



SYNERGY INSTITUTE OF ENGINEERING & TECHNOLOGY
Department of Computer Science & Engineering
Academic Session 2026-27
LECTURE NOTE

SINCE 1999
SYNERGY
A Modern Day Gurukul

Name of Faculty	: Mr. SMRUTI RANJAN DASH
Name of Subject	: DATA SCIENCE FOUNDATIONS
Subject Code	: PCAC2002
Subject Credit	: 3
Semester	: III
Year	: 2nd
Course	: B.TECH
Branch	: COMPUTER SCIENCE & ENGINEERING
Admission Batch	: 2025-29

INDEX

TOPIC	PAGE NO
INTRODUCTION DATA, EXPRESSIONS, STATEMENTS	1
Introduction to Python and installation	1
data types: Int	6
float	7
Boolean	8
string	8
List	10
variables	11
expressions	13
statements	16
precedence of operators	17
comments	18
modules	19
functions ----- function and its use	20
flow of execution	21
parameters and arguments	26
CONTROL FLOW, LOOPS	35
Conditionals: Boolean values and operators,	35
conditional (if)	36
alternative (if-else)	37
chained conditional (if-elif-else)	39
Iteration: while, for, break, continue.	41
FUNCTIONS, ARRAYS	55
Fruitful functions: return values	55
parameters	57
local and global scope	59
function composition	62
recursion	63
Strings: string slices	64
immutability	66
string functions and methods	67
string module	72
Python arrays	73
Access the Elements of an Array	75
Array methods	76

IV	LISTS, TUPLES, DICTIONARIES	78
	Lists	78
	list operations	79
	list slices	80
	list methods	81
	list loop	83
	mutability	85
	aliasing	87
	cloning lists	88
	list parameters	89
	list comprehension	90
	Tuples	91
	tuple assignment	94
	tuple as return value	95
	tuple comprehension	96
	Dictionaries	97
	operations and methods	97
	comprehension	102
V	FILES, EXCEPTIONS, MODULES, PACKAGES	103
	Files and exception: text files	103
	reading and writing files	104
	command line arguments	109
	errors and exceptions	112
	handling exceptions	114
	modules (datetime, time, OS , calendar, math module)	121
	Explore packages	134

UNIT – I

INTRODUCTION DATA, EXPRESSIONS, STATEMENTS

Introduction to Python and installation, data types: Int, float, Boolean, string, and list; variables, expressions, statements, precedence of operators, comments; modules, functions--
- function and its use, flow of execution, parameters and arguments.

Introduction to Python and installation:

Python is a widely used general-purpose, high level programming language. It was initially designed by **Guido van Rossum in 1991** and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Python is a programming language that lets you work quickly and integrate systems more efficiently.

There are two major Python versions- **Python 2 and Python 3**.

- On 16 October 2000, Python 2.0 was released with many new features.
- On 3rd December 2008, Python 3.0 was released with more testing and includes new features.

Beginning with Python programming:

1) Finding an Interpreter:

Before we start Python programming, we need to have an interpreter to interpret and run our programs. There are certain online interpreters like <https://ide.geeksforgeeks.org/>, <http://ideone.com/> or <http://codepad.org/> that can be used to start Python without installing an interpreter.

Windows: There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from <http://python.org/downloads/>

2) Writing first program:

```
# Script Begins
```

```
Statement1
```

Statement2

Statement3

Script Ends

Differences between scripting language and programming language:

SCRIPTING LANGUAGE	PROGRAMMING LANGUAGE
A programming language that supports scripts: programs written for a special run-time environment that automate the execution of tasks	A formal language, which comprises a set of instructions used to produce various kinds of output
Execution speed is slow	Compiler-based languages are executed much faster while interpreter-based languages are executed slower
Can be divided into client-side scripting languages and server-side scripting languages	Can be divided into high-level, low-level languages or compiler-based or interpreter-based languages
Easier to learn	Not as easy to learn
Ex: JavaScript, Perl, PHP, Python and Ruby	Ex: C, C++, and Assembly
Mostly used for web development	Used to develop various applications such as desktop, web, mobile, etc.

Why to use Python:

The following are the primary factors to use python in day-to-day life:

1. Python is object-oriented

Structure supports such concepts as polymorphism, operation overloading and multiple inheritance.

2. Indentation

Indentation is one of the greatest feature in python

3. It's free (open source)

Downloading python and installing python is free and easy

4. It's Powerful

- Dynamic typing
- Built-in types and tools
- Library utilities
- Third party utilities (e.g. Numeric, NumPy, sciPy)
- Automatic memory management

5. It's Portable

- Python runs virtually every major platform used today
- As long as you have a compatible python interpreter installed, python programs will run in exactly the same manner, irrespective of platform.

6. It's easy to use and learn

- No intermediate compile
- Python Programs are compiled automatically to an intermediate form called byte code, which the interpreter then reads.
- This gives python the development speed of an interpreter without the performance loss inherent in purely interpreted languages.
- Structure and syntax are pretty intuitive and easy to grasp.

7. Interpreted Language

Python is processed at runtime by python Interpreter

8. Interactive Programming Language

Users can interact with the python interpreter directly for writing the programs

9. Straight forward syntax

The formation of python syntax is simple and straight forward which also makes it popular.

Installation:

There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from <http://python.org/downloads/>

Steps to be followed and remembered:

Step 1: Select Version of Python to Install.

Step 2: Download Python Executable Installer.

Step 3: Run Executable Installer.

Step 4: Verify Python Was Installed On Windows.

Step 5: Verify Pip Was Installed.

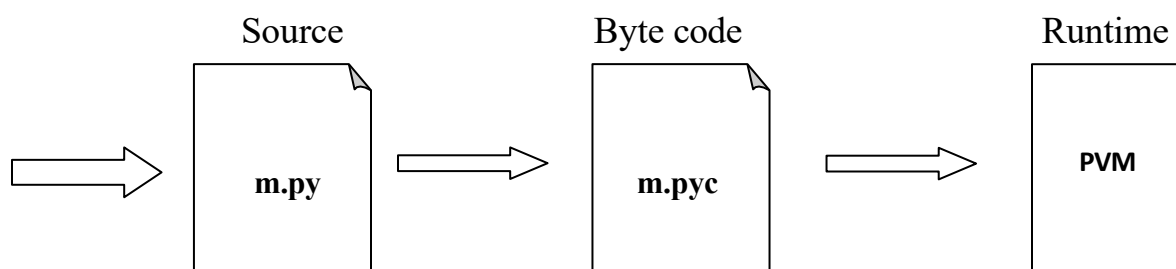
Step 6: Add Python Path to Environment Variables (Optional)



Working with Python

Python Code Execution:

Python's traditional runtime execution model: Source code you type is translated to byte code, which is then run by the Python Virtual Machine (PVM). Your code is automatically compiled, but then it is interpreted.



Source code extension is .py
Byte code extension is .pyc (Compiled python code)

There are two modes for using the Python interpreter:

- Interactive Mode
- Script Mode

Running Python in interactive mode:

Without passing python script file to the interpreter, directly execute code to Python prompt. Once you're inside the python interpreter, then you can start.

```
>>> print("hello world")
```

```
hello world
```

Relevant output is displayed on subsequent lines without the >>> symbol

```
>>> x=[0,1,2]
```

Quantities stored in memory are not displayed by default.

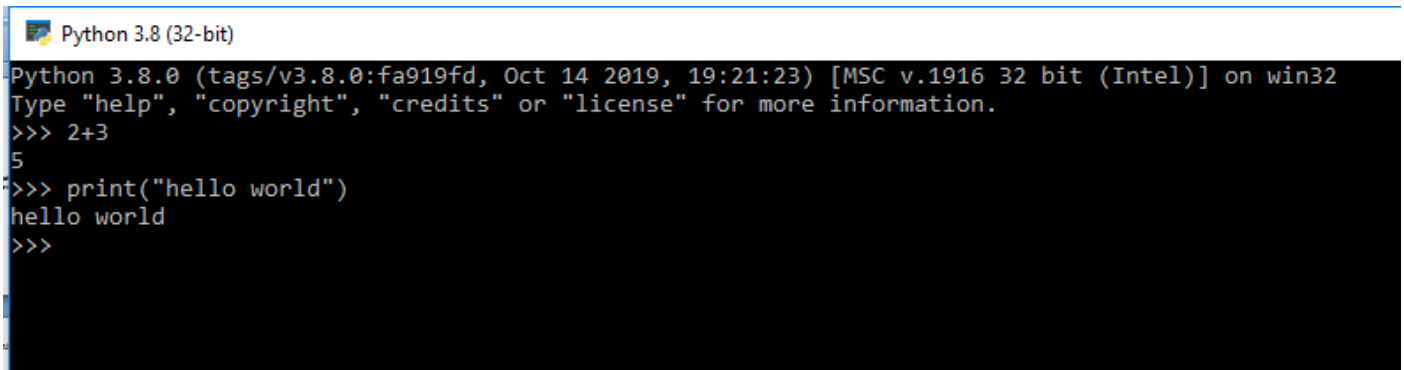
```
>>> x
```

#If a quantity is stored in memory, typing its name will display it.

```
[0, 1, 2]
```

```
>>> 2+3
```

```
5
```

A screenshot of a Python 3.8 (32-bit) interpreter window. The window title is "Python 3.8 (32-bit)". The content shows the interpreter's startup message: "Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32". Below this, the prompt ">>>" is followed by the command "2+3", which results in the output "5". Then, the prompt ">>>" is followed by the command "print('hello world')", which results in the output "hello world". Finally, the prompt ">>>" is shown on a new line without any output yet.

```
Python 3.8 (32-bit)
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> print("hello world")
hello world
>>>
```

The chevron at the beginning of the 1st line, i.e., the symbol >>> is a prompt the python interpreter uses to indicate that it is ready. If the programmer types 2+6, the interpreter replies 8.

Running Python in script mode:

Alternatively, programmers can store Python script source code in a file with the .py extension, and use the interpreter to execute the contents of the file. To execute the script by the interpreter, you have to tell the interpreter the name of the file. For example, if you have a script name MyFile.py and you're working on Unix, to run the script you have to type:

python MyFile.py

Working with the interactive mode is better when Python programmers deal with small pieces of code as you can type and execute them immediately, but when the code is more than 2-4 lines, using the script for coding can help to modify and use the code in future.

Example:

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy>python e1.py
resource open
the no cant be divisibile zero division by zero
resource close
finished
```

Data types:

The data stored in memory can be of many types. For example, a student roll number is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Int:

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

```
>>> print(24656354687654+2)
```

```
24656354687656
```

```
>>> print(20)
```

```
20
```

```
>>> print(0b10)
```

```
2
```

```
>>> print(0B10)
```

```
2
```

```
>>> print(0X20)
```

```
32
```

```
>>> 20
```

```
20
```

```
>>> 0b10
```

```
2
```

```
>>> a=10
```

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

```
10
```

To verify the type of any object in Python, use the type() function:

```
>>> type(10)
```

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

```
>>> a=11
```

```
>>> print(type(a))
```

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

Float:

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Float can also be scientific numbers with an "e" to indicate the power of 10.

```
>>> y=2.8
```

```
>>> y
```

```
2.8
```

```
>>> y=2.8
```

```
>>> print(type(y))
```

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

```
>>> type(.4)
```

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

```
>>> 2.
```

2.0

Example:

```
x = 35e3  
y = 12E4  
z = -87.7e100
```

```
print(type(x))  
print(type(y))  
print(type(z))
```

Output:

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

Boolean:

Objects of Boolean type may have one of two values, True or False:

```
>>> type(True)  
  
<class 'bool'>  
  
>>> type(False)  
  
<class 'bool'>
```

String:

1. Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.
 - 'hello' is the same as "hello".
 - Strings can be output to screen using the print function. **For example: print("hello").**

```
>>> print("mrcet college")  
  
mrcet college  
  
>>> type("mrcet college")  
  
<class 'str'>
```

```
>>> print('mrcet college')
```

```
mrcet college
```

```
>>> " "
```

```
' '
```

If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

```
>>> print("mrcet is an autonomous (') college")
```

```
mrcet is an autonomous (') college
```

```
>>> print('mrcet is an autonomous (") college')
```

```
mrcet is an autonomous (") college
```

Suppressing Special Character:

Specifying a backslash (\) in front of the quote character in a string “escapes” it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

```
>>> print("mrcet is an autonomous (\') college")
```

```
mrcet is an autonomous (') college
```

```
>>> print('mrcet is an autonomous (\") college')
```

```
mrcet is an autonomous (") college
```

The following is a table of escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

```
>>> print('a\
```

```
....b')
```

```
a....b
```

```
>>> print('a\
```

```
b\
```

```
c')
```

abc

```
>>> print('a \n b')
```

a

b

```
>>> print("mrcet \n college")
```

mrcet

college

Escape Sequence	Usual Interpretation of Character(s) After Backslash	“Escaped” Interpretation
\'	Terminates string with single quote opening delimiter	Literal single quote (') character
\"	Terminates string with double quote opening delimiter	Literal double quote (") character
\newline	Terminates input line	Newline is ignored
\\	Introduces escape sequence	Literal backslash (\) character

In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence `\t`:

```
>>> print("a\tb")
```

a b

List:

- It is a general purpose most widely used in data structures
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Ex:

```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
```

```
>>> print(list1)
```

[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]

```
>>> x=list()
```

```
>>> x
```

```
[]
```

```
>>> tuple1=(1,2,3,4)
```

```
>>> x=list(tuple1)
```

```
>>> x
```

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

Variables:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Assigning Values to Variables:

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

For example –

```
a= 100      # An integer assignment
```

```
b = 1000.0    # A floating point
```

```
c = "John"    # A string
```

```
print (a)
```

```
print (b)
```

```
print (c)
```

This produces the following result –

```
100
```

```
1000.0
```

```
John
```

Multiple Assignment:

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

For example :

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example –

```
a,b,c = 1,2,"mrcet"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Output Variables:

The Python print statement is often used to output variables.

Variables do not need to be declared with any particular type and can even change type after they have been set.

```
x = 5          # x is of type int
x = "mrcet "   # x is now of type str
print(x)
```

Output: mrcet

To combine both text and a variable, Python uses the “+” character:

Example

```
x = "awesome"
print("Python is " + x)
```

Output

Python is awesome

You can also use the + character to add a variable to another variable:

Example

```
x = "Python is "
y = "awesome"
z = x + y
print(z)
```

Output:

Python is awesome

Expressions:

An expression is a combination of values, variables, and operators. An expression is evaluated using assignment operator.

Examples: $Y = x + 17$

```
>>> x=10
```

```
>>> z=x+20
```

```
>>> z
```

30

```
>>> x=10
```

```
>>> y=20
```

```
>>> c=x+y
```

```
>>> c
```

```
30
```

A value all by itself is a simple expression, and so is a variable.

```
>>> y=20
```

```
>>> y
```

```
20
```

Python also defines expressions only contain identifiers, literals, and operators. So,

Identifiers: Any name that is used to define a class, function, variable module, or object is an identifier.

Literals: These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.

Operators: In Python you can implement the following operations using the corresponding tokens.

Operator	Token
add	+
subtract	-
multiply	*
Integer Division	/
remainder	%
Binary left shift	<<
Binary right shift	>>
and	&
or	\
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Check equality	==
Check not equal	!=

Some of the python expressions are:

Generator expression:

Syntax: (compute(var) for var in iterable)

```
>>> x = (i for i in 'abc') #tuple comprehension
>>> x
<generator object <genexpr> at 0x033EEC30>

>>> print(x)
<generator object <genexpr> at 0x033EEC30>
```

You might expect this to print as ('a', 'b', 'c') but it prints as <generator object <genexpr> at 0x02AAD710> The result of a tuple comprehension is not a tuple: it is actually a generator. The only thing that you need to know now about a generator now is that you can iterate over it, but ONLY ONCE.

Conditional expression:

Syntax: true_value if Condition else false_value

```
>>> x = "1" if True else "2"

>>> x

'1'
```

Statements:

A statement is an instruction that the Python interpreter can execute. We have normally two basic statements, the assignment statement and the print statement. Some other kinds of statements that are if statements, while statements, and for statements generally called as control flows.

Examples:

An assignment statement creates new variables and gives them values:

```
>>> x=10
```

```
>>> college="mrcet"
```

An print statement is something which is an input from the user, to be printed / displayed on to the screen (or) monitor.

```
>>> print("mrcet colege")
```

```
mrcet college
```

Precedence of Operators:

Operator precedence affects how an expression is evaluated.

For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first multiplies $3*2$ and then adds into 7.

Example 1:

```
>>> 3+4*2
```

```
11
```

Multiplication gets evaluated before the addition operation

```
>>> (10+10)*2
```

```
40
```

Parentheses () overriding the precedence of the arithmetic operators

Example 2:

```
a = 20
```

```
b = 10
```

```
c = 15
```

```
d = 5
```

```
e = 0
```

```
e = (a + b) * c / d    #( 30 * 15 ) / 5
```

```
print("Value of (a + b) * c / d is ", e)
```

```
e = ((a + b) * c) / d    # (30 * 15 ) / 5
```

```
print("Value of ((a + b) * c) / d is ", e)
```

```
e = (a + b) * (c / d);    # (30) * (15/5)
```

```
print("Value of (a + b) * (c / d) is ", e)
```

```
e = a + (b * c) / d;    # 20 + (150/5)
print("Value of a + (b * c) / d is ", e)
```

Output:

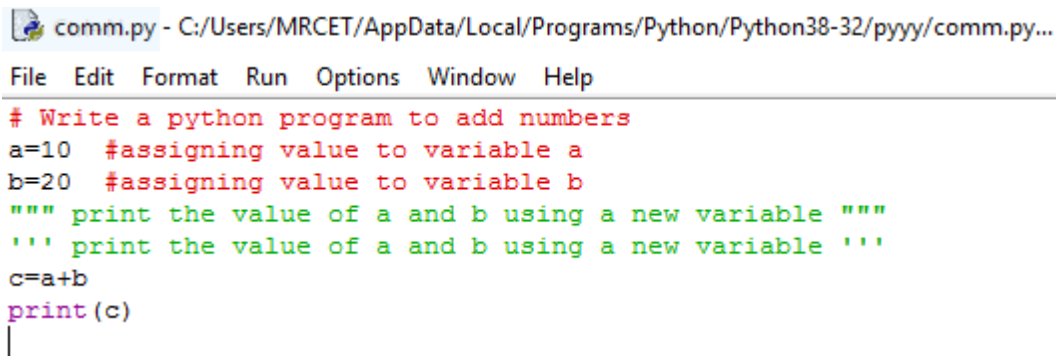
```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/opprec.py
Value of (a + b) * c / d is 90.0
Value of ((a + b) * c) / d is 90.0
Value of (a + b) * (c / d) is 90.0
Value of a + (b * c) / d is 50.0
```

Comments:

Single-line comments begins with a hash(#) symbol and is useful in mentioning that the whole line should be considered as a comment until the end of line.

A Multi line comment is useful when we need to comment on many lines. In python, triple double quote(“ “ “) and single quote(‘ ‘ ‘)are used for multi-line commenting.

Example:



```
comm.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/comm.py...
File Edit Format Run Options Window Help
# Write a python program to add numbers
a=10 #assigning value to variable a
b=20 #assigning value to variable b
''' print the value of a and b using a new variable '''
''' print the value of a and b using a new variable '''
c=a+b
print(c)
|
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/comm.py
```

```
30
```

Modules:

Modules: Python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module. A module in Python provides us the flexibility to organize the code in a logical way. To use the functionality of one module into another, we must have to **import** the specific module.

Syntax:

```
import <module-name>
```

Every module has its own functions, those can be accessed with . (dot)

Note: In python we have help ()

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

Some of the modules like os, date, and calendar so on.....

```
>>> import sys
```

```
>>> print(sys.version)
```

```
3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)]
```

```
>>> print(sys.version_info)
```

```
sys.version_info(major=3, minor=8, micro=0, releaselevel='final', serial=0)
```

```
>>> print(calendar.month(2021,5))
```

```
    May 2021
Mo Tu We Th Fr Sa Su
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

```
>>> print(calendar.isleap(2020))
```

```
True
```

```
>>> print(calendar.isleap(2017))
```

```
False
```

Functions:

Functions and its use: Function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. It avoids repetition and makes code reusable.

Basically, we can divide functions into the following two types:

1. **Built-in functions** - Functions that are built into Python.

Ex: abs(),all().ascii(),bool().....so on....

```
integer = -20
```

```
print('Absolute value of -20 is:', abs(integer))
```

Output:

Absolute value of -20 is: 20

2. **User-defined functions** - Functions defined by the users themselves.

