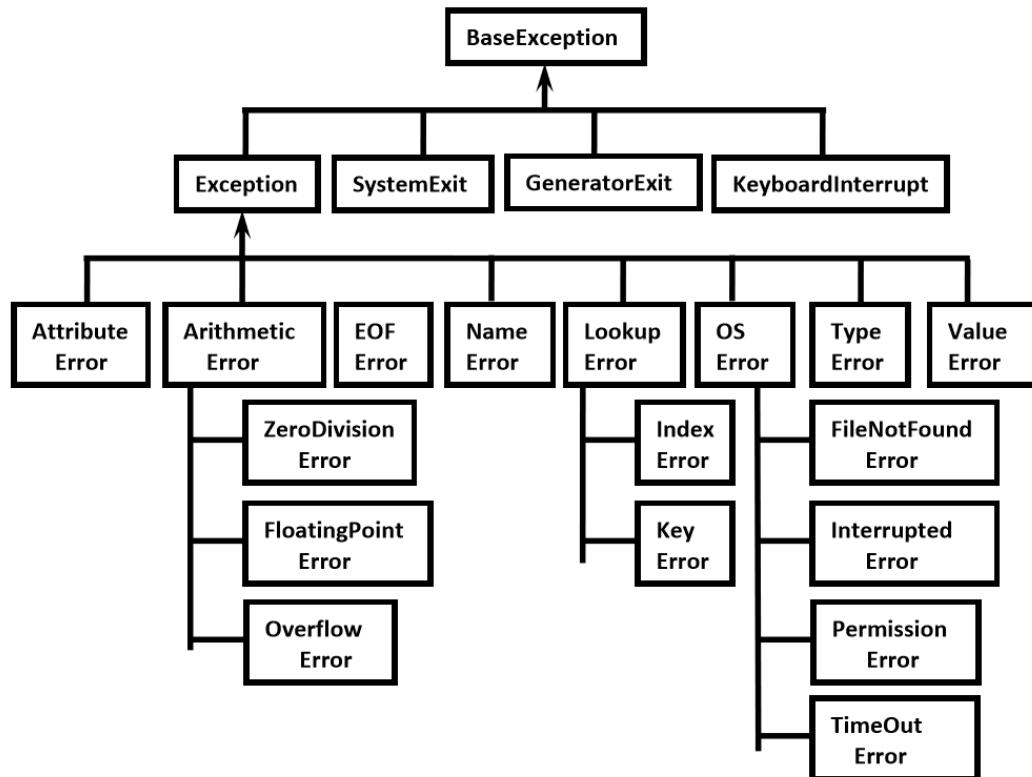
```
print("One")
print("Two")
print(10/0)
print("Four")
print("Five")
```

## Output

```
One
Two
Traceback (most recent call last):
  File "demo9.py", line 3, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

Exception hierarchy diagram

## Python's Exception Hierarchy



- ✓ Every Exception in Python is a class.
- ✓ **BaseException** class is a root for all exception classes in python exception hierarchy.
- ✓ All exception classes are child classes of BaseException

Programmer focus

- ✓ Programmer needs to focus and understand clearly the Exception and child classes.

## Handling exceptions by using try except

- ✓ We can handle exceptions by using **try** and **except**

### **try** block

- ✓ **try** is a keyword in python
- ✓ The code which may raise an exception, that code we need to write inside **try** block.

### **except** block

- ✓ **except** is a keyword in python
- ✓ The corresponding handling code for exception we need write inside **except** block.

### Make a note

- ✓ **try-except** flow:
  - If any exception raised in **try** block, then only execution flow goes to **except** block for handling code.
  - If there is no exception, then execution flow won't go to except block

### Syntax

```
try :  
    code which may raise an exception  
  
except :  
    exception handling code
```

**Program Name** Without **try** and **except**, abnormal termination  
`demo10.py`

```
print("One")  
print("Two")  
print(10/0)  
print("Four")  
print("Five")
```

**Output**  
One  
Two  
Traceback (most recent call last):  
 File "demo10.py", line 3, in <module>  
 print(10/0)  
**ZeroDivisionError**: division by zero

**Program Name** Handling exception by using try and except, normal termination  
demo11.py

```
print("One")
print("Two")

try:
    print(10/0)

except ZeroDivisionError:
    print(10/2)

print("Four")
```

**Output**

```
One
Two
5.0
Four
```

Control flow in **try** and **except**

**try except case 1**

- ✓ In our program if there is no exception, then its normal termination.
- ✓ If it is normal termination, then **except block won't execute**.

**Program Name** No exception, so except block wont execute  
demo12.py

```
print("One")
print("Two")

try :
    print("try block")

except ZeroDivisionError :
    print("except block: Handling code")

print("Three")
```

**Output**

```
One
Two
try block
```

Three

## try except case 2

- ✓ In our program if there is an exception before **try** block then its program terminates immediately its abnormal termination.

Program Name

Exception raised in before try block, abnormal termination  
demo13.py

```
print(10/0)
print("Two")
```

```
try :
    print("try block")
```

```
except ZeroDivisionError :
    print("except block: Handling code")
```

```
print("Three")
```

Output

```
ZeroDivisionError: division by zero
```

## try except case 3

- ✓ If an exception raised inside **try** block, then execution flow goes to **except** block.
- ✓ If **except** block having corresponding exception handling code then only **except** block will be execute, its normal termination

Program Name

Exception inside try block and handling code exists in except block  
demo14.py

```
print("One")
print("Two")
```

```
try:
```

```
    print("try block")
    print(10/0)
```

```
except ZeroDivisionError:
    print("except block: Handling code")
```

```
print("Four")
```

## Output

```
One  
Two  
try block  
except block: Handling code  
Four
```

## try except case 3a

- ✓ If an exception raised inside **try** block, then execution flow goes to **except** block.
- ✓ If **except** block cannot having corresponding exception handling code then **except** block won't execute, its abnormal termination

**Program Name** Exception inside try block and handling code not exists in except block  
`demo15.py`

```
print("One")  
print("Two")  
  
try:  
    print("try block")  
    print(10/0)  
  
except TypeError:  
    print("except block: Handling code")  
  
print("Four")
```

## Output

```
One  
Two  
try block  
Traceback (most recent call last):  
  File "demo15.py", line 3, in <module>  
    print(10/0)  
ZeroDivisionError: division by zero
```

## try except case 4

- ✓ If an exception raised in **try** block first statement, then corresponding **except** block will execute, its normal termination.
- ✓ After exception raised in **try** block, if **try** block contains any other statements, then those statements won't execute.
- ✓ **Conclusion:**
  - Within the try block if any exception raised then **rest of code** in **try** block won't be executed even though we handled that exception.
  - So, we should take only exception raised code inside **try** block.

### Program

After exception raised in **try** block, if **try** block contains any other statements, then those statements won't execute.

### Name

demo16.py

```
print("One")
print("Two")
```

```
try:
    print("try block")
    print(10/0)
    print("rest of the code")
```

```
except ZeroDivisionError:
    print("except block: Handling code")
```

```
print("Four")
```

### Output

```
One
Two
try block
except block: Handling code
Four
```

## try except case 5

- ✓ If there is no exception inside **try** block, then **except** block won't execute.
- ✓ If flow didn't enter into **except** block, then even **except** block won't throw exception even if it contains exception code.

**Program Name** Exception raised inside except block  
demo17.py

```
print("One")
print("Two")

try:
    print("try block")

except ZeroDivisionError:
    print(10/0)

print("Four")
```

**Output**

```
One
Two
try block
Four
```

## try except case 6

- ✓ If an exception raised inside **try** block, then corresponding **except** block also raising exception then it's abnormal termination.

**Program Name** try and except both blocks raising exceptions  
demo18.py

```
print("One")
print("Two")

try:
    print("try block")
    print(10/0)

except ZeroDivisionError:
    print(10/0)

print("Four")
```

**Output**

```
One
Two
try block
Traceback (most recent call last):
  File " demo18.py", line 5, in <module>
    print(10/0)
```

**ZeroDivisionError:** division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):

```
  File " demo18.py", line 8, in <module>
    print(10/0)
```

**ZeroDivisionError:** division by zero

## try except case 7

- ✓ If an exception raised inside **try** block, then corresponding **except** block will executed.
- ✓ After **except** block executed, if any other statements raising any exception again its abnormal termination.

<b>Program Name</b>	Abnormal termination demo19.py
	<pre>print("One") print("Two")</pre>
	<pre>try:     print(10/0)</pre>
	<pre>except ZeroDivisionError:     print("except block: Handling code")</pre>
	<pre>print(10/0)</pre>

## Output

```
One
Two
except block: Handling code
```

Traceback (most recent call last):  
 File " demo19.py", line 8, in <module>
 print(10/0)  
**ZeroDivisionError:** division by zero

**Program Name** Normal termination  
demo19a.py

```
print("One")
print("Two")

try:
    print(10/0)

except ZeroDivisionError:
    print("except block1: Handling code")

try:
    print(10/0)

except ZeroDivisionError:
    print("except block2: Handling code")
```

**Output**

```
One
Two
except block1: Handling code
except block2: Handling code
```

**Printing exception information**

- ✓ We can create reference to Exception class and print to see the information.

**Program Name** Printing exception information  
demo20.py

```
try:
    print(10/0)
except ZeroDivisionError as z:
    print("Handling code : ", z)
```

**Output**

```
Handling code: division by zero
```

**try with multiple except blocks**

- ✓ **try** with multiple **except** blocks are allowed.
- ✓ The way of handling exception is different from exception to exception.
- ✓ So, for every exception type a separate **except** block we have to write.

## Syntax

```
try:  
    -----  
    -----  
    -----  
except ZeroDivisionError:  
    Handle arithmetic operations related exceptions  
  
except ValueError:  
    Handle run time values related exceptions
```

**Program Name** try with multiple except blocks  
demo21.py

```
try:  
    x=int(input("Enter First Number: "))  
    y=int(input("Enter Second Number: "))  
    print(x/y)  
  
except ZeroDivisionError:  
    print("Can't Divide with Zero")  
  
except ValueError:  
    print("please provide int value only")
```

## Output

```
C:\Users\Nireekshan\Programs>py demo.py  
Enter First Number: 4  
Enter Second Number: 2  
2.0
```

```
C:\Users\Nireekshan\Programs>py demo.py  
Enter First Number: 4  
Enter Second Number: 0  
Can't Divide with Zero
```

```
C:\Users\Nireekshan\Programs>py demo.py  
Enter First Number: 4  
Enter Second Number: hello  
please provide int value only
```

## Make a note

- ✓ If **try** with multiple **except** blocks available, if **try** block raised any exception then starts searching from top to bottom for matched **except** block

**Program Name** multiple catch block  
demo22.py

```
try:  
    x=int(input("Enter First Number: "))  
    y=int(input("Enter Second Number: "))  
    print(x/y)  
  
except ArithmeticError:  
    print("ArithmaticError")  
  
except ZeroDivisionError:  
    print("ZeroDivisionError")
```

**Output**

```
C:\Users\Nireekshan\Desktop>py demo22.py  
Enter First Number: 34  
Enter Second Number: 0  
ArithmaticError
```

**Program Name** multiple catch block  
demo22a.py

```
try:  
    x=int(input("Enter First Number: "))  
    y=int(input("Enter Second Number: "))  
    print(x/y)  
  
except ZeroDivisionError:  
    print("ZeroDivisionError")  
  
except ArithmeticError:  
    print("ArithmaticError")
```

**Output**

```
C:\Users\Nireekshan\Desktop>py demo22.py  
Enter First Number: 34  
Enter Second Number: 0  
ZeroDivisionError
```

Single except block can handle multiple exceptions:

- ✓ We can write a single except block that can handle multiple different types of exceptions.

## Syntax 1

```
except (Exception1, Exception2, exception3,..):
```

## Syntax 2

```
except (Exception1, Exception2, exception3,..) as msg:
```

- ✓ Parenthesis are mandatory, and this group of exceptions internally considered as tuple.

**Program Name** single except block is handling multiple exceptions  
demo23.py

```
try:  
    x=int(input("Enter First Number: "))  
    y=int(input("Enter Second Number: "))  
    print(x/y)  
  
except (ZeroDivisionError, ValueError) as z:  
    print("Please Provide valid numbers only and problem is: ", z)
```

## Output

```
C:\Users\Nireekshan\Desktop>py demo23.py  
Enter First Number: 4  
Enter Second Number: 0  
Please Provide valid numbers only and problem is: division by zero
```

```
C:\Users\Nireekshan\Desktop>py demo23.py  
Enter First Number: 4  
Enter Second Number: hello  
Please Provide valid numbers only and problem is: invalid literal for  
int() with base 10: 'hello'
```

## Default except block

- ✓ We can use default `except` block to handle any type of exceptions.
- ✓ In default `except` block generally we can print normal error messages.

Syntax:

```
except:  
    statements
```

Program  
Name

```
default except block  
demo24.py
```

```
try:  
    x=int(input("Enter First Number: "))  
    y=int(input("Enter Second Number: "))  
    print(x/y)  
  
except ZeroDivisionError:  
    print("ZeroDivisionError: Can't divide with zero")  
  
except:  
    print("Default Except: Please provide valid input only")
```

Output

```
C:\Users\Nireekshan\Desktop>py demo24.py  
Enter First Number: 4  
Enter Second Number: 0  
ZeroDivisionError: Can't divide with zero
```

```
C:\Users\Nireekshan\Desktop>py demo24.py  
Enter First Number: 4  
Enter Second Number: hello  
Default Except: Please provide valid input only
```

## Make a note

- ✓ If try with multiple `except` blocks available then default `except` block should be last, otherwise we will get `SyntaxError`.

Program  
Name

```
default except block  
demo25.py
```

```
try:  
    print(10/0)  
  
except ZeroDivisionError:  
    print("ZeroDivisionError")
```

```
except:  
    print("Default Except")
```

Output

```
ZeroDivisionError
```

Program Name      default except block  
demo25a.py

```
try:  
    print(10/0)  
  
except:  
    print("Default Except")  
  
except ZeroDivisionError:  
    print("ZeroDivisionError")
```

Output

```
SyntaxError: default 'except:' must be last
```

Make a note

The following are various possible combinations of except blocks

1. `except ZeroDivisionError:`
2. `except ZeroDivisionError as msg:`
3. `except (ZeroDivisionError, ValueError) :`
4. `except (ZeroDivisionError, ValueError) as msg:`
5. `except :`

## Make a note

- ✓ In any project after using all resource its good practice to do clean-up activities.
- ✓ **Example:** If I've open data base connection after used that connection then I should close that connection.
- ✓ So, a separate place is required to do all clean-up activities.
- ✓ These kinds of activities will be done inside **finally** block in python

## finally block

- ✓ **finally**, is a keyword in python.
- ✓ We will use finally block to do clean-up activities.

## Why separate block for clean-up activities, can't we write inside try and except blocks?

- ✓ We cannot write clean-up activities inside try and except block because...

## Coming to the try block

- ✓ There is no guarantee like every statement will be execute inside **try** block.
- ✓ **Example:** After exception raised in **try** block, if **try** block having any rest of the code after exception code then it won't get execute those statements.
- ✓ It is not recommended to write clean up code inside **try** block.

<b>Program Name</b>	try and except blocks demotryexcept.py
	<pre>try:     print(10/0)     print("rest of the code") except:     print("except block: handling code")</pre>
<b>Output</b>	except block: handling code

## Coming to the except block

- ✓ There is no guarantee like **except** block will be execute
- ✓ **Example:** If there is no exception then **except** block won't be executed.
- ✓ It is not recommended to write clean up code inside **except** block.

## Conclusion

- ✓ So, a separate block is required to write clean-up activities.
- ✓ That code should execute always.

- If no exception, then clean-up code **should execute**.
  - If exception raised and handled that exception, then clean-up code **should execute**.
  - If exception raised but not handled that exception, then also clean-up code **should execute**.
- ✓ Hence the main purpose of finally block is to maintain clean up code.

What is the speciality of final block?

- ✓ The speciality of finally block is it will be executed always, irrespective of exception raised or not, exception handled or not

## Syntax

```
try:  
    Risky Code  
except:  
    Handling Code  
finally:  
    Clean-up code
```

## finally case-1

- ✓ If there is no exception, then finally block will get execute.

**Program Name**      finally block always executes  
                          demo26.py

```
try:  
    print("try block")  
  
except:  
    print("except block")  
  
finally:  
    print("finally block")
```

**Output**  
try block  
finally block

## finally case-2:

- ✓ If an exception raised and handled with except block then also finally block will get execute

**Program Name** finally block always executes  
demo27.py

```
try:  
    print("try block")  
    print(10/0)  
  
except ZeroDivisionError:  
    print("except block")  
  
finally:  
    print("finally block")
```

**Output**

```
try block  
except block  
finally block
```

## finally case-3:

- ✓ If an exception raised inside try block but except block not handled, then also finally block will get execute
- ✓ Means an exception raised but no corresponding handling code in except block, then also finally block will get execute.

**Conclusion:**

- If an exception raised but not handled, then program supposed to terminate abnormally, but before terminating the program **finally** block will get execute

**Program Name** finally block always executes  
demo28.py

```
try:  
    print("try")  
    print(10/0)  
  
except NameError:  
    print("except block")
```

```
finally:  
    print("finally block")
```

## Output

```
try block  
finally block  
Traceback (most recent call last):  
  File "demo28.py", line 4, in <module>  
    print(10/0)  
ZeroDivisionError: division by zero
```

Any situation like, finally will not execute?

- ✓ `_exit(0)` is a predefined function available in `os` module.
- ✓ If we call `os._exit(0)` then Python Virtual Machine itself will be shutdown.
- ✓ In this case finally block won't execute.
- ✓ So, only in this scenario finally block will be not executed.

**Program Name** Only one situation which finally block won't get execute  
`demo29.py`

```
import os  
  
try:  
    print("try block")  
    os._exit(0)  
  
except NameError:  
    print("except")  
  
finally:  
    print("finally")
```

## Output

```
try block
```

## Control flow in try-except-finally

### Case 1: try-except-finally

- ✓ If there is no exception, then **try**, **finally** blocks will execute and **except** block won't execute, its normal termination.

<b>Program Name</b>	if no exception then try and finally executes but not except block demo30.py
	print("One") print("Two")
	<b>try:</b> print("try block")
<b>Output</b>	<b>except ZeroDivisionError:</b> print("except block: Handling code")
	<b>finally:</b> print("finally block: clean-up activities")
	print("Four")
	One Two try block finally block: clean-up activities Four

### Case 2: try-except-finally

- ✓ If an exception raised inside **try** block and corresponding **except** block is handling the exception then **try**, **except**, **finally** blocks will execute, its normal termination

<b>Program Name</b>	if an exception raised then try, except and finally blocks executes demo31.py
	print("One") print("Two")
	<b>try:</b> print("try block") print(10/0) print("rest of the code after exception raised in try")
	<b>except ZeroDivisionError:</b>

```
print("except block: Handling code")

finally :
    print("finally block: clean-up activities")

print("Four")
```

## Output

```
One
Two
try block
except block: Handling code
finally block: clean-up activities
Four
```

## Case 3: try-except-finally

- ✓ If an exception raised inside **try** block and corresponding **except** block matched then **try**, **except**, **finally** blocks will execute.
- ✓ After exception raised in **try** block then rest of the code (inside try block) won't execute.
- ✓ Its normal termination.

## Program

After exception raised in try block then rest of the code (inside try block) won't execute.

## Name

demo32.py

```
print("One")
print("Two")

try:
    print("try block ")
    print(10/0)
    print("rest of the code after exception raised in try")

except ZeroDivisionError:
    print("except block: Handling code")

finally :
    print("finally block: clean-up activities")

print("Four")
```

## Output

```
One
Two
try block
```

except block: Handling code  
finally block: clean-up activities  
Four

## Case 4: try-except-finally

- ✓ If an exception raised inside **try** block and corresponding **except** block not matched then **try**, **finally** blocks will execute, It's an abnormal termination.

**Program Name** If exception raised and corresponding except block not matched then demo33.py

```
print("One")
print("Two")

try:
    print("try block ")
    print(10/0)

except NameError:
    print("except block: Handling code")

finally :
    print("finally block: clean-up activities")

print("Four")
```

## Output

```
One
Two
try block
finally: clean-up activities
Traceback (most recent call last):
  File "demo33.py", line 6, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

## Case 5: try-except-finally

- ✓ If an exception raised inside **try** block, then flow goes to **except** block.
- ✓ So here **except** block should handle the exception.
- ✓ Instead of handling exception, if **except** block also raising any exception then program will terminate abnormally.
- ✓ As we know before terminating the program finally block will get execute.