```
def add_numbers(x,y):  
    sum = x + y  
    return sum
```

```
print("The sum is", add_numbers(5, 20))
```

Output:

The sum is 25

Flow of Execution:

1. The order in which statements are executed is called the flow of execution
2. Execution always begins at the first statement of the program.
3. Statements are executed one at a time, in order, from top to bottom.
4. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
5. Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called.

Example:

#example for flow of execution

```
print("welcome")
for x in range(3):
    print(x)
print("Good morning college")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py

welcome

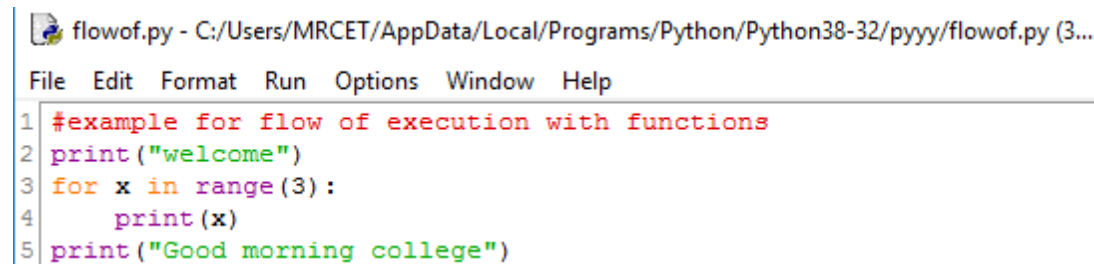
0

1

2

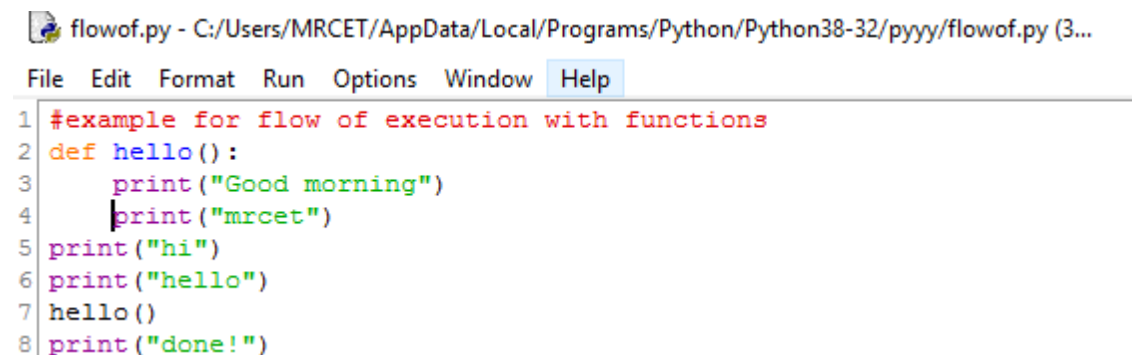
Good morning college

The flow/order of execution is: 2,3,4,3,4,3,4,5



flowof.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py (3...

```
File Edit Format Run Options Window Help
1 #example for flow of execution with functions
2 print("welcome")
3 for x in range(3):
4     print(x)
5 print("Good morning college")
```



flowof.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py (3...

```
File Edit Format Run Options Window Help
1 #example for flow of execution with functions
2 def hello():
3     print("Good morning")
4     print("mrcet")
5 print("hi")
6 print("hello")
7 hello()
8 print("done!")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py

hi

hello

Good morning

mrcet

done!

The flow/order of execution is: 2,5,6,7,2,3,4,7,8

Parameters and arguments:

Parameters are passed during the definition of function while Arguments are passed during the function call.

Example:

#here a and b are parameters

```
def add(a,b): #function definition
    return a+b
```

#12 and 13 are arguments

#function call

```
result=add(12,13)
```

```
print(result)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/paraarg.py

25

There are three types of Python function arguments using which we can call a function.

1. Default Arguments
2. Keyword Arguments
3. Variable-length Arguments

Syntax:

```
def functionname():
```

statements

•
•
•

functionname()

Function definition consists of following components:

1. Keyword **def** indicates the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A **colon (:)** to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

Example:

```
def hf():
```

```
    hello world
```

```
hf()
```

In the above example we are just trying to execute the program by calling the function. So it will not display any error and no output on to the screen but gets executed.

To get the statements of function need to be use print().

#calling function in python:

```
def hf():
```

```
    print("hello world")
```

```
hf()
```

Output:

```
hello world
```

```
-----
```

```
def hf():  
    print("hw")  
    print("gh kfjg 66666")
```

```
hf()
```

```
hf()
```

```
hf()
```

Output:

```
hw  
gh kfjg 66666  
hw  
gh kfjg 66666  
hw  
gh kfjg 66666
```

```
def add(x,y):
```

```
    c=x+y
```

```
    print(c)
```

```
add(5,4)
```

Output:

```
9
```

```
def add(x,y):
```

```
    c=x+y
```

```
    return c
```

```
print(add(5,4))
```

Output:

```
9
```

```
def add_sub(x,y):  
    c=x+y  
    d=x-y  
    return c,d  
  
print(add_sub(10,5))
```

Output:

(15, 5)

The **return** statement is used to exit a function and go back to the place from where it was called. This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the **None** object.

```
def hf():  
    return "hw"  
  
print(hf())
```

Output:

hw

```
def hf():  
    return "hw"  
  
hf()
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu.py

>>>

```
-----  
def hello_f():  
    return "hellocollege"  
print(hello_f().upper())
```

Output:

HELLOCOLLEGE

Passing Arguments

```
def hello(wish):  
    return '{}'.format(wish)  
print(hello("mrcet"))
```

Output:

mrcet

Here, the function wish() has two parameters. Since, we have called this function with two arguments, it runs smoothly and we do not get any error. If we call it with different number of arguments, the interpreter will give errors.

```
def wish(name,msg):  
    """This function greets to  
    the person with the provided message"""  
    print("Hello",name + ' ' + msg)  
wish("MRCET","Good morning!")
```

Output:

Hello MRCET Good morning!

Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> wish("MRCET")  # only one argument
TypeError: wish() missing 1 required positional argument: 'msg'
>>> wish()  # no arguments
TypeError: wish() missing 2 required positional arguments: 'name' and 'msg'
```

```
-----

def hello(wish,hello):

    return "hi" '{}{}'.format(wish,hello)

print(hello("mrcet","college"))
```

Output:

```
himrcet,college
```

#Keyword Arguments

When we call a function with some values, these values get assigned to the arguments according to their position.

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed.

(Or)

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called **keyword arguments** - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two *advantages* - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters which we want, provided that the other parameters have default argument values.

```
def func(a, b=5, c=10):
    print 'a is', a, 'and b is', b, 'and c is', c
```

```
func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output:

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

Note:

The function named func has one parameter without default argument values, followed by two parameters with default argument values.

In the first usage, func(3, 7), the parameter a gets the value 3, the parameter b gets the value 5 and c gets the default value of 10.

In the second usage func(25, c=24), the variable a gets the value of 25 due to the position of the argument. Then, the parameter c gets the value of 24 due to naming i.e. keyword arguments. The variable b gets the default value of 5.

In the third usage func(c=50, a=100), we use keyword arguments completely to specify the values. Notice, that we are specifying value for parameter c before that for a even though a is defined before c in the function definition.

For example: if you define the function like below

```
def func(b=5, c=10,a): # shows error : non-default argument follows default argument
```

```
-----
```

```
def print_name(name1, name2):
```

```
    """ This function prints the name """
```

```
    print (name1 + " and " + name2 + " are friends")
```

```
#calling the function
```

```
print_name(name2 = 'A',name1 = 'B')
```

Output:

B and A are friends

#Default Arguments

Function arguments can have default values in Python.

We can provide a default value to an argument by using the assignment operator (=)

```
def hello(wish,name='you'):
    return '{} {}'.format(wish,name)

print(hello("good morning"))
```

Output:

good morning,you

```
def hello(wish,name='you'):
    return '{} {}'.format(wish,name)    //print(wish + ' ' + name)

print(hello("good morning","nirosha")) // hello("good morning","nirosha")
```

Output:

good morning,nirosha // good morning nirosha

Note: Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

```
def hello(name='you', wish):
```

Syntax Error: non-default argument follows default argument

```
def sum(a=4, b=2): #2 is supplied as default argument
```

```
""" This function will print sum of two numbers
    if the arguments are not supplied
    it will add the default value """
```

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

```
sum(1,2) #calling with arguments
```

```
sum( ) #calling without arguments
```

Output:

3

6

Variable-length arguments

Sometimes you may need more arguments to process function then you mentioned in the definition. If we don't know in advance about the arguments needed in function, we can use variable-length arguments also called arbitrary arguments.

For this an asterisk (*) is placed before a parameter in function definition which can hold non-keyworded variable-length arguments and a double asterisk (**) is placed before a parameter in function which can hold keyworded variable-length arguments.

If we use one asterisk (*) like *var, then all the positional arguments from that point till the end are collected as a tuple called 'var' and if we use two asterisks (**) before a variable like **var, then all the positional arguments from that point till the end are collected as a dictionary called 'var'.

```
def wish(*names):
```

```
    """This function greets all
    the person in the names tuple."""
```

```
    # names is a tuple with arguments
    for name in names:
        print("Hello",name)
```

```
wish("MRCET","CSE","SIR","MADAM")
```

Output:

Hello MRCET
Hello CSE
Hello SIR
Hello MADAM

#Program to find area of a circle using function use single return value function with argument.

```
pi=3.14
def areaOfCircle(r):

    return pi*r*r
r=int(input("Enter radius of circle"))

print(areaOfCircle(r))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ful.py
Enter radius of circle 3
28.259999999999998
```

#Program to write sum different product and using arguments with return value function.

```
def calculete(a,b):

    total=a+b

    diff=a-b

    prod=a*b

    div=a/b

    mod=a%b
```

```

    return total,diff,prod,div,mod

a=int(input("Enter a value"))
b=int(input("Enter b value"))

#function call
s,d,p,q,m = calculate(a,b)

print("Sum= ",s,"diff= ",d,"mul= ",p,"div= ",q,"mod= ",m)

#print("diff= ",d)

#print("mul= ",p)

#print("div= ",q)

#print("mod= ",m)

```

Output:

```

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
Enter a value 5
Enter b value 6
Sum= 11 diff= -1 mul= 30 div= 0.8333333333333334 mod= 5

```

#program to find biggest of two numbers using functions.

```

def biggest(a,b):
    if a>b :
        return a
    else :
        return b

a=int(input("Enter a value"))
b=int(input("Enter b value"))
#function call
big= biggest(a,b)
print("big number= ",big)

```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

Enter a value 5

Enter b value-2

big number= 5

#program to find biggest of two numbers using functions. (nested if)

```
def biggest(a,b,c):
```

```
    if a>b :
```

```
        if a>c :
```

```
            return a
```

```
        else :
```

```
            return c
```

```
    else :
```

```
        if b>c :
```

```
            return b
```

```
        else :
```

```
            return c
```

```
a=int(input("Enter a value"))
```

```
b=int(input("Enter b value"))
```

```
c=int(input("Enter c value"))
```

```
#function call
```

```
big=      biggest(a,b,c)
```

```
print("big number= ",big)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

Enter a value 5

Enter b value -6

Enter c value 7

big number= 7

#Writer a program to read one subject mark and print pass or fail use single return values function with argument.

```
def result(a):
```

```
    if a>40:
```

```
        return "pass"
```

```
else:
    return "fail"
a=int(input("Enter one subject marks"))

print(result(a))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
Enter one subject marks 35
fail
```

#Write a program to display mrcet cse dept 10 times on the screen. (while loop)

```
def usingFunctions():
    count =0
    while count<10:
        print("mrcet cse dept",count)
        count=count+1

usingFunctions()
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
mrcet cse dept 0
mrcet cse dept 1
mrcet cse dept 2
mrcet cse dept 3
mrcet cse dept 4
mrcet cse dept 5
mrcet cse dept 6
mrcet cse dept 7
mrcet cse dept 8
mrcet cse dept 9
```

UNIT – II

CONTROL FLOW, LOOPS

Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: while, for, break, continue.

Control Flow, Loops:

Boolean Values and Operators:

A boolean expression is an expression that is either true or false. The following examples use the operator `==`, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 5 == 5
```

```
True
```

```
>>> 5 == 6
```

```
False
```

True and False are special values that belong to the type `bool`; they are not strings:

```
>>> type(True)
```

```
<class 'bool'>
```

```
>>> type(False)
```

```
<class 'bool'>
```

The `==` operator is one of the relational operators; the others are: `x != y` # x is not equal to y

`x > y` # x is greater than y `x < y` # x is less than y

`x >= y` # x is greater than or equal to y `x <= y` # x is less than or equal to y

Note:

All expressions involving relational and logical operators will evaluate to either true or false

Conditional (if):

The if statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax:

if expression:

 statement(s)

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

if Statement Flowchart:

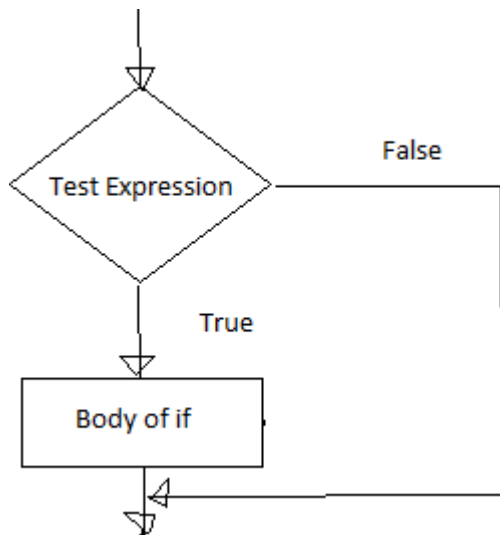


Fig: Operation of if statement

Example: Python if Statement

```
a = 3
if a > 2:
    print(a, "is greater")
print("done")
```

```
a = -1
if a < 0:
    print(a, "a is smaller")
print("Finish")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/if1.py

3 is greater

done

-1 a is smaller

Finish

a=10

if a>9:

 print("A is Greater than 9")

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/if2.py

A is Greater than 9

Alternative if (If-Else):

An else statement can be combined with an if statement. An else statement contains the block of code (false block) that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

The else statement is an optional statement and there could be at most only one else Statement following if.

Syntax of if - else :

 if test expression:

 Body of if stmts

 else:

 Body of else stmts

If - else Flowchart :

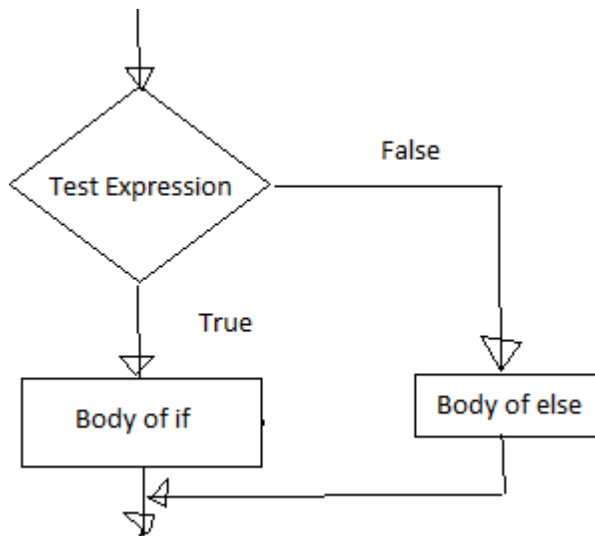


Fig: Operation of if – else statement

Example of if - else:

```
a=int(input('enter the number'))
if a>5:
    print("a is greater")
else:
    print("a is smaller than the input given")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py

enter the number 2

a is smaller than the input given

```
a=10
```

```
b=20
```

```
if a>b:
```

```
    print("A is Greater than B")
```

```
else:
```

```
    print("B is Greater than A")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/if2.py

B is Greater than A

Chained Conditional: (If-elif-else):

The elif statement allows us to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE. Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

Syntax of if – elif - else :

If test expression:

 Body of if stmts

elif test expression:

 Body of elif stmts

else:

 Body of else stmts

Flowchart of if – elif - else:

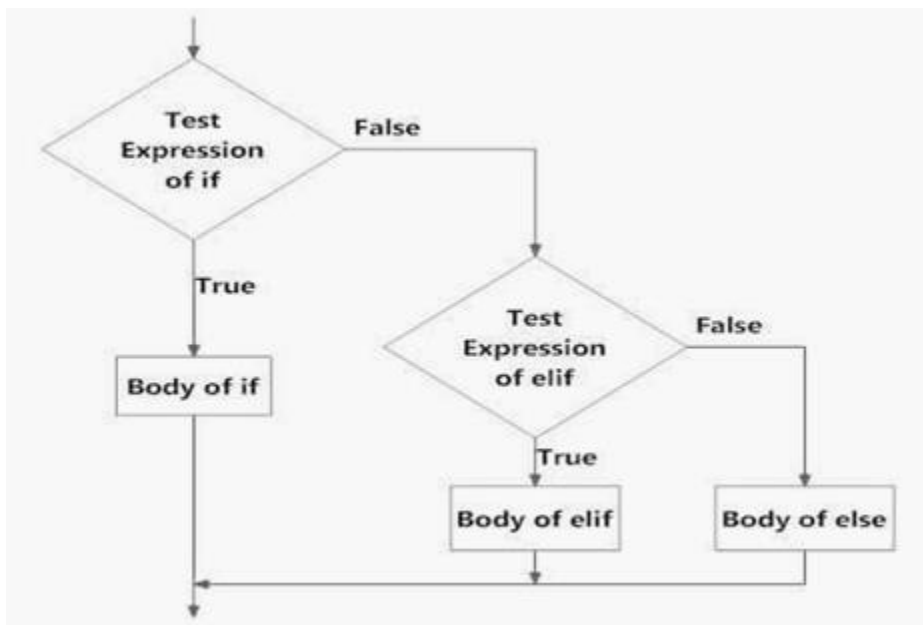


Fig: Operation of if – elif - else statement

Example of if - elif – else:

```
a=int(input('enter the number'))
b=int(input('enter the number'))
c=int(input('enter the number'))
if a>b:
```

```
    print("a is greater")
elif b>c:
    print("b is greater")
else:
    print("c is greater")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py

enter the number5

enter the number2

enter the number9

a is greater

>>>

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py

enter the number2

enter the number5

enter the number9

c is greater

```
var = 100
if var == 200:
    print("1 - Got a true expression value")
    print(var)
elif var == 150:
    print("2 - Got a true expression value")
    print(var)
elif var == 100:
    print("3 - Got a true expression value")
    print(var)
else:
    print("4 - Got a false expression value")
    print(var)
print("Good bye!")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelif.py

3 - Got a true expression value

100

Good bye!

Iteration:

A loop statement allows us to execute a statement or group of statements multiple times as long as the condition is true. Repeated execution of a set of statements with the help of loops is called iteration.

Loops statements are used when we need to run same code again and again, each time with a different value.

Statements:

In Python Iteration (Loops) statements are of three types:

1. While Loop
2. For Loop
3. Nested For Loops

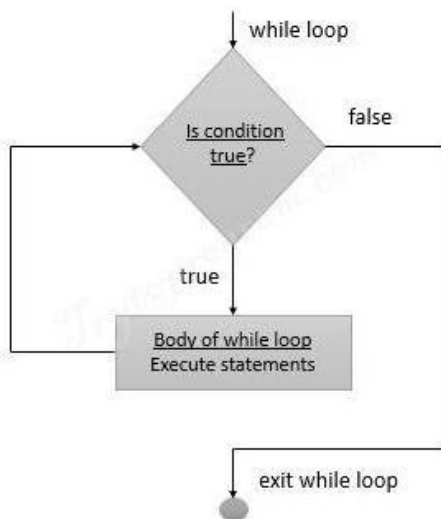
While loop:

- Loops are either infinite or conditional. Python while loop keeps reiterating a block of code defined inside it until the desired condition is met.
- The while loop contains a boolean expression and the code inside the loop is repeatedly executed as long as the boolean expression is true.
- The statements that are executed inside while can be a single line of code or a block of multiple statements.

Syntax:

```
while(expression):  
    Statement(s)
```

Flowchart:



Example Programs:

```
1. -----  
   i=1  
   while i<=6:  
       print("Mrcet college")  
       i=i+1
```

output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh1.py

Mrcet college

Mrcet college

Mrcet college

Mrcet college

Mrcet college

Mrcet college

```
2. -----  
   i=1  
  
   while i<=3:  
       print("MRCET",end=" ")  
       j=1  
       while j<=1:  
           print("CSE DEPT",end="")  
           j=j+1  
       i=i+1  
       print()
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh2.py

MRCET CSE DEPT

MRCET CSE DEPT

MRCET CSE DEPT

```
3. -----  
  
   i=1
```

```

j=1
while i<=3:
    print("MRCET",end=" ")

    while j<=1:
        print("CSE DEPT",end="")
        j=j+1
    i=i+1
    print()

```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh3.py

MRCET CSE DEPT
MRCET
MRCET

4. -----

```

i = 1
while (i < 10):
    print (i)
    i = i+1

```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh4.py

1
2
3
4
5
6
7
8
9

2. -----

```

a = 1
b = 1
while (a<10):
    print ('Iteration',a)
    a = a + 1
    b = b + 1

```

```

    if (b == 4):
        break
print ('While loop terminated')

```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh5.py

Iteration 1

Iteration 2

Iteration 3

While loop terminated

```
count = 0
```

```
while (count < 9):
```

```
    print("The count is:", count)
```

```
    count = count + 1
```

```
print("Good bye!")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh.py =

The count is: 0

The count is: 1

The count is: 2

The count is: 3

The count is: 4

The count is: 5

The count is: 6

The count is: 7

The count is: 8

Good bye!

For loop:

Python **for loop** is used for repeated execution of a group of statements for the desired number of times. It iterates over the items of lists, tuples, strings, the dictionaries and other iterable objects

Syntax: for var in sequence:

Statement(s)

Holds the value of item
in sequence in each iteration

A sequence of values assigned to var in each iteration

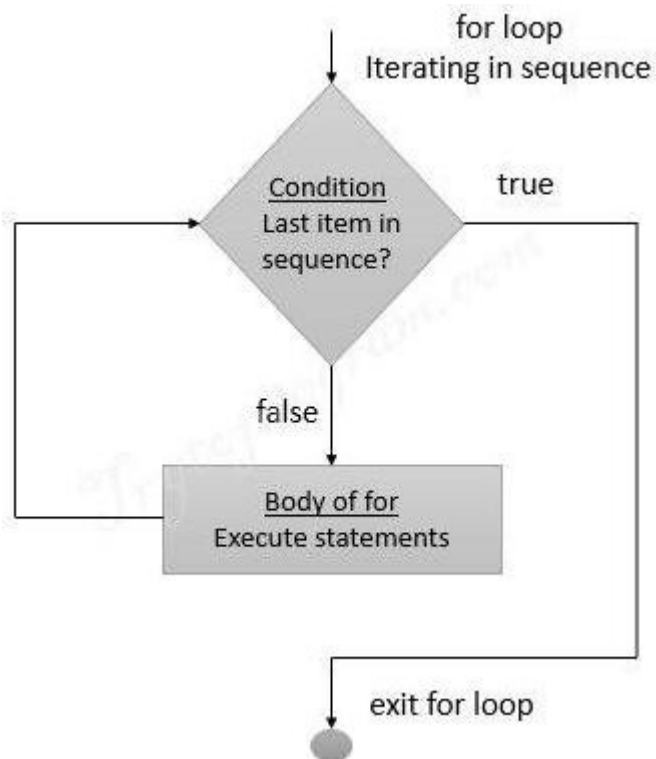
Sample Program:

```
numbers = [1, 2, 4, 6, 11, 20]
seq=0
for val in numbers:
    seq=val*val
    print(seq)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/fr.py
1
4
16
36
121
400
```

Flowchart:



Iterating over a list:

```
#list of items
list = ['M','R','C','E','T']
i = 1

#Iterating over the list
for item in list:
    print ('college ',i,' is ',item)
    i = i+1
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/lis.py
college 1 is M
college 2 is R
college 3 is C
college 4 is E
college 5 is T
```

Iterating over a Tuple:

```
tuple = (2,3,5,7)
print ('These are the first four prime numbers ')
#Iterating over the tuple
for a in tuple:
    print (a)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fr3.py
These are the first four prime numbers
2
3
5
7
```

Iterating over a dictionary:

```
#creating a dictionary
college = {"ces":"block1","it":"block2","ece":"block3"}

#Iterating over the dictionary to print keys
print ('Keys are:')
```

```
for keys in college:  
    print (keys)
```

```
#Iterating over the dictionary to print values  
print ('Values are:')  
for blocks in college.values():  
    print(blocks)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/dic.py

Keys are:

ces

it

ece

Values are:

block1

block2

block3

Iterating over a String:

```
#declare a string to iterate over  
college = 'MRCET'
```

```
#Iterating over the string  
for name in college:  
    print (name)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/strr.py

M

R

C

E

T

Nested For loop:

When one Loop defined within another Loop is called Nested Loops.

Syntax:

```
for val in sequence:
```

```
    for val in sequence:
```

statements

statements

Example 1 of Nested For Loops (Pattern Programs)

```
for i in range(1,6):  
    for j in range(0,i):  
        print(i, end=" ")  
    print("")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/nesforr.py

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

Example 2 of Nested For Loops (Pattern Programs)

```
for i in range(1,6):  
    for j in range(5,i-1,-1):  
        print(i, end=" ")  
    print("")
```

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/nesforr.py

Output:

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

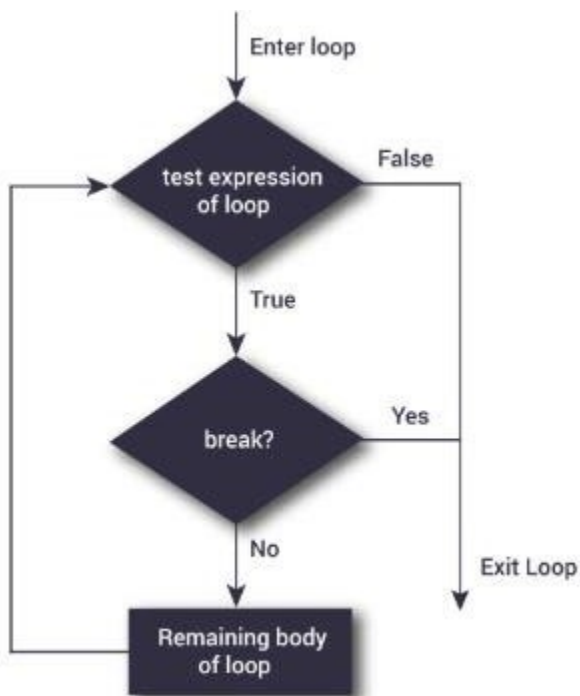
Break and continue:

In Python, **break** and **continue** statements can alter the flow of a normal loop. Sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. The break and continue statements are used in these cases.

Break:

The break statement terminates the loop containing it and control of the program flows to the statement immediately after the body of the loop. If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

Flowchart:



The following shows the working of break statement in for and while loop:

for var in sequence:

 # code inside for loop

 If condition:

 break (if break condition satisfies it jumps to outside loop)

 # code inside for loop

code outside for loop

while test expression

 # code inside while loop

 If condition:

 break (if break condition satisfies it jumps to outside loop)

 # code inside while loop

code outside while loop

Example:

```
for val in "MRCET COLLEGE":
```

```
    if val == " ":
```

```
        break
```

```
    print(val)
```

```
print("The end")
```

Output:

M

R

C

E

T

The end

Program to display all the elements before number 88

```
for num in [11, 9, 88, 10, 90, 3, 19]:
```

```
    print(num)
```

```
    if(num==88):
```

```
        print("The number 88 is found")
```

```
        print("Terminating the loop")
```

```
        break
```

Output:

11

9

88

The number 88 is found

Terminating the loop

```
#-----  
for letter in "Python": # First Example  
    if letter == "h":  
        break  
    print("Current Letter :", letter)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/br.py =

Current Letter : P

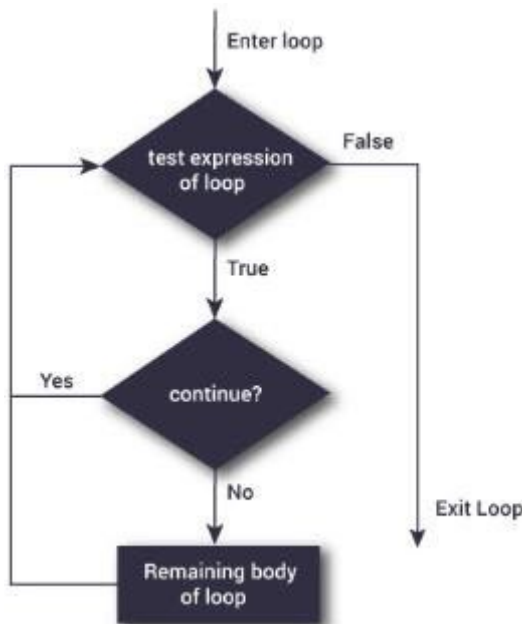
Current Letter : y

Current Letter : t

Continue:

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Flowchart:



The following shows the working of break statement in for and while loop:

for var in sequence:

 # code inside for loop

 If condition:

 continue (if break condition satisfies it jumps to outside loop)

 # code inside for loop

code outside for loop

while test expression

 # code inside while loop

 If condition:

 continue (if break condition satisfies it jumps to outside loop)

 # code inside while loop

code outside while loop

Example:

Program to show the use of continue statement inside loops

```
for val in "string":
```

```
    if val == "i":
```

```
        continue
```

```
    print(val)
```

```
print("The end")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/cont.py

s

t

r

n

g

The end

program to display only odd numbers

```
for num in [20, 11, 9, 66, 4, 89, 44]:
```

```
# Skipping the iteration when number is even
if num%2 == 0:
    continue
# This statement will be skipped for all even numbers
print(num)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/cont2.py

11

9

89

```
#-----
```

```
for letter in "Python": # First Example
    if letter == "h":
        continue
    print("Current Letter :", letter)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/con1.py

Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : o

Current Letter : n

Pass:

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored.

pass is just a placeholder for functionality to be added later.

Example:

```
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fl.y.py

>>>

Similarly we can also write,

```
def f(arg): pass    # a function that does nothing (yet)
```

```
class C: pass      # a class with no methods (yet)
```

UNIT – III

FUNCTIONS, ARRAYS

Fruitful functions: return values, parameters, local and global scope, function composition, recursion; Strings: string slices, immutability, string functions and methods, string module; Python arrays, Access the Elements of an Array, array methods.

Functions, Arrays:

Fruitful functions:

We write functions that return values, which we will call fruitful functions. We have seen the return statement before, but in a fruitful function the return statement includes a return value. This statement means: "Return immediately from this function and use the following expression as a return value."

(or)

Any function that returns a value is called Fruitful function. A Function that does not return a value is called a void function

Return values:

The Keyword return is used to return back the value to the called function.

returns the area of a circle with the given radius:

```
def area(radius):  
    temp = 3.14 * radius**2  
    return temp  
print(area(4))
```

(or)

```
def area(radius):  
    return 3.14 * radius**2  
print(area(2))
```

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
def absolute_value(x):  
    if x < 0:
```

```
    return -x
else:
    return x
```

Since these return statements are in an alternative conditional, only one will be executed.

As soon as a return statement executes, the function terminates without executing any subsequent statements. Code that appears after a return statement, or any other place the flow of execution can never reach, is called dead code.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. For example:

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

This function is incorrect because if x happens to be 0, both conditions is true, and the function ends without hitting a return statement. If the flow of execution gets to the end of a function, the return value is None, which is not the absolute value of 0.

```
>>> print absolute_value(0)
None
```

By the way, Python provides a built-in function called abs that computes absolute values.

Write a Python function that takes two lists and returns True if they have at least one common member.

```
def common_data(list1, list2):
    for x in list1:
        for y in list2:
            if x == y:
                result = True
                return result
print(common_data([1,2,3,4,5], [1,2,3,4,5]))
print(common_data([1,2,3,4,5], [1,7,8,9,510]))
print(common_data([1,2,3,4,5], [6,7,8,9,10]))
```

Output:

C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy\ful.py

True
True
None

#-----

```
def area(radius):  
    b = 3.14159 * radius**2  
    return b
```

Parameters:

Parameters are passed during the definition of function while Arguments are passed during the function call.

Example:

#here a and b are parameters

```
def add(a,b): ##function definition  
    return a+b
```

#12 and 13 are arguments

#function call

```
result=add(12,13)  
print(result)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/paraarg.py

25

Some examples on functions:

To display vandemataram by using function use no args no return type

#function defination

```
def display():  
    print("vandemataram")  
print("i am in main")
```

```
#function call
display()
print("i am in main")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
i am in main
vandemataram
i am in main
```

#Type1 : No parameters and no return type

```
def Fun1() :
    print("function 1")
Fun1()
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

function 1
```

#Type 2: with param with out return type

```
def fun2(a) :
    print(a)
fun2("hello")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

Hello
```

#Type 3: without param with return type

```
def fun3():
    return "welcome to python"
print(fun3())
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
```

```
welcome to python
```

#Type 4: with param with return type

```
def fun4(a):  
    return a  
print(fun4("python is better then c"))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
```

```
python is better then c
```

Local and Global scope:

Local Scope:

A variable which is defined inside a function is local to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing

Global Scope:

A variable which is defined in the main body of a file is called a global variable. It will be visible throughout the file, and also inside any file which imports that file.

- The variable defined inside a function can also be made global by using the global statement.

```
def function_name(args):  
    .....  
    global x    #declaring global variable inside a function  
    .....
```

create a global variable

```
x = "global"

def f():
    print("x inside :", x)

f()
print("x outside:", x)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

x inside : global

x outside: global

create a local variable

```
def f1():
    y = "local"
    print(y)

f1()
```

Output:

local

- If we try to access the local variable outside the scope for example,

```
def f2():
    y = "local"

f2()
print(y)
```

Then when we try to run it shows an error,

Traceback (most recent call last):

File "C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py", line 6, in <module>

```
print(y)
NameError: name 'y' is not defined
```

The output shows an error, because we are trying to access a local variable y in a global scope whereas the local variable only works inside f2() or local scope.

use local and global variables in same code

```
x = "global"

def f3():
    global x
    y = "local"
    x = x * 2
    print(x)
    print(y)
```

```
f3()
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
globalglobal
local
```

- In the above code, we declare x as a global and y as a local variable in the f3(). Then, we use multiplication operator * to modify the global variable x and we print both x and y.
- After calling the f3(), the value of x becomes global global because we used the x * 2 to print two times global. After that, we print the value of local variable y i.e local.

use Global variable and Local variable with same name

```
x = 5

def f4():
    x = 10
    print("local x:", x)

f4()
print("global x:", x)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

local x: 10

global x: 5

Function Composition:

Having two (or more) functions where the output of one function is the input for another. So for example if you have two functions FunctionA and FunctionB you compose them by doing the following.

FunctionB(FunctionA(x))

Here x is the input for FunctionA and the result of that is the input for FunctionB.

Example 1:**#create a function compose2**

```
>>> def compose2(f, g):  
    return lambda x:f(g(x))
```

```
>>> def d(x):  
    return x*2
```

```
>>> def e(x):  
    return x+1
```

```
>>> a=compose2(d,e) # FunctionC = compose(FunctionB,FunctionA)
```

```
>>> a(5) # FunctionC(x)
```

```
12
```

In the above program we tried to compose n functions with the main function created.

Example 2:

```
>>> colors=('red','green','blue')
```

```
>>> fruits=['orange','banana','cherry']
```

```
>>> zip(colors,fruits)
```

```
<zip object at 0x03DAC6C8>
```

```
>>> list(zip(colors,fruits))
```

```
[('red', 'orange'), ('green', 'banana'), ('blue', 'cherry')]
```

Recursion:

Recursion is the process of defining something in terms of itself.

Python Recursive Function

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is $1*2*3*4*5*6 = 720$.

Following is an example of recursive function to find the factorial of an integer.

```
# Write a program to factorial using recursion
```

```
def fact(x):
```

```
    if x==0:
```

```
        result = 1
```

```
    else :
```

```
        result = x * fact(x-1)
```

```
    return result
```

```
print("zero factorial",fact(0))
```

```
print("five factorial",fact(5))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/rec.py
```

```
zero factorial 1
```

```
five factorial 120
```

```
-----
```

```
def calc_factorial(x):
```

```
    """This is a recursive function
```

```
    to find the factorial of an integer"""
```

```
    if x == 1:
```

```
        return 1
```

```
    else:
```

```
        return (x * calc_factorial(x-1))
```

```
num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/rec.py

The factorial of 4 is 24

Strings:

A string is a group/ a sequence of characters. Since Python has no provision for arrays, we simply use strings. This is how we declare a string. We can use a pair of single or double quotes. Every string object is of the type 'str'.

```
>>> type("name")
<class 'str'>
>>> name=str()
>>> name
''
>>> a=str('mrcet')
>>> a
'mrcet'
>>> a=str(mrcet)
>>> a[2]
'c'
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement selects character number 1 from fruit and assigns it to letter. The expression in brackets is called an index. The index indicates which character in the sequence we want

String slices:

A segment of a string is called a slice. Selecting a slice is similar to selecting a character:

Subsets of strings can be taken using the slice operator (**[] and [:]**) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

Slice out substrings, sub lists, sub Tuples using index.

Syntax: [Start: stop: steps]

- Slicing will start from index and will go up to **stop** in **step** of steps.
- Default value of start is 0,

- Stop is last index of list
- And for step default is 1

For example 1–

```
str = 'Hello World!'
```

```
print str # Prints complete string
```

```
print str[0] # Prints first character of the string
```

```
print str[2:5] # Prints characters starting from 3rd to 5th
```

```
print str[2:] # Prints string starting from 3rd character print
```

```
str * 2 # Prints string two times
```

```
print str + "TEST" # Prints concatenated string
```

Output:

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

Example 2:

```
>>> x='computer'
```

```
>>> x[1:4]
```

```
'omp'
```

```
>>> x[1:6:2]
```

```
'opt'
```

```
>>> x[3:]
```

```
'puter'
>>> x[:5]
'compu'
>>> x[-1]
'r'
>>> x[-3:]
'ter'
>>> x[:-2]
'comput'
>>> x[::-2]
'rtpo'
>>> x[::-1]
'retupmoc'
```

Immutability:

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string.

For example:

```
>>> greeting='mrcet college!'
>>> greeting[0]='n'
```

TypeError: 'str' object does not support item assignment

The reason for the error is that strings are **immutable**, which means we can't change an existing string. The best we can do is creating a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

Note: The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator

String functions and methods:

There are many methods to operate on String.

S.no	Method name	Description
1.	isalnum()	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
2.	isalpha()	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
3.	isdigit()	Returns true if string contains only digits and false otherwise.
4.	islower()	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
5.	isnumeric()	Returns true if a string contains only numeric characters and false otherwise.
6.	isspace()	Returns true if string contains only whitespace characters and false otherwise.
7.	istitle()	Returns true if string is properly “titlecased” and false otherwise.
8.	isupper()	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
9.	replace(old, new [, max])	Replaces all occurrences of old in string with new or at most max occurrences if max given.
10.	split()	Splits string according to delimiter str (space if not provided) and returns list of substrings;
11.	count()	Occurrence of a string in another string
12.	find()	Finding the index of the first occurrence of a string in another string
13.	swapcase()	Converts lowercase letters in a string to uppercase and viceversa
14.	startswith(str, beg=0, end=len(string))	Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.

Note:

All the string methods will be returning either true or false as the result

1. isalnum():

Isalnum() method returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

Syntax:

String.isalnum()

Example:

```
>>> string="123alpha"
>>> string.isalnum() True
```

2. isalpha():

isalpha() method returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

Syntax:

String.isalpha()

Example:

```
>>> string="nikhil"
>>> string.isalpha()
True
```

3. isdigit():

isdigit() returns true if string contains only digits and false otherwise.

Syntax:

String.isdigit()

Example:

```
>>> string="123456789"
>>> string.isdigit()
True
```

4. islower():

Islower() returns true if string has characters that are in lowercase and false otherwise.

Syntax:

String.islower()

Example:

```
>>> string="nikhil"  
>>> string.islower()  
True
```

5. isnumeric():

isnumeric() method returns true if a string contains only numeric characters and false otherwise.

Syntax:

String.isnumeric()

Example:

```
>>> string="123456789"  
>>> string.isnumeric()  
True
```

6. isspace():

isspace() returns true if string contains only whitespace characters and false otherwise.

Syntax:

String.isspace()

Example:

```
>>> string=" "  
>>> string.isspace()  
True
```

7. istitle()

istitle() method returns true if string is properly “titlecased”(starting letter of each word is capital) and false otherwise

Syntax:

String.istitle()

Example:

```
>>> string="Nikhil Is Learning"
>>> string.istitle()
True
```

8. isupper()

isupper() returns true if string has characters that are in uppercase and false otherwise.

Syntax:

String.isupper()

Example:

```
>>> string="HELLO"
>>> string.isupper()
True
```

9. replace()

replace() method replaces all occurrences of old in string with new or at most max occurrences if max given.

Syntax:

String.replace()

Example:

```
>>> string="Nikhil Is Learning"
>>> string.replace('Nikhil','Neha')
'Neha Is Learning'
```

10. split()

split() method splits the string according to delimiter str (space if not provided)

Syntax:

String.split()

Example:

```
>>> string="Nikhil Is Learning"
>>> string.split()
```

['Nikhil', 'Is', 'Learning']

11.count()

count() method counts the occurrence of a string in another string Syntax:

String.count()

Example:

```
>>> string='Nikhil Is Learning'
```

```
>>> string.count('i')
```

```
3
```

12.find()

Find() method is used for finding the index of the first occurrence of a string in another string

Syntax:

String.find(„string“)

Example:

```
>>> string="Nikhil Is Learning"
```

```
>>> string.find('k')
```

```
2
```

13.swapcase()

converts lowercase letters in a string to uppercase and viceversa

Syntax:

String.find(„string“)

Example:

```
>>> string="HELLO"
```

```
>>> string.swapcase()
```

```
'hello'
```

14.startswith()

Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.

Syntax:

String.startswith(„string“)

Example:

```
>>> string="Nikhil Is Learning"
>>> string.startswith('N')
True
```

15.endswith()

Determines if string or a substring of string (if starting index beg and ending index end are given) ends with substring str; returns true if so and false otherwise.

Syntax:

String.endswith(„string“)

Example:

```
>>> string="Nikhil Is Learning"
>>> string.endswith('g')
True
```

String module:

This module contains a number of functions to process standard Python strings. In recent versions, most functions are available as string methods as well.

It's a built-in module and we have to **import** it before using any of its constants and classes

Syntax: import string

Note:

help(string) --- gives the information about all the variables ,functions, attributes and classes to be used in string module.

Example:

```
import string
print(string.ascii_letters)
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.digits)
```

```
print(string.hexdigits)
#print(string.whitespace)
print(string.punctuation)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/strrmodl.py

```
=====
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
0123456789abcdefABCDEF
!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
```

Python String Module Classes

Python string module contains two classes – Formatter and Template.

Formatter

It behaves exactly same as str.format() function. This class becomes useful if you want to subclass it and define your own format string syntax.

Syntax: from string import Formatter

Template

This class is used to create a string template for simpler string substitutions

Syntax: from string import Template

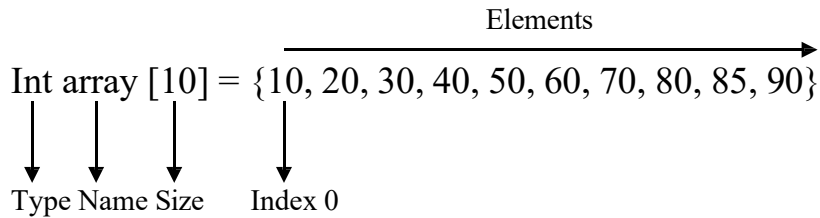
Python arrays:

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element**– Each item stored in an array is called an element.
- **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. Below is an illustration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 70

Basic Operations

Following are the basic operations supported by an array.

- Traverse – print all the array elements one by one.
- Insertion – Adds an element at the given index.
- Deletion – Deletes an element at the given index.
- Search – Searches an element using the given index or by the value.
- Update – Updates an element at the given index.

Array is created in Python by importing array module to the python program. Then the array is declared as shown below.

```
from array import *  
  
arrayName=array(typecode, [initializers])
```

Typecode are the codes that are used to define the type of value the array will hold. Some common typecodes used are:

Typecode	Value
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte

c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
l	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Creating an array:

```
from array import *
array1 = array('i', [10,20,30,40,50])
for x in array1:
    print(x)
```

Output:

```
>>>
```

```
RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/arr.py
```

```
10
20
30
40
50
```

Access the elements of an Array:

Accessing Array Element

We can access each element of an array using the index of the element.