**Program Name** raised exception inside except block  
demo34.py

```
print("One")
print("Two")

try:
    print("try block")
    print(10/0)

except ZeroDivisionError:
    print(10/0)

finally :
    print("finally block: clean-up activities")

print("Four")
```

## Output

```
One
Two
try block
finally block: clean-up activities
```

```
Traceback (most recent call last):
  File "demo34.py", line 6, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "demo34.py", line 9, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

## Case 6: try-except-finally

- ✓ If an exception raised inside **finally** block, then it's always abnormal termination.

**Program Name** raised exception inside finally block, abnormal termination  
demo35.py

```
print("One")
print("Two")
```

```
try:
```

```
print("try block")

except NameError:
    print("except block: Handling code")

finally:
    print(10/0)

print("Four")
```

## Output

```
One
Two
try block
Traceback (most recent call last):
  File "demo35.py", line 6, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

## Nested try-except-finally blocks

- ✓ In python nested **try except finally** blocks are allowed.
- ✓ We can take try-except-finally blocks inside **try** block
- ✓ We can take try-except-finally blocks inside **except** block
- ✓ We can take try-except-finally blocks inside **finally** block

## Syntax

```
try:
    try outer statements

try:
    inner try statements

except:
    inner except statements

except:
    except outer statements

finally:
    finally outer statements
```

## Case 1

- ✓ If no exception raised then outer **try**, inner **try**, inner **finally**, outer **finally** blocks will get executes

## Program

If no exception raised then outer try, inner try, inner finally, outer finally blocks will get executes

## Name

demo36.py

```
try:  
    print("outer try block")  
  
    try:  
        print("Inner try block")  
  
    except ZeroDivisionError:  
        print("Inner except block")  
  
    finally:  
        print("Inner finally block")  
  
except:  
    print("outer except block")  
  
finally:  
    print("outer finally block")
```

## Output

outer try block  
Inner try block  
Inner finally block  
outer finally block

## Case 2

- ✓ If an exception raised in outer **try**, then outer **except** blocks is responsible to handle that exception.

## Program

If an exception raised inside outer try, then outer except blocks is responsible to handle that exception.

## Name

demo37.py

```
try:  
    print("outer try block")  
    print(10/0)  
  
    try:  
        print("Inner try block")
```

```
except ZeroDivisionError:  
    print("Inner except block")  
  
finally:  
    print("Inner finally block")  
  
except:  
    print("outer except block")  
  
finally:  
    print("outer finally block")
```

## Output

```
outer try block  
outer except block  
outer finally block
```

## Case 3

- ✓ If an exception raised in inner **try** block then inner **except** block is responsible to handle, if it is unable to handle then outer except block is responsible to handle.

## Program

If an exception raised in inner try block then inner except block is responsible to handle, if it is unable to handle then outer except block is responsible to handle.

## Name

demo38.py

```
try:  
    print("outer try block")  
  
    try:  
        print("Inner try block")  
        print(10/0)  
  
    except ZeroDivisionError:  
        print("Inner except block")  
  
    finally:  
        print("Inner finally block")  
  
except:  
    print("outer except block")  
  
finally:  
    print("outer finally block")
```

## Output

```
outer try block  
Inner try block  
Inner except block  
Inner finally block  
outer finally block
```

## Case 4

- ✓ If an exception raised in inner **try** block then inner **except** block is responsible to handle, if it is unable to handle then outer except block is responsible to handle.

### Program

If an exception raised in inner try block then inner except block is responsible to handle, if it is unable to handle outer except block is responsible to handle.

### Name

demo39.py

```
try:  
    print("outer try block")  
  
    try:  
        print("Inner try block")  
        print(10/0)  
  
    except NameError:  
        print("Inner except block")  
  
    finally:  
        print("Inner finally block")  
  
except:  
    print("outer except block")  
  
finally:  
    print("outer finally block")
```

### Output

```
outer try block  
Inner try block  
Inner finally block  
outer except block  
outer finally block
```

## else block with try-except-finally:

- ✓ We can use **else** block with try-except-finally blocks.
- ✓ **else** block will be executed if and only if there are no exceptions inside **try** block.

At what time which block?

```
try:          Exception raised code
except:       will be executed if exception raised inside try
else:         will be executed if there is no exception inside try
finally:      will be executed always.
```

## Case 1

- ✓ If no exception then **try**, **else** and **finally** blocks will get executed.

### Program

If no exception then try, else and finally blocks will get executed.

### Name

demo40.py

```
try:          print("try block")
except:       print("except: Handling code")
else:         print("else block")
finally:      print("finally block")
```

### Output

```
try block
else block
finally block
```

## Case 2

- ✓ If an exception raised inside **try** block, then **except** block will get executed but **else** block won't executed.

### Program

If an exception raised inside try block, then except block will get executed but else block won't executed.

### Name

demo41.py

```
try:  
    print("try block")  
    print(10/0)  
  
except:  
    print("except: Handling code")  
  
else:  
    print("else block")  
  
finally:  
    print("finally block")
```

### Output

```
try block  
except: Handling code  
finally block
```

Various possible combinations of **try-except-else-finally**:

## Case 1:

- ✓ Only **try** block is invalid

### Program Name

Error: only try is invalid  
demo42.py

```
try:  
    print("try block")
```

### Output

SyntaxError: unexpected EOF while parsing

## Case 2:

- ✓ Only **except** block is invalid

<b>Program Name</b>	<b>Error:</b> only except is invalid demo43.py
	<b>except</b> NameError: print("except: Handling code")
<b>Output</b>	<b>SyntaxError:</b> invalid syntax

## Case 3:

- ✓ Only **finally** block is invalid

<b>Program Name</b>	<b>Error:</b> only finally is invalid demo44.py
	<b>finally:</b> print("finally block")
<b>Output</b>	<b>SyntaxError:</b> invalid syntax

## Case 4:

- ✓ **try** with **except** combination is valid.
- ✓ **try** block follows **except** block

<b>Program Name</b>	try block follows except block demo45.py
	<b>try:</b> print("try block") <b>except:</b> print("except block")
<b>Output</b>	try block

## Case 4:

- ✓ `try` with `finally` combination is valid.
- ✓ `try` block follows `finally` block

**Program Name** try block follows finally block  
demo46.py

```
try:  
    print("try block")  
finally:  
    print("finally block")
```

**Output**

```
try block  
finally block
```

## Case 4:

- ✓ `try, except` and `else` combination is valid.
- ✓ `try, except` blocks follows `else` block

**Program Name** try block follows finally block  
demo47.py

```
try:  
    print("try block")  
except:  
    print("except block")  
else:  
    print("else block")
```

**Output**

```
try block  
else block
```

## Types of Exceptions

- ✓ In Python there are 2 types of exceptions are possible.
  1. Predefined Exceptions
  2. User Defined Exceptions

## 1. Predefined Exceptions:

- ✓ If any specific error occurs, then for those errors Python Virtual Machine will handle by using predefined exceptions.
- ✓ Predefined exceptions are called as In-built exceptions.

### Example 1

- ✓ Whenever we are trying to perform Division by zero, automatically Python will raise ZeroDivisionError.

Program Name	ZeroDivisionError demo48.py
Output	print(10/0)  ZeroDivisionError

### Example 2

- ✓ Whenever we are trying to convert input value to int type and if input value is not int value then Python will raise ValueError automatically

Program Name	ZeroDivisionError demo49.py
Output	x=int("ten") print(x)  ValueError: invalid literal for int() with base 10: 'ten'

## 2. User Defined Exceptions:

### User defined exceptions or Custom Exceptions

- ✓ A programmer can create own exceptions to raise explicitly.
- ✓ Sometimes based on project requirement, programmer needs to create his own exceptions and raise explicitly for corresponding scenarios.
- ✓ These are called as customized Exceptions or Programmatic Exceptions

## How to raise customised exception?

- ✓ By using `raise` keyword

### Example

- ✓ `InSufficientFundsException`
- ✓ `InvalidInputException`

## Steps to follow to Define and Raise Customized Exceptions:

### 1. Customised exception class should be sub class of predefined Exception class

- ✓ Since all exceptions are classes, the programmer is supposed to create his own exception as a class.
- ✓ Also, he should make his class as a sub class to the in-built '`Exception`' class

#### Syntax

```
class MyException(Exception):
    def __init__(self, arg):
        self.arg = arg
```

- ✓ `MyException` is the sub class for `Exception`, this class has a constructor which having one argument.

### 2. raise the exception where are all required

- ✓ When the programmer suspects the possibility of exception, he should raise his own exception using `raise` as:

#### Syntax

```
raise MyException('message')
```

We can raise the exception info simply as well

#### Program

Program to add two numbers if both are an integer type, if not integer please throw customized error

#### Name

`demo50.py`

```
def add_numbers(a, b):
```

```
if not (isinstance(a, int) and isinstance(b, int)):
    raise TypeError("Dude cool, please pass integers.")
return a + b

print(add_numbers(10, 30))
print(add_numbers(10, "chiru"))
```

## Output

```
40
Traceback (most recent call last):
  File "demo50.py", line 7, in <module>
    print(add_numbers(10, "chiru"))
  File "demo50.py", line 3, in add_numbers
    raise TypeError("Dude cool, please pass integers.")
TypeError: Dude cool, please pass integers.
```

## Program Name

```
raising exception by using raise
demo51.py

class SymbolicError(Exception):
    def __init__(self, data):
        self.data = data

    def __str__():
        return repr(self.data)

try:
    if not (isinstance("Anushka", int)):
        raise SymbolicError(2000)

except SymbolicError:
    print("Dude: please provide integer values only")
```

## Output

```
Dude: please provide integer values only
```

## Program Name

```
Customised exception
demo53.py

class TooYoungException(Exception):
    def __init__(self, arg):
        self.msg=arg

class TooOldException(Exception):
    def __init__(self, arg):
        self.msg=arg
```

```
age=int(input("Enter Age:"))

if age>60:
    raise TooYoungException("Plz wait some more time you will get
best match soon!!!")

elif age<18:
    raise TooOldException("Your age already crossed marriage
age...no chance of getting marriage")

else:
    print("You will get match details soon by email!!!")
```

## Output

```
C:\Users\Nireekshana\Desktop>py demo53.py
Enter Age:10
__main__.TooOldException: Your age already crossed marriage age...no
chance of getting marriage

C:\Users\Nireekshana\Desktop>py demo53.py
Enter Age:30
You will get match details soon by email!!!

C:\Users\Nireekshana\Desktop>py demo53.py
Enter Age:90
__main__.TooYoungException: Plz wait some more time you will get
best match soon!!!
```

## 22. File io in Python

### What is a file?

- ✓ A file contains data which stores permanently in computer devices.
- ✓ As the part of programming requirement, we have to store our data permanently for future purpose.
- ✓ For this requirement we should go for files.
- ✓ Files are very common permanent storage areas to store our data.

### Types of Files:

- ✓ There are 2 types of files

#### Text files

- ✓ Normally, text files are used to store characters or strings.
- ✓ Text files store the data in the form of characters

#### Binary Files

- ✓ Binary files store entire data in the form of bytes, i.e a group of 8 bits each
- ✓ When the data is retrieved from the binary file, the programmer can retrieve the data as bytes.
- ✓ Binary files can be used to store text, images, audio and video.

### file modes and their meanings

File open mode	Description
w	<ul style="list-style-type: none"><li>✓ Open an existing file for write operation.</li><li>✓ If the file already contains some data, then it will be overridden.</li><li>✓ If the specified file is not already available, then this mode will create that file.</li></ul>
r	<ul style="list-style-type: none"><li>✓ Open an existing file for read operation.</li><li>✓ The file pointer is positioned at the beginning of the file.</li><li>✓ If the specified file does not exist, then we will get FileNotFoundError.</li><li>✓ This is default mode.</li></ul>
a	<ul style="list-style-type: none"><li>✓ Open an existing file for append operation.</li><li>✓ It won't override existing data.</li><li>✓ If the specified file is not already available, then this mode will create a new file.</li></ul>

w++	<ul style="list-style-type: none"> <li>✓ To write and read data of a file.</li> <li>✓ The previous data will be deleted from file</li> </ul>
r++	<ul style="list-style-type: none"> <li>✓ To read and write the file.</li> <li>✓ Previous data will be deleted</li> <li>✓ Pointer used to stand at first line</li> </ul>
a++	<ul style="list-style-type: none"> <li>✓ To append and read data of a file</li> <li>✓ File pointer is end of the file if the file exists, it creates new file for reading and writing</li> </ul>
x	<ul style="list-style-type: none"> <li>✓ To open the file in exclusive creation mode</li> <li>✓ The file creation fails if the file already exists</li> </ul>

### Make a note

- ✓ All the above modes are applicable for text files. If the above modes suffixed with 'b' then these represents for binary files.

### Example

rb, wb, ab, r+b, w+b, a+b, xb

### Opening a File:

- ✓ Before performing any operation (like read or write) on the file, first we must open that file.
- ✓ `open()` is a predefined function in python.
- ✓ By using `open()` function we can open the file.
- ✓ But at the time of open, we must specify mode, which represents the purpose of opening file.

### Syntax

```
f = open(filename, mode)
```

### Closing a File:

- ✓ After completing our operations on the file, it is highly recommended to close the file. For this we have to use `close()` function.

### Syntax

```
f.close()
```

## Various properties of File Object

- Once we opened a file and we got file object we can get various details related to that file by using its properties.

✓ name	:	Name of opened file mode
✓ mode	:	Mode in which the file is opened
✓ closed	:	Returns boolean value that file is closed or not
✓ readable()	:	Returns boolean value that whether file is readable or not
✓ writable()	:	Returns boolean value that whether file is writable or not.

**Program Name** opening a file and closing a file  
demo1.py

```
f=open("abc.txt", 'w')

print("File Name: ", f.name)
print("File Mode: ", f.mode)

print("Is File Readable: ", f.readable())
print("Is File Writable: ", f.writable())
print("Is File Closed : ", f.closed)
f.close()
print("Is File Closed : ", f.closed)
```

**Output**

```
File Name: abc.txt
File Mode: w
Is File Readable: False
Is File Writable: True
Is File Closed: False
Is File Closed: True
```

abc.txt

An empty file created with the name of abc.txt file  
This file created in current directory

## Writing data to text files

- ✓ We can write character data to the text files by using the following 2 methods.

1. `write(str)`
2. `writelines(list of lines)`

**Program Name** opening a file and writing data into that file  
`demo2.py`

```
f=open("wish.txt", 'w')
f.write("Welcome\n")
f.write("to\n")
f.write("python world\n")

print("Data written to the file successfully")
f.close()
```

### Output

Data written to the file successfully

`wish.txt`

```
Welcome
to
python world
```

## Instead of overriding the content, how to append the content to the file

- ✓ In the above program, data present in the file will be overridden every time if we run the program.
- ✓ Instead of overriding if we want append operation then we should open the file as follows.

### Syntax

```
f = open("wish.txt", "a")
```

**Program Name** Instead of overriding the content, appending the text to file  
demo3.py

```
f=open("wish1.txt", 'a')

f.write("Welcome\n")
f.write("to\n")
f.write("python world\n")

print("Data written to the file successfully")

f.close()
```

## Output

Data written to the file successfully

### wish1.txt

```
Welcome
to
python world

Welcome
to
python world

Welcome
to
python world
```

## Make a note

- ✓ While writing data by using write() methods, compulsory we have to provide line separator(\n),otherwise total data should be written to a single line.

**Program Name** Instead of overriding the content, appending the text to file  
demo4.py

```
f=open("wish1.txt", 'a')

f.write("Welcome ")
f.write("to ")
f.write("python world")
```

```
print("Data written to the file successfully")  
f.close()
```

## Output

Data written to the file successfully

## wish1.txt

Welcome to python world

## writelines(argument)

- ✓ We can write lines of text into the file by using writelines method

<b>Program Name</b>	writing text into file by using writelines method demo5.py
	f=open("names.txt", 'w')
	list=["Ramesh\n", "Arjun\n", "Senthil\n", "Vignesh"]
	f.writelines(list)
	print("List of lines written to the file successfully")
	f.close()

## Output

List of lines written to the file successfully

## names.txt

Ramesh  
Arjun  
Senthil  
Vignesh

## Reading Character Data from text files:

- ✓ We can read character data from text file by using the following read methods.

- ✓ `read()` : To read total data from the file
- ✓ `read(n)` : To read 'n' characters from the file
- ✓ `readline()` : To read only one line
- ✓ `readlines()` : To read all lines into a list

**Program Name**      Reading data from file  
demo6.py

```
f=open("input.txt", 'r')  
data=f.read()  
print(data)  
f.close()
```

**Output**

Hi good morning

**input.txt**      Hi good morning

**Program Name**      Reading only first 10 characters of file  
demo7.py

```
f=open("log.txt", 'r')  
data=f.read(10)  
print(data)  
f.close()
```

**Output**

Hi good

**log.txt**      Hi good morning

## readline() method

- ✓ By using readline() we can read line by line from file

**Program Name** Reading data line by line

demo8.py

```
f=open("info.txt", 'r')

line1=f.readline()
print(line1, end="")

line2=f.readline()
print(line2, end="")

line3=f.readline()
print(line3, end="")

f.close()
```

## Output

```
Hello everyone
This topic is important
Please don't sleep
```

## info.txt

```
Hello everyone
This topic is important
Please don't sleep
Once this topic done
Then happily you can sleep
Thanks for cooperating
```

## readlines() method

- ✓ By using readlines() method we can read all lines

**Program Name** Reading all lines into list

demo9.py

```
f=open("employees.txt", 'r')
lines=f.readlines()
for line in lines:
    print(line, end="")
f.close()
```

## Output

```
Ramesh  
Senthil  
Chaithanya  
Nireekshan
```

## log.txt

```
Ramesh  
Senthil  
Chaithanya  
Nireekshan
```

## with keyword

- ✓ **with** is a keyword in python.
- ✓ The **with** keyword can be used while opening a file.
- ✓ We can use this to group file operation statements within a block.