```
from array import *
array1 = array('i', [10,20,30,40,50])
print (array1[0])
print (array1[2])
```

Output:

RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/arr2.py
10
30

Array methods:

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position
<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the first item with the specified value
<u>reverse()</u>	Reverses the order of the list
<u>sort()</u>	Sorts the list

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

Example:

```
>>> college=["mrcet","it","cse"]

>>> college.append("autonomous")

>>> college

['mrcet', 'it', 'cse', 'autonomous']

>>> college.append("eee")

>>> college.append("ece")

>>> college

['mrcet', 'it', 'cse', 'autonomous', 'eee', 'ece']

>>> college.pop()

'ece'

>>> college

['mrcet', 'it', 'cse', 'autonomous', 'eee']

>>> college.pop(4)

'eee'

>>> college

['mrcet', 'it', 'cse', 'autonomous']

>>> college.remove("it")

>>> college

['mrcet', 'cse', 'autonomous']
```

UNIT – IV

LISTS, TUPLES, DICTIONARIES

Lists: list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters, list comprehension; **Tuples:** tuple assignment, tuple as return value, tuple comprehension; **Dictionaries:** operations and methods, comprehension;

Lists, Tuples, Dictionaries:

List:

- It is a general purpose most widely used in data structures
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Ex:

```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
```

```
>>> print(list1)
```

```
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
```

```
-----
```

```
>>> x=list()
```

```
>>> x
```

```
[]
```

```
-----
```

```
>>> tuple1=(1,2,3,4)
```

```
>>> x=list(tuple1)
```

```
>>> x
```

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

List operations:

These operations include indexing, slicing, adding, multiplying, and checking for membership

Basic List Operations:

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

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

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input –

`L = ['mrcet', 'college', 'MRCET!']`

Python Expression	Results	Description
<code>L[2]</code>	MRCET	Offsets start at zero

L[-2]	college	Negative: count from the right
L[1:]	['college', 'MRCET!']	Slicing fetches sections

List slices:

```
>>> list1=range(1,6)
>>> list1
range(1, 6)
>>> print(list1)
range(1, 6)
>>> list1=[1,2,3,4,5,6,7,8,9,10]
>>> list1[1:]
[2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list1[:1]
[1]
>>> list1[2:5]
[3, 4, 5]
>>> list1[:6]
[1, 2, 3, 4, 5, 6]
>>> list1[1:2:4]
[2]
>>> list1[1:8:2]
[2, 4, 6, 8]
```

List methods:

The list data type has some more methods. Here are all of the methods of list objects:

- Del()

- Append()
- Extend()
- Insert()
- Pop()
- Remove()
- Reverse()
- Sort()

Delete: Delete a list or an item from a list

```
>>> x=[5,3,8,6]
>>> del(x[1])      #deletes the index position 1 in a list
>>> x
[5, 8, 6]
-----
>>> del(x)
>>> x              # complete list gets deleted
```

Append: Append an item to a list

```
>>> x=[1,5,8,4]
>>> x.append(10)
>>> x
[1, 5, 8, 4, 10]
```

Extend: Append a sequence to a list.

```
>>> x=[1,2,3,4]
>>> y=[3,6,9,1]
>>> x.extend(y)
>>> x
[1, 2, 3, 4, 3, 6, 9, 1]
```

Insert: To add an item at the specified index, use the insert () method:

```
>>> x=[1,2,4,6,7]
```

```
>>> x.insert(2,10) #insert(index no, item to be inserted)
```

```
>>> x
```

```
[1, 2, 10, 4, 6, 7]
```

```
-----
```

```
>>> x.insert(4,['a',11])
```

```
>>> x
```

```
[1, 2, 10, 4, ['a', 11], 6, 7]
```

Pop: The pop() method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

```
>>> x=[1, 2, 10, 4, 6, 7]
```

```
>>> x.pop()
```

```
7
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
-----
```

```
>>> x=[1, 2, 10, 4, 6]
```

```
>>> x.pop(2)
```

```
10
```

```
>>> x
```

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

Remove: The remove() method removes the specified item from a given list.

```
>>> x=[1,33,2,10,4,6]
```

```
>>> x.remove(33)
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
>>> x.remove(4)
```

```
>>> x
```

```
[1, 2, 10, 6]
```

Reverse: Reverse the order of a given list.

```
>>> x=[1,2,3,4,5,6,7]
```

```
>>> x.reverse()
```

```
>>> x
```

```
[7, 6, 5, 4, 3, 2, 1]
```

Sort: Sorts the elements in ascending order

```
>>> x=[7, 6, 5, 4, 3, 2, 1]
```

```
>>> x.sort()
```

```
>>> x
```

```
[1, 2, 3, 4, 5, 6, 7]
```

```
-----
```

```
>>> x=[10,1,5,3,8,7]
```

```
>>> x.sort()
```

```
>>> x
```

```
[1, 3, 5, 7, 8, 10]
```

List loop:

Loops are control structures used to repeat a given section of code a certain number of times or until a particular condition is met.

Method #1: For loop

#list of items

```
list = ['M','R','C','E','T']
```

```
i = 1
```

```
#Iterating over the list  
for item in list:  
    print ('college ',i,' is ',item)  
    i = i+1
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/lis.py  
college 1 is M  
college 2 is R  
college 3 is C  
college 4 is E  
college 5 is T
```

Method #2: For loop and range()

In case we want to use the traditional for loop which iterates from number x to number y.

Python3 code to iterate over a list

```
list = [1, 3, 5, 7, 9]
```

```
# getting length of list  
length = len(list)
```

```
# Iterating the index  
# same as 'for i in range(len(list))'  
for i in range(length):  
    print(list[i])
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/listloop.py  
1  
3  
5  
7  
9
```

Method #3: using while loop

Python3 code to iterate over a list

```
list = [1, 3, 5, 7, 9]
```

```
# Getting length of list
```

```
length = len(list)
i = 0
```

```
# Iterating using while loop
while i < length:
    print(list[i])
    i += 1
```

Mutability:

A mutable object can be changed after it is created, and an immutable object can't.

Append: Append an item to a list

```
>>> x=[1,5,8,4]
>>> x.append(10)
>>> x
[1, 5, 8, 4, 10]
```

Extend: Append a sequence to a list.

```
>>> x=[1,2,3,4]
>>> y=[3,6,9,1]
>>> x.extend(y)
>>> x
```

Delete: Delete a list or an item from a list

```
>>> x=[5,3,8,6]
>>> del(x[1])      #deletes the index position 1 in a list
>>> x
[5, 8, 6]
```

Insert: To add an item at the specified index, use the insert () method:

```
>>> x=[1,2,4,6,7]
>>> x.insert(2,10) #insert(index no, item to be inserted)
```

```
>>> x
```

```
[1, 2, 10, 4, 6, 7]
```

```
-----
```

```
>>> x.insert(4,['a',11])
```

```
>>> x
```

```
[1, 2, 10, 4, ['a', 11], 6, 7]
```

Pop: The `pop()` method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

```
>>> x=[1, 2, 10, 4, 6, 7]
```

```
>>> x.pop()
```

```
7
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
-----
```

```
>>> x=[1, 2, 10, 4, 6]
```

```
>>> x.pop(2)
```

```
10
```

```
>>> x
```

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

Remove: The `remove()` method removes the specified item from a given list.

```
>>> x=[1,33,2,10,4,6]
```

```
>>> x.remove(33)
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
>>> x.remove(4)
```

```
>>> x
```

```
[1, 2, 10, 6]
```

Reverse: Reverse the order of a given list.

```
>>> x=[1,2,3,4,5,6,7]
```

```
>>> x.reverse()
```

```
>>> x
```

```
[7, 6, 5, 4, 3, 2, 1]
```

Sort: Sorts the elements in ascending order

```
>>> x=[7, 6, 5, 4, 3, 2, 1]
```

```
>>> x.sort()
```

```
>>> x
```

```
[1, 2, 3, 4, 5, 6, 7]
```

```
-----
```

```
>>> x=[10,1,5,3,8,7]
```

```
>>> x.sort()
```

```
>>> x
```

```
[1, 3, 5, 7, 8, 10]
```

Aliasing:

1. An alias is a second name for a piece of data, often easier (and more useful) than making a copy.
2. If the data is immutable, aliases don't matter because the data can't change.
3. But if data can change, aliases can result in lot of hard – to – find bugs.
4. Aliasing happens whenever one variable's value is assigned to another variable.

For ex:

```
a = [81, 82, 83]
```

```
b = [81, 82, 83]
print(a == b)
print(a is b)
b = a
print(a == b)
print(a is b)
b[0] = 5
print(a)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/alia.py
True
False
True
True
[5, 82, 83]
```

Because the same list has two different names, a and b, we say that it is **aliased**. Changes made with one alias affect the other. In the example above, you can see that a and b refer to the same list after executing the assignment statement b = a.

Cloning Lists:

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called cloning, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the slice operator. Taking any slice of a creates a new list. In this case the slice happens to consist of the whole list.

Example:

```
a = [81, 82, 83]

b = a[:]    # make a clone using slice

print(a == b)

print(a is b)

b[0] = 5

print(a)

print(b)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/clo.py

True

False

[81, 82, 83]

[5, 82, 83]

Now we are free to make changes to b without worrying about a

List parameters:

Passing a list as an argument actually passes a reference to the list, not a copy of the list. Since lists are mutable, changes made to the elements referenced by the parameter change the same list that the argument is referencing.

for example, the function below takes a list as an argument and multiplies each element in the list by 2:

```
def doubleStuff(List):
```

```
    """ Overwrite each element in aList with double its value. """
```

```
    for position in range(len(List)):
```

```
        List[position] = 2 * List[position]
```

```
things = [2, 5, 9]
```

```
print(things)
```

```
doubleStuff(things)
```

```
print(things)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/lipar.py ==

[2, 5, 9]

[4, 10, 18]

List comprehension:

List:

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
>>> list1=[]

>>> for x in range(10):

    list1.append(x**2)

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

(or)

This is also equivalent to

```
>>> list1=list(map(lambda x:x**2, range(10)))

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

(or)

Which is more concise and readable.

```
>>> list1=[x**2 for x in range(10)]

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Similarly some examples:

```
>>> x=[m for m in range(8)]
```

```
>>> print(x)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> x=[z**2 for z in range(10) if z>4]
```

```
>>> print(x)
```

```
[25, 36, 49, 64, 81]
```

```
>>> x=[x ** 2 for x in range (1, 11) if x % 2 == 1]
```

```
>>> print(x)
```

```
[1, 9, 25, 49, 81]
```

```
>>> a=5
```

```
>>> table = [[a, b, a * b] for b in range(1, 11)]
```

```
>>> for i in table:
```

```
    print(i)
```

```
[5, 1, 5]
```

```
[5, 2, 10]
```

```
[5, 3, 15]
```

```
[5, 4, 20]
```

```
[5, 5, 25]
```

```
[5, 6, 30]
```

```
[5, 7, 35]
```

```
[5, 8, 40]
```

```
[5, 9, 45]
```

```
[5, 10, 50]
```

Tuples:

A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.

- Supports all operations for sequences.
- Immutable, but member objects may be mutable.
- If the contents of a list shouldn't change, use a tuple to prevent items from

accidentally being added, changed, or deleted.

- Tuples are more efficient than list due to python's implementation.

We can construct tuple in many ways:

```
X=() #no item tuple
```

```
X=(1,2,3)
```

```
X=tuple(list1)
```

```
X=1,2,3,4
```

Example:

```
>>> x=(1,2,3)
```

```
>>> print(x)
```

```
(1, 2, 3)
```

```
>>> x
```

```
(1, 2, 3)
```

```
-----
```

```
>>> x=()
```

```
>>> x
```

```
()
```

```
-----
```

```
>>> x=[4,5,66,9]
```

```
>>> y=tuple(x)
```

```
>>> y
```

```
(4, 5, 66, 9)
```

```
-----
```

```
>>> x=1,2,3,4
```

```
>>> x
```

```
(1, 2, 3, 4)
```

Some of the operations of tuple are:

- Access tuple items
- Change tuple items
- Loop through a tuple
- Count()
- Index()
- Length()

Access tuple items: Access tuple items by referring to the index number, inside square brackets

```
>>> x=('a','b','c','g')
```

```
>>> print(x[2])
```

c

Change tuple items: Once a tuple is created, you cannot change its values. Tuples are unchangeable.

```
>>> x=(2,5,7,'4',8)
```

```
>>> x[1]=10
```

Traceback (most recent call last):

File "<pyshell#41>", line 1, in <module>

x[1]=10

TypeError: 'tuple' object does not support item assignment

```
>>> x
```

```
(2, 5, 7, '4', 8) # the value is still the same
```

Loop through a tuple: We can loop the values of tuple using for loop

```
>>> x=4,5,6,7,2,'aa'
```

```
>>> for i in x:
```

```
    print(i)
```

4

5

6

7

2

aa

Count (): Returns the number of times a specified value occurs in a tuple

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
```

```
>>> x.count(2)
```

4

Index (): Searches the tuple for a specified value and returns the position of where it was found

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> x.index(2)
1
```

(Or)

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=x.index(2)
>>> print(y)
1
```

Length (): To know the number of items or values present in a tuple, we use len().

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=len(x)
>>> print(y)
12
```

Tuple Assignment

Python has tuple assignment feature which enables you to assign more than one variable at a time. In here, we have assigned tuple 1 with the college information like college name, year, etc. and another tuple 2 with the values in it like number (1, 2, 3... 7).

For Example,

Here is the code,

- >>> tup1 = ('mrcet', 'eng college','2004','cse', 'it','csit');
- >>> tup2 = (1,2,3,4,5,6,7);
- >>> print(tup1[0])
- mrcet
- >>> print(tup2[1:4])
- (2, 3, 4)

Tuple 1 includes list of information of mrcet

Tuple 2 includes list of numbers in it

We call the value for [0] in tuple and for tuple 2 we call the value between 1 and 4

Run the above code- It gives name mrcet for first tuple while for second tuple it gives number (2, 3, 4)

Tuple as return values:

A Tuple is a comma separated sequence of items. It is created with or without (). Tuples are immutable.

A Python program to return multiple values from a method using tuple

This function returns a tuple

```
def fun():
```

```
    str = "mrcet college"
```

```
    x = 20
```

```
    return str, x; # Return tuple, we could also
```

```
        # write (str, x)
```

Driver code to test above method

```
str, x = fun() # Assign returned tuple
```

```
print(str)
```

```
print(x)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/tupretval.py

mrcet college

20

- Functions can return tuples as return values.

```
def circleInfo(r):
```

```
    """ Return (circumference, area) of a circle of radius r """
```

```
    c = 2 * 3.14159 * r
```

```
    a = 3.14159 * r * r
```

```
    return (c, a)
```

```
print(circleInfo(10))
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/functupretval.py

(62.8318, 314.159)

```

-----
def f(x):
    y0 = x + 1
    y1 = x * 3
    y2 = y0 ** y3
    return (y0, y1, y2)

```

Tuple comprehension:

Tuple Comprehensions are special: The result of a tuple comprehension is special. You might expect it to produce a tuple, but what it does is produce a special "generator" object that we can iterate over.

For example:

```

>>> x = (i for i in 'abc') #tuple comprehension
>>> x
<generator object <genexpr> at 0x033EEC30>

>>> print(x)
<generator object <genexpr> at 0x033EEC30>

```

You might expect this to print as ('a', 'b', 'c') but it prints as <generator object <genexpr> at 0x02AAD710> The result of a tuple comprehension is not a tuple: it is actually a generator. The only thing that you need to know now about a generator now is that you can iterate over it, but ONLY ONCE.

So, given the code

```

>>> x = (i for i in 'abc')
>>> for i in x:
    print(i)

```

```

a
b
c

```

Create a list of 2-tuples like (number, square):

```

>>> z=[(x, x**2) for x in range(6)]
>>> z
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]

```

Set:

Similarly to list comprehensions, set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

```
>>> x={3*x for x in range(10) if x>5}
>>> x
{24, 18, 27, 21}
```

Dictionaries:

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

- Key-value pairs
- Unordered

We can construct or create dictionary like:

```
X={1:'A',2:'B',3:'c'}
```

```
X=dict([('a',3) ('b',4)])
```

```
X=dict('A'=1,'B' =2)
```

Example:

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
```

```
>>> dict1
```

```
{'brand': 'mrcet', 'model': 'college', 'year': 2004}
```

Operations and methods:

Methods that are available with dictionary are tabulated below. Some of them have already been used in the above examples.

Method	Description
clear()	Remove all items form the dictionary.

<code>copy()</code>	Return a shallow copy of the dictionary.
<code>fromkeys(seq[, v])</code>	Return a new dictionary with keys from seq and value equal to v (defaults to None).
<code>get(key[,d])</code>	Return the value of key. If key doesnot exit, return d (defaults to None).
<code>items()</code>	Return a new view of the dictionary's items (key, value).
<code>keys()</code>	Return a new view of the dictionary's keys.
<code>pop(key[,d])</code>	Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError.
<code>popitem()</code>	Remove and return an arbitrary item (key, value). Raises KeyError if the dictionary is empty.
<code>setdefault(key[,d])</code>	If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None).
<code>update([other])</code>	Update the dictionary with the key/value pairs from other, overwriting existing keys.
<code>values()</code>	Return a new view of the dictionary's values

Below are some dictionary operations:

To access specific value of a dictionary, we must pass its key,

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> x=dict1["brand"]
>>> x
'mrcet'
```

To access keys and values and items of dictionary:

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1.keys()
dict_keys(['brand', 'model', 'year'])
>>> dict1.values()
dict_values(['mrcet', 'college', 2004])
>>> dict1.items()
dict_items([('brand', 'mrcet'), ('model', 'college'), ('year', 2004)])
```

>>> for items in dict1.values():
 print(items)

```
mrcet
college
2004
```

```
>>> for items in dict1.keys():
    print(items)
```

```
brand
model
year
```

```
>>> for i in dict1.items():
    print(i)
```

```
('brand', 'mrcet')
('model', 'college')
('year', 2004)
```

Some more operations like:

- Add/change

- Remove
- Length
- Delete

Add/change values: You can change the value of a specific item by referring to its key name

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1["year"]=2005
>>> dict1
{'brand': 'mrcet', 'model': 'college', 'year': 2005}
```

Remove(): It removes or pop the specific item of dictionary.

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> print(dict1.pop("model"))
college
>>> dict1
{'brand': 'mrcet', 'year': 2005}
```

Delete: Deletes a particular item.

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> del x[5]
>>> x
```

Length: we use len() method to get the length of dictionary.

```
>>> {1: 1, 2: 4, 3: 9, 4: 16}
{1: 1, 2: 4, 3: 9, 4: 16}
>>> y=len(x)
>>> y
4
```

Iterating over (key, value) pairs:

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> for key in x:
    print(key, x[key])
```

```
1 1
2 4
3 9
```

```
4 16
5 25
>>> for k,v in x.items():
    print(k,v)
```

```
1 1
2 4
3 9
4 16
5 25
```

List of Dictionaries:

```
>>> customers = [{"uid":1,"name":"John"},
    {"uid":2,"name":"Smith"},
    {"uid":3,"name":"Andersson"},
    ]
>>> >>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'Andersson'}]
```

Print the uid and name of each customer

```
>>> for x in customers:
    print(x["uid"], x["name"])
```

```
1 John
2 Smith
3 Andersson
```

Modify an entry, This will change the name of customer 2 from Smith to Charlie

```
>>> customers[2]["name"]="charlie"
>>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'charlie'}]
```

Add a new field to each entry

```
>>> for x in customers:
    x["password"]="123456" # any initial value
```

```
>>> print(customers)
```

```
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 2, 'name': 'Smith', 'password': '123456'}, {'uid': 3, 'name': 'charlie', 'password': '123456'}]
```

Delete a field

```
>>> del customers[1]
```

```
>>> print(customers)
```

```
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 3, 'name': 'charlie', 'password': '123456'}]
```

```
>>> del customers[1]
```

```
>>> print(customers)
```

```
[{'uid': 1, 'name': 'John', 'password': '123456'}]
```

Delete all fields

```
>>> for x in customers:
```

```
    del x["uid"]
```

```
>>> x
```

```
{'name': 'John', 'password': '123456'}
```

Comprehension:

Dictionary comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> z={x: x**2 for x in (2,4,6)}
```

```
>>> z
```

```
{2: 4, 4: 16, 6: 36}
```

```
>>> dict11 = {x: x*x for x in range(6)}
```

```
>>> dict11
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

UNIT – V

FILES, EXCEPTIONS, MODULES, PACKAGES

Files and exception: text files, reading and writing files, command line arguments, errors and exceptions, handling exceptions, modules (datetime, time, OS , calendar, math module), Explore packages.