## Advantage

- ✓ Generally, when we open a file, we need to close the file.
- ✓ The advantage of **with** keyword is it will take care closing of file.
- ✓ After completing all operations automatically even in the case of exceptions also, and we are not required to close explicitly.

## Program Name

```
with keyword  
demo10.py  
  
with open("abc.txt", "w") as f:  
    f.write("Welcome\n")  
    f.write("to\n")  
    f.write("Python\n")  
    print("Is File Closed: ", f.closed)  
print("Is File Closed: ", f.closed)
```

## Output

```
Is File Closed: False  
Is File Closed: True
```

## The seek() and tell() methods:

### tell()

- ✓ We can use **tell()** method to return current position of the cursor(file pointer) from beginning of the file.[ can you please tell current cursor position]
- ✓ The position(index) of first character in files is zero just like string index.

<b>Program Name</b>	<b>tell() method</b>
	<b>demo11.py</b>
	<pre>f=open("input.txt", "r")  print(f.tell()) print(f.read(2)) print(f.tell()) print(f.read(3)) print(f.tell())</pre>
<b>Output</b>	<pre>0 We 2 lco 5</pre>
<b>input.txt</b>	<pre>Welcome to Python</pre>

### seek() method

- ✓ We can use **seek()** method to move cursor(file pointer) to specified location.  
[Can you please seek the cursor to a specific location]

<b>Syntax</b>	<b>f.seek(offset, fromwhere)</b>
---------------	----------------------------------

- ✓ offset represents the number of positions

The allowed values for second attribute (from where) are

- 0---->From beginning of file (default value)
- 1---->From current position

2--->From end of the file

**Program Name** seek() method  
demo12.py

```
data="SampurneshBabu movie is excellent"
f=open("abc.txt", "w")
f.write(data)
with open("abc.txt", "r+") as f:
    text=f.read()
    print(text)
    print("The Current Cursor Position: ",f.tell())
    f.seek(24)
    print("The Current Cursor Position: ",f.tell())
    f.write("Britania Bisket")
    f.seek(0)
    text=f.read()
    print("Data After Modification:")
    print(text)
```

**Output**

```
SampurneshBabu movie is excellent
The Current Cursor Position: 33
The Current Cursor Position: 24
Data After Modification:
SampurneshBabu movie is Britania Bisket
```

**How to check a specific file exists or not?**

- ✓ We can use **os** library to get information about files in our computer.
- ✓ **os** module has path sub module, which contains **isFile()** function to check whether a particular file exists or not

**Syntax**

```
os.path.isfile(fname)
```

**Program Name** Checking file exists or not  
demo13.py

```
import os, sys
fname=input("Enter File Name: ")

if os.path.isfile(fname):
    print("File exists:", fname)
    f=open(fname, "r")
```

```
else:  
    print("File does not exist:", fname)  
    sys.exit(0)  
  
print("The content of file is:")  
data=f.read()  
print(data)
```

## Output

```
Enter File Name: abc.txt  
File exists: abc.txt  
The content of file is:  
SampurneshBabu movie is Britania Bisket
```

## sys.exit(0)

- ✓ sys.exit(0): To exit system without executing rest of the program.
- ✓ argument represents status code
- ✓ 0 means normal termination and it is the default value.

## Q. Program to print the number of lines, words and characters present in the given file?

Program Name	print the number of lines, words and characters present in the given file demo14.py
--------------	--

```
import os, sys  
fname=input("Enter File Name: ")  
  
if os.path.isfile(fname):  
    print("File exists:", fname)  
    f=open(fname, "r")  
  
else:  
    print("File does not exist:", fname)  
    sys.exit(0)  
  
lcount=wcount=ccount=0  
  
for line in f:  
    lcount=lcount+1  
    ccount=ccount+len(line)  
    words=line.split()  
    wcount=wcount+len(words)  
  
print("The number of Lines:", lcount)  
print("The number of Words:", wcount)  
print("The number of Characters:", ccount)
```

## Output

```
Enter File Name: abc.txt
File exists: abc.txt
The number of Lines: 1
The number of Words: 5
The number of Characters: 39
```

## Working with Binary data:

- ✓ It is very common requirement to read or write binary data like images, video files, audio files etc.

<b>Program Name</b>	Read image file and write to a new image file demo15.py
	<pre>f1=open("rossum.jpg", "rb") f2=open("newpic.jpg", "wb") bytes=f1.read() f2.write(bytes) print("New Image is available with the name: newpic.jpg")</pre>

## Output

```
New Image is available with the name: newpic.jpg
```

## Working with csv files

- ✓ csv -> Comma separated values
- ✓ As the part of programming, it is very common requirement to write and read data wrt csv files. Python provides csv module to handle csv files.

<b>Program Name</b>	Working with csv files. demo16.py
	<pre>import csv with open("emp.csv", "w", newline="") as f:     w=csv.writer(f)     w.writerow(["EMP NO","EMP NAME","EMP SAL","EMP ADDR"])     n=int(input("Enter Number of Employees:"))      for i in range(n):         eno=input("Enter Employee No:")         ename=input("Enter Employee Name:")         esal=input("Enter Employee Salary:")         eaddr=input("Enter Employee Address:")         w.writerow([eno, ename, esal, eaddr])  print("Total Employees data written to csv file successfully")</pre>

## Output

```
Enter Number of Employees:2
Enter Employee No:1
Enter Employee Name: senthil
Enter Employee Salary:100
Enter Employee Address: Banglore
Enter Employee No:2
Enter Employee Name: mohan
Enter Employee Salary:200
Enter Employee Address: vijayawada
Total Employees data written to csv file successfully
```

## emp.csv

EMP NO	EMP NAME	EMP SAL	EMP ADDR
1	senthil	100	Banglore
2	mohan	200	vijayawada

## Make a note

- ✓ Observe the difference with newline attribute and without newline attribute

```
with open("emp.csv", "w", newline="") as f:
with open("emp.csv", "w") as f:
```

## Make a note

- ✓ If we are not using newline attribute, then in the csv file blank lines will be included between data.
- ✓ To prevent these blank lines, newline attribute is required in Python-3.
- ✓ In Python-2 just we can specify mode as 'wb' and we are not required to use newline attribute.

**Program Name** demo17.py

```
import csv
f=open("emp.csv", 'r')
r=csv.reader(f)
data=list(r)
#print(data)

for line in data:
    for word in line:
        print(word, "\t", end="")
```

```
print()
```

## Output

EMP NO	EMP NAME	EMP SAL	EMP ADDR
1	senthil	100	Banglore
2	mohan	200	vijayawada

## Zipping and Unzipping Files:

- ✓ It is very common requirement to zip and unzip files.

The main advantages are:

1. To improve memory utilization
2. We can reduce transport time
3. We can improve performance.

- ✓ To perform zip and unzip operations, Python contains one in-built module `zipfile`. This module contains a class: `ZipFile`

## To create Zip file:

- ✓ We have to create object to `ZipFile` class with name of the zip file, mode and constant `ZIP_DEFLATED`.
- ✓ This constant represents we are creating zip file.

```
f = ZipFile("files.zip", "w", "ZIP_DEFLATED")
```

- ✓ Once we create `ZipFile` object, we can add files by using `write()` method.

```
f.write(filename)
```

Program Name      creating zip file.  
                      demo18.py

```
from zipfile import *
f=ZipFile("files.zip", 'w', ZIP_DEFLATED)
f.write("file1.txt")
f.write("file2.txt")
f.write("file3.txt")
f.close()
```

```
print("files.zip file created successfully")
```

## Output

```
files.zip file created successfully
```

## To perform unzip operation:

- ✓ We have to create object to ZipFile class as follows

```
f = ZipFile("files.zip", "r", ZIP_STORED)
```

- ✓ ZIP\_STORED represents unzip operation.
- ✓ This is default value and hence we are not required to specify.
- ✓ Once we created ZipFile object for unzip operation, we can get all file names present in that zip file by using namelist() method.

```
names = f.namelist()
```

**Program Name**      unzipping zip file.  
                          demo19.py

```
from zipfile import *
f=ZipFile("files.zip", 'r', ZIP_STORED)
names=f.namelist()

for name in names:
    print( "File Name: ",name)
    print("The Content of this file is:")
    f1=open(name, 'r')
    print(f1.read())
    print()
```

## Output

```
File Name: file1.txt
The Content of this file is:

File Name: file2.txt
The Content of this file is:

File Name: file3.txt
The Content of this file is:
```

## Working with Directories:

- ✓ It is very common requirement to perform operations for directories like
  1. To know current working directory
  2. To create a new directory
  3. To remove an existing directory
  4. To rename a directory
  5. To list contents of the directory etc...
- ✓ To perform these operations, Python provides inbuilt module name as **os**.
- ✓ **os** module contains several functions to perform directory related operations.

<b>Program Name</b>	To know current working directory. demo20.py
<b>Output</b>	<pre>import os cwd=os.getcwd() print("Current Working Directory:" , cwd)</pre> Current Working Directory: C:\Users\Nireekshana\Desktop\Files_in_python

<b>Program Name</b>	To create sub directory in current working directory. demo21.py
<b>Output</b>	<pre>import os os.mkdir("mysub") print("mysub directory created in current working directory")</pre> mysub directory created in cwd

## Make a note

- ✓ Assuming that **mysub** already present in **cwd**.

```
cwd
 |-mysub
   |-mysub2
```

**Program Name** To create sub directory in mysub directory.  
demo22.py

```
import os  
os.mkdir("mysub/mysub2")  
print("mysub2 created inside mysub")
```

**Output**

mysub2 created inside mysub

**Program Name** Create multiple directories like sub1, in that sub2, in that sub3  
demo23.py

```
import os  
os.makedirs("sub1/sub2/sub3")  
print("sub1 and in that sub2 and in that sub3 directories created")
```

**Output**

sub1 and in that sub2 and in that sub3 directories created

**Program Name** Removing a directory  
demo24.py

```
import os  
os.rmdir("mysub/mysub2")  
print("mysub2 directory deleted")
```

**Output**

mysub2 directory deleted

**Program Name** Removing all directory  
demo25.py

```
import os  
os.removedirs("sub1/sub2/sub3")  
print("All 3 directories sub1,sub2 and sub3 removed")
```

**Output**

All 3 directories sub1,sub2 and sub3 removed

**Program Name** Removing directory  
demo26.py

```
import os
os.rename("mysub", "newdir")
print("mysub directory renamed to newdir")
```

**Output**

```
mysub directory renamed to newdir
```

To know contents of directory:

- ✓ **os** module provides `listdir()` to list out the contents of the specified directory.
- ✓ It won't display the contents of sub directory.

**Program Name** To know the contents of directory  
demo27.py

```
import os
print(os.listdir("."))
```

**Output**

```
['abc.txt', 'demo1.py', 'demo2.py', 'demo3.py']
```

- ✓ The above program display contents of current working directory but not contents of sub directories.

Pickling and Unpickling of Objects:

- ✓ Sometimes we have to write total state of object to the file and we have to read total object from the file.
- ✓ The process of writing state of object to the file is called pickling and the process of reading state of an object from the file is called unpickling.
- ✓ We can implement pickling and unpickling by using pickle module of Python.
- ✓ pickle module contains `dump()` function to perform pickling.

```
pickle.dump(object, file)
```

- ✓ pickle module contains `load()` function to perform unpickling

```
obj=pickle.load(file)
```

## DVS Technologies

**Program Name** Example by using pickle.  
demo28.py

```
import pickle

class Employee:
    def __init__(self, eno, ename, esal, eaddr):
        self.eno=eno
        self.ename=ename
        self.esal=esal
        self.eaddr=eaddr
    def display(self):
        print(self.eno, "\t", self.ename, "\t", self.esal, "\t",
        self.eaddr)

with open("emp.dat", "wb") as f:
    e=Employee(100, "Nireekshan", 1000, "Hyd")
    pickle.dump(e,f)
    print("Pickling of Employee Object completed...")

with open("emp.dat", "rb") as f:
    obj=pickle.load(f)
    print("Printing Employee Information after unpickling")
    obj.display()
```

### Output

```
Pickling of Employee Object completed...
Printing Employee Information after unpickling
1      Nireekshan    10000  Banglore
```

**Program Name** Writing multiple objects to the file.  
demo29.py

```
class Employee:

    def __init__(self, eno, ename, esal, eaddr):
        self.eno=eno
        self.ename=ename
        self.esal=esal
        self.eaddr=eaddr

    def display(self):
        print(self.eno, "\t", self.ename, "\t", self.esal, "\t",
        self.eaddr)
```

### Output

<b>Program Name</b>	Writing multiple objects to the file. demo30.py
	<pre>import emp, pickle f=open("emp.dat","wb") n=int(input("Enter The number of Employees:")) for i in range(n):     eno=int(input("Enter Employee Number:"))     ename=input("Enter Employee Name:")     esal=float(input("Enter Employee Salary:"))     eaddr=input("Enter Employee Address:")     e=demo29.Employee(eno, ename, esal, eaddr)     pickle.dump(e, f) print("Employee Objects pickled successfully")</pre>
<b>Output</b>	

<b>Program Name</b>	Loading by using pickle demo32.py
	<pre>import emp, pickle f=open("emp.dat","rb") print("Employee Details:")  while True:     try:         obj=pickle.load(f)         obj.display()     except EOFError:         print("All employees Completed")         break f.close()</pre>
<b>Output</b>	

## 26. Database connectivity

### Data Storage Areas

As the Part of our Applications, we required to store our Data like,

- ✓ Customers Information,
- ✓ Billing Information,
- ✓ Calls Information etc.

To store this Data, we required Storage Areas. There are mainly 2 types of Storage Areas.

- 1) Temporary Storage Areas
- 2) Permanent Storage Areas

#### 1. Temporary Storage Areas:

- ✓ These are the Memory Areas where Data will be stored temporarily.
- ✓ Example: Python objects like list, tuple, dictionary.
- ✓ Once Python program completes its execution then these objects will be destroyed automatically, and data will be lost.

#### 2. Permanent Storage Areas:

- ✓ Also known as Persistent Storage Areas.
- ✓ Here we can store Data permanently.
- ✓ Example: File Systems, Databases, Data warehouses, Big Data Technologies etc

### File Systems:

- ✓ File Systems can be provided by Local operating System.
- ✓ File Systems are best suitable to store very less Amount of Information.

### Limitations:

- ✓ We cannot store huge Amount of Information.
- ✓ There is no Query Language support and hence operations will become very complex.
- ✓ There is no Security for Data.
- ✓ There is no Mechanism to prevent duplicate Data. Hence there may be a chance of data Inconsistency Problems.

### How to overcome limitations?

- ✓ To overcome the above Problems of File Systems, we should go for Databases.

### Databases:

- ✓ We can store the data in database in a table.
- ✓ We can store Huge Amount of Information in the Databases.
- ✓ Query Language Support is available for every Database and hence we can perform Database Operations very easily.

## Accessing data from Database

- ✓ To access Data present in the Database, compulsory **username** and **password** must be required. Hence Data is secured.
- ✓ Inside Database Data will be stored in the form of Tables.
- ✓ While creating Database Table Schemas, Database Admin follow various Normalization Techniques and can implement various Constraints like Unique Key Constraints, Primary Key Constraints etc which prevent Data Duplication.
- ✓ Hence there is no chance of Data Inconsistency Problems.

## Limitations of Databases:

- ✓ Database cannot hold very Huge Amount of Information like Terabytes of Data.
- ✓ Database can provide support only for Structured Data (Tabular Data OR Relational Data)
- ✓ Database cannot provide support for Semi Structured Data (like XML Files) and Unstructured Data (like Video Files, Audio Files, Images etc)

## How to overcome limitations?

- ✓ To overcome these Problems, we should go for more Advanced Storage Areas like Big Data storage related like HDFS (Hadoop), Data warehouses etc.
  - ✓ Sometimes as the part of Programming requirement we have to connect to the database and we have to perform several operations like,
    - creating tables,
    - inserting data,
    - updating data,
    - deleting data,
    - selecting data etc.
  - ✓ We can use SQL Language to talk to the database and we can use Python to send those SQL commands to the database.
  - ✓ Python provides inbuilt support for several databases like,
    - Oracle,
    - MySql,
    - SqlServer,
    - GadFly,
    - sqlite, etc.
1. Python has separate module for each database.
    1. **cx\_Oralce** module for communicating with Oracle database
    2. **pymssql** module for communicating with Microsoft Sql Server

## Standard Steps for Python database Programming:

1. import database specific module
2. Establish Connection.
3. Create Cursor object
4. In-built methods to execute the sql queries
5. commit or rollback
6. Fetch the result from the Cursor
7. close the resources

## Standard Steps for Python database Programming:

### 1. import database specific module

1. We need to import specific database module to work with database
2. We can import specific module by using import keyword

#### Syntax

```
import cx_Oracle
```

**Program Name**      importing cx\_Oracle module  
                         demo1.py

```
import cx_Oracle
print("Step 1: Successfully imported cx_Oracle module")
```

#### Output

```
Step 1: Successfully imported cx_Oracle module
```

### 2. Establish connection

- ✓ We can establish connection between python program and database by using connect() method.
- ✓ While we are calling this connect() method then its returns Connection object.

#### Syntax

```
con = cx_Oracle.connect("username/password@systeminfo")
```

#### Example

```
con=cx_Oracle.connect('system/root@localhost')
```

**Program Name**

importing cx\_Oracle module and connecting to database  
demo2.py

```
import cx_Oracle

print("Step 1: Successfully imported cx_Oracle module")

con=cx_Oracle.connect('system/root@localhost')

print("Step 2: Successfully connected to Oracle Database")

print("Oracle database version is: ",con.version)

con.close()

print("Successfully closed database connection")
```

**Output**

```
Step 1: Successfully imported cx_Oracle module
Step 2: Successfully connected to Oracle Database
Oracle database version is: 11.2.0.2.0
Successfully closed database connection
```

### 3. Create Cursor object

- ✓ After connection object created then the next step is, we need to create Cursor object.
- ✓ With connection object we need to call cursor() method.
- ✓ Whenever we are calling cursor() method then it returns Cursor object

**Syntax**

```
cursor_object_name = connection_object_name.cursor()
```

**Example**

```
cursor=con.cursor()
```

**Program Name**

importing cx\_Oracle module and connecting and creating Cursor object  
demo3.py

```
import cx_Oracle

print("Step 1: Successfully imported cx_Oracle module")
```

```
con=cx_Oracle.connect('system/root@localhost')
print("Step 2: Successfully connected to Oracle Database")
cursor = con.cursor()
print("Step 3: Successfully created cursor object")
con.close()
print("Successfully closed database connection")
```

## Output

```
Step 1: Successfully imported cx_Oracle module
Step 2: Successfully connected to Oracle Database
Step 3: Successfully created cursor object
Successfully closed database connection
```

## 4. Execute the sql queries

- ✓ To execute sql queries we need to use three predefined methods based on requirement,
  1. execute("sql query")
  2. executescript()
  3. executemany()
- ✓ These methods we need to call by using Cursor object.

### 1. execute("sql query")

- ✓ This method is used to execute a single sql query

#### Syntax

```
cursor_object_name.execute("sql query")
```

#### Example

```
cursor.execute()
```

### 2. executescript("sql queries")

- ✓ This method is used to execute sql queries separated by semi colon.

## Syntax

```
cursor_object_name.executescript("sql queries ")
```

## Example

```
cursor.executescript ()
```

## 3. executemany()

- ✓ This method is used to execute parameterised query.

## Syntax

```
cursor_object_name.executemany("sql queries ")
```

## Example

```
cursor.executemany("sql queries")
```

## 5. commit or rollback

- ✓ We can commit or rollback the changes based on requirement in case of DML queries means for insert, update and delete queries.
- ✓ We can call commit or rollback methods by using connection object

### commit()

- ✓ commit() method saves the changes into the database.
- ✓ commit() method we need to call by using connection object

## Syntax

```
connection_object_name.commit()
```

## Example

```
con.commit()
```

### rollback()

- ✓ rollback() method roll the temporary changes to back.
- ✓ rollback() method we need to call by using connection object

## Syntax

```
connection_object_name.rollback()
```

## Example

```
con.rollback()
```

## 6. Fetch the results from Cursor object

- ✓ We can fetch or get the result from cursor object in case of select queries.
- ✓ While fetching the result we can use three types of methods based on requirement.

### 1. `fetchone()`

- ✓ This method is used to fetch one record.
- ✓ Internally this method returns one record as a tuple.
- ✓ If there are no more records, then it returns `None`

## Syntax

```
cursor_object_name.fetchone()
```

## Example

```
cursor.fetchone()
```

### 2. `fetchmany(number of records)`

- ✓ By using this method, we can fetch or get number of records.
- ✓ This method accepts **number of records (integer value)** to fetch and returns tuple where each record itself is a tuple.
- ✓ If we pass number of records 2 then it returns 2 records.
- ✓ If there are not more records, then it returns an empty tuple.

## Syntax

```
cursor_object_name.fetchmany(n)
```

## Example

```
cursor.fetchmany(2)
```

## 7. close the resources

- ✓ Once if programmer opened any database connection or created any cursor object then programmer should take responsibility to close opened database connection and close the cursor object.

### When should we close the resources?

- ✓ After completing all operations then programmer should close the resources.

### How to close the resources?

- ✓ By using close() method we can close the resources.

#### Closing cursor

##### Syntax

```
cursor_object_name.close()
```

##### Example

```
cursor.close()
```

#### Closing connection

##### Syntax

```
connection_object_name.close()
```

##### Example

```
con.close()
```

#### Important methods while working with Database

- ✓ The following is the list of important methods which can be used for python database programming.
  1. connect()
  2. cursor()
  3. execute()
  4. executescript()

5. executemany()
6. commit()
7. rollback()
8. fetchone()
9. fetchall()
10. fetchmany(n)
11. close()

#### Make a note

- ✓ These methods are common for all databases.
- ✓ So, these methods won't be changed from database to database.

#### Working with Oracle Database:

##### Driver

- ✓ Driver is a program which works like interface or translator between python program and database.

##### Translator

- ✓ Translator is a program which helps to communicate to database from python program.
- ✓ Translator translates python calls into database specific calls and database specific calls into python calls.
- ✓ This translator is nothing but Driver/Connector.

##### cx\_Oracle

- ✓ For Oracle database the name of driver needed is [cx\\_Oracle](#).
- ✓ [cx\\_Oracle](#) is a python module that enables access to Oracle Database.
- ✓ It can be used for both Python2 and Python3.
- ✓ It can work with any version of Oracle database like 9,10,11 and 12.

#### Installing cx\_Oracle:

#### Make a note

- ✓ From normal command prompt (But not from Python console) execute the following command.

```
E:\python_programs>pip install cx_Oracle
Collecting cx_Oracle
  Downloading cx_Oracle-6.0.2-cp36-cp36m-win32.whl (100kB)
  100% |-----| 102kB 256kB/s
Installing collected packages: cx-Oracle
```

Successfully installed cx-Oracle-6.0.2

## How to Test Installation:

- ✓ From python console execute the following command:

```
>>> help("modules")
```

- ✓ In the output we can see **cx\_Oracle** among all modules

**Program Name** Connect with Oracle database and print its version.  
demo4.py

```
import cx_Oracle  
con=cx_Oracle.connect('system/root@localhost')  
print(con.version)  
con.close()
```

**Output**  
11.2.0.2.0

**Program Name** Create employees table in the oracle database  
demo5.py

```
import cx_Oracle  
try:  
    con=cx_Oracle.connect('system/root@localhost')  
    cursor=con.cursor()  
    cursor.execute("create table employees(eno number, ename  
varchar2(10),esal number(10,2),eaddr varchar2(10))")  
    print("Table created successfully")  
  
except cx_Oracle.DatabaseError as e:  
    if con:  
        con.rollback()  
        print(e)  
  
finally:  
    if cursor:  
        cursor.close()  
    if con:  
        con.close()
```

**Output**  
Table created successfully

**Confirmation** Open oracle SQL command Line

```
SQL> conn  
Enter user-name: system  
Enter password: ****
```

```
SQL> desc employees;
```

Name	Null?	Type
ENO		NUMBER
ENAME		VARCHAR2(10)
ESAL		NUMBER(10,2)
EADDR		VARCHAR2(10)

```
SQL> select * from employees;
```

```
no rows selected
```

```
SQL>
```

**Expected common errors while executing programs**

Reasons for common error	Common errors names
1. If username or password is wrong then	invalid username/password; logon denied
2. If table does not exist or not created	table or view does not exist

**Program Name** Insert single row in the employees table  
demo6.py

```
import cx_Oracle  
  
try:  
    con=cx_Oracle.connect('system/root@localhost')  
    cursor=con.cursor()  
    cursor.execute("insert into employees  
values(1,'Ramesh',10000,'Banglore')")  
    con.commit()  
    print("Employee Record Inserted Successfully")  
  
except cx_Oracle.DatabaseError as e:
```

```
if con:  
    con.rollback()  
    print(e)  
  
finally:  
  
    if cursor:  
        cursor.close()  
  
    if con:  
        con.close()
```

## Output

Employee Record Inserted Successfully

## Confirmation

Open oracle SQL command Line

```
SQL> conn  
Enter user-name: system  
Enter password: ****  
  
SQL> select * from employees;  
  
ENO ENAME      ESAL EADDR  
-----  
1 Ramesh       10000 Banglore  
  
SQL>
```

## Program Name

Insert single row in the employees table  
demo7.py

```
import cx_Oracle  
  
try:  
    con=cx_Oracle.connect('system/root@localhost')  
    cursor=con.cursor()  
    cursor.execute("insert into employees  
values(2,'Prasad',20000,'Banglore')")  
    con.commit()  
    print("Employee Record Inserted Successfully")  
  
except cx_Oracle.DatabaseError as e:  
  
    if con:  
        con.rollback()
```

```
    print(e)

finally:

    if cursor:
        cursor.close()

    if con:
        con.close()
```

## Output

Employee Record Inserted Successfully

**Confirmation** Open oracle SQL command Line

```
SQL> conn
Enter user-name: system
Enter password: ****

SQL> select * from employees;

ENO ENAME      ESAL EADDR
-----
1 Ramesh       10000 Banglore
2 Prasad       20000 Banglore

SQL>
```

## Make a note

- ✓ While performing DML Operations (insert | update | delete), compulsory we have to use commit() method then only the results will be reflected in the database.

<b>Program Name</b>	Insert single row in the employees table, without using commit() demo8.py
---------------------	--

```
import cx_Oracle

try:
    con=cx_Oracle.connect('system/root@localhost')
    cursor=con.cursor()

    cursor.execute("insert into employees
values(3,'Hari',30000,'Banglore')")
    print("Employee Record Inserted Successfully")

except cx_Oracle.DatabaseError as e:
```

```
if con:  
    con.rollback()  
    print(e)  
  
finally:  
  
    if cursor:  
        cursor.close()  
  
    if con:  
        con.close()
```

## Output

Employee Record Inserted Successfully

## Confirmation

Open oracle SQL command Line

```
SQL> conn  
Enter user-name: system  
Enter password: ****  
  
SQL> select * from employees;  
  
ENO ENAME          ESAL EADDR  
-----  
1 Ramesh           10000 Banglore  
2 Prasad           20000 Banglore  
  
SQL>
```

## Conclusion:

- ✓ In above program while inserting the single row of the data we didn't use commit() method, so data not inserted to database.
- ✓ So, while performing DML Operations (insert | update | delete), compulsory we have to use commit() method then only the results will be reflected in the database.

**Program**      Insert multiple rows in the employees table by using executemany() method.

**Name**      demo9.py

```
import cx_Oracle  
try:  
    con=cx_Oracle.connect('system/root@localhost')  
    cursor=con.cursor()
```

```
sql="insert into employees values(:eno, :ename, :esal, :eaddr)"

records=[(3,'Hari',30000,'Mumbai'),
         (4,'Hema',40000,'BZA'),
         (5,'Mohan',50000,'Banglore')]

cursor.executemany(sql, records)
con.commit()
print("Records Inserted Successfully")

except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print(e)

finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

## Output

Records Inserted Successfully

**Confirmation** Open oracle SQL command Line

```
SQL> conn
Enter user-name: system
Enter password: ****

SQL> select * from employees;

ENO ENAME      ESAL EADDR
-----
1 Ramesh       10000 Banglore
2 Prasad       20000 Banglore
3 Hari          30000 Mumbai
4 Hema          40000 BZA
5 Mohan         50000 Banglore

5 rows selected.

SQL>
```

**Program Name** Insert multiple rows in the employees table with dynamic input from the keyboard  
demo10.py

```
import cx_Oracle
try:
    con=cx_Oracle.connect('system/root@localhost')
    cursor=con.cursor()

    while True:

        eno=int(input("Enter Employee Number:"))
        ename=input("Enter Employee Name:")
        esal=float(input("Enter Employee Salary:"))
        eaddr=input("Enter Employee Address:")
        sql="insert into employees values(%d, '%s', %f, '%s')"
        cursor.execute(sql %(eno, ename, esal, eaddr))
        print("Record Inserted Successfully")

        option=input("Do you want to insert one more record[Yes | No] :")

        if option=="No":
            con.commit()
            break
except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem with sql :",e)

finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

**Output**

```
C:\Users\Nireekshan\Desktop>py demo.py
Enter Employee Number:6
Enter Employee Name: Arjun
Enter Employee Salary:60000
Enter Employee Address: Guntur
Record Inserted Successfully
Do you want to insert one more record [Yes | No] : No
```

**Confirmation** Open oracle SQL command Line

```
SQL> conn
Enter user-name: system
Enter password: ****

SQL> select * from employees;

    ENO ENAME          ESAL EADDR
-----  -----
1 Ramesh      10000 Banglore
2 Prasad       20000 Banglore
3 Hari         30000 Mumbai
4 Hema          40000 BZA
5 Mohan        50000 Banglore
6 Arjun        60000 Guntur

6 rows selected.

SQL>
```

**Program Name**

Increment all employee salaries by 600 whose salary < 50000  
demo11.py

```
import cx_Oracle
try:
    con=cx_Oracle.connect('system/root@localhost')
    cursor=con.cursor()

    increment=float(input("How much amount need to Increment:"))
    salrange=float(input("Enter Salary Range:"))
    sql="update employees set esal=esal+%f where esal<%f"
    cursor.execute(sql %(increment, salrange))
    print("Records Updated Successfully")
    con.commit()
except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem with sql : ",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

**Output**

```
C:\Users\Nireekshan\Desktop>py demo.py
How much amount need to Increment : 600
```

Enter Salary Range:50000  
Records Updated Successfully

**Confirmation** Open oracle SQL command Line

```
SQL> conn
Enter user-name: system
Enter password: ****

SQL> select * from employees;

    ENO ENAME          ESAL EADDR
-----  -----
      1 Ramesh        10600 Banglore
      2 Prasad        20600 Banglore
      3 Hari          30600 Mumbai
      4 Hema          40600 BZA
      5 Mohan         50000 Banglore
      6 Arjun         60000 Guntur

6 rows selected.

SQL>
```

**Program Name**

select all employees info by using fetchone() method  
demo12.py

```
import cx_Oracle
try:
    con=cx_Oracle.connect('system/root@localhost')
    cursor=con.cursor()
    cursor.execute("select * from employees")
    row=cursor.fetchone()

    while row is not None:
        print(row)
        row=cursor.fetchone()

except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem with sql : ",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

**Output**

```
(1, 'Ramesh', 10600.0, 'Banglore')
(2, 'Prasad', 20600.0, 'Banglore')
(3, 'Hari', 30600.0, 'Hyd')
(4, 'Hema', 40600.0, 'BZA')
(5, 'Mohan', 50000.0, 'Banglore')
(6, 'Arjun', 60000.0, 'Guntur')
```

**Program Name** select employees info by using fetchmany() method and the required number of rows will be provided as dynamic input  
demo13.py

```
import cx_Oracle
try:
    con=cx_Oracle.connect('system/root@localhost')
    cursor=con.cursor()
    cursor.execute("select * from employees")
    n=int(input("Enter the number of required rows:"))
    data=cursor.fetchmany(n)

    for row in data:
        print(row)
except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem with sql : ",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

**Output**

```
C:\Users\Nireekshana\Desktop>py demo13.py
```

```
Enter the number of required rows:2
(1, 'Ramesh', 10600.0, 'Banglore')
(2, 'Prasad', 20600.0, 'Banglore')
```

**Program Name** select all employees info by using fetchall() method  
demo14.py

```
import cx_Oracle
try:
    con=cx_Oracle.connect('system/root@localhost')
    cursor=con.cursor()
    cursor.execute("select * from employees")
    data=cursor.fetchall()
```

```
    print(data)

except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem with sql :",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

## Output

```
[(1, 'Ramesh', 10600.0, 'Banglore'), (2, 'Prasad', 20600.0, 'Banglore'),
(3, 'Hari', 30600.0, 'Hyd'), (4, 'Hema', 40600.0, 'BZA')
(5, 'Mohan', 50000.0, 'Banglore'), (6, 'Arjun', 60000.0, 'Guntur')]
```

**Program Name** select a specific record by index  
demo15.py

```
import cx_Oracle
try:
    con=cx_Oracle.connect('system/root@localhost')
    cursor=con.cursor()
    cursor.execute("select * from employees")
    data=cursor.fetchall()

    print(data[0])

except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem with sql :",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

## Output

```
(1, 'Ramesh', 10600.0, 'Banglore')
```

**Program Name** select a specific field from record by index  
demo16.py

```
import cx_Oracle
try:
    con=cx_Oracle.connect('system/root@localhost')
    cursor=con.cursor()
    cursor.execute("select * from employees")
    data=cursor.fetchall()

    print(data[0][0])
    print(data[0][1])

except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem with sql : ",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

**Output**

```
1
Ramesh
```

## 24. Regular Expressions

What is Regular expression?

- ✓ A regular expression is a sequence of characters that define a search pattern in theoretical computer science and formal language.

When should we chose?

- ✓ If we want to represent a group of Strings according to a particular format/pattern then we should go for Regular Expressions.
- ✓ Regular Expressions is a declarative mechanism to represent a group of Strings according to particular format/pattern.

Example 1: We can write a regular expression to represent all mobile numbers

Example 2: We can write a regular expression to represent all mail ids.

The main important application areas of Regular Expressions are

1. To develop validation frameworks/validation logic.
  2. To develop Pattern matching applications (ctrl-f in windows, grep in UNIX etc)
  3. To develop Translators like compilers, interpreters etc
  4. To develop digital circuits
  5. To develop communication protocols like TCP/IP, UDP etc.
- ✓ We can develop Regular Expression Based applications by using python module: **re**
  - ✓ This module contains several inbuilt functions to use Regular Expressions very easily in our applications.

1. **compile()**

- ✓ re module contains compile() function to compile a pattern into RegexObject.

```
pattern = re.compile("ab")
```

2. **finditer():**

- ✓ Returns an Iterator object which yields Match object for every Match

```
matcher = pattern.finditer("abaababa")
```

On Match object we can call the following methods.

- |            |   |                                  |
|------------|---|----------------------------------|
| 1. start() | : | Returns start index of the match |
| 2. end()   | : | Returns end+1 index of the match |
| 3. group() | : | Returns the matched string       |

**Program Name** importing re module and working with methods  
demo1.py

```
import re

count=0
pattern=re.compile("ab")
matcher=pattern.finditer("abaababa")

for match in matcher:
    count+=1
    print(match.start(),"...",match.end(),"...", match.group())
    print("The number of occurrences: ", count)
```

**Output**

```
0 ... 2 ... ab
The number of occurrences: 1
3 ... 5 ... ab
The number of occurrences: 2
5 ... 7 ... ab
The number of occurrences: 3
```

### Make a note

- ✓ We can pass pattern directly as argument to finditer() function.