Files, Exceptions, Modules, Packages:

Files and exception:

A **file** is some information or data which stays in the computer storage devices. Python gives you easy ways to manipulate these files. Generally files divide in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

An **exception** is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

Text files:

We can create the text files by using the syntax:

Variable name=open (“file.txt”, file mode)

For ex: f= open ("hello.txt","w+")

- We declared the variable f to open a file named hello.txt. **Open** takes 2 arguments, the file that we want to open and a string that represents the kinds of permission or operation we want to do on the file
- Here we used "w" letter in our argument, which indicates write and the plus sign that means it will create a file if it does not exist in library

- The available option beside "w" are "r" for read and "a" for append and plus sign means if it is not there then create it

File Modes in Python:

Mode	Description
'r'	This is the default mode. It Opens file for reading.
'w'	This Mode Opens file for writing. If file does not exist, it creates a new file. If file exists it truncates the file.
'x'	Creates a new file. If file already exists, the operation fails.
'a'	Open file in append mode. If file does not exist, it creates a new file.
't'	This is the default mode. It opens in text mode.
'b'	This opens in binary mode.
'+'	This will open a file for reading and writing (updating)

Reading and Writing files:

The following image shows how to create and open a text file in notepad from command prompt

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

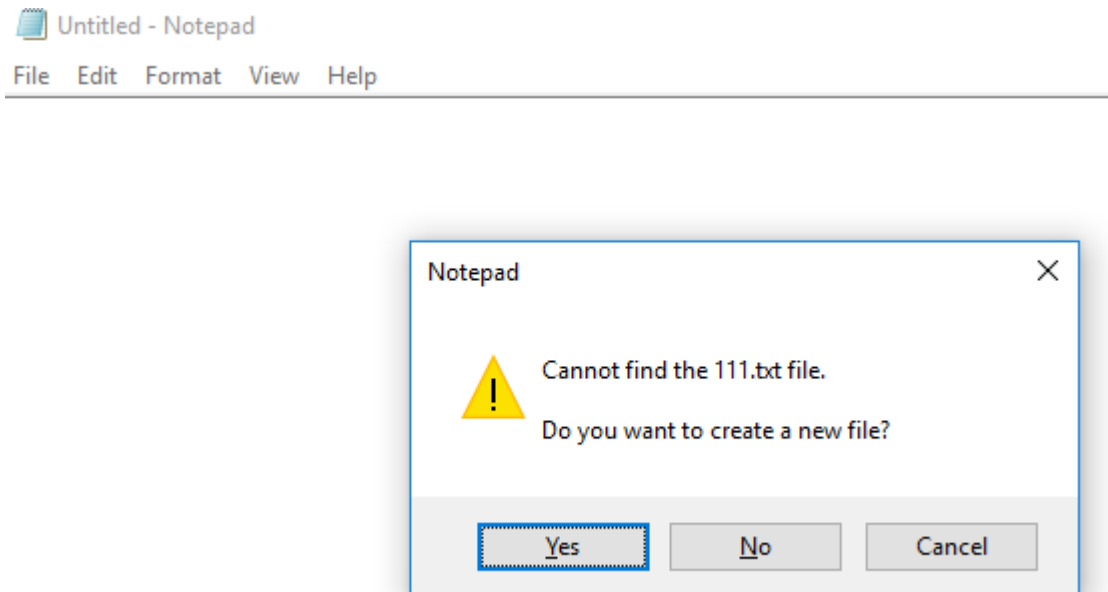
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>start notepad hello.txt

C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type hello.txt
Hello mrcet
good morning
how r u
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>
```

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>notepad 111.txt
```

Hit on enter then it shows the following whether to open or not?



Click on “yes” to open else “no” to cancel

Write a python program to open and read a file

```
a=open(“one.txt”,”r”)
```

```
print(a.read())
```

a.close()

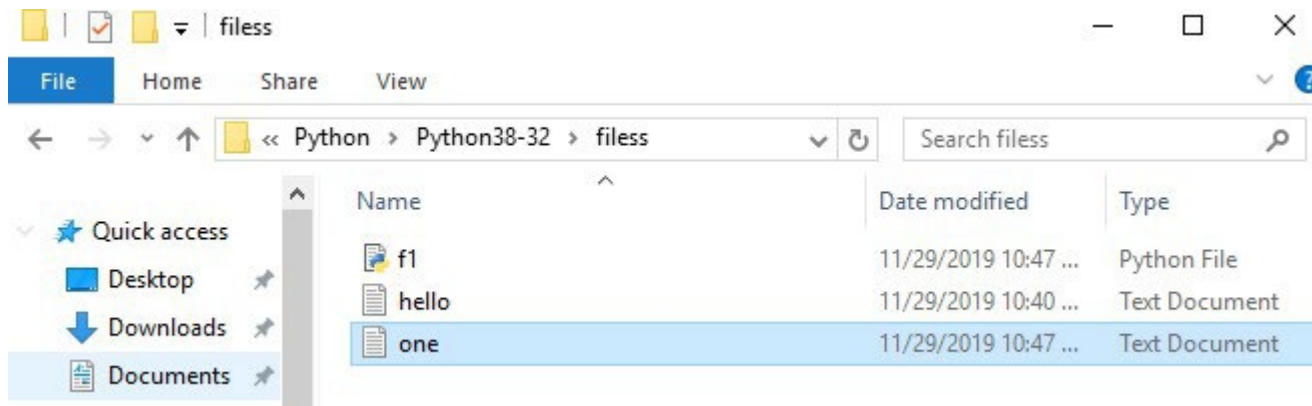
Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/filess/f1.py
welcome to python programming

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>python f1.py  
welcome to python programming
```

Note: All the program files and text files need to be saved together in a particular file then only the program performs the operations in the given file mode



f.close() ---- This will close the instance of the file somefile.txt stored

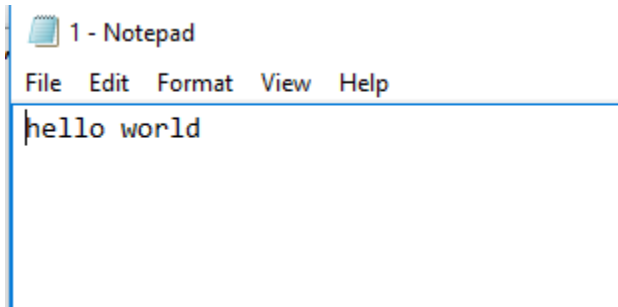
Write a python program to open and write “hello world” into a file?

```
f=open("1.txt","a")
```

```
f.write("hello world")
```

```
f.close()
```

Output:



(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type 1.txt
hello world
```

Note: In the above program the 1.txt file is created automatically and adds hello world into txt file

If we keep on executing the same program for more than one time then it append the data that many times

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type 1.txt
hello worldhello world
```

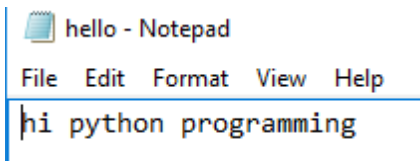
Write a python program to write the content “hi python programming” for the existing file.

```
f=open("1.txt",'w')
```

```
f.write("hi python programming")
```

```
f.close()
```

Output:



In the above program the hello.txt file consist of data like

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type hello.txt
Hello mrcet
good morning
how r u
```

But when we try to write some data on to the same file it overwrites and saves with the current data (check output)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type hello.txt
hi python programming
```

Write a python program to open and write the content to file and read it.

```
fo=open("abc.txt","w+")
```

```
fo.write("Python Programming")
```

```
print(fo.read())
```

```
fo.close()
```

Output:

abc - Notepad

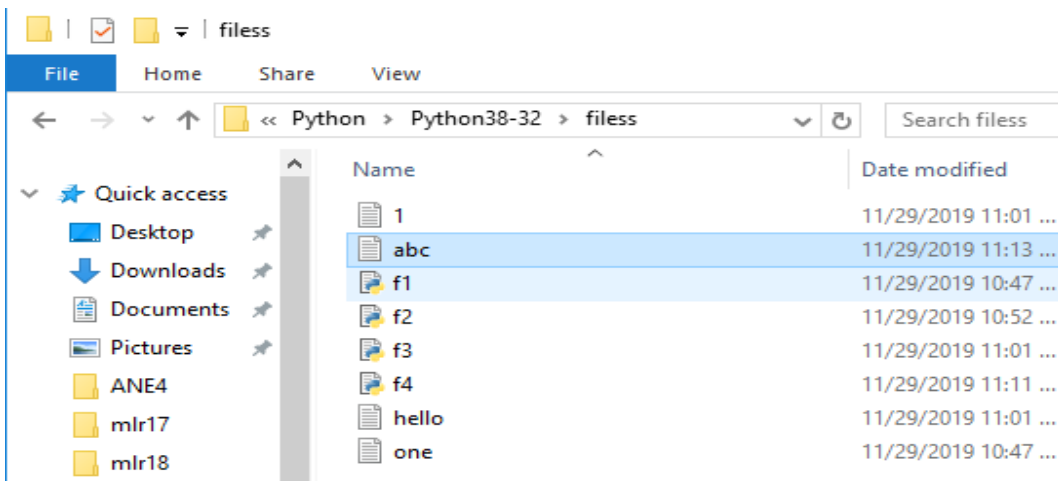
File Edit Format View Help

Python Programming introduced for III years

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type abc.txt
Python Programming introduced for III years
```

Note: It creates the abc.txt file automatically and writes the data into it



Command line arguments:

The command line arguments must be given whenever we want to give the input before the start of the script, while on the other hand, `raw_input()` is used to get the input while the python program / script is running.

The command line arguments in python can be processed by using either 'sys' module, 'argparse' module and 'getopt' module.

'sys' module :

Python sys module stores the command line arguments into a list, we can access it using `sys.argv`. This is very useful and simple way to read command line arguments as String.

sys.argv is the list of commandline arguments passed to the Python program. argv represents all the items that come along via the command line input, it's basically an array holding the command line arguments of our program

```
>>> sys.modules.keys() -- this prints so many dict elements in the form of list.
```

```
# Python code to demonstrate the use of 'sys' module for command line arguments
```

```
import sys
```

```
# command line arguments are stored in the form
```

```
# of list in sys.argv
```

```
argumentList = sys.argv
```

```
print(argumentList)
```

```
# Print the name of file
```

```
print(sys.argv[0])
```

```
# Print the first argument after the name of file
```

```
#print(sys.argv[1])
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/cmdnlinarg.py
```

```
['C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/cmdnlinarg.py']
```

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/cmdnlinarg.py
```

Note: Since my list consist of only one element at '0' index so it prints only that list element, if we try to access at index position '1' then it shows the error like,

IndexError: list index out of range

```
import sys
```

```
print(type(sys.argv))
```

```
print('The command line arguments are:')
```

```
for i in sys.argv:
```

```
    print(i)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/symod.py ==

<class 'list'>

The command line arguments are:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/symod.py

write a python program to get python version.

```
import sys
```

```
print("System version is:")
```

```
print(sys.version)
```

```
print("Version Information is:")
```

```
print(sys.version_info)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/s1.py =

System version is:

3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)]

Version Information is:

sys.version_info(major=3, minor=8, micro=0, releaselevel='final', serial=0)

'argparse' module :

Python getopt module is very similar in working as the C getopt() function for parsing command-line parameters. Python getopt module is useful in parsing command line arguments where we want user to enter some options too.

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

```
#-----
```

```
import argparse
```

```
parser = argparse.ArgumentParser()  
print(parser.parse_args())
```

‘getopt’ module :

Python argparse module is the preferred way to parse command line arguments. It provides a lot of option such as positional arguments, default value for arguments, help message, specifying data type of argument etc

It parses the command line options and parameter list. The signature of this function is mentioned below:

```
getopt.getopt(args, shortopts, longopts=[ ])
```

- args are the arguments to be passed.
- shortopts is the options this script accepts.
- Optional parameter, longopts is the list of String parameters this function accepts which should be supported. Note that the -- should not be prepended with option names.

```
-h----- print help and usage message  
-m----- accept custom option value  
-d----- run the script in debug mode
```

```
import getopt  
import sys
```

```
argv = sys.argv[0:]
```

```
try:
```

```
    opts, args = getopt.getopt(argv, 'hm:d', ['help', 'my_file='])  
    #print(opts)  
    print(args)
```

```
except getopt.GetoptError:
```

```
    # Print a message or do something useful  
    print('Something went wrong!')  
    sys.exit(2)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/gtopt.py ==
```

```
['C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/gtopt.py']
```

Errors and Exceptions:

Python Errors and Built-in Exceptions: Python (interpreter) raises exceptions when it encounters **errors**. When writing a program, we, more often than not, will encounter errors. Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error

ZeroDivisionError:

ZeroDivisionError in Python indicates that the second argument used in a division (or modulo) operation was zero.

OverflowError:

OverflowError in Python indicates that an arithmetic operation has exceeded the limits of the current Python runtime. This is typically due to excessively large float values, as integer values that are too big will opt to raise memory errors instead.

ImportError:

It is raised when you try to import a module which does not exist. This may happen if you made a typing mistake in the module name or the module doesn't exist in its standard path. In the example below, a module named "non_existing_module" is being imported but it doesn't exist, hence an import error exception is raised.

IndexError:

An IndexError exception is raised when you refer a sequence which is out of range. In the example below, the list abc contains only 3 entries, but the 4th index is being accessed, which will result an IndexError exception.

TypeError:

When two unrelated type of objects are combined, TypeErrorexception is raised. In example below, an int and a string is added, which will result in TypeError exception.

IndentationError:

Unexpected indent. As mentioned in the "expected an indentedblock" section, Python not only insists on indentation, it insists on consistent indentation. You are free to choose the number of spaces of indentation to use, but you then need to stick with it.

Syntax errors:

These are the most basic type of error. They arise when the Python parser is unable to understand a line of code. Syntax errors are almost always fatal, i.e. there is almost never a way to successfully execute a piece of code containing syntax errors.

Run-time error:

A run-time error happens when Python understands what you are saying, but runs into trouble when following your instructions.

Key Error :

Python raises a `KeyError` whenever a `dict()` object is requested (using the format `a = adict[key]`) and the key is not in the dictionary.

Value Error:

In Python, a value is the information that is stored within a certain object. To encounter a `ValueError` in Python means that is a problem with the content of the object you tried to assign the value to.

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong. In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from `Exception` class.

Different types of exceptions:

- `ArrayIndexOutOfBoundsException.`
- `ClassNotFoundException.`
- `FileNotFoundException.`
- `IOException.`
- `InterruptedException.`
- `NoSuchFieldException.`
- `NoSuchMethodException`

Handling Exceptions:

The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc. Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files. Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.

Python uses try and except keywords to handle exceptions. Both keywords are followed by indented blocks.

Syntax:

try :

 #statements in try block

except :

 #executed when error in try block

Typically we see, most of the times

- **Syntactical errors** (wrong spelling, colon (:) missing),
At developer level and compile level it gives errors.
- **Logical errors** (2+2=4, instead if we get output as 3 i.e., wrong output,),
As a developer we test the application, during that time logical error may obtained.
- **Run time error** (In this case, if the user doesn't know to give input, 5/6 is ok but if the user say 6 and 0 i.e., 6/0 (shows error a number cannot be divided by zero))
This is not easy compared to the above two errors because it is not done by the system, it is (mistake) done by the user.

The things we need to observe are:

1. You should be able to understand the mistakes; the error might be done by user, DB connection or server.
2. Whenever there is an error execution should not stop.
Ex: Banking Transaction
3. The aim is execution should not stop even though an error occurs.

For ex:

```
a=5
```

```
b=2
```

```
print(a/b)
```

```
print("Bye")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex1.py
```

```
2.5
```

```
Bye
```

- **The above is normal execution with no error, but if we say when b=0, it is a critical and gives error, see below**

```
a=5
```

```
b=0
```

```
print(a/b)
```

```
print("bye") #this has to be printed, but abnormal termination
```

Output:

Traceback (most recent call last):

File "C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex2.py", line 3, in <module>

```
print(a/b)
```

ZeroDivisionError: division by zero

- **To overcome this we handle exceptions using except keyword**

```
a=5
```

```
b=0
```

```
try:
    print(a/b)
except Exception:
    print("number can not be divided by zero")
    print("bye")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex3.py

number can not be divided by zero

bye

- **The except block executes only when try block has an error, check it below**

a=5

b=2

```
try:
```

```
    print(a/b)
```

```
except Exception:
```

```
    print("number can not be divided by zero")
```

```
    print("bye")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex4.py

2.5

- **For example if you want to print the message like what is an error in a program then we use “e” which is the representation or object of an exception.**

a=5

b=0

```
try:
```

```
print(a/b)
```

except Exception as e:

```
print("number can not be divided by zero",e)
```

```
print("bye")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex5.py

number can not be divided by zero **division by zero**

bye



(Type of error)

Let us see some more examples:

I don't want to print bye but I want to close the file whenever it is opened.

```
a=5
```

```
b=2
```

```
try:
```

```
print("resource opened")
```

```
print(a/b)
```

```
print("resource closed")
```

except Exception as e:

```
print("number can not be divided by zero",e)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex6.py

resource opened

2.5

resource closed

- **Note: the file is opened and closed well, but see by changing the value of b to 0,**

a=5

b=0

try:

```
print("resource opened")
```

```
print(a/b)
```

```
print("resource closed")
```

except Exception as e:

```
print("number can not be divided by zero",e)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex7.py

resource opened

number can not be divided by zero division by zero

- **Note: resource not closed**
- **To overcome this, keep print("resource closed") in except block, see it**

a=5

b=0

try:

```
print("resource opened")
```

```
print(a/b)
```

except Exception as e:

```
print("number can not be divided by zero",e)
```

```
print("resource closed")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex8.py

resource opened

number can not be divided by zero division by zero

resource closed

- **The result is fine that the file is opened and closed, but again change the value of b to back (i.e., value 2 or other than zero)**

```
a=5
```

```
b=2
```

```
try:
```

```
    print("resource opened")
```

```
    print(a/b)
```

```
except Exception as e:
```

```
    print("number can not be divided by zero",e)
```

```
    print("resource closed")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex9.py

resource opened

2.5

- **But again the same problem file/resource is not closed**
- **To overcome this python has a feature called **finally:****



This block gets executed though we get an error or not

Note: **Except block executes, only when **try** block has an error, but **finally** block executes, even though you get an exception.**

```
a=5
```

```
b=0
```

```
try:
```

```

        print("resource open")
        print(a/b)
        k=int(input("enter a number"))
        print(k)
except ZeroDivisionError as e:
    print("the value can not be divided by zero",e)
finally:
    print("resource closed")

```

Output:

```

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py
resource open
the value can not be divided by zero division by zero
resource closed

```

- **change the value of b to 2 for above program, you see the output like**

```

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py
resource open
2.5
enter a number 6
6
resource closed

```

- **Instead give input as some character or string for above program, check the output**

```

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py
resource open
2.5
enter a number p
resource closed

```

Traceback (most recent call last):

File "C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py", line 7, in <module>

```

    k=int(input("enter a number"))

```

ValueError: invalid literal for int() with base 10: ' p'

```

#-----
a=5
b=0

try:
    print("resource open")
    print(a/b)
    k=int(input("enter a number"))
    print(k)
except ZeroDivisionError as e:
    print("the value can not be divided by zero",e)
except ValueError as e:
    print("invalid input")
except Exception as e:
    print("something went wrong...",e)

finally:
    print("resource closed")

```

Output:

```

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex11.py
resource open
the value can not be divided by zero division by zero
resource closed

```

- **Change the value of b to 2 and give the input as some character or string (other than int)**

```

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex12.py
resource open
2.5
enter a number p
invalid input
resource closed

```

Modules (Date, Time, os, calendar, math):

- Modules refer to a file containing Python statements and definitions.

- We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.
- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.
- **Modular programming** refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or **modules**.

Advantages :

- **Simplicity:** Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem. If you're working on a single module, you'll have a smaller problem domain to wrap your head around. This makes development easier and less error-prone.
- **Maintainability:** Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program. This makes it more viable for a team of many programmers to work collaboratively on a large application.
- **Reusability:** Functionality defined in a single module can be easily reused (through an appropriately defined interface) by other parts of the application. This eliminates the need to recreate duplicate code.
- **Scoping:** Modules typically define a separate **namespace**, which helps avoid collisions between identifiers in different areas of a program.
- **Functions, modules and packages** are all constructs in Python that promote code modularization.

A file containing Python code, for e.g.: example.py, is called a module and its module name would be example.

```
>>> def add(a,b):
    result=a+b
    return result
    """This program adds two numbers and return the result"""
```

```
example.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/example...
File Edit Format Run Options Window Help
def add(a,b):
    """This program adds two numbers and return the result"""
    result=a+b
    return result
```

Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

How to import the module is:

- We can import the definitions inside a module to another module or the Interactive interpreter in Python.
- We use the import keyword to do this. To import our previously defined module example we type the following in the Python prompt.
- Using the module name we can access the function using dot (.) operation. For Eg:

```
>>> import example
```

```
>>> example.add(5,5)
```

```
10
```

- Python has a ton of standard modules available. Standard modules can be imported the same way as we import our user-defined modules.

Reloading a module:

```
def hi(a,b):
```

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

```
hi(4,4)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/add.py
```

```
8
```

```
>>> import add
```

```
8
```

```
>>> import add
```

```
>>> import add
```

```
>>>
```

Python provides a neat way of doing this. We can use the `reload()` function inside the `imp` module to reload a module. This is how its done.

- `>>> import imp`
- `>>> import my_module`
- This code got executed `>>> import my_module >>> imp.reload(my_module)` This code got executed `<module 'my_module' from '!.\\my_module.py'>` how its done.

```
>>> import imp
```

```
>>> import add
```

```
>>> imp.reload(add)
```

```
8
```

```
<module 'add' from 'C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy\\add.py'>
```

The `dir()` built-in function

```
>>> import example
```

```
>>> dir(example)
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'add']
```

```
>>> dir()
```

```
['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '__warningregistry__', 'add', 'example', 'hi', 'imp']
```

It shows all built-in and user-defined modules.

For ex:

```
>>> example.__name__
```

```
'example'
```

Datetime module:

Write a python program to display date, time

```
>>> import datetime
```

```
>>> a=datetime.datetime(2019,5,27,6,35,40)
```

```
>>> a
```

```
datetime.datetime(2019, 5, 27, 6, 35, 40)
```

write a python program to display date

```
import datetime
```

```
a=datetime.date(2000,9,18)
```

```
print(a)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =
```

```
2000-09-18
```

write a python program to display time

```
import datetime
```

```
a=datetime.time(5,3)
```

```
print(a)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =
```

```
05:03:00
```

#write a python program to print date, time for today and now.

```
import datetime
```

```
a=datetime.datetime.today()
```

```
b=datetime.datetime.now()
```

```
print(a)
```

```
print(b)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =
```

```
2019-11-29 12:49:52.235581
```

```
2019-11-29 12:49:52.235581
```

#write a python program to add some days to your present date and print the date added.

```
import datetime
```

```
a=datetime.date.today()
```

```
b=datetime.timedelta(days=7)
```

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

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =
```

```
2019-12-06
```

#write a python program to print the no. of days to write to reach your birthday

```
import datetime
```

```
a=datetime.date.today()
```

```
b=datetime.date(2020,5,27)
```

```
c=b-a
```

```
print(c)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =

180 days, 0:00:00

#write an python program to print date, time using date and time functions

```
import datetime
```

```
t=datetime.datetime.today()
```

```
print(t.date())
```

```
print(t.time())
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =

2019-11-29

12:53:39.226763

Time module:

#write a python program to display time.