**Program Name** importing re module and working with methods  
demo2.py

```
import re

count=0
matcher=re.finditer("ab", "abaababa")

for match in matcher:
    count+=1
    print(match.start(),"...",match.end(),"...",match.group())
    print("The number of occurrences: ",count)
```

## Output

```
0 ... 2 ... ab
The number of occurrences: 1
3 ... 5 ... ab
The number of occurrences: 2
5 ... 7 ... ab
The number of occurrences: 3
```

## Character classes:

- ✓ We can use character classes to search a group of characters

1. [abc]	:	Either a or b or c
2. [^abc]	:	Except a and b and c
3. [a-z]	:	Any Lower-case alphabet symbol
4. [A-Z]	:	Any upper-case alphabet symbol
5. [a-zA-Z]	:	Any alphabet symbol
6. [0-9]	:	Any digit from 0 to 9
7. [a-zA-Z0-9]	:	Any alphanumeric character
8. [^a-zA-Z0-9]	:	Except alphanumeric characters (Special Characters)

**Program Name** importing re module and working with methods  
demo3.py

```
import re

matcher=re.finditer("[abc]","a7b@k9z")

for match in matcher:
    print(match.start(),".....",match.group())
```

## Output

```
0 ..... a
2 ..... b
```

**Program Name** importing re module and working with methods  
demo3.py

```
import re

matcher=re.finditer("[^abc]","a7b@k9z")
```

```
for match in matcher:  
    print(match.start(),".....",match.group())
```

**Output**

```
1 ..... 7  
3 ..... @  
4 ..... k  
5 ..... 9  
6 ..... z
```

**Program Name** importing re module and working with methods  
demo4.py

```
import re  
  
matcher=re.finditer("[a-z]","a7b@k9z")  
  
for match in matcher:  
    print(match.start(),".....",match.group())
```

**Output**

```
0 ..... a  
2 ..... b  
4 ..... k  
6 ..... z
```

**Program Name** importing re module and working with methods  
demo5.py

```
import re  
  
matcher=re.finditer("[0-9]","a7b@k9z")  
  
for match in matcher:  
    print(match.start(),".....",match.group())
```

**Output**

```
1 ..... 7  
5 ..... 9
```

**Program Name** importing re module and working with methods  
demo6.py

```
import re

matcher=re.finditer("[a-zA-Z0-9]","a7b@k9z")

for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

```
0 ..... a
1 ..... 7
2 ..... b
4 ..... k
5 ..... 9
6 ..... z
```

**Program Name** importing re module and working with methods  
demo7.py

```
import re

matcher=re.finditer("[^a-zA-Z0-9]","a7b@k9z")

for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

```
3 ..... @
```

## Pre-defined Character classes:

✓ \s	:	Space character
✓ \S	:	Any character except space character
✓ \d	:	Any digit from 0 to 9
✓ \D	:	Any character except digit
✓ \w	:	Any word character [a-zA-Z0-9]
✓ \W	:	Any character except word character (Special Characters)
✓ .	:	Any character including special characters

**Program Name** importing re module and working with methods  
demo8.py

```
import re

matcher=re.finditer("\s","a7b k@9z")

for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

```
3 .....
```

**Program Name** importing re module and working with methods  
demo9.py

```
import re
matcher=re.finditer("\S","a7b k@9z")
for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

```
0 ..... a
1 ..... 7
2 ..... b
4 ..... k
5 ..... @
6 ..... 9
7 ..... z
```

**Program Name** importing re module and working with methods  
demo10.py

```
import re
matcher=re.finditer("\d","a7b k@9z")
for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

```
1 ..... 7
6 ..... 9
```

**Program Name** importing re module and working with methods  
demo11.py

```
import re

matcher=re.finditer("\D","a7b k@9z")

for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

```
0 ..... a
2 ..... b
3 .....
4 ..... k
5 ..... @
7 ..... z
```

**Program Name** importing re module and working with methods  
demo12.py

```
import re
matcher=re.finditer("\w","a7b k@9z")
for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

```
0 ..... a
1 ..... 7
2 ..... b
4 ..... k
6 ..... 9
7 ..... z
```

**Program Name** importing re module and working with methods  
demo13.py

```
import re
matcher=re.finditer("\W","a7b k@9z")
for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

```
3 .....
5 ..... @
```

**Program Name** importing re module and working with methods  
demo14.py

```
import re

matcher=re.finditer(".","a7b k@9z")

for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

```
0 ..... a
1 ..... 7
2 ..... b
3 ..... 
4 ..... k
5 ..... @
6 ..... 9
7 ..... z
```

## Quantifiers:

- ✓ We can use quantifiers to specify the number of occurrences to match.

✓ a	:	Exactly one 'a'
✓ a+	:	At least one 'a'
✓ a*	:	Any number of a's including zero number
✓ a?	:	At most one 'a' i.e either zero number or one number
✓ a{m}	:	Exactly m number of a's
✓ a{m, n}:		Minimum m number of a's and Maximum n number of a's

**Program Name** importing re module and working with methods  
demo15.py

```
import re
matcher=re.finditer("a","abaabaaab")
for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

```
0 ..... a
2 ..... a
3 ..... a
5 ..... a
6 ..... a
```

7 ..... a

**Program Name** importing re module and working with methods  
demo16.py

```
import re
matcher=re.finditer("a+","abaabaaab")
for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

0 ..... a  
2 ..... aa  
5 ..... aaa

**Program Name** importing re module and working with methods  
demo17.py

```
import re
matcher=re.finditer("a*","abaabaaab")

for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

0 ..... a  
1 .....  
2 ..... aa  
4 .....  
5 ..... aaa  
8 .....  
9 .....

**Program Name** importing re module and working with methods  
demo18.py

```
import re
matcher=re.finditer("a?", "abaabaaab")

for match in matcher:
    print(match.start(),".....",match.group())
```

**Output**

```
0 ..... a  
1 .....  
2 ..... a  
3 ..... a  
4 .....  
5 ..... a  
6 ..... a  
7 ..... a  
8 .....  
9 .....
```

**Program Name** importing re module and working with methods  
demo19.py

```
import re  
  
matcher=re.finditer("a{3}", "abaabaaab")  
  
for match in matcher:  
    print(match.start(),".....",match.group())
```

**Output**

```
5 ..... aaa
```

**Program Name** importing re module and working with methods  
demo20.py

```
import re  
  
matcher=re.finditer("a{2,4}", "abaabaaab")  
  
for match in matcher:  
    print(match.start(),".....",match.group())
```

**Output**

```
2 ..... aa  
5 ..... aaa
```

## Make a note

- ✓ ^x : It will check whether target string starts with x or not
- ✓ x\$ : It will check whether target string ends with x or not

## Important functions of re module:

1. match()
2. fullmatch()
3. search()
4. findall()
5. finditer()
6. sub()
7. subn()
8. split()
9. compile()

### 1. match():

- ✓ We can use match function to check the given pattern at beginning of target string.
- ✓ If the match is available then we will get Match object, otherwise we will get None.

<b>Program Name</b>	importing re module and working with methods demo21.py
	<pre>import re  s=input("Enter pattern to check: ") m=re.match(s, "abcabdefg")  if m!= None:     print("Match is available at the beginning of the String")     print("Start Index:", m.start(), "and End Index:", m.end())  else:     print("Match is not available at the beginning of the String")</pre>
<b>Output</b>	Enter pattern to check: abc Match is available at the beginning of the String Start Index: 0 and End Index: 3

## 2. fullmatch():

- ✓ We can use fullmatch() function to match a pattern to all of target string. i.e complete string should be matched according to given pattern.
- ✓ If complete string matched, then this function returns Match object otherwise it returns **None**.

**Program Name** importing re module and working with methods  
demo22.py

```
import re

s=input("Enter pattern to check: ")
m=re.fullmatch(s, "ababab")

if m!= None:
    print("Full String Matched")

else:
    print("Full String not Matched")
```

**Output**

```
C:\Users\Nireekshan\Desktop>py demo22.py
Enter pattern to check: abcdefg
Full String not Matched

C:\Users\Nireekshan\Desktop>py demo22.py
Enter pattern to check: ababab
Full String Matched
```

## 3. search():

- ✓ We can use search() function to search the given pattern in the target string.
- ✓ If the match is available, then it returns the Match object which represents first occurrence of the match.
- ✓ If the match is not available, then it returns None

**Program Name** importing re module and working with methods  
demo23.py

```
import re

s=input("Enter pattern to check: ")
m=re.search(s, "abaaaba")

if m!= None:
    print("Match is available")
    print("First Occurrence of match with start index:",
          m.start(),"and end index:", m.end())
```

```
else:  
    print("Match is not available")
```

## Output

```
Enter pattern to check: aaa  
Match is available  
First Occurrence of match with start index: 2 and end index: 5  
  
Enter pattern to check: bbb  
Match is not available
```

## 4. findall():

- ✓ To find all occurrences of the match.
- ✓ This function returns a list object which contains all occurrences.

**Program Name** importing re module and working with methods  
demo24.py

```
import re  
l=re.findall("[0-9]","a7b9c5kz")  
print(l)
```

## Output

```
['7', '9', '5']
```

## 5. finditer():

- ✓ Returns the iterator yielding a match object for each match.
- ✓ On each match object we can call start(), end() and group() functions.

**Program Name** importing re module and working with methods  
demo25.py

```
import re  
  
itr=re.finditer("[a-z]","a7b9c5k8z")  
  
for m in itr:  
    print(m.start(),"...",m.end(),"...", m.group())
```

## Output

```
0 ... 1 ... a
```

```
2 ... 3 ... b  
4 ... 5 ... c  
6 ... 7 ... k  
8 ... 9 ... z
```

## 6. sub():

- ✓ sub means substitution or replacement

```
re.sub(regex, replacement, targetstring)
```

- ✓ In the target string every matched pattern will be replaced with provided replacement.

**Program Name** importing re module and working with methods  
`demo26.py`

```
import re  
s=re.sub("[a-z]","#","a7b9c5k8z")  
print(s)
```

**Output**

```
#7#9#5#8#
```

## 7. subn():

- ✓ It is exactly same as sub except it can also returns the number of replacements.
- ✓ This function returns a tuple where first element is result string and second element is number of replacements. (resultstring, number of replacements)

**Program Name** importing re module and working with methods  
`demo27.py`

```
import re  
  
t=re.subn("[a-z]","#","a7b9c5k8z")  
  
print(t)  
print("The Result String:", t[0])  
print("The number of replacements:", t[1])
```

**Output**

```
('#7#9#5#8#', 5)  
The Result String: #7#9#5#8#  
The number of replacements: 5
```

## 8. split():

- ✓ If we want to split the given target string according to a particular pattern then we should go for split() function.
- ✓ This function returns list of all tokens.

**Program Name** importing re module and working with methods  
demo28.py

```
import re

l=re.split(", ", "nireekshan,ramesh,arjun")

print(l)
for t in l:
    print(t)
```

**Output**

```
['nireekshan', 'ramesh', 'arjun']
nireekshan
ramesh
arjun
```

**Program Name** importing re module and working with methods  
demo29.py

```
import re

l=re.split("\.", "www.nireekshan.com")

for t in l:
    print(t)
```

**Output**

```
www
nireekshan
com
```

## ^ symbol:

- ✓ We can use ^ symbol to check whether the given target string starts with our provided pattern or not.

```
res=re.search("^Learn", s)
```

- ✓ if the target string starts with Learn then it will return Match object, otherwise returns None.

<b>Program Name</b>	importing re module and working with methods demo30.py
	<pre>import re  s="Learning Python is Very Easy"  res=re.search("^Learn", s)  if res != None:     print("Target String starts with Learn")  else:     print("Target String Not starts with Learn")</pre>
<b>Output</b>	Target String starts with Learn

## \$ symbol:

- ✓ We can use \$ symbol to check whether the given target string ends with our provided pattern or not

```
res=re.search("Easy$", s)
```

- ✓ If the target string ends with Easy then it will return Match object, otherwise returns None.

<b>Program Name</b>	importing re module and working with methods demo31.py
	<pre>import re  s="Learning Python is Very Easy" res=re.search("Easy\$", s)</pre>

```
if res != None:  
    print("Target String ends with Easy")  
  
else:  
    print("Target String Not ends with Easy")
```

Output

Target String ends with Easy

Make a note

- ✓ If we want to ignore case then we have to pass 3rd argument re.IGNORECASE for search() function.

```
res = re.search("easy$", s, re.IGNORECASE)
```

Program Name importing re module and working with methods  
demo32.py

```
import re  
  
s="Learning Python is Very Easy"  
res=re.search("easy$", s, re.IGNORECASE)  
  
if res != None:  
    print("Target String ends with Easy by ignoring case")  
  
else:  
    print("Target String Not ends with Easy by ignoring case")
```

Output

Target String ends with Easy by ignoring case

## Requirement rules:

1. The allowed characters are **a-z, A-Z, 0-9, #**
2. The first character should be a lower-case alphabet symbol from a to k
3. The second character should be a digit divisible by 3
4. The length of identifier should be at least 2.

Regular expression for above requirement

✓ [a-k][0369][a-zA-Z0-9#]\*

1. Write a python program to check whether the given string is following requirement1 rules or not?

<b>Program Name</b>	Requirement demo33.py
<pre>import re  s=input("Enter string:") m=re.fullmatch("[a-k][0369][a-zA-Z0-9#]*",s)  if m!= None:     print(s, "Entered regular expression is matched")  else:     print(s, " Entered regular expression is not matched ")</pre>	
<b>Output</b>	<pre>C:\Users\Nireekshan\Desktop&gt;py demo33.py Enter string: b6jj8y## b6jj8y## Entered regular expression is matched  C:\Users\Nireekshan\Desktop&gt;py demo33.py Enter string:abcdefddfdf73834## abcdefddfdf73834## Entered regular expression is not matched</pre>

2. Write a Regular Expression to represent all 10 digit mobile numbers.

Rules:

- ✓ Every number should contain exactly 10 digits
- ✓ The first digit should be 7 or 8 or 9

## Regular expression

[7-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]

or

[7-9][0-9]{9}

or

[7-9]\d{9}

Program  
Name

Requirement  
demo34.py

```
import re

n=input("Enter number:")
m=re.fullmatch("[7-9]\d{9}",n)

if m!= None:
    print("Valid Mobile Number and this is Anushka number")

else:
    print("Please enter valid Mobile Number")
```

Output

```
C:\Users\Nireekshan\Desktop>py demo34.py
Enter number:9123456789
Valid Mobile Number and this is Anushka number
```

```
C:\Users\Nireekshan\Desktop>py demo34.py
Enter number:91234567891
Please enter valid Mobile Number
```

## DVS Technologies

3. Write a python program to extract all mobile numbers present in input.txt where numbers are mixed with normal text data

**Program Name** Requirement  
demo35.py

```
import re

f1=open("input.txt", "r")
f2=open("output.txt", "w")

for line in f1:
    list=re.findall("[7-9]\d{9}",line)

    for n in list:
        f2.write(n+"\n")

print("Extracted all Mobile Numbers into output.txt")

f1.close()
f2.close()
```

**Output**

Extracted all Mobile Numbers into output.txt

**input.txt**

9912345678  
1234567890  
9876543210

**onput.txt**

9912345678  
9876543210

4. Write a Python Program to check whether the given mail id is valid gmail id or not?

**Program Name** Requirement  
demo36.py

```
import re

s=input("Enter Mail id:")
m=re.fullmatch("\w[a-zA-Z0-9_.]*@[gmail\.]com", s)

if m!=None:
    print("Valid Mail Id")
```

```
else:  
    print("Invalid Mail id")
```

## Output

```
C:\Users\Nireekshan\Desktop>py demo36t.py  
Enter Mail id: nireekshan@gmail.com  
Valid Mail Id
```

```
C:\Users\Nireekshan\Desktop>py demo36.py  
Enter Mail id: nireekshan@fmail.com  
Invalid Mail id
```

5. Write a python program to check whether given car registration number is valid Karnataka State Registration number or not?

Program Name      Requirement  
demo37.py

```
import re  
  
s=input("Enter Vehicle Registration Number:")  
m=re.fullmatch("KA[012][0-9][A-Z]{2}\d{4}",s)  
  
if m!=None:  
    print("Valid Vehicle Registration Number")  
  
else:  
    print("Invalid Vehicle Registration Number")
```

## Output

```
C:\Users\Nireekshan\Desktop>py demo37.py  
Enter Vehicle Registration Number: KA01JA2787  
Valid Vehicle Registration Number
```

```
C:\Users\Nireekshan\Desktop>py demo37.py  
Enter Vehicle Registration Number: AP011234  
Invalid Vehicle Registration Number
```

## 25. Multithreading

### Multi-Tasking:

- ✓ Executing several tasks simultaneously is the concept of multitasking.

There are 2 types of Multi-Tasking

1. Process based Multi-Tasking
2. Thread based Multi-Tasking

### 1. Process based Multi-Tasking:

- ✓ Executing several tasks simultaneously where each task is a separate independent process is called process based multi-tasking.

#### Examples

- ✓ In a same computer,
  - Typing the program
  - Download the file
  - These tasks are executing simultaneously and independent of each other.
  - It is process based multi-tasking.

#### Where multi-tasking fits?

- ✓ This type of multi-tasking is best suitable at operating system level.

### 2. Thread based Multi-tasking:

- ✓ Executing several tasks simultaneously where each task is a separate independent part of the same program, is called Thread based multi-tasking, and each independent part is called a Thread.

#### Where multi-tasking fits?

- ✓ This type of multi-tasking is best suitable at programmatic level.

### Make a note

- ✓ Whether it is process based or thread based, the main advantage of multi-tasking is to improve performance of the system by reducing response time.

### Areas of multi-threading

The main important application areas of multi-threading are:

- ✓ To implement Multimedia graphics
- ✓ To develop animations
- ✓ To develop video games
- ✓ To develop web and application servers etc...

## Make a note

- ✓ Where-ever a group of independent jobs are available, then it is highly recommended to execute simultaneously instead of executing one by one.
- ✓ For such type of cases we should go for Multi-Threading.
  - In Hadoop MapReduce while processing the huge file, Hadoop process the file in simultaneously.
- ✓ Python provides one inbuilt module "threading" to provide support for developing threads.
- ✓ So, developing multi-threaded Programs is very easy in python.

## Key point

- ✓ Every Python Program by default contains one thread which is nothing but **MainThread**.

<b>Program Name</b>	Every python program having one Main thread demo1.py
	<pre>import threading print("Current Executing Thread:", threading.current_thread().getName())</pre>
<b>Output</b>	Current Executing Thread: MainThread

## Make a note

- ✓ **threading** module contains function `current_thread()` which returns the current executing Thread object.
- ✓ We need to call `getName()` method by using thread object to know current executing thread name

## The ways of Creating Thread in Python:

We can create a thread in Python by using 3 ways

1. Creating a Thread without using any class
2. Creating a Thread by inheriting Thread class
3. Creating a Thread without inheriting Thread class

### 1. Creating a Thread without using any class:

#### Thread class

- ✓ Thread is a pre-defined class present in threading module which can be used to create our own Threads.

**Program Name** creating thread without using any class  
demo2.py

```
from threading import *

def display():
    for i in range(6):
        print("Child Thread")

t=Thread(target=display)
t.start()
for i in range(6):
    print("Main Thread")
```

## Output

```
C:\Users\Nireekshana\Desktop>py demo2.py
Child Thread
Main Thread
```

```
C:\Users\Nireekshan\Desktop>py demo2.py
Child Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Child Thread
Child Thread
Child Thread
```

## Make a note

- ✓ If multiple threads present in our program, then we cannot expect exact execution order in output.
- ✓ Because of this we cannot provide exact output order for the above program.
- ✓ It is varied from machine to machine while running.

## 2. Creating a Thread by inheriting Thread class

- ✓ We can create a thread by predefined **Thread** class.
- ✓ Here our class should **inherit** Thread predefined class in python.
- ✓ Predefined Thread class contains **run()** method.
- ✓ In our child class we need to **override** run() method with our required functionality.

- ✓ Whenever we call `start()` method then automatically `run()` method will be executed and performs our job.

**Program Name** Creating a thread by inheriting `Thread` class  
`demo3.py`

```
from threading import *
class MyThread(Thread):
    def run(self):
        for i in range(6):
            print("Child Thread")

t=MyThread()
t.start()
for i in range(6):
    print("Main Thread")
```

**Output**

```
C:\Users\Nireekshan\Desktop>py demo3.py
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
```

```
C:\Users\Nireekshan\Desktop>py demo3.py
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
```

### 3. Creating a Thread without inheriting Thread class:

- ✓ We can create a thread without inheriting predefined `Thread` class.

**Program Name** Creating a thread without inheriting **Thread** class  
demo4.py

```
from threading import *

class Demo:
    def display(self):
        for i in range(5):
            print("Child Thread")

obj=Demo()
t=Thread(target=obj.display)
t.start()
for i in range(5):
    print("Main Thread")
```

## Output

```
C:\Users\Nireekshan\Desktop>py demo4.py
Child Thread
Main Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
Main Thread
Main Thread
Main Thread
Main Thread
```

```
C:\Users\Nireekshan\Desktop>py demo4.py
Child Thread
Child Thread
Child Thread
Main Thread
Main Thread
Child Thread
Child Thread
Main Thread
Main Thread
Main Thread
```

## Without multi-threading:

- ✓ We can finish our task without multi-threading also.
- ✓ But it's a sequential execution but not simultaneous.

**Program Name** Creating a thread without applying multi-threading concept  
demo5.py

```
from threading import *
import time

def doubles(numbers):
    for n in numbers:
        time.sleep(1)
        print("Double:", 2*n)

def squares(numbers):
    for n in numbers:
        time.sleep(1)
        print("Square:", n*n)

numbers=[1,2,3,4,5,6]
begintime=time.time()
doubles(numbers)
squares(numbers)
print("The total time taken:", time.time()-begintime)
```

**Output**

```
Double: 2
Double: 4
Double: 6
Double: 8
Double: 10
Double: 12
Square: 1
Square: 4
Square: 9
Square: 16
Square: 25
Square: 36
The total time taken: 12.091077327728271
```

**With multithreading:**

**Program Name** Finishing task quickly with applying multi-threading concept  
demo5.py

```
from threading import *
import time

def doubles(numbers):
    for n in numbers:
        time.sleep(1)
```

```
print("Double:", 2*n)

def squares(numbers):
    for n in numbers:
        time.sleep(1)
        print("Square:", n*n)

numbers=[1,2,3,4,5,6]
begintime=time.time()

t1=Thread(target=doubles, args=(numbers,))
t2=Thread(target=squares, args=(numbers,))

t1.start()
t2.start()

t1.join()
t2.join()

print("The total time taken:", time.time()-begintime)
```

## Output

```
Square: 1
Double: 2
Double: 4
Square: 4
Square: 9
Double: 6
Double: 8
Square: 16
Square: 25
Double: 10
Double: 12
Square: 36
The total time taken: 6.05490779876709
```

## Setting and Getting Name of a Thread:

- ✓ Every thread in python has name.
- ✓ It may be default name generated by Python.
- ✓ We can customize the thread name.

## How to get and set name of the thread?

- ✓ We can get and set name of thread by using the following Thread class methods.

t.getName()	:	Returns Name of Thread
t.setName(newName)	:	To set our own name

**Program Name** set and get name of the thread  
demo6.py

```
from threading import *
print("Main thread name is: ",current_thread().getName())
current_thread().setName("Nireekshan")
print("After customise the thread name: ",current_thread().getName())
print(current_thread().name)
```

**Output**

```
Main thread name is: MainThread
After customise the thread name: Nireekshan
Nireekshan
```

**Make a note**

- ✓ Every Thread has implicit variable "name" to represent name of Thread.

**Thread Identification Number (ident):**

- ✓ For every thread internally, a unique identification number is available. We can access this id by using implicit variable "ident"

**Program Name** Thread identification number getting.  
demo7.py

```
from threading import *
def m():
    print("Child Thread")
t=Thread(target=m)
t.start()
print("Main Thread Identification Number:", current_thread().ident)
print("Child Thread Identification Number:", t.ident)
```

**Output**

```
Child Thread
Main Thread Identification Number: 14832
Child Thread Identification Number: 12856
```

## active\_count():

- ✓ This function returns the number of active threads currently running.

**Program Name**

active threads count.  
demo8.py

```
from threading import *
import time

def display():
    print(current_thread().getName(), "...started")
    time.sleep(3)
    print(current_thread().getName(), "...ended")

print("The Number of active Threads:", active_count())

t1=Thread(target=display, name="ChildThread1")
t2=Thread(target=display, name="ChildThread2")
t3=Thread(target=display, name="ChildThread3")

t1.start()
t2.start()
t3.start()

print("The Number of active Threads:", active_count())
time.sleep(5)
print("The Number of active Threads:", active_count())
```

**Output**

```
The Number of active Threads: 1
ChildThread1 ...started
ChildThread2 ...started
ChildThread3 ...started
The Number of active Threads: 4
ChildThread3 ...ended
ChildThread1 ...ended
ChildThread2 ...ended
The Number of active Threads: 1
```

## enumerate() function:

- ✓ This function returns a list of all active threads currently running.

**Program Name**

active threads count.  
demo9.py

```
from threading import *
import time

def display():
    print(current_thread().getName(),"...started")
    time.sleep(3)
    print(current_thread().getName(),"...ended")

t1=Thread(target=display, name="ChildThread1")
t2=Thread(target=display, name="ChildThread2")
t3=Thread(target=display, name="ChildThread3")

t1.start()
t2.start()
t3.start()

l=enumerate()

for t in l:
    print("Thread Name:", t.name)

time.sleep(5)
l=enumerate()

for t in l:
    print("Thread Name:", t.name)
```

## Output

```
ChildThread1 ...started
ChildThread2 ...started
ChildThread3 ...started
Thread Name: MainThread
Thread Name: ChildThread1
Thread Name: ChildThread2
Thread Name: ChildThread3
ChildThread3 ...ended
ChildThread2 ...ended
ChildThread1 ...ended
Thread Name: MainThread
```

## isAlive():

- ✓ isAlive() method checks whether a thread is still executing or not.

<b>Program Name</b>	isAlive() method demo10.py
---------------------	-------------------------------

```
from threading import *
import time
```

```
def display():
    print(current_thread().getName(),"...started")
    time.sleep(3)
    print(current_thread().getName(),"...ended")

t1=Thread(target=display, name="ChildThread1")
t2=Thread(target=display, name="ChildThread2")

t1.start()
t2.start()

print(t1.name,"is Alive :",t1.isAlive())
print(t2.name,"is Alive :",t2.isAlive())

time.sleep(5)

print(t1.name,"is Alive :",t1.isAlive())
print(t2.name,"is Alive :",t2.isAlive())
```

## Output

```
ChildThread1 ...started
ChildThread2 ...started
ChildThread1 is Alive : True
ChildThread2 is Alive : True
ChildThread2 ...ended
ChildThread1 ...ended
ChildThread1 is Alive : False
ChildThread2 is Alive : False
```

## join() method:

- ✓ If a thread wants to wait until completing some other thread then we should go for join() method.

Program Name	join() method demo11.py
<pre>from threading import * import time  def display():     for i in range(5):         print("first Thread")         time.sleep(2)  t=Thread(target=display) t.start()</pre>	

```
t.join()#This Line executed by Main Thread  
  
for i in range(5):  
    print("second Thread")
```

## Output

```
first Thread  
first Thread  
first Thread  
first Thread  
first Thread  
second Thread  
second Thread  
second Thread  
second Thread  
second Thread
```

## join(seconds) method:

- ✓ We can call join(seconds) method with time period also.
- ✓ In this case thread will wait only specified amount of time

## Syntax

```
t.join(seconds)
```

**Program Name** join(seconds) method  
demo12.py

```
from threading import *  
import time  
  
def display():  
    for i in range(5):  
        print("first Thread")  
        time.sleep(2)  
  
t=Thread(target=display)  
t.start()  
t.join(5)#This Line executed by Main Thread  
  
for i in range(5):  
    print("next Thread")
```

## Output

```
first Thread  
first Thread  
first Thread  
next Thread
```

```
next Thread  
next Thread  
next Thread  
next Thread  
first Thread  
first Thread
```

## Daemon Threads:

- ✓ The threads which are running in the background are called Daemon Threads.
- ✓ The main objective of Daemon Threads is to provide support for Non-Daemon Threads (like main thread)

## Example: Garbage Collector

- ✓ Whenever Main Thread runs with low memory, immediately PVM runs Garbage Collector to destroy useless objects and to provide free memory, so that Main Thread can continue its execution without having any memory problems.
- ✓ We can check whether thread is Daemon or not by using `t.isDaemon()` method of Thread class or by using daemon property.

Program Name	Checking weather the Thread is Daemon or not <code>demo13.py</code>
--------------	--

```
from threading import *
print(current_thread().isDaemon())
print(current_thread().daemon)
```

Output	False False
--------	----------------

## `setDaemon()` method

- ✓ We can change Daemon nature by using `setDaemon()` method of Thread class.

Syntax	<code>t.setDaemon(True)</code>
--------	--------------------------------

## Warning

- ✓ But we can use this method before starting of Thread.
- ✓ i.e once a thread started, we cannot change its Daemon nature.
- ✓ Otherwise we will get `RuntimeError`: cannot set daemon status of active thread

Program Name	Checking weather the Thread is Daemon or not demo14.py
Output	<pre>from threading import * print(current_thread().isDaemon()) current_thread().setDaemon(True)</pre> <b>RuntimeError:</b> cannot set daemon status of active thread

## Default Nature

- ✓ By default, Main Thread is always non-daemon.
- ✓ But for the remaining threads Daemon nature will be inherited from parent to child.
- ✓ If the Parent Thread is Daemon, then child thread is also Daemon.
- ✓ If the Parent Thread is Non-Daemon, then ChildThread is also Non Daemon.

Program Name	Daemon nature will be inherited from parent thread to child thread. demo15.py
Output	<pre>from threading import *  def job():     print("Child Thread")  t=Thread(target=job) print(t.isDaemon())#False  t.setDaemon(True) print(t.isDaemon()) #True</pre> False True

## Main Thread is always Non-Daemon

- ✓ Main Thread is always Non-Daemon and we cannot change its Daemon Nature because it is already started at the beginning only.
- ✓ Whenever the last Non-Daemon Thread terminates automatically all Daemon Threads will be terminated

**Program Name** Main Thread is always Non-Daemon  
demo16.py

```
from threading import *
import time

def job():
    for i in range(10):
        print("Lazy Thread")
        time.sleep(2)

t=Thread(target=job)
#t.setDaemon(True)====>Line-1
t.start()

time.sleep(5)
print("End Of Main Thread")
```

**Output**

```
Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread
End Of Main Thread
Lazy Thread
```

## Scenarios from previous program

- ✓ In the above program if we comment Line-1 then both Main Thread and Child Threads are Non-Daemon and hence both will be executed until their completion.  
In this case output is:

```
Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread
End Of Main Thread
Lazy Thread
```

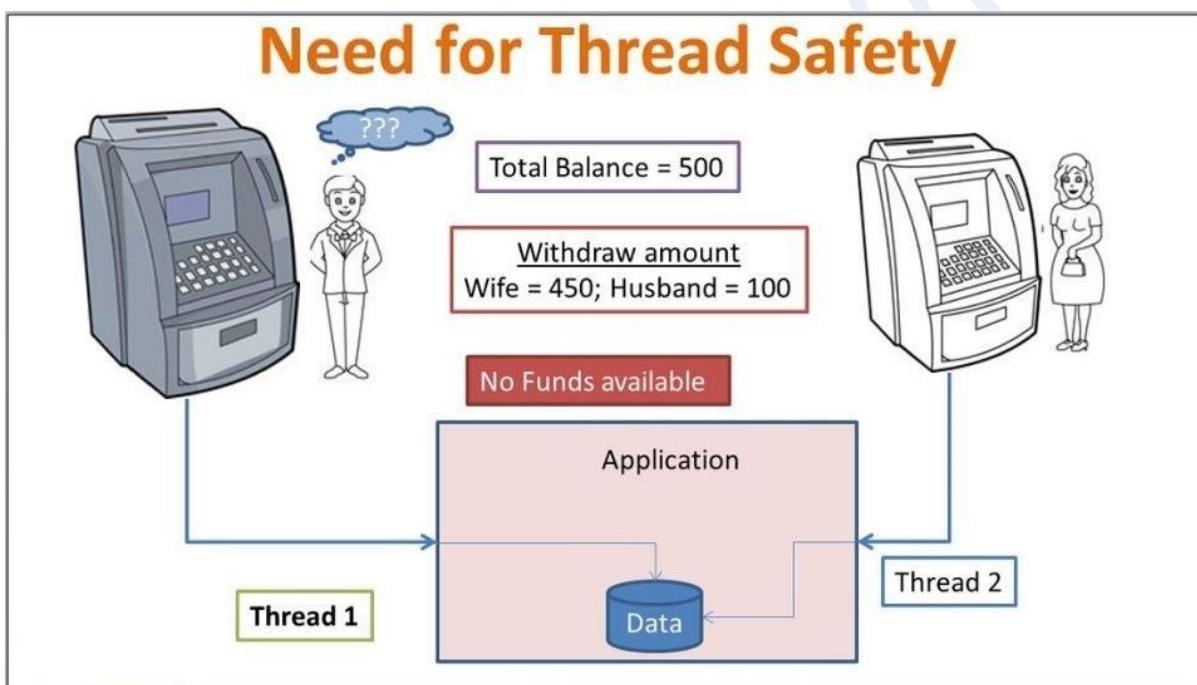
- ✓ If we are not commenting Line-1 then Main Thread is Non-Daemon and Child Thread is Daemon.
- ✓ Because whenever MainThread terminates automatically child thread will be terminated. In this case output is

```
Lazy Thread  
Lazy Thread  
Lazy Thread  
End of Main Thread
```

## Synchronization

- ✓ If multiple threads are executing simultaneously on object or data then there may be a chance of data inconsistency problems.

### Example 1



- ✓ Consider an example

- A couple who have a Joint account where both have their ATM cards.
- They come to different ATMs and try to withdraw some amount at the same time.
- Now the total balance in the account is 500.
- **Wife** tries to withdraw 450 and husband tried to withdraw 100.
- Both were shown the balance as 500.
- When trying to withdraw, the two threads assigned for the transactions try to withdraw the amount.
- But only one succeeds and other fails.
- This creates a confusion among the users.

- This happens because there is no restriction that only one thread can work on the data at a given time.

## Example 2



Let us consider another example

- ✓ where we have a travel website happybus.com through which you can book your bus tickets.
- ✓ Now there are only three seats left and two people are trying to book the tickets at the same time.
- ✓ Only one will be able to proceed and other will fail.

## Understandings

- ✓ If you consider in both the cases, the trouble arises when different threads try to work on the same data at same time.
- ✓ To avoid this problem, the code synchronization should be implemented which restricts multiple threads to work on the same code at the same time.

Program Name	Two threads work on same data and prints irregular output demo17.py
	<pre>from threading import * import time  def wish(name):     for i in range(10):         print("Hi : " ,end="")         time.sleep(2)</pre>

```
print(name)

t1=Thread(target=wish, args=("Nireekshan",))
t2=Thread(target=wish, args=("Arjun",))

t1.start()
t2.start()
```

## Output

```
Hi : Hi : Nireekshan
Arjun
Hi : Hi : Arjun
Nireekshan
Hi : Hi : Arjun
Nireekshan
Hi : Hi : Arjun
Nireekshan
Hi : Hi : Nireekshan
Arjun
Hi : Hi : Nireekshan
Hi : Arjun
Hi : Nireekshan
Arjun
```

- ✓ We are getting irregular output because both threads are executing simultaneously on wish() function.
- ✓ To overcome this problem, we should go for synchronization.

## How to overcome data inconsistency problems?

- ✓ In synchronization the threads will be executed one by one so that we can overcome data inconsistency problems.
- ✓ Synchronization means at a time only one Thread will be executed
- ✓ The main application areas of synchronization are,
  - Online Reservation system
  - Funds Transfer from joint accounts etc.

## How to implement synchronization

- ✓ In Python, we can implement synchronization by using the following concepts

1. Lock
2. RLock
3. Semaphore

### Synchronization By using Lock concept:

- ✓ Locks are the most fundamental synchronization mechanism provided by threading module.
- ✓ We can create Lock object as follows,
  - o `l=Lock()`
- ✓ The Lock object can be hold by only one thread at a time.
- ✓ If any other thread required, the same lock then it will wait until thread releases lock. (similar like, waiting in line to book train ticket, public telephone booth etc)

#### acquire() method

- ✓ A Thread can acquire the lock by using acquire() method.
  - o `l.acquire()`

#### release() method

- ✓ A Thread can release the lock by using release() method.
  - o `l.release()`

#### Make a note

- ✓ To call release() method compulsory thread should be owner of that lock. i.e thread should has the lock already, otherwise we will get Runtime Exception saying

**RuntimeError:** release unlocked lock

<b>Program Name</b>	lock acquiring and releasing demo18a.py
	<pre>from threading import * l=Lock() l.acquire() print("lock acquired") l.release() print("lock released")</pre>
<b>Output</b>	

lock acquired  
lock released

**Program Name** trying to release the lock without acquiring  
`demo18b.py`

```
from threading import *
l=Lock()
l.release()
print("lock released")
```

**Output**

```
RuntimeError: release unlocked lock
```

## 1. Lock()

- ✓ We can create Lock object as follows,
  - `l=Lock()`
- ✓ The Lock object can be hold by only one thread at a time.
- ✓ If any other thread required, the same lock then it will wait until thread releases lock. (similar like, waiting in line to book train ticket, public telephone booth etc)

**Program Name** Example by using Lock  
`demo18.py`

```
from threading import *
import time
l=Lock()

def wish(name):
    l.acquire()
    for i in range(10):
        print("Hi: ", end="")
        time.sleep(2)
        print(name)
    l.release()

t1=Thread(target=wish, args=("Nireekshan",))
t2=Thread(target=wish, args=("Arjun",))
t3=Thread(target=wish, args=("Ramesh",))

t1.start()
```

```
t2.start()  
t3.start()
```

## Output

```
Hi: Nireekshan  
Hi: Arjun  
Hi: Ramesh  
Hi: Ramesh
```

- ✓ In the above program at a time only one thread is allowed to execute wish() method and hence we will get regular output.

## Problem with Simple Lock:

- ✓ The standard Lock object does not care which thread is currently holding that lock.
- ✓ If the lock is held and any thread attempts to acquire lock, then it will be blocked, even the same thread is already holding that lock.

<b>Program Name</b>	problem with simple lock demo19.py
<pre>from threading import *</pre>	

```
I=Lock()  
  
print("Main Thread trying to acquire Lock")  
I.acquire()  
  
print("Main Thread trying to acquire Lock Again")  
I.acquire()
```

## Output

```
Main Thread trying to acquire Lock  
Main Thread trying to acquire Lock Again  
.....
```

- ✓ In the above Program main thread will be blocked because it is trying to acquire the lock second time.

## Make a note

- ✓ To kill the blocking thread from windows command prompt we have to use ctrl+break. Here ctrl+C won't work.
- ✓ If the Thread calls recursive functions or nested access to resources, then the thread may try to acquire the same lock again and again, which may block our thread.
- ✓ Hence Traditional Locking mechanism won't work for executing recursive functions.