```
import time
```

```
print(time.time())
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =

1575012547.1584706

#write a python program to get structure of time stamp.

```
import time
```

```
print(time.localtime(time.time()))
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =

```
time.struct_time(tm_year=2019, tm_mon=11, tm_mday=29, tm_hour=13, tm_min=1,
tm_sec=15, tm_wday=4, tm_yday=333, tm_isdst=0)
```

#write a python program to make a time stamp.

```
import time

a=(1999,5,27,7,20,15,1,27,0)

print(time.mktime(a))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =
927769815.0
```

#write a python program using sleep().

```
import time

time.sleep(6)    #prints after 6 seconds

print("Python Lab")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =
Python Lab (#prints after 6 seconds)
```

os module:

```
>>> import os

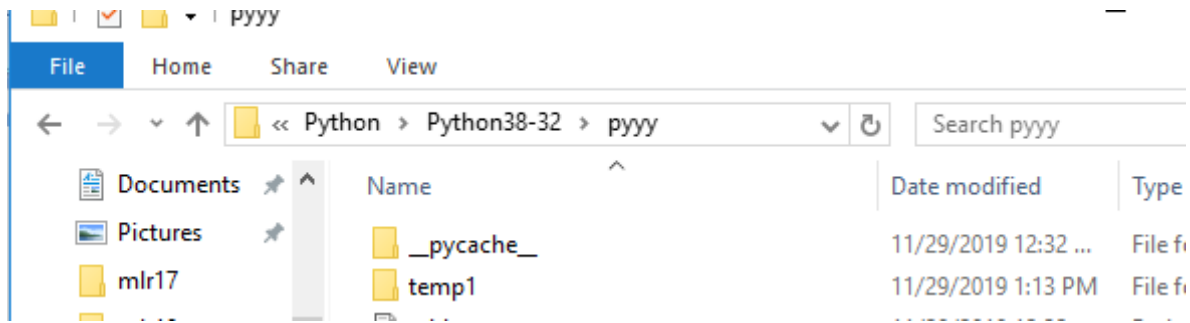
>>> os.name

'nt'

>>> os.getcwd()

'C:\\Users\\MRCET\\AppData\\Local\\Programs\\Python\\Python38-32\\pyyy'

>>> os.mkdir("temp1")
```



Note: temp1 dir is created

```
>>> os.getcwd()
```

```
'C:\\Users\\MRCET\\AppData\\Local\\Programs\\Python\\Python38-32\\pyyy'
```

```
>>> open("t1.py","a")
```

```
<_io.TextIOWrapper name='t1.py' mode='a' encoding='cp1252'>
```

```
>>> os.access("t1.py",os.F_OK)
```

```
True
```

```
>>> os.access("t1.py",os.W_OK)
```

```
True
```

```
>>> os.rename("t1.py","t3.py")
```

```
>>> os.access("t1.py",os.F_OK)
```

```
False
```

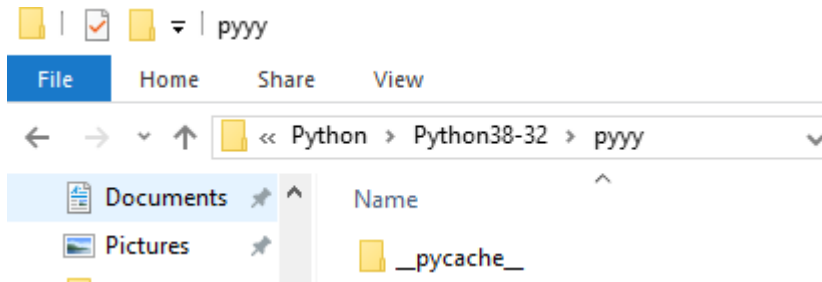
```
>>> os.access("t3.py",os.F_OK)
```

```
True
```

```
>>> os.rmdir('temp1')
```

(or)

```
os.rmdir('C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/temp1')
```



Note: Temp1dir is removed

```
>>> os.remove("t3.py")
```

Note: We can check with the following cmd whether removed or not

```
>>> os.access("t3.py",os.F_OK)
```

False

```
>>> os.listdir()
```

```
['add.py', 'ali.py', 'alia.py', 'arr.py', 'arr2.py', 'arr3.py', 'arr4.py', 'arr5.py', 'arr6.py', 'br.py', 'br2.py', 'bubb.py', 'bubb2.py', 'bubb3.py', 'bubb4.py', 'bubbdesc.py', 'clo.py', 'cmndlinarg.py', 'comm.py', 'con1.py', 'cont.py', 'cont2.py', 'd1.py', 'dic.py', 'e1.py', 'example.py', 'fl.y.py', 'flowof.py', 'fr.py', 'fr2.py', 'fr3.py', 'fu.py', 'fu1.py', 'if1.py', 'if2.py', 'ifelif.py', 'ifelse.py', 'iff.py', 'insertdesc.py', 'inserti.py', 'k1.py', 'l1.py', 'l2.py', 'link1.py', 'linklisttt.py', 'lis.py', 'listlooop.py', 'm1.py', 'merg.py', 'nesforr.py', 'nestedif.py', 'opprec.py', 'paraarg.py', 'qucksort.py', 'qukdsc.py', 'quu.py', 'r.py', 'rec.py', 'ret.py', 'rn.py', 's1.py', 'scoglo.py', 'selecasce.py', 'selectdecs.py', 'stk.py', 'strmodl.py', 'strr.py', 'strr1.py', 'strr2.py', 'strr3.py', 'strr4.py', 'strrmodl.py', 'wh.py', 'wh1.py', 'wh2.py', 'wh3.py', 'wh4.py', 'wh5.py', '__pycache__']
```

```
>>> os.listdir('C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32')
```

```
['argpar.py', 'br.py', 'bu.py', 'cmndlinarg.py', 'DLLs', 'Doc', 'fl.py', 'fl.txt', 'filess', 'functupretval.py', 'funturet.py', 'gtopt.py', 'include', 'Lib', 'libs', 'LICENSE.txt', 'lisparam.py', 'mysite', 'NEWS.txt', 'niru', 'python.exe', 'python3.dll', 'python38.dll', 'pythonw.exe', 'pyyy', 'Scripts', 'srp.py', 'sy.py', 'symod.py', 'tcl', 'the_weather', 'Tools', 'tupretval.py', 'vcruntime140.dll']
```

Calendar module:

#write a python program to display a particular month of a year using calendar module.

```
import calendar  
  
print(calendar.month(2020,1))
```

Output:

```
>>>  
= RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/c11.py  
    January 2020  
Mo Tu We Th Fr Sa Su  
      1  2  3  4  5  
 6  7  8  9 10 11 12  
13 14 15 16 17 18 19  
20 21 22 23 24 25 26  
27 28 29 30 31  
,
```

Activate Windows

write a python program to check whether the given year is leap or not.

```
import calendar  
  
print(calendar.isleap(2021))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/c11.py  
  
False
```

#write a python program to print all the months of given year.

```
import calendar  
  
print(calendar.calendar(2020,1,1,1))
```

Output:

```
>>>
= RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32,
2020
```

January							February							March						
Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su
		1	2	3	4	5						1	2							1
6	7	8	9	10	11	12	3	4	5	6	7	8	9	2	3	4	5	6	7	8
13	14	15	16	17	18	19	10	11	12	13	14	15	16	9	10	11	12	13	14	15
20	21	22	23	24	25	26	17	18	19	20	21	22	23	16	17	18	19	20	21	22
27	28	29	30	31			24	25	26	27	28	29	23	24	25	26	27	28	29	
													30	31						

April							May							June							
Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	
		1	2	3	4	5						1	2	3	1	2	3	4	5	6	7
6	7	8	9	10	11	12	4	5	6	7	8	9	10	8	9	10	11	12	13	14	
13	14	15	16	17	18	19	11	12	13	14	15	16	17	15	16	17	18	19	20	21	
20	21	22	23	24	25	26	18	19	20	21	22	23	24	22	23	24	25	26	27	28	
27	28	29	30				25	26	27	28	29	30	31	29	30						

July							August							September						
Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su
		1	2	3	4	5						1	2	1	2	3	4	5	6	
6	7	8	9	10	11	12	3	4	5	6	7	8	9	7	8	9	10	11	12	13
13	14	15	16	17	18	19	10	11	12	13	14	15	16	14	15	16	17	18	19	20
20	21	22	23	24	25	26	17	18	19	20	21	22	23	21	22	23	24	25	26	27
27	28	29	30	31			24	25	26	27	28	29	30	28	29	30				
							31													

October							November							December						
Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su
			1	2	3	4							1	1	2	3	4	5	6	
5	6	7	8	9	10	11	2	3	4	5	6	7	8	7	8	9	10	11	12	13
12	13	14	15	16	17	18	9	10	11	12	13	14	15	14	15	16	17	18	19	20
19	20	21	22	23	24	25	16	17	18	19	20	21	22	21	22	23	24	25	26	27
26	27	28	29	30	31		23	24	25	26	27	28	29	28	29	30	31			
							30													

Activate Windows

math module:

write a python program which accepts the radius of a circle from user and computes the area.

```
import math

r=int(input("Enter radius:"))

area=math.pi*r*r

print("Area of circle is:",area)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/m.py =

Enter radius:4

Area of circle is: 50.26548245743669

```
>>> import math
```

```
>>> print("The value of pi is", math.pi)
```

O/P: The value of pi is 3.141592653589793

Import with renaming:

- We can import a module by renaming it as follows.
- **For Eg:**

```
>>> import math as m
```

```
>>> print("The value of pi is", m.pi)
```

O/P: The value of pi is 3.141592653589793

- We have renamed the math module as m. This can save us typing time in some cases.
- Note that the name math is not recognized in our scope. Hence, math.pi is invalid, m.pi is the correct implementation.

Python from...import statement:

- We can import specific names from a module without importing the module as a whole. Here is an example.

```
>>> from math import pi
```

```
>>> print("The value of pi is", pi)
```

O/P: The value of pi is 3.141592653589793

- We imported only the attribute pi from the module.
- In such case we don't use the dot operator. We could have imported multiple attributes as follows.

```
>>> from math import pi, e
```

```
>>> pi
```

```
3.141592653589793
```

```
>>> e
```

```
2.718281828459045
```

Import all names:

- We can import all names(definitions) from a module using the following construct.

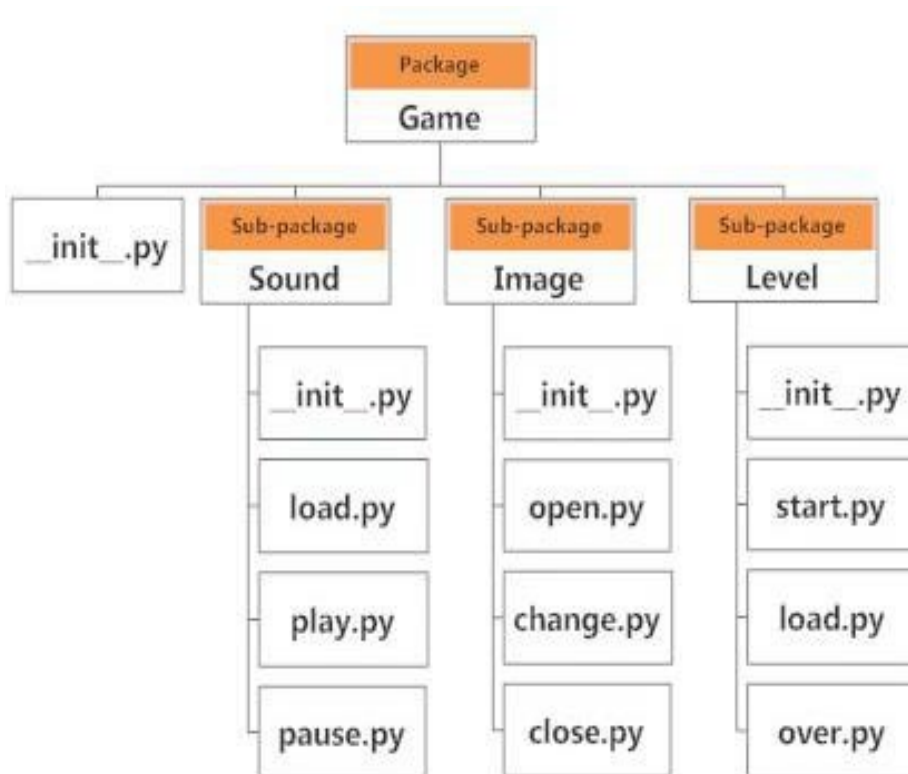
```
>>>from math import *
```

```
>>>print("The value of pi is", pi)
```

- We imported all the definitions from the math module. This makes all names except those beginning with an underscore, visible in our scope.

Explore packages:

- We don't usually store all of our files in our computer in the same location. We use a well-organized hierarchy of directories for easier access.
- Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.
- As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.
- Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.
- A directory must contain a file named `__init__.py` in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.
- Here is an example. Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.



- If a file named `_init_.py` is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.
- For example `__init__.py`
- A **module** in the package can access the global by importing it in turn
- We can import modules from packages using the dot (.) operator.
- For example, if want to import the start module in the above example, it is done as follows.
- `import Game.Level.start`
- Now if this module contains a function named `select_difficulty()`, we must use the full name to reference it.
- `Game.Level.start.select_difficulty(2)`

- If this construct seems lengthy, we can import the module without the package prefix as follows.
- `from Game.Level import start`
- We can now call the function simply as follows.
- **`start.select_difficulty(2)`**
- Yet another way of importing just the required function (or class or variable) from a module within a package would be as follows.
- **`from Game.Level.start import select_difficulty`**
- Now we can directly call this function.
- **`select_difficulty(2)`**

Examples:

#Write a python program to create a package (II YEAR),sub-package(CSE),modules(student) and create read and write function to module

```
def read():
```

```
    print("Department")
```

```
def write():
```

```
    print("Student")
```

Output:

```
>>> from IIYEAR.CSE import student
```

```
>>> student.read()
```

```
Department
```

```
>>> student.write()
```

```
Student
```

```
>>> from IIYEAR.CSE.student import read
```

```
>>> read
```

<function read at 0x03BD1070>

```
>>> read()
```

Department

```
>>> from IIYEAR.CSE.student import write
```

```
>>> write()
```

Student

Write a program to create and import module?

```
def add(a=4,b=6):
```

```
    c=a+b
```

```
    return c
```

Output:

C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\IIYEAR\modul.py

```
>>> from IIYEAR import modul
```

```
>>> modul.add()
```

10

Write a program to create and rename the existing module.

```
def a():
```

```
    print("hello world")
```

```
a()
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/IIYEAR/exam.py

hello world

```
>>> import exam as ex
```

hello world

Introduction to NumPy, Pandas, Exploratory Data Analysis (EDA), Data Science life cycle, Descriptive Statistics, Basic tools (plots, graphs and summary statistics) of EDA, Philosophy of EDA. Data Visualization: Scatter plot, bar chart, histogram, boxplot, heat maps, etc.

NumPy:-

NumPy is a Python library used for working with arrays.

It also has functions for working in domain of linear algebra, fourier transform, and matrices.

NumPy stands for Numerical Python.

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.

Arrays are very frequently used in data science.

Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

```
import numpy
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

output –

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

Dimensions in Arrays -

A dimension in arrays is one level of array depth (nested arrays).

0-D Arrays:

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

1-D Arrays:

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

2-D Arrays:

An array that has 1-D arrays as its elements is called a 2-D array.

3-D arrays:

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

Example

```
import numpy as np
```

```
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
```

```
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

output –

```
0
1
2
3
```

Array Indexing -

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

output – 1

```
[1 2]
1
```

Array Slicing -

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: `[start:end]`.

We can also define the step, like this: `[start:end:step]`.

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

```
arr = np.array([1, 2, 3, 4])  
print(arr[0:2])
```

output -

[1,2]

Use the minus operator to refer to an index from the end.

```
import numpy as np  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
print(arr[-3:-1])
```

output –

[5,6]

Introduction to Pandas:-

Pandas is a Software Library in Computer Programming and it is written for the Python Programming Language its work to do data analysis and manipulation.

import pandas as pd

Here, pd is referred to as an alias to the Pandas. However, it is not necessary to import the library using the alias, it just helps in writing less amount code every time a method or property is called.

Pandas generally provide two data structures for manipulating data, They are:

- Series

- DataFrame

Series:

Pandas Series is a one-dimensional labelled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called indexes. Pandas Series is nothing but a column in an excel sheet. Labels need not be unique but must be a hashable type.

The object supports both integer and label-based indexing and provides a host of methods for performing operations involving the index.

```
import pandas as pd
import numpy as np
ser = pd.Series()
print(ser)
data = np.array(['g', 'e', 'e', 'k', 's'])
ser = pd.Series(data)
print(ser)
```

output -

```
Series([], dtype: float64)
0    g
1    e
2    e
3    k
4    s
dtype: object
```

DataFrame:

Pandas DataFrame is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the data, rows, and columns.

```
import pandas as pd
```

```
studentData = {
    'name' : ['jack', 'Riti', 'Aadi'],
    'age' : [34, 30, 16],
    'city' : ['Sydney', 'Delhi', 'New york'] }
```

```
dfObj = pd.DataFrame(studentData)
print(dfObj)
```

output –

	age	city	name
0	34	Sydney	jack
1	30	Delhi	Riti
2	16	New york	Aadi

Exploratory Data Analysis (EDA):-

What is exploratory data analysis?

The “exploratory” aspect means that your understanding of the problem you are solving, or might solve, is changing as you go.

Exploratory data analysis (EDA) is used by data scientists to analyze and investigate data sets and summarize their main characteristics, often employing data visualization methods. It helps determine how best to manipulate data sources to get the answers you need, making it easier for data scientists to discover patterns, spot anomalies, test a hypothesis, or check assumptions.

EDA assists Data science professionals in various ways:-

- 1 Getting a better understanding of data
- 2 Identifying various data patterns
- 3 Getting a better understanding of the problem statement

Why is exploratory data analysis important in data science?

The main purpose of EDA is to help look at data before making any assumptions. It can help identify obvious errors, as well as better understand patterns within the data, detect outliers or anomalous events, find interesting relations among the variables.

Data scientists can use exploratory analysis to ensure the results they produce are valid and applicable to any desired business outcomes and goals. EDA also helps stakeholders by confirming they are asking the right questions. EDA can help answer questions about standard deviations, categorical variables, and confidence intervals. Once EDA is complete and insights are drawn, its features can then be used for more sophisticated data analysis or modeling, including machine learning.

Exploratory data analysis tools

Specific statistical functions and techniques you can perform with EDA tools include:

- Clustering and dimension reduction techniques, which help create graphical displays of high-dimensional data containing many variables.
- Univariate visualization of each field in the raw dataset, with summary statistics.
- Bivariate visualizations and summary statistics that allow you to assess the relationship between each variable in the dataset and the target variable you're looking at.
- Multivariate visualizations, for mapping and understanding interactions between different fields in the data.
- K-means Clustering is a clustering method in unsupervised learning where data points are assigned into K groups, i.e. the number of clusters, based on the distance from each group's centroid. The data points closest to a particular centroid will be clustered under the same category. K-means Clustering is commonly used in market segmentation, pattern recognition, and image compression.
- Predictive models, such as linear regression, use statistics and data to predict outcomes.

Types of exploratory data analysis

There are four primary types of EDA:

- **Univariate non-graphical.** This is simplest form of data analysis, where the data being analyzed consists of just one variable. Since it's a single variable, it doesn't deal with

causes or relationships. The main purpose of univariate analysis is to describe the data and find patterns that exist within it.

- **Univariate graphical.** Non-graphical methods don't provide a full picture of the data. Graphical methods are therefore required. Common types of univariate graphics include:
 - Stem-and-leaf plots, which show all data values and the shape of the distribution.
 - Histograms, a bar plot in which each bar represents the frequency (count) or proportion (count/total count) of cases for a range of values.
 - Box plots, which graphically depict the five-number summary of minimum, first quartile, median, third quartile, and maximum.
- **Multivariate nongraphical:** Multivariate data arises from more than one variable. Multivariate non-graphical EDA techniques generally show the relationship between two or more variables of the data through cross-tabulation or statistics.
- **Multivariate graphical:** Multivariate data uses graphics to display relationships between two or more sets of data. The most used graphic is a grouped bar plot or bar chart with each group representing one level of one of the variables and each bar within a group representing the levels of the other variable.

Other common types of multivariate graphics include:

- Scatter plot, which is used to plot data points on a horizontal and a vertical axis to show how much one variable is affected by another.
- Multivariate chart, which is a graphical representation of the relationships between factors and a response.
- Run chart, which is a line graph of data plotted over time.
- Bubble chart, which is a data visualization that displays multiple circles (bubbles) in a two-dimensional plot.
- Heat map, which is a graphical representation of data where values are depicted by color.

Some of the most common data science tools used to create an EDA include:

- **Python:** An interpreted, object-oriented programming language with dynamic semantics. Its high-level, built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for rapid application development, as well as for use as a scripting or glue language to connect existing components together. Python and EDA can be used together to identify missing values in a data set, which is important so you can decide how to handle missing values for machine learning.

- **R:** An open-source programming language and free software environment for statistical computing and graphics supported by the R Foundation for Statistical Computing. The R language is widely used among statisticians in data science in developing statistical observations and data analysis.

Data Science Lifecycle:-

Data Science Lifecycle revolves around the use of machine learning and different analytical strategies to produce insights and predictions from information in order to acquire a commercial enterprise objective. The complete method includes a number of steps like data cleaning, preparation, modelling, model evaluation, etc. It is a lengthy procedure and may additionally take quite a few months to complete. So, it is very essential to have a generic structure to observe for each and every hassle at hand.

Let us understand what is the need for Data Science?

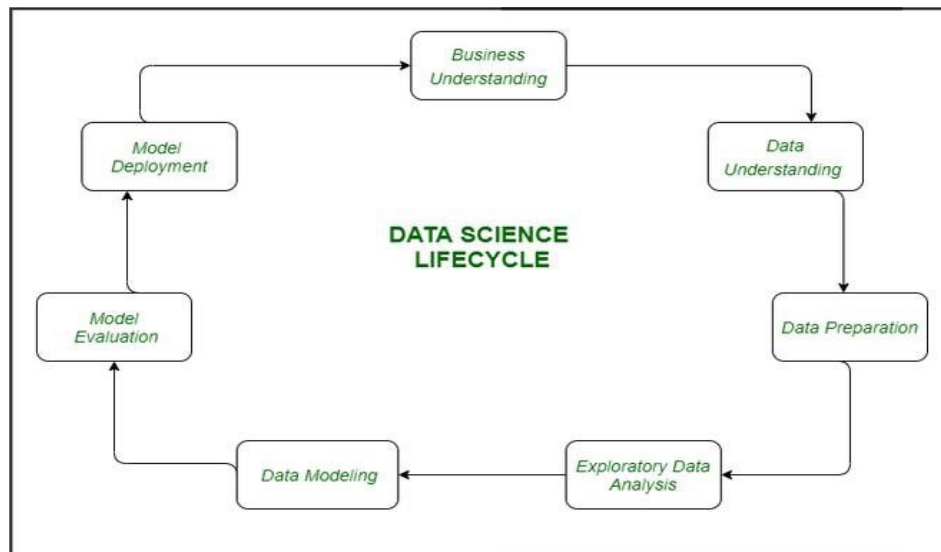
Earlier data used to be much less and generally accessible in a well-structured form, that we could save effortlessly and easily in Excel sheets, and with the help of Business Intelligence tools data can be processed efficiently. But Today we used to deals with large amounts of data every day, which ultimately results in an explosion of records and data.

So it is a very big challenge for any organization to deal with such a massive amount of data generating every second. For handling and evaluating this data we required some very powerful, complex algorithms and technologies and this is where Data science comes into the picture.

The following are some primary motives for the use of Data science technology:

- It helps to convert the big quantity of uncooked and unstructured records into significant insights.
- It can assist in unique predictions such as a range of surveys, elections, etc.
- It also helps in automating transportation such as growing a self-driving car, we can say which is the future of transportation.
- Companies are shifting towards Data science and opting for this technology. Amazon, Netflix, etc, which cope with the big quantity of data, are the use of information science algorithms for higher consumer experience.

The lifecycle of Data Science



1. Business Understanding:

The complete cycle revolves around the enterprise goal. What will you resolve if you do no longer have a specific problem? It is extraordinarily essential to apprehend the commercial enterprise goal sincerely due to the fact that will be your ultimate aim of the analysis. After desirable perception only we can set the precise aim of evaluation that is in sync with the enterprise objective. You need to understand if the customer desires to minimize savings loss, or if they prefer to predict the rate of a commodity, etc.

2. Data Understanding:

After enterprise understanding, the subsequent step is data understanding. This includes a series of all the reachable data. Here you need to intently work with the commercial enterprise group as they are certainly conscious of what information is present, what facts should be used for this commercial enterprise problem, and different information. This step includes describing the data, their structure, their relevance, their records type. Explore the information using graphical plots. Basically, extracting any data that you can get about the information through simply exploring the data.

3. Preparation of Data:

Next comes the data preparation stage. This consists of steps like choosing the applicable data, integrating the data by means of merging the data sets, cleaning it, treating the lacking values through either eliminating them or imputing them, treating inaccurate data through eliminating them, additionally test for outliers the use of box plots and cope with them. Constructing new data, derive new elements from present ones. Format the data into the preferred structure, eliminate undesirable columns and features. Data preparation is the most time-consuming but arguably the most essential step in the complete existence cycle. Your model will be as accurate as your data.

4. Exploratory Data Analysis:

This step includes getting some concept about the answer and elements affecting it, earlier than constructing the real model. Distribution of data inside distinctive variables of a character is explored graphically the usage of bar-graphs, Relations between distinct aspects are captured via graphical representations like scatter plots and warmth maps. Many data visualization strategies are considerably used to discover each and every characteristic individually and by means of combining them with different features.

5. Data Modeling:

Data modeling is the coronary heart of data analysis. A model takes the organized data as input and gives the preferred output. This step consists of selecting the suitable kind of model, whether the problem is a classification problem, or a regression problem or a clustering problem. After deciding on the model family, amongst the number of algorithms amongst that family, we need to cautiously pick out the algorithms to put into effect and enforce them. We need to tune the hyperparameters of every model to obtain the preferred performance. We additionally need to make positive there is the right stability between overall performance and generalizability. We do no longer desire the model to study the data and operate poorly on new data.

6. Model Evaluation:

Here the model is evaluated for checking if it is geared up to be deployed. The model is examined on an unseen data, evaluated on a cautiously thought out set of assessment metrics. We additionally need to make positive that the model conforms to reality. If we do not acquire a quality end result in the evaluation, we have to re-iterate the complete modelling procedure until the preferred stage of metrics is achieved. Any data science solution, a machine learning model, simply like a human, must evolve, must be capable to enhance itself with new data,

adapt to a new evaluation metric. We can construct more than one model for a certain phenomenon, however, a lot of them may additionally be imperfect. The model assessment helps us select and construct an ideal model.

7. Model Deployment:

The model after a rigorous assessment is at the end deployed in the preferred structure and channel. This is the last step in the data science life cycle. Each step in the data science life cycle defined above must be laboured upon carefully. If any step is performed improperly, and hence, have an effect on the subsequent step and the complete effort goes to waste. For example, if data is no longer accumulated properly, you'll lose records and you will no longer be constructing an ideal model. If information is not cleaned properly, the model will no longer work. If the model is not evaluated properly, it will fail in the actual world. Right from Business perception to model deployment, every step has to be given appropriate attention, time, and effort.

Descriptive Statistics:-

Statistics is a branch of mathematics that deals with collecting, interpreting, organization and interpretation of data.

Within statistics, there are two main categories:

1. Descriptive Statistics: In Descriptive Statistics your are describing, presenting, summarizing and organizing your data (population), either through numerical calculations or graphs or tables.

2. Inferential statistics: Inferential Statistics are produced by more complex mathematical calculations, and allow us to infer trends and make assumptions and predictions about a population based on a study of a sample taken from it.

Data availability in today's world is a big boon. However, analyzing them to our needs is the biggest challenge. To be able to analyze the vast resources of data, understanding and describing the data is crucial.

There are different methods through which we can describe data.

What is Descriptive Statistics?

Descriptive Statistics, as the name suggests, describes data. It is a method to collect, organize, summarize, display and analyze sample data taken from a population. Descriptive Statistics, unlike inferential statistics, is not based on probability theory. It paves the way to understand and visualize data better.

Types of Descriptive Statistics

Descriptive Statistics is classified into Measures of Central Tendency and Measure of Dispersion.

A. Measures of Central Tendency

1. Mean/ Average

This measure of central tendency summarizes the data, by considering a value which is an estimate of the total data set. It helps us to ascertain the spread in variables between the minimum and maximum values.

Sample Mean

$$\bar{x} = \frac{\sum x_i}{n}$$

Population Mean

$$\mu = \frac{\sum x_i}{N}$$

Sample Data: 12,18,25,69,45

Sample Mean: [(12+18+25+69+45)/5] = 33.80

Population Data: 55,46,78,12,18,33,28,45,25,69,66

Population Mean: [(55+46+78+12+18+33+28+45+25+69+66)/11] = 43.18

2. Median

- Median is the middle item in a data set arranged in ascending/descending order.
- If there are n observations then the Median = (n+1)/2 th observation.
- Computational Rule.
- If n is odd, then (n+1)/2 is an integer.
- If n is even, then use an average of n/2 and (n/2) +1th observation.

3. Mode

- Mode is the highest occurring observation.
- The greatest frequency can occur at two or more different values.
- If the data has only two modes, the data is bimodal.
- If the data has more than two modes, the data is multimodal.

4. Percentiles and Quartiles

- The P^{th} percentile in the ordered set is that value below which lies $P\%$ (P percent) of the observations in the set.
- The position of the P^{th} percentile is given by $(n + 1) P/100$, where n is the number of observations in the set.
- Quartiles are special names to percentiles.

Q1 = 25th percentile

Q2 = 50th percentile = median

Q3 = 75th percentile

B. Measures of Dispersion

1. Range

- The range of a data set is the difference between the largest and smallest data values.
- It is the simplest measure of variability.
- It is very sensitive to the smallest and largest data values.
- $\text{Range} = X_{\text{max}} - X_{\text{min}}$

2. Interquartile Range (IQR)

- The interquartile range of a data set is the difference between the third quartile and the first quartile.
- It is the range for the middle 50% of the data.
- It overcomes the sensitivity to extreme data values.

3. Variance

- The variance is a measure of variability that utilizes all the data.

- It is based on the difference between the value of each observation (x_i) and the mean ($\bar{x}$ for a sample, μ for a population).

$$\sigma^2 = \frac{\sum (x_i - \mu)^2}{N}$$

< - Population variance

Sample variance - >

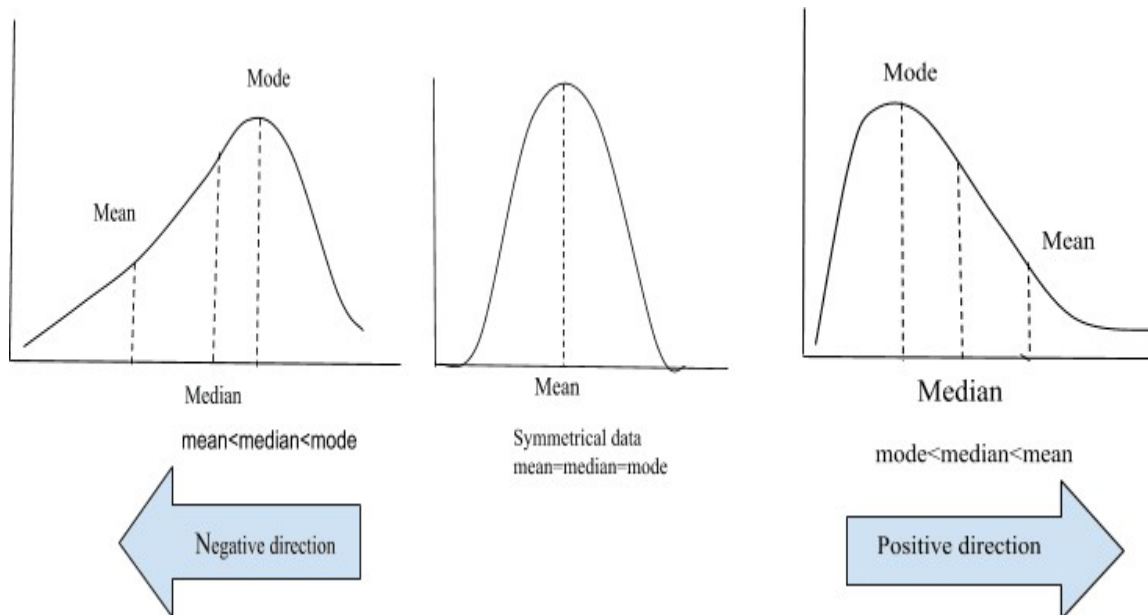
$$s^2 = \frac{\sum (x_i - \bar{x})^2}{n-1}$$

4. Standard Deviation

- The standard deviation of a data set is the positive square root of the variance.
- It is measured in the same units as the data, making it more easily comparable, than the variance, to the mean.
- If the data set is a sample, the standard deviation is denoted s .
- If the data set is a population, the standard deviation is denoted σ (sigma).

Skewness -

Skewness characterizes the degree of asymmetry of a distribution around its mean.



The concept of skewness is mainly used to understand the distribution of the data and steps taken to normalize the data for further building of machine learning models.

In case of negatively skewed data, Mean<Median<Mode. This indicates, more data points are to the right of the curve where the data has very high values in large numbers.
In case of positively skewed data, Mode<Median<Mean. This means that more data points are to the left of the curve where the data has very low values in large numbers.

Measure of Skewness

$$\gamma_1 = \frac{\sum (X - \mu)^3}{N\sigma^3}$$

Kurtosis -

Kurtosis characterizes the symmetric distribution through the relative peaks or flatness of the curve.

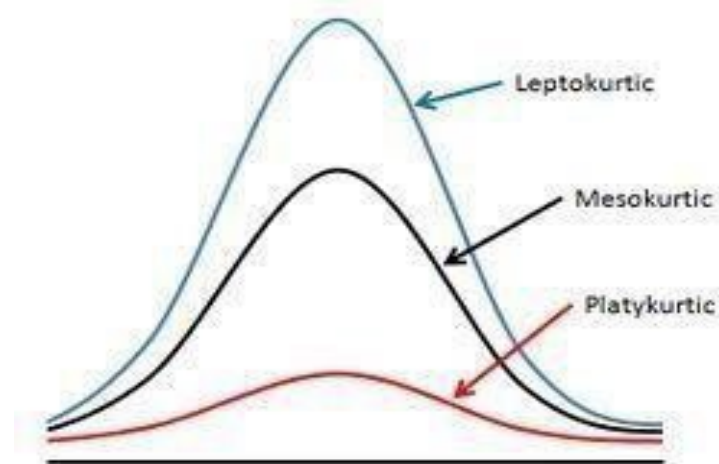
The main difference between Skewness and Kurtosis is:

Skewness measures the degree of the slope in the frequency distribution.

Kurtosis measures the degree of thickness in the tails of the distribution curve.

There are 3 types of Kurtosis:

- Platykurtic (relatively flat)
- Mesokurtic (normal)
- Leptokurtic (relatively peaked)



Measure of Kurtosis

$$\eta = \beta_2 - 3$$

Where,

$$\beta_2 = \frac{\sum (X - \mu)^4}{N\sigma^4}$$

is the fourth standardized moment or the fourth degree.

Basic tools of EDA:-

The basic tools of EDA are plots, graphs and summary statistics.

Generally speaking, it's a method of systematically going through the data, plotting distributions of all variables (using box plots), plotting time series of data, transforming variables, looking at all pairwise relationships between variables using scatterplot matrices, and generating summary statistics for all of them. At the very least that would mean computing their mean, minimum, maximum, the upper and lower quartiles, and identifying outliers.

But as much as EDA is a set of tools, it's also a mindset. And that mindset is about your relationship with the data. You want to understand the data—gain intuition, understand the shape of it, and try to connect your understanding of the process that generated the data to the data itself. EDA happens between you and the data and isn't about proving anything to anyone else yet.

Typical graphical tools used in EDA are:

- Box plot
- Histogram
- Multi-vari chart
- Scatter plot
- Stem-and-leaf plot

Note:

Also explain about different 'plots' in Machine Learning and 'Descriptive statistics'.

Philosophy of EDA:-

There are important reasons anyone working with data should do EDA. Namely, to gain intuition about the data; to make comparisons between distributions; for sanity checking

(making sure the data is on the scale you expect, in the format you thought it should be); to find out where data is missing or if there are outliers; and to summarize the data.

In the context of data generated from logs, EDA also helps with debugging the logging process. For example, “patterns” you find in the data could actually be something wrong in the logging process that needs to be fixed. If you never go to the trouble of debugging, you’ll continue to think your patterns are real. The engineers we’ve worked with are always grateful for help in this area.

In the end, EDA helps you make sure the product is performing as intended.

Although there’s lots of visualization involved in EDA, we distinguish between EDA and data visualization in that EDA is done toward the beginning of analysis, and data visualization as it’s used in our vernacular, is done toward the end to communicate one’s findings. With EDA, the graphics are solely done for you to understand what’s going on. With EDA, you can also use the understanding you get to inform and improve the development of algorithms. For example, suppose you are trying to develop a ranking algorithm that ranks content that you are showing to users. To do this you might want to develop a notion of “popular.”

Before you decide how to quantify popularity (which could be, for example, highest frequency of clicks, or the post with the most number of comments, or comments above some threshold, or some weighted average of many metrics), you need to understand how the data is behaving, and the best way to do that is looking at it and getting your hands dirty.

Plotting data and making comparisons can get you extremely far, and is far better to do than getting a dataset and immediately running a regression just because you know how. It’s been a disservice to analysts and data scientists that EDA has not been enforced as a critical part of the process of working with data.

Data Visualization:-

Data visualization is the process of creating interactive visuals to understand trends, variations, and derive meaningful insights from the data.

This visualization technique aims to identify the Patterns, Trends, Correlations, and Outliers of data sets.

Data visualization is used mainly for data checking and cleaning, exploration and discovery, and communicating results to business stakeholders.

The basic uses of the Data Visualization technique are as follows:

- It is a powerful technique to explore the data with **presentable** and **interpretable** results.
- In the **data mining process**, it acts as a primary step in the pre-processing portion.
- It supports the **data cleaning process** by finding incorrect data and corrupted or missing values.
- It also helps to **construct and select variables**, which means we have to determine which variable to include and discard in the analysis.
- In the process of **Data Reduction**, it also plays a crucial role while combining the categories.

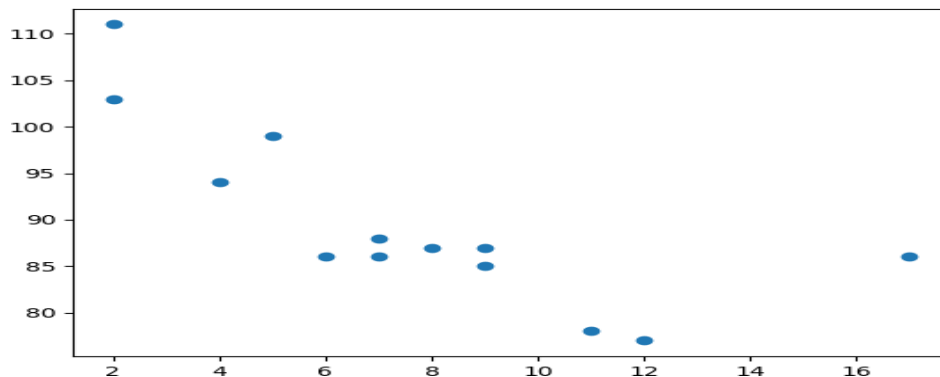
Scatter plot –

A scatter plot is a chart type that is normally used to observe and visually display the relationship between variables. The values of the variables are represented by dots. The positioning of the dots on the vertical and horizontal axis will inform the value of the respective data point; hence, scatter plots make use of Cartesian coordinates to display the values of the variables in a data set.

The relationships observed can either be positive or negative, non-linear or linear, and/or, strong or weak.

The data points or dots, which appear on a scatter plot, represent the individual values of each of the data points and also allow pattern identification when looking at the data holistically.

```
import matplotlib.pyplot as plt
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
plt.scatter(x, y)
plt.show()
```



The x-axis represents ages, and the y-axis represents speeds.

What we can read from the diagram is that the two fastest cars were both 2 years old, and the slowest car was 12 years old.

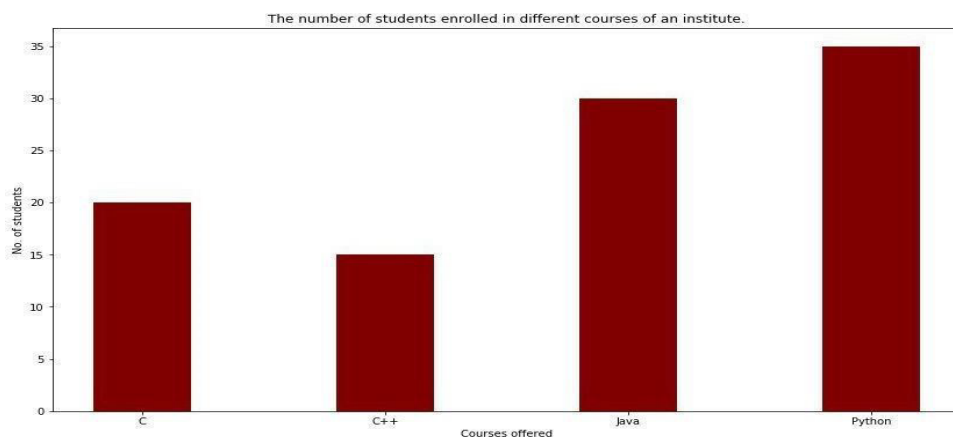
Bar chart –

A bar plot or bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. The bar plots can be plotted horizontally or vertically. A bar chart describes the comparisons between the discrete categories. One of the axis of the plot represents the specific categories being compared, while the other axis represents the measured values corresponding to those categories.

The matplotlib API in Python provides the `bar()` function which can be used in MATLAB style use or as an object-oriented API.

```
import numpy as np
import matplotlib.pyplot as plt

data = {'C':20, 'C++':15, 'Java':30, 'Python':35}
courses = list(data.keys())
values = list(data.values())
# creating the bar plot
plt.bar(courses, values, color='maroon',width = 0.4)
plt.xlabel("Courses offered")
plt.ylabel("No. of students enrolled")
plt.title("Students enrolled in different courses")
plt.show()
```



Histogram –

A histogram is basically used to represent data provided in a form of some groups. It is an accurate method for the graphical representation of numerical data distribution. It is a type of bar plot where X-axis represents the bin ranges while Y-axis gives information about frequency.

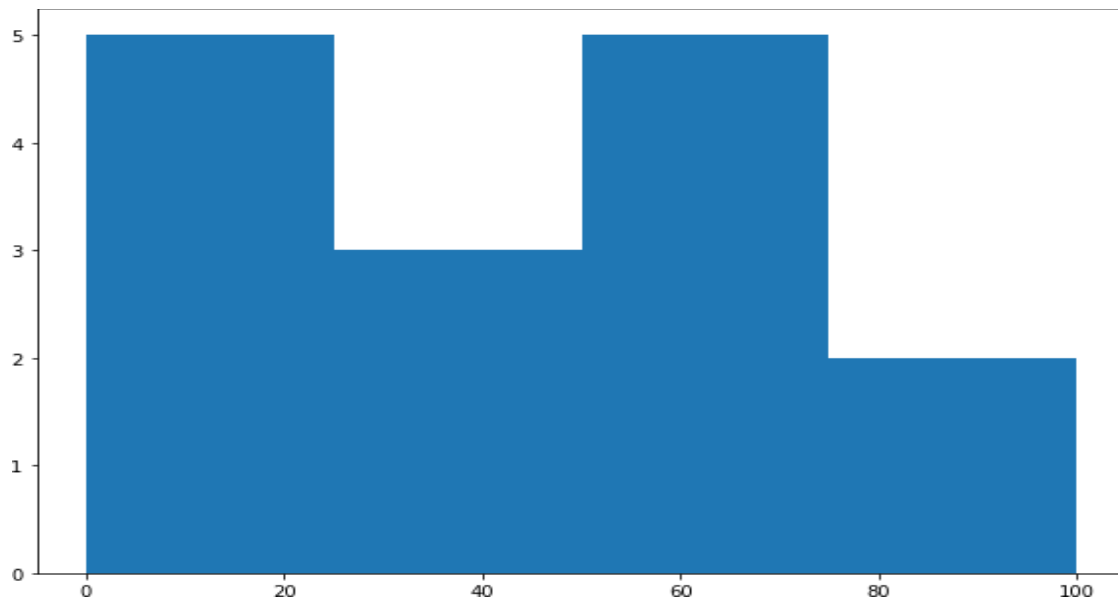
Creating a Histogram

To create a histogram the first step is to create bins of the ranges, then distribute the whole range of the values into a series of intervals, and count the values which fall into each of the intervals. Bins are clearly identified as consecutive, non-overlapping intervals of variables. The `matplotlib.pyplot.hist()` function is used to compute and create histogram of `x`.