## RLock()

- ✓ To overcome this problem, we should go for **RLock**(Reentrant Lock).
- ✓ Reentrant means the thread can acquire the same lock again and again.
- ✓ If the lock is held by other threads, then only the thread will be blocked.
- ✓ Reentrant facility is available only for owner thread but not for other threads.

<b>Program Name</b>	problem with simple RLock demo20.py
	<pre>from threading import *  I=RLock()  print("Main Thread trying to acquire Lock") I.acquire()  print("Main Thread trying to acquire Lock Again") I.acquire()</pre>

## Output

```
Main Thread trying to acquire Lock
```

Main Thread trying to acquire Lock Again

- ✓ In this case Main Thread won't be Locked because thread can acquire the lock any number of times.
  - ✓ This RLock keeps track of recursion level and hence for every acquire() call compulsory release() call should be available.
  - ✓ The number of acquire() calls and release() calls should be matched then only lock will be released.

## Example

```
I=RLock()  
I.acquire()  
I.acquire()  
I.release()  
I.release()
```

- ✓ After 2 release() calls only the Lock will be released.

**Make a note**

- ✓ Only owner thread can acquire the lock multiple times
  - ✓ The number of acquire() calls and release() calls should be matched.

Demo Program for synchronization by using RLock:

Program Name	Applying synchronization by using RLock demo21.py
	<pre>from threading import * import time  l=RLock()  def factorial(n):     l.acquire()     if n==0:         result=1     else:         result=n*factorial(n-1)     l.release()     return result  def results(n):     print("The Factorial of", n, "is:", factorial(n))  t1=Thread(target=results, args=(5,)) t2=Thread(target=results, args=(9,))</pre>

```
t1.start()  
t2.start()
```

## Output

```
The Factorial of 5 is: 120  
The Factorial of 9 is: 362880
```

## Make a note

- ✓ In the above program instead of RLock if we use normal Lock then the thread will be blocked.

## Difference between Lock and RLock

Lock	RLock
<ul style="list-style-type: none"><li>✓ Lock object can be acquired by only one thread at a time. Even owner thread also cannot acquire multiple times.</li></ul>	<ul style="list-style-type: none"><li>✓ RLock object can be acquired by only one thread at a time, but owner thread can acquire same lock object multiple times.</li></ul>
<ul style="list-style-type: none"><li>✓ Not suitable to execute recursive functions and nested access calls</li></ul>	<ul style="list-style-type: none"><li>✓ Best suitable to execute recursive functions and nested access calls</li></ul>
<ul style="list-style-type: none"><li>✓ In this case Lock object will takes care only Locked or unlocked and it never takes care about owner thread and recursion level.</li></ul>	<ul style="list-style-type: none"><li>✓ In this case RLock object will takes care whether Locked or unlocked and owner thread information, recursion level.</li></ul>

## Synchronization by using Semaphore:

- ✓ In the case of Lock and RLock, at a time only one thread is allowed to execute.
- ✓ Sometimes our requirement is at a time a particular number of threads are allowed to access like
  - At a time 10 members are allowed to access database server;
  - At a time 4 members are allowed to access Network connection.
- ✓ To handle this requirement, we cannot use Lock and RLock concepts and we should go for Semaphore concept.
- ✓ Semaphore can be used to limit the access to the shared resources with limited capacity.
- ✓ Semaphore is advanced Synchronization Mechanism.

## How to create Semaphore object?

- ✓ We can create Semaphore object as follows.

```
s=Semaphore(counter)
```

- ✓ Here counter represents the maximum number of threads are allowed to access simultaneously.
- ✓ The default value of counter is 1.
- ✓ Whenever thread executes acquire() method then the counter value will be decremented by 1 and if thread executes release() method then the counter value will be incremented by 1.

## Conclusion

- ✓ For every acquire() call counter value will be decremented
- ✓ For every release() call counter value will be incremented.

### Case-1:

- ✓ s=Semaphore() In this case counter value is 1 and at a time only one thread is allowed to access.
- ✓ It is exactly same as Lock concept.

### Case-2:

- ✓ s=Semaphore(3) In this case Semaphore object can be accessed by 3 threads at a time.
- ✓ The remaining threads have to wait until releasing the semaphore.
- ✓ In below program at a time 2 threads are allowed to access semaphore and hence 2 threads are allowed to execute wish() function.

<b>Program Name</b>	Synchronization by using Semaphore demo22.py
	<pre>from threading import * import time  s=Semaphore(2) def wish(name):     s.acquire()     for i in range(5):         print("Hi:", end="")         time.sleep(2)         print(name)     s.release()  t1=Thread(target=wish, args=("Nireekshan",)) t2=Thread(target=wish, args=("Mohan",))</pre>

```
t3=Thread(target=wish, args=("Ramesh",))
t4=Thread(target=wish, args=("Arjun",))
t5=Thread(target=wish, args=("Hari",))

t1.start()
t2.start()
t3.start()
t4.start()
t5.start()
```

### Output

```
Hi:Hi:Nireekshan
Mohan
Hi:Hi:Mohan
Nireekshan
Hi:Hi:Mohan
Hi:Nireekshan
Hi:Mohan
Hi:Nireekshan
Hi:Mohan
Hi:Nireekshan
Hi:Ramesh
Hi:Arjun
Hi:Ramesh
Hi:Arjun
Hi:Ramesh
Hi:Arjun
Hi:Ramesh
Hi:Arjun
Hari
Hi:Hari
Hi:Hari
Hi:Hari
```

## BoundedSemaphore:

- ✓ Normal Semaphore is an unlimited semaphore which allows us to call release() method any number of times to increment counter.
- ✓ The number of release() calls can exceed the number of acquire() calls also.

**Program Name** BoundedSemaphore  
demo23.py

```
from threading import *
s=Semaphore(2)
s.acquire()
s.acquire()
s.release()
s.release()
s.release()
s.release()
print("End")
```

**Output**

End

- ✓ It is valid because in normal semaphore we can call release() any number of times.
- ✓ BoundedSemaphore is exactly same as Semaphore except that the number of release() calls should not exceed the number of acquire() calls, otherwise we will get
- ✓ **ValueError**: Semaphore released too many times

<b>Program Name</b>	BoundedSemaphore demo24.py
	<pre>from threading import *  s=BoundedSemaphore(2)  s.acquire() s.acquire()  s.release() s.release() s.release() s.release()  print("End")</pre>

## Output

**ValueError:** Semaphore released too many times

- ✓ It is invalid because the number of release() calls should not exceed the number of acquire() calls in BoundedSemaphore.

## Make a note

- ✓ To prevent simple programming mistakes, it is recommended to use BoundedSemaphore over normal Semaphore.

## Difference between Lock and Semaphore:

- ✓ Lock(): At a time Lock object can be acquired by only one thread
- ✓ Semaphore: At a time Semaphore object can be acquired by fixed number of threads specified by counter value.

## Conclusion:

- ✓ The main advantage of synchronization is we can overcome data inconsistency problems.
- ✓ But the main disadvantage of synchronization is it increases waiting time of threads and creates performance problems.
- ✓ Hence if there is no specific requirement then it is not recommended to use synchronization.

## Inter Thread Communication:

- ✓ Sometime one thread may communicate to other thread as per the requirement.
- ✓ This concept is nothing but interthread communication.

### Example:

- ✓ After producing items Producer thread has to communicate with Consumer thread to notify about new item.
- ✓ Then consumer thread can consume that new item.

In Python, we can implement interthread communication by using the following ways

1. Event
2. Condition
3. Queue
- etc...

## Interthread communication by using Event Objects:

- ✓ Event object is the simplest communication mechanism between the threads.
- ✓ One thread signals an event and other threads wait for it.
- ✓ We can create Event object as follows,

```
event = threading.Event()
```

- ✓ Event manages an internal flag that can set() or clear() Threads can wait until event set.

### Methods of Event class:

#### 1. set()

- ✓ internal flag value will become True and it represents GREEN signal for all waiting threads.

#### 2. clear()

- ✓ internal flag value will become False and it represents RED signal for all waiting threads.

#### 3. isSet()

- ✓ This method can be used whether the event is set or not

#### 4. wait() | wait(seconds)

- ✓ Thread can wait until event is set

## Pseudo Code:

```
event = threading.Event()

#consumer thread has to wait until event is set
event.wait()

#producer thread can set or clear event
event.set()
event.clear()
```

<b>Program Name</b>	Inter thread communication demo25.py
	<pre>from threading import * import time  def producer():     time.sleep(5)     print("Producer thread producing items:")     print("Producer thread giving notification by setting event")     event.set()  def consumer():     print("Consumer thread is waiting for updation")     event.wait()     print("Consumer thread got notification and consuming items")  event=Event() t1=Thread(target=producer) t2=Thread(target=consumer)  t1.start() t2.start()</pre>
<b>Output</b>	Consumer thread is waiting for updation Producer thread producing items: Producer thread giving notification by setting event Consumer thread got notification and consuming items

Program Name	Inter thread communication demo26.py
	<pre>from threading import * import time  def traffic_police():     while True:         time.sleep(5)         print("Traffic Police Giving GREEN Signal")         event.set()         time.sleep(10)         print("Traffic Police Giving RED Signal")         event.clear()  def driver():     num=0     while True:         print("Drivers waiting for GREEN Signal")         event.wait()         print("Traffic Signal is GREEN...Vehicles can move")          while event.isSet():             num=num+1             print("Vehicle No:", num, " Crossing the Signal")             time.sleep(2)             print("Traffic Signal is RED...Drivers have to wait")      event=Event()  t1=Thread(target=traffic_police) t2=Thread(target=driver)  t1.start() t2.start()</pre>

## Output

```
Drivers waiting for GREEN Signal
Traffic Police Giving GREEN Signal
Traffic Signal is GREEN...Vehicles can move
Vehicle No: 1 Crossing the Signal
Traffic Signal is RED...Drivers have to wait
Vehicle No: 2 Crossing the Signal
Traffic Signal is RED...Drivers have to wait
Vehicle No: 3 Crossing the Signal
Traffic Signal is RED...Drivers have to wait
Vehicle No: 4 Crossing the Signal
Traffic Signal is RED...Drivers have to wait
Vehicle No: 5 Crossing the Signal
Traffic Police Giving RED Signal
Traffic Signal is RED...Drivers have to wait
```

.....

- ✓ In the above program driver thread has to wait until Trafficpolice thread sets event. i.e until giving **GREEN** signal.
- ✓ Once Traffic police thread sets event (giving **GREEN** signal), vehicles can cross the signal.
- ✓ Once traffic police thread clears event (giving **RED** Signal) then the driver thread has to wait.

## Interthread communication by using Condition Object:

- ✓ Condition is the more advanced version of Event object for interthread communication.
- ✓ A condition represents some kind of state change in the application like producing item or consuming item.
- ✓ Threads can wait for that condition and threads can be notified once condition happened
- ✓ i.e Condition object allows one or more threads to wait until notified by another thread.
- ✓ Condition is always associated with a lock (ReentrantLock).
- ✓ A condition has acquire() and release() methods that call the corresponding methods of the associated lock.
- ✓ We can create Condition object as follows,
  - `condition = threading.Condition()`

## Methods of Condition:

### 1. `acquire()`

- ✓ To acquire Condition object before producing or consuming items. i.e thread acquiring internal lock.

### 2. `release()`

- ✓ To release Condition object after producing or consuming items. i.e thread releases internal lock

### 3. `wait()`|`wait(time)`

- ✓ To wait until getting Notification or time expired

### 4. `notify()`

- ✓ To give notification for one waiting thread

### 5. `notifyAll()`

- ✓ To give notification for all waiting threads

## Case Study:

- ✓ The producing thread needs to acquire the Condition before producing item to the resource and notifying the consumers.

### #Producer Thread

```
...generate item..  
condition.acquire()...  
add item to the resource...  
condition.notify()#signal that a new item is available(notifyAll())  
condition.release()
```

- ✓ The Consumer must acquire the Condition and then it can consume items from the resource

### #Consumer Thread

```
condition.acquire()  
condition.wait()  
consume item  
condition.release()
```

## Program Name

Interthread communication by using Condition Object  
demo27.py

```
from threading import *  
  
def consume(c):  
    c.acquire()  
    print("Consumer waiting for updation")  
    c.wait()  
    print("Consumer got notification & consuming the item")  
    c.release()  
  
def produce(c):  
    c.acquire()  
    print("Producer Producing Items")  
    print("Producer giving Notification")  
    c.notify()  
    c.release()  
  
c=Condition()  
  
t1=Thread(target=consume, args=(c,))  
t2=Thread(target=produce, args=(c,))  
  
t1.start()
```

```
t2.start()
```

## Output

```
Consumer waiting for updation  
Producer Producing Items  
Producer giving Notification  
Consumer got notification & consuming the item
```

## Program Name

Interthread communication by using Condition Object  
demo28.py

```
from threading import *
import time
import random

items=[]

def produce(c):
    while True:
        c.acquire()
        item=random.randint(1,10)
        print("Producer Producing Item:", item)
        items.append(item)
        print("Producer giving Notification")
        c.notify()
        c.release()
        time.sleep(5)

def consume(c):
    while True:
        c.acquire()
        print("Consumer waiting for updation")
        c.wait()
        print("Consumer consumed the item", items.pop())
        c.release()
        time.sleep(5)

c=Condition()

t1=Thread(target=consume, args=(c,))
t2=Thread(target=produce, args=(c,))

t1.start()
t2.start()
```

## Output

```
Consumer waiting for updation  
Producer Producing Item: 1  
Producer giving Notification  
Consumer consumed the item 1
```

```
Consumer waiting for updation
Producer Producing Item: 4
Producer giving Notification
Consumer consumed the item 4
Producer Producing Item: 6
Producer giving Notification
Consumer waiting for updation
Producer Producing Item: 6
Producer giving Notification
Consumer consumed the item 6
```

- ✓ In the above program Consumer thread expecting updation and hence it is responsible to call wait() method on Condition object.
- ✓ Producer thread performing updation and hence it is responsible to call notify() or notifyAll() on Condition object.

## Interthread communication by using Queue:

- ✓ Queues Concept is the most enhanced Mechanism for interthread communication and to share data between threads.
- ✓ Queue internally has Condition and that Condition has Lock. Hence whenever we are using Queue we are not required to worry about Synchronization.
- ✓ If we want to use Queues first we should import queue module.

### Syntax

```
import queue
```

- ✓ We can create Queue object as follows

```
q = queue.Queue()
```

## Important Methods of Queue:

1. put(): Put an item into the queue.
  2. get(): Remove and return an item from the queue.
- ✓ Producer Thread uses put() method to insert data in the queue.
  - ✓ Internally this method has logic to acquire the lock before inserting data into queue.
  - ✓ After inserting data lock will be released automatically.
  - ✓ put() method also checks whether the queue is full or not and if queue is full then the Producer thread will enter in to waiting state by calling wait() method internally.
  - ✓ Consumer Thread uses get() method to remove and get data from the queue.
  - ✓ Internally this method has logic to acquire the lock before removing data from the queue.

- ✓ Once removal completed then the lock will be released automatically.
- ✓ If the queue is empty then consumer thread will enter into waiting state by calling wait() method internally.
- ✓ Once queue updated with data then the thread will be notified automatically.

## Make a note

- ✓ The queue module takes care of locking for us which is a great advantage.

### Program Name

Interthread communication by using Queue  
demo29.py

```
from threading import *
import time
import random
import queue

def produce(q):
    while True:
        item=random.randint(1,100)
        print("Producer Producing Item:", item)
        q.put(item)
        print("Producer giving Notification")
        time.sleep(5)

def consume(q):
    while True:
        print("Consumer waiting for updation")
        print("Consumer consumed the item:", q.get())
        time.sleep(5)

q=queue.Queue()

t1=Thread(target=consume, args=(q,))
t2=Thread(target=produce, args=(q,))

t1.start()
t2.start()
```

### Output

```
Consumer waiting for updation
Producer Producing Item: 45
Producer giving Notification
Consumer consumed the item: 45
Consumer waiting for updation
Producer Producing Item: 88
Producer giving Notification
Consumer consumed the item: 88
Producer Producing Item: 95
Consumer waiting for updation
Producer giving Notification
```

Consumer consumed the item: 95

## Types of Queues:

Python Supports 3 Types of Queues.

1. FIFO Queue
2. LIFO Queue
3. Proprietary Queue

### 1. FIFO Queue:

#### Syntax

```
q = queue.Queue()
```

- ✓ This is Default Behaviour. In which order we put items in the queue, in the same order the items will come out (FIFO-First In First Out).

Program Name	FIFO Queue demo30.py
	<pre>import queue  q=queue.Queue()  q.put(10) q.put(5) q.put(20) q.put(15)  while not q.empty():     print(q.get(),end=' ')</pre>
Output	10 5 20 15

### 2. LIFO Queue:

- ✓ The removal will be happened in the reverse order of insertion(Last In First Out)

Program Name	LIFO Queue demo31.py
	<pre>import queue  q=queue.LifoQueue()</pre>

```
q.put(10)
q.put(5)
q.put(20)
q.put(15)

while not q.empty():
    print(q.get(),end=' ')
```

## Output

15 20 5 10

### 3. Priority Queue:

- ✓ The elements will be inserted based on some priority order.

Program Name      Priority Queue  
demo32.py

```
import queue

q=queue.PriorityQueue()

q.put(10)
q.put(5)
q.put(20)
q.put(15)

while not q.empty():
    print(q.get(),end=' ')
```

## Output

5 10 15 20

### Make a note

- ✓ If the data is non-numeric, then we have to provide our data in the form of tuple.

(x, y)

- ✓ x is priority
- ✓ y is our element

Program Name      Priority Queue  
demo33.py

```
import queue  
  
q=queue.PriorityQueue()  
  
q.put((1,"AAA"))  
q.put((3,"CCC"))  
q.put((2,"BBB"))  
q.put((4,"DDD"))  
  
while not q.empty():  
    print(q.get()[1],end=' ')
```

## Output

AAA BBB CCC DDD

## Good Programming Practices with usage of Locks:

### Case-1:

- ✓ It is highly recommended to write code of releasing locks inside finally block.
- ✓ The advantage is lock will be released always whether exception raised or not raised and whether handled or not handled.

```
l=threading.Lock()  
l.acquire()  
  
try:  
    perform required safe operations  
finally:  
    l.release()
```

<b>Program Name</b>	Using Lock concept demo34.py
	<pre>from threading import * import time  l=Lock()  def wish(name):     l.acquire()     try:         for i in range(5):             print("Hi:", end="")             time.sleep(2)             print(name)</pre>

```
finally:  
    l.release()  
  
t1=Thread(target=wish, args=("Nireekshan",))  
t2=Thread(target=wish, args=("Ramesh",))  
t3=Thread(target=wish, args=("Arjun",))  
  
t1.start()  
t2.start()  
t3.start()
```

## Output

```
Hi:Nireekshan  
Hi:Nireekshan  
Hi:Nireekshan  
Hi:Nireekshan  
Hi:Nireekshan  
Hi:Nireekshan  
Hi:Ramesh  
Hi:Ramesh  
Hi:Ramesh  
Hi:Ramesh  
Hi:Ramesh  
Hi:Ramesh  
Hi:Ramesh  
Hi:Arjun  
Hi:Arjun  
Hi:Arjun  
Hi:Arjun  
Hi:Arjun
```

## Case-2:

- ✓ It is highly recommended to acquire lock by using **with** statement.
- ✓ The main advantage of with statement is the lock will be released automatically once control reaches end of with block and we are not required to release explicitly.

This is exactly same as usage of with statement for files.

### Example for File:

```
with open('demo.txt', 'w') as f:  
    f.write("Hello...")
```

### Example for Lock:

```
lock=threading.Lock()  
with lock:  
    perform required safe operations  
lock will be released automatically
```

<p>Program Name</p>	by using with concept demo35.py
	<pre>from threading import * import time  lock=Lock()  def wish(name):     with lock:         for i in range(10):             print("Hi:", end="")             time.sleep(2)             print(name)  t1=Thread(target=wish, args=("Nireekshan",)) t2=Thread(target=wish, args=("Arjun",)) t3=Thread(target=wish, args=("Ramesh",))  t1.start() t2.start() t3.start()</pre>

```
Hi:Ramesh  
Hi:Ramesh  
Hi:Ramesh
```

**Question.** What is the advantage of using **with** statement to acquire a lock in threading?

**Answer:** Lock will be released automatically once control reaches end of with block and We are not required to release explicitly.

**with keyword in multithreading**

- ✓ We can use with statement in multithreading for the following cases:

1. Lock
2. RLock
3. Semaphore
4. Condition

## 26. Logging module

### Why Logging is required?

- ✓ Generally, in real time application Logging is important.
- ✓ When you transfer money, internally application will execute corresponding step.
- ✓ Its good to keep save those steps in log file to monitor if something goes wrong while transferring.
- ✓ When an airplane is flying, black box (flight data recorder) is recording everything.
- ✓ If something goes wrong, people can read the log file and has a chance to figure out what happened.
- ✓ When a program crashes, if there is no logging record then it's more pain to track or understand what's happened.

### Logging

- ✓ It is highly recommended to save or store complete application flow and exceptions information to a file.
- ✓ This process is called logging.

### The main advantages of logging

- ✓ We can use log files while performing debugging if any issue occurs.
- ✓ We can find like, a user how many times logged into the specific application.
- ✓ We can provide statistics like number of requests per day etc.

### Can we use print statement for debugging?

- ✓ I'm sorry for saying, print statement is not a good idea for debugging
- ✓ Anyway, its works better for small scale application.
- ✓ we need to remove the print statements while delivering application to client.
- ✓ For large scale applications print statement will fits

Program Name	using print statement for debugging demo1.py
	<pre>print("Execution started")  print("one") print("first step started") print("two") print("second step started") print("three") print("third step started") print("four") print("fourth step started")  print("Execution finished")</pre>

### Output

```
Execution started
one
first step started
two
second step started
three
third step started
four
fourth step started
Execution finished
```

## logging module

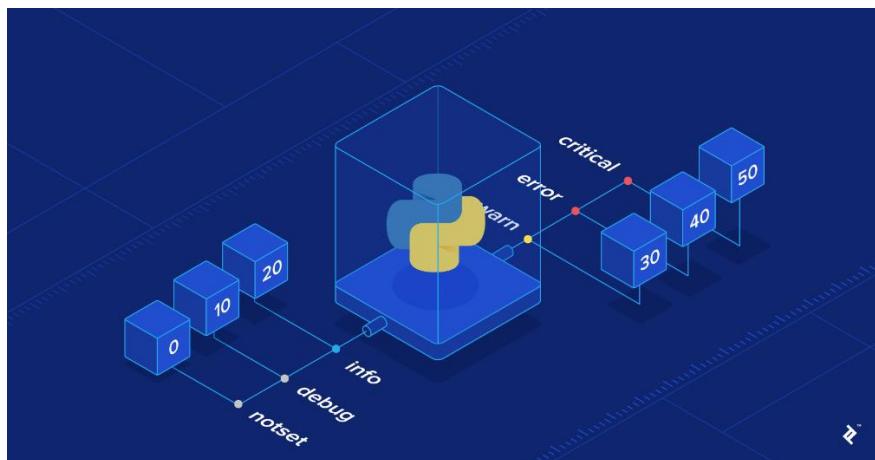
- ✓ To implement logging, Python provides inbuilt module logging.

### Logging Levels:

Level	Numeric value	Description
CRITICAL	50	Represents very serious error needs to high attention
ERROR	40	Represents serious error
WARNING	30	Represents a warning message, some caution is needed
INFO	20	Represents a message with some important information.
DEBUG	10	Represents a message with debugging information
NOTSET	0	Represents that the level is not set

## Make a note

- ✓ By default, while executing Python program only WARNING and higher-level messages will be displayed.



## How to implement Logging

- ✓ To perform logging, first we required to create a file to store messages and we have to specify which level messages required to store.
- ✓ We can do this by using basicConfig() function of logging module.

```
logging.basicConfig(filename='log.txt', level=logging.WARNING)
```

- ✓ The above line will create a file log.txt and we can store either WARNING level or higher-level messages to that file.
- ✓ After creating log file, we can write messages to that file by using the following methods

```
logging.debug(message)
logging.info(message)
logging.warning(message)
logging.error(message)
logging.critical(message)
```

<b>Program Name</b>	logging program demo2.py
	<code>import logging</code>
	<code>logging.basicConfig(filename='demo2log.txt', level=logging.WARNING)</code> <code>print('Logging started')</code>

```
logging.debug('Debug Information')
logging.info('info Information')
logging.warning('warning Information')
logging.error('error Information')
logging.critical('critical Information')
print('Logging end')
```

## Output

```
Logging started
Logging end
```

## demo2log.txt

```
WARNING:root:warning Information
ERROR:root:error Information
CRITICAL:root:critical Information
```

## level=logging.WARNING

- ✓ In the above program we set as, level=logging.WARNING, then it prints only WARNING and higher-level messages will be written to the log file.

## level=logging.DEBUG

- ✓ If we set level as DEBUG then all messages will be written to the log file.

<b>Program Name</b>	logging program demo3.py
	<pre>import logging  logging.basicConfig(filename=' demo3log.txt',level=logging.DEBUG)  print('Logging started')  logging.debug('Debug Information') logging.info('info Information') logging.warning('warning Information') logging.error('error Information') logging.critical('critical Information')  print('Logging end')</pre>

## Output

```
Logging started
Logging end
```

## demo3log.txt

```
DEBUG:root:Debug Information  
INFO:root:info Information  
WARNING:root:warning Information  
ERROR:root:error Information  
CRITICAL:root:critical Information
```

## How to configure log file in over writing mode:

- ✓ In the above program by default data will be appended to the log file.
  - i.e append is the default mode.
- ✓ Instead of appending if we want to over write data then we have to use filemode property.
- ✓ `logging.basicConfig(filename='log786.txt', level=logging.WARNING)`
  - above piece of code by default appending mode
- ✓ `logging.basicConfig(filename='log786.txt', level=logging.WARNING,filemode='a')`
  - above piece of code explicitly we are specifying as appending mode.
- ✓ `logging.basicConfig(filename='log786.txt', level=logging.WARNING,filemode='w')`
  - above piece of code explicitly we are specifying as over write previous data.

## Make a note

```
logging.basicConfig(filename='log.txt',level=logging.DEBUG)
```

- ✓ If we are not specifying level then the default level is WARNING(30)
- ✓ If we are not specifying file name, then the messages will be printed to the console.

Program Name	logging program <code>demo4.py</code>
	<code>import logging</code>
	<code>logging.basicConfig()</code>
	<code>print('Logging started')</code>
	<code>logging.debug('Debug Information')</code>
	<code>logging.info('info Information')</code>
	<code>logging.warning('warning Information')</code>
	<code>logging.error('error Information')</code>
	<code>logging.critical('critical Information')</code>
	<code>print('Logging end')</code>

## Output

```
Logging starts
WARNING:root:warning Information
ERROR:root:error Information
CRITICAL:root:critical Information
Logging ends
```

## How to Format log messages:

- ✓ By using format keyword-argument, we can format messages.

### 1. To display only level name:

```
logging.basicConfig(format='%(levelname)s')
```

Program Name      logging program  
demo5.py

```
import logging

logging.basicConfig(format='%(levelname)s')

print('Logging started')

logging.debug('Debug Information')
logging.info('info Information')
logging.warning('warning Information')
logging.error('error Information')
logging.critical('critical Information')

print('Logging end')
```

Output  
Logging started  
WARNING  
ERROR  
CRITICAL  
Logging end

### 2. To displaylevelname and message:

```
logging.basicConfig(format='%(levelname)s:%(message)s')
```

Program Name	logging program demo6.py
Output	<pre>import logging  logging.basicConfig(format='%(levelname)s:%(message)s')  print('Logging started')  logging.debug('Debug Information') logging.info('info Information') logging.warning('warning Information') logging.error('error Information') logging.critical('critical Information')  print('Logging end')</pre> <p>Logging started WARNING:warning Information ERROR:error Information CRITICAL:critical Information Logging end</p>

Add timestamp in the log messages:

```
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s')
```

Program Name	logging program demo7.py
Output	<pre>import logging  logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s')  print('Logging started')  logging.debug('Debug Information') logging.info('info Information') logging.warning('warning Information') logging.error('error Information') logging.critical('critical Information')  print('Logging end')</pre> <p>Logging started 2018-09-12 15:04:45,040:WARNING:warning Information</p>

```
2018-09-12 15:04:45,040:ERROR:error Information  
2018-09-12 15:04:45,040:CRITICAL:critical Information  
Logging end
```

change date and time format:

- ✓ We have to use special keyword argument: datefmt

```
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s',  
datefmt='%d/%m/%Y %I:%M:%S %p')
```

- ✓ datefmt='%d/%m/%Y %I:%M:%S %p' -> case is important

Make a note

- ✓ %I ---> means 12 Hours' time scale
- ✓ %H ---> means 24 Hours' time scale

Program Name

```
logging program  
demo8.py
```

```
import logging
```

```
logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s',  
datefmt='%d/%m/%Y %I:%M:%S %p')  
  
print('Logging started')  
  
logging.debug('Debug Information')  
logging.info('info Information')  
logging.warning('warning Information')  
logging.error('error Information')  
logging.critical('critical Information')  
  
print('Logging end')
```

Output

```
Logging started  
12/09/2018 03:09:32 PM:WARNING:warning Information  
12/09/2018 03:09:32 PM:ERROR:error Information  
12/09/2018 03:09:32 PM:CRITICAL:critical Information  
Logging end
```

**Program Name** logging program demo9.py

```
import logging

logging.basicConfig(format='%(asctime)s:%(levelname)s:%(message)s',
datefmt='%d/%m/%Y %H:%M:%S')

print('Logging started')

logging.debug('Debug Information')
logging.info('info Information')
logging.warning('warning Information')
logging.error('error Information')
logging.critical('critical Information')

print('Logging end')
```

**Output**

```
Logging started
12/09/2018 15:11:45:WARNING:warning Information
12/09/2018 15:11:45:ERROR:error Information
12/09/2018 15:11:45:CRITICAL:critical Information
Logging end
```

### Writing Python program exceptions to the log file:

- ✓ By using the following function, we can write exception information to the log file.

```
logging.exception(msg)
```

**Program Name** logging program demo10.py

```
import logging

logging.basicConfig(filename=
'demo10log.txt',level=logging.INFO,format='%(asctime)s:%(levelname)s:%(message)s', datefmt='%d/%m/%Y:%M:%S %p')

logging.info('A new Request Came')

try:
    x=int(input('Enter First Number:'))
    y=int(input('Enter Second Number:'))
```

```
print('The Result:', x/y)

except ZeroDivisionError as msg:
    print('cannot divide with zero')
    logging.exception(msg)

except ValueError as msg:
    print('Please provide int values only')
    logging.exception(msg)

logging.info('Request Processing Completed')
```

## Output

```
C:\Users\Nireekshana\Desktop\python logging>py demo10.py
Enter First Number:2
Enter Second Number:4
The Result: 0.5
```

```
C:\Users\Nireekshana\Desktop\python logging>py demo10.py
Enter First Number:1
Enter Second Number:0
cannot divide with zero
```

## demo10log.txt

```
30/10/2018 06:13:40 PM:INFO:A new Request Came
30/10/2018 06:13:43 PM:INFO:Request Processing Completed
30/10/2018 06:13:52 PM:INFO:A new Request Came
30/10/2018 06:13:55 PM:ERROR:division by zero
Traceback (most recent call last):
  File "demo10.py", line 11, in <module>
    print('The Result:', x/y)
ZeroDivisionError: division by zero
30/10/2018 06:13:55 PM:INFO:Request Processing Completed
```

## Problems with root logger:

- ✓ If we are not defining our own logger, then by default root logger will be considered.
- ✓ Once we perform basic configuration to root logger then the configurations are fixed, and we cannot change.

## student.log

```
INFO:root:info message from student module
```

- ✓ In the above application the configurations performed in test module won't be reflected, because root logger is already configured in **student** module.

## Need of Our own customized logger:

The problems with root logger are:

- ✓ Once we set basic configuration then that configuration is final, and we cannot change.
- ✓ It will always work for only one handler at a time, either console or file, but not both simultaneously.
- ✓ It is not possible to configure logger with different configurations at different levels.
- ✓ We cannot specify multiple log files for multiple modules/classes/methods.

To overcome these problems, we should go for our own customized loggers

## Advanced logging Module Features:

### Logger

- ✓ Logger is more advanced than basic logging.
- ✓ It is highly recommended to use, and it provides several extra features.

### Steps for Advanced Logging

#### 1. Creation of **Logger object** and set log level

```
logger = logging.getLogger('demologger')
logger.setLevel(logging.INFO)
```

#### 2. Creation of **Handler object** and set log level.

- ✓ There are several types of Handlers like StreamHandler, FileHandler etc

```
consoleHandler = logging.StreamHandler()
consoleHandler.setLevel(logging.INFO)
```

### Make a note

- ✓ If we use StreamHandler then log messages will be printed to console

#### 3. Creation of **Formatter object**

```
formatter = logging.Formatter('%(asctime)s - %(name)s -
    %(levelname)s: %(message)s', datefmt='%d/%m/%Y %I:%M:%S %p')
```

## 4. Add Formatter to Handler

```
consoleHandler.setFormatter(formatter)
```

## 5. Add Handler to Logger

```
logger.addHandler(consoleHandler)
```

## 6. Write messages by using logger object and the following methods

```
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

### Make a note

- ✓ By default, logger will set to WARNING level. But we can set our own level based on our requirement.

```
logger = logging.getLogger('demologger')
logger.setLevel(logging.INFO)
```

- ✓ logger log level by default available to console and file handlers.
- ✓ If we are not satisfied with logger level, then we can set log level explicitly at console level and file levels.

```
consoleHandler = logging.StreamHandler()
consoleHandler.setLevel(logging.WARNING)

fileHandler=logging.FileHandler('abc.log', mode='a')
fileHandler.setLevel(logging.ERROR)
```

### Make a note

- ✓ console and file log levels should be supported by logger. i.e logger log level should be lower than console and file levels.
- ✓ Otherwise only logger log level will be considered.

## Example

```
logger      ==> DEBUG  
console     ==> INFO
```

Valid and INFO will be considered

```
logger      ==> INFO  
console     ==> DEBUG
```

Invalid and only INFO will be considered to the console.

**Program Name** Logging program demo11.py

```
import logging  
  
class LoggerDemoConsole:  
    def testLog(self):  
        logger = logging.getLogger('demologger')  
        logger.setLevel(logging.INFO)  
        consoleHandler = logging.StreamHandler()  
        consoleHandler.setLevel(logging.INFO)  
        formatter = logging.Formatter('%(asctime)s -  
        %(name)s %(levelname)s: %(message)s',  
        datefmt='%m/%d/%Y %I:%M:%S %p')  
        consoleHandler.setFormatter(formatter)  
        logger.addHandler(consoleHandler)  
        logger.debug('debug message')  
        logger.info('info message')  
        logger.warn('warn message')  
        logger.error('error message')  
        logger.critical('critical message')  
  
    demo = LoggerDemoConsole()  
    demo.testLog()
```

## Output

```
09/12/2018 04:09:06 PM - demologger INFO: info message  
09/12/2018 04:09:06 PM - demologger WARNING: warn message  
09/12/2018 04:09:06 PM - demologger ERROR: error message  
09/12/2018 04:09:06 PM - demologger CRITICAL: critical message
```

## Make a note

- ✓ If we want to use class name as logger name, then we have to create logger object as follows

```
logger = logging.getLogger(LoggerDemoConsole.__name__)
```

In this case output is:

```
09/12/2018 04:16:46 PM - LoggerDemoConsole INFO: info message
09/12/2018 04:16:46 PM - LoggerDemoConsole WARNING: warn message
09/12/2018 04:16:46 PM - LoggerDemoConsole ERROR: error message
09/12/2018 04:16:46 PM - LoggerDemoConsole CRITICAL: critical message
```

<b>Program Name</b>	Logging program demo12.py
	<pre>import logging  class LoggerDemoConsole:     def testLog(self):         logger = logging.getLogger('demologger')         logger.setLevel(logging.INFO)          consoleHandler = logging.StreamHandler()         consoleHandler.setLevel(logging.INFO)         formatter = logging.Formatter('%(asctime)s - %(name)s %(levelname)s: %(message)s',datefmt='%m/%d/%Y %I:%M:%S %p')         consoleHandler.setFormatter(formatter)         logger.addHandler(consoleHandler)         logger.debug('debug message')         logger.info('info message')         logger.warn('warn message')         logger.error('error message')         logger.critical('critical message')         logging.getLogger(LoggerDemoConsole.__name__)      demo = LoggerDemoConsole()     demo.testLog()</pre>
<b>Output</b>	<pre>10/30/2018 06:16:52 PM - LoggerDemoConsole INFO: info message 10/30/2018 06:16:52 PM - LoggerDemoConsole WARNING: warn message 10/30/2018 06:16:52 PM - LoggerDemoConsole ERROR: error message 10/30/2018 06:16:52 PM - LoggerDemoConsole CRITICAL: critical message</pre>

## Demo Program for File Handler:

<b>Program Name</b>	Logging program by using file handler demo13.py
	<pre>import logging  class LoggerDemoConsole:     def testLog(self):         logger = logging.getLogger('demologger')         logger.setLevel(logging.INFO)         fileHandler =             logging.FileHandler('demo13log.txt', mode='a')         fileHandler.setLevel(logging.INFO)         formatter = logging.Formatter('%(asctime)s - %(name)s %(levelname)s: %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')          fileHandler.setFormatter(formatter)          logger.addHandler(fileHandler)         logger.debug('debug message')         logger.info('info message')         logger.warn('warn message')         logger.error('error message')         logger.critical('critical message')  demo = LoggerDemoConsole() demo.testLog() print("done")</pre>
<b>Output</b>	done

DVS Technologies