```
from matplotlib import pyplot as plt
import numpy as np

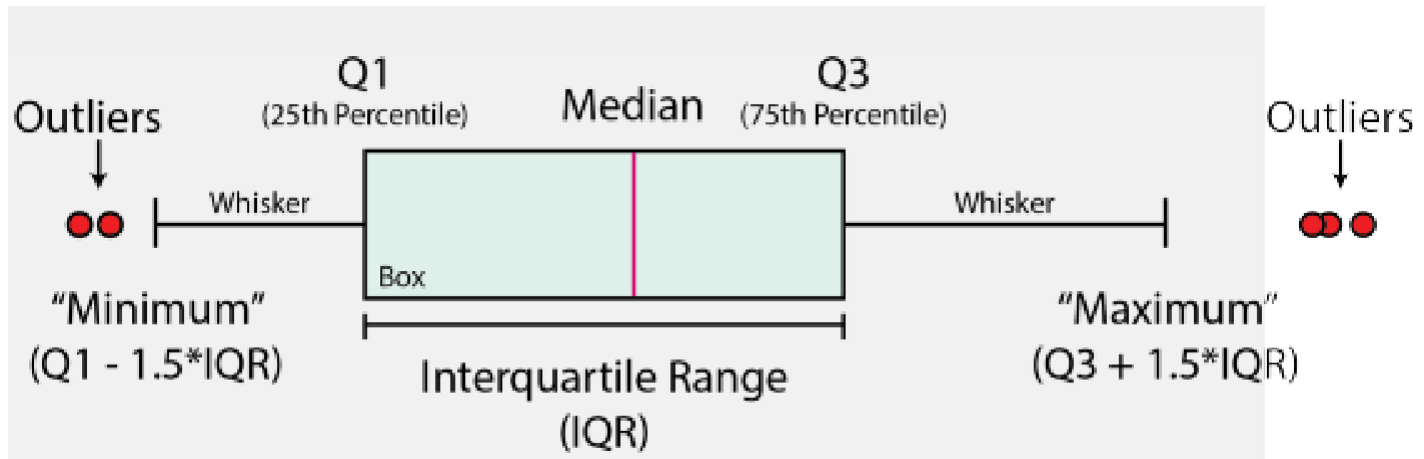
a = np.array([22, 87, 5, 43, 56, 73, 55, 54, 11, 20, 51, 5, 79, 31, 27])
# Creating histogram
plt.hist(a, bins = [0, 25, 50, 75, 100])

# Show plot
plt.show()
```



Boxplot –

A boxplot is a graphical and standardised way to display the distribution of data based on five key numbers: The “minimum”, 1st Quartile (25th percentile), median (2nd Quartile./ 50th Percentile), the 3rd Quartile (75th percentile), and the “maximum”. The minimum and maximum values are defined as $Q1 - 1.5 * IQR$ and $Q3 + 1.5 * IQR$ respectively. Any points that fall outside of these limits are referred to as outliers.



Graphical depiction of a boxplot highlighting key components, including the median, quartiles, outliers, and Interquartile Range. Image created by author.

Boxplots can be used to:

- Identify outliers or anomalous data points
- To determine if our data is skewed
- To understand the spread/range of the data
- To construct a boxplot, we first start with the median value / 50th percentile (Q2). This represents the middle value within our data.
- A box is then formed between the 25th and 75th percentiles (Q1 and Q3 respectively). The range represented by this box is known as the interquartile range (IQR).
- From this box extends lines, which are also known as the whiskers. These extend to $Q1 - 1.5 * IQR$ and $Q3 + 1.5 * IQR$ or to the last data point if it is less than this value.

Heat maps –

Heatmap is defined as a graphical representation of data using colors to visualize the value of the matrix. In this, to represent more common values or higher activities brighter colors basically reddish colors are used and to represent less common or activity values, darker colors are preferred. Heatmap is also defined by the name of the shading matrix. Heatmaps in Seaborn can be plotted by using the `seaborn.heatmap()` function.

Syntax:

```
seaborn.heatmap(data, *, vmin=None, vmax=None, cmap=None, center=None,  
annot_kws=None, linewidths=0, linecolor='white', cbar=True, **kwargs)
```

Important Parameters:

`data`: 2D dataset that can be coerced into an ndarray.

`vmin`, `vmax`: Values to anchor the colormap, otherwise they are inferred from the data and other keyword arguments.

`cmap`: The mapping from data values to color space.

`center`: The value at which to center the colormap when plotting divergent data.

`annot`: If True, write the data value in each cell.

`fmt`: String formatting code to use when adding annotations.

`linewidths`: Width of the lines that will divide each cell.

`linecolor`: Color of the lines that will divide each cell.

`cbar`: Whether to draw a colorbar.

All the parameters except data are optional.

S. No	Topic	Pg.No
1	INTRODUCTION TO BIG DATA AND ANALYTICS Classification of Digital Data, Structured and Unstructured Data -Introduction to Big Data	1
2	Why Big Data Traditional Business Intelligence versus Big Data - DataWarehouse and Hadoop	4
3	Environment Big Data Analytics: Classification of Analytics – Challenges - Big Data Analytics importance	5
4	Data Science - Data Scientist - Terminologies used in Big Data Environme	10
5	Basically, Available Soft State Eventual Consistency -Top Analytics Tools	12
7	INTRODUCTION TO TECHNOLOGY LANDSCAPE NoSQL, Comparison of SQL and NoSQL, Hadoop -RDBMS VersusHadoop - Distributed Computing	15
8	Challenges – Hadoop Overview - Hadoop Distributed File System - Processing Data with Hadoop -	20
9	Managing Resources and Applications with Hadoop YARN -Interacting with Hadoop Ecosystem	22
10	INTRODUCTION TO MONGODB AND MAPREDUCEPROGRAMMING MongoDB: Why Mongo DB - Terms used in RDBMS and Mongo DB - Data Types - MongoDB Query Language	25
11	MapReduce: Mapper – Reducer – Combiner – Partitioner – Searching –Sorting – Compression	36
12	INTRODUCTION TO HIVE AND PIG Hive: Introduction – Architecture - Data Types - File Formats - HiveQuery Language Statements	66
13	Partitions – Bucketing – Views - Sub- Query – Joins – Aggregations -Group by and Having - RCFile	70
14	Implementation – Hive User Defined Function - Serialization andDeserialization. Pig: Introduction	75
15	Anatomy – Features – Philosophy - Use Case for Pig - Pig LatinOverview - Pig Primitive Data Types	76
16	Running Pig - Execution Modes of Pig - HDFS Commands -RelationalOperators - Eval Function	79
17	Complex Data Types - Piggy Bank - User-Defined Functions -Parameter Substitution – Diagnostic	82

18	IV	Operator - Word Count Example using Pig - Pig at Yahoo! - Pig VersusHive	93
19	V	INTRODUCTION TO DATA ANALYTICS WITH R Machine Learning: Introduction, Supervised Learning, Unsupervised Learning, Machine Learning	83
20	V	Algorithms: Regression Model, Clustering, Collab orativeFiltering, Associate Rule Making, Decision Tree, Big Data Analytics with BigR.	97

What is Big Data?

According to Gartner, the definition of Big Data –

“Big data” is high-volume, velocity, and variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making.”

This definition clearly answers the “What is Big Data?” question – Big Data refers to complex and large data sets that have to be processed and analyzed to uncover valuable information that can benefit businesses and organizations.

However, there are certain basic tenets of Big Data that will make it even simpler to answer what is Big Data:

- It refers to a massive amount of data that keeps on growing exponentially with time.
- It is so voluminous that it cannot be processed or analyzed using conventional data processing techniques.
- It includes data mining, data storage, data analysis, data sharing, and data visualization.
- The term is an all-comprehensive one including data, data frameworks, along with the tools and techniques used to process and analyze the data.

The History of Big Data

Although the concept of big data itself is relatively new, the origins of large data sets go back to the 1960s and '70s when the world of data was just getting started with the first data centers and the development of the relational database.

Around 2005, people began to realize just how much data users generated through Facebook, YouTube, and other online services. Hadoop (an open-source framework created specifically to store and analyze big data sets) was developed that same year. NoSQL also began to gain popularity during this time.

The development of open-source frameworks, such as Hadoop (and more recently, Spark) was essential for the growth of big data because they make big data easier to work with and cheaper to store. In the years since then, the volume of big data has skyrocketed. Users are still generating huge amounts of data—but it's not just humans who are doing it.

With the advent of the Internet of Things (IoT), more objects and devices are connected to the internet, gathering data on customer usage patterns and product performance. The emergence of machine learning has produced still more data.

While big data has come far, its usefulness is only just beginning. Cloud computing has expanded big data possibilities even further. The cloud offers truly elastic scalability, where developers can simply spin up ad hoc clusters to test a subset of data.

Benefits of Big Data and Data Analytics

- Big data makes it possible for you to gain more complete answers because you have more information.
- More complete answers mean more confidence in the data—which means a completely different approach to tackling problems.

Types of Big Data

Now that we are on track with what is big data, let's have a look at the types of big data:

a) Structured

Structured is one of the types of big data and By structured data, we mean data that can be processed, stored, and retrieved in a fixed format. It refers to highly organized information that can be readily and seamlessly stored and accessed from a database by simple search engine algorithms. **For instance, the employee table in a company database will be structured as the employee details, their job positions, their salaries, etc.,** will be present in an organized manner.

b) Unstructured

Unstructured data refers to the data that lacks any specific form or structure whatsoever. This makes it very difficult and time-consuming to process and analyze unstructured data. Email is an example of unstructured data. Structured and unstructured are two important types of big data.

c) Semi-structured

Semi structured is the third type of big data. Semi-structured data pertains to the data containing both the formats mentioned above, that is, structured and unstructured data. To be precise, it refers to the data that although has not been classified under a particular repository (database), yet contains vital information or tags that segregate individual elements within the data. Thus we come to the end of types of data.

Characteristics of Big Data

Back in 2001, Gartner analyst Doug Laney listed the **3 'V's of Big Data – Variety, Velocity, and Volume**. Let's discuss the characteristics of big data. These characteristics, isolated, are enough to know what big data is. Let's look at them in depth:

a) Variety

Variety of Big Data refers to structured, unstructured, and semi-structured data that is gathered from multiple sources. While in the past, data could only be collected from spreadsheets and databases, today data comes in an array of forms such as emails, PDFs, photos, videos, audios, SM posts, and so much more. Variety is one of the important characteristics of big data.

b) Velocity

Velocity essentially refers to the speed at which data is being created in real-time. In a broader prospect, it comprises the rate of change, linking of incoming data sets at varying speeds, and activity bursts.

c) Volume

Volume is one of the characteristics of big data. We already know that Big Data indicates huge 'volumes' of data that is being generated on a daily basis from various sources like social media platforms, business processes, machines, networks, human interactions, etc. Such a large amount of data is stored in data warehouses. Thus comes to the end of characteristics of big data.

Why is Big Data Important?

The importance of big data does not revolve around how much data a company has but how a company utilizes the collected data. Every company uses data in its own way; the more efficiently a company uses its data, the more potential it has to grow. The company can take data from any source and analyze it to find answers which will enable:

1. **Cost Savings:** Some tools of Big Data like Hadoop and Cloud-Based Analytics can bring cost advantages to business when large amounts of data are to be stored and these tools also help in identifying more efficient ways of doing business.
2. **Time Reductions:** The high speed of tools like Hadoop and in-memory analytics can easily identify new sources of data which helps businesses analyzing data immediately and make quick decisions based on the learning.
3. **Understand the market conditions:** By analyzing big data you can get a better understanding of current market conditions. For example, by analyzing customers' purchasing behaviors, a company can find out the products that are sold the most and produce products according to this trend. By this, it can get ahead of its competitors.
4. **Control online reputation:** Big data tools can do sentiment analysis. Therefore, you can get feedback about who is saying what about your company. If you want to monitor and improve the online presence of your business, then, big data tools can help in all this.
5. **Using Big Data Analytics to Boost Customer Acquisition and Retention**
The customer is the most important asset any business depends on. There is no single business that can claim success without first having to establish a solid customer base. However, even with a customer base, a business cannot afford to disregard the high competition it faces. If a business is slow to learn what customers are looking for, then it is very easy to begin offering poor quality products. In the end, loss of clientele will result, and this creates an adverse overall effect on business success. The use of big data allows businesses to observe various customer related patterns and trends. Observing customer behavior is important to trigger loyalty.
6. **Using Big Data Analytics to Solve Advertisers Problem and Offer Marketing Insights**

Big data analytics can help change all business operations. This includes the ability to match customer expectation, changing company's product line and of course ensuring that the marketing campaigns are powerful.

7. Big Data Analytics As a Driver of Innovations and Product Development

Another huge advantage of big data is the ability to help companies innovate and redevelop their products.

Business Intelligence vs Big Data

Although Big Data and Business Intelligence are two technologies used to analyze data to help companies in the decision-making process, there are differences between both of them. They differ in the way they work as much as in the type of data they analyze.

Traditional BI methodology is based on the principle of grouping all business data into a central server. Typically, this data is analyzed in offline mode, after storing the information in an environment called Data Warehouse. The data is structured in a conventional relational database with an additional set of indexes and forms of access to the tables (multidimensional cubes).

A Big Data solution differs in many aspects to BI to use. These are the main differences between Big Data and Business Intelligence:

1. In a Big Data environment, information is stored on a distributed file system, rather than on a central server. It is a much safer and more flexible space.
2. Big Data solutions carry the processing functions to the data, rather than the data to the functions. As the analysis is centered on the information, it's easier to handle larger amounts of information in a more agile way.
3. Big Data can analyze data in different formats, both structured and unstructured. The volume of unstructured data (those not stored in a traditional database) is growing at levels much higher than the structured data. Nevertheless, its analysis carries different challenges. Big Data solutions solve them by allowing a global analysis of various sources of information.
4. Data processed by Big Data solutions can be historical or come from real-time sources. Thus, companies can make decisions that affect their business in an agile and efficient way.
5. Big Data technology uses parallel mass processing (MPP) concepts, which improves the speed of analysis. With MPP many instructions are executed simultaneously, and since the various jobs are divided into several parallel execution parts, at the end the overall results are reunited and presented. This allows you to analyze large volumes of information quickly.

Big Data vs Data Warehouse

Big Data has become the reality of doing business for organizations today. There is a boom in the amount of structured as well as raw data that floods every organization daily. If this data is managed well, it can lead to powerful insights and quality decision making.

Big data analytics is the process of examining large data sets containing a variety of data types to discover some knowledge in databases, to identify interesting patterns and establish relationships to solve problems, market trends, customer preferences, and other useful information. Companies and businesses that implement Big Data Analytics often reap several business benefits. Companies implement Big Data Analytics because they want to make more informed business decisions.

A data warehouse (DW) is a collection of corporate information and data derived from operational systems and external data sources. A data warehouse is designed to support business decisions by allowing data consolidation, analysis and reporting at different aggregate levels. Data is populated into the Data Warehouse through the processes of extraction, transformation and loading (ETL tools). Data analysis tools, such as business intelligence software, access the data within the warehouse.

Hadoop Environment Big Data Analytics

Hadoop is changing the perception of handling Big Data especially the unstructured data. Let's know how Apache Hadoop software library, which is a framework, plays a vital role in handling Big Data. Apache Hadoop enables surplus data to be streamlined for any distributed processing system across clusters of computers using simple programming models. It truly is made to scale up from single servers to a large number of machines, each and every offering local computation, and storage space. Instead of depending on hardware to provide high-availability, the library itself is built to detect and handle breakdowns at the application layer, so providing an extremely available service along with a cluster of computers, as both versions might be vulnerable to failures.

Hadoop Community Package Consists of

- File system and OS level abstractions
- A MapReduce engine (either MapReduce or YARN)
- The Hadoop Distributed File System (HDFS)
- Java ARchive (JAR) files
- Scripts needed to start Hadoop
- Source code, documentation and a contribution section

Activities performed on Big Data

- **Store** – Big data need to be collected in a seamless repository, and it is not necessary to store in a single physical database.
- **Process** – The process becomes more tedious than traditional one in terms of cleansing, enriching, calculating, transforming, and running algorithms.
- **Access** – There is no business sense of it at all when the data cannot be searched, retrieved easily, and can be virtually showcased along the business lines.

Classification of analytics

Descriptive analytics

Descriptive analytics is a statistical method that is used to search and summarize historical data in order to identify patterns or meaning.

Data aggregation and **data mining** are two techniques used in descriptive analytics to discover historical data. Data is first gathered and sorted by data aggregation in order to make the datasets more manageable by analysts.

Data mining describes the next step of the analysis and involves a search of the data to identify patterns and meaning. Identified patterns are analyzed to discover the specific ways that learners interacted with the learning content and within the learning environment.

Advantages:

- Quickly and easily report on the Return on Investment (ROI) by showing how performance achieved business or target goals.
- Identify gaps and performance issues early - before they become problems.
- Identify specific learners who require additional support, regardless of how many students or employees there are.
- Identify successful learners in order to offer positive feedback or additional resources.
- Analyze the value and impact of course design and learning resources.

Predictive analytics

Predictive Analytics is a statistical method that utilizes algorithms and machine learning to identify trends in data and predict future behaviors

The software for predictive analytics has moved beyond the realm of statisticians and is becoming more affordable and accessible for different markets and industries, including the field of learning & development.

For online learning specifically, predictive analytics is often found incorporated in the Learning Management System (LMS), but can also be purchased separately as specialized software.

For the learner, predictive forecasting could be as simple as a dashboard located on the main screen after logging in to access a course. Analyzing data from past and current progress, visual indicators in the dashboard could be provided to signal whether the employee was on track with training requirements.

Advantages:

- **Personalize the training needs** of employees by identifying their gaps, strengths, and weaknesses; specific learning resources and training can be offered to support individual needs.
- **Retain Talent** by tracking and understanding employee career progression and forecasting what skills and learning resources would best benefit their career paths. Knowing what skills employees need also benefits the design of future training.
- **Support employees** who may be falling behind or not reaching their potential by offering intervention support before their performance puts them at risk.
- **Simplified reporting** and visuals that keep everyone updated when predictive forecasting is required.

Prescriptive analytics

Prescriptive analytics is a statistical method used to generate recommendations and make decisions based on the computational findings of algorithmic models.

Generating automated decisions or recommendations requires specific and unique algorithmic models and clear direction from those utilizing the analytical technique. A recommendation cannot be generated without knowing what to look for or what problem is desired to be solved. In this way, prescriptive analytics begins with a problem.

Example

A Training Manager uses predictive analysis to discover that most learners without a particular skill will not complete the newly launched course. What could be done? Now prescriptive analytics can be of assistance on the matter and help determine options for action. Perhaps an algorithm can detect the learners who require that new course, but lack that particular skill, and send an automated recommendation that they take an additional training resource to acquire the missing skill.

The accuracy of a generated decision or recommendation, however, is only as good as the quality of data and the algorithmic models developed. What may work for one company's training needs may not make sense when put into practice in another company's training department. Models are generally recommended to be tailored for each unique situation and need.

Descriptive vs Predictive vs Prescriptive Analytics

Descriptive Analytics is focused solely on historical data.

You can think of Predictive Analytics as then using this historical data to develop statistical models that will then forecast about future possibilities.

Prescriptive Analytics takes Predictive Analytics a step further and takes the possible forecasted outcomes and predicts consequences for these outcomes.

What Big Data Analytics Challenges

1. Need For Synchronization Across Disparate Data Sources

As data sets are becoming bigger and more diverse, there is a big challenge to incorporate them into an analytical platform. If this is overlooked, it will create gaps and lead to wrong messages and insights.

2. Acute Shortage Of Professionals Who Understand Big Data Analysis

The analysis of data is important to make this voluminous amount of data being produced in every minute, useful. With the exponential rise of data, a huge demand for big data scientists and Big Data analysts has been created in the market. It is important for business organizations to hire a data scientist having skills that are varied as the job of a data scientist is multidisciplinary. Another major challenge faced by businesses is the shortage of professionals who understand Big Data analysis. There is a sharp shortage of data scientists in comparison to the massive amount of data being produced.

3. Getting Meaningful Insights Through The Use Of Big Data Analytics

It is imperative for business organizations to gain important insights from Big Data analytics, and also it is important that only the relevant department has access to this information. A big challenge faced by the companies in the Big Data analytics is mending this wide gap in an effective manner.

4. Getting Voluminous Data Into The Big Data Platform

It is hardly surprising that data is growing with every passing day. This simply indicates that business organizations need to handle a large amount of data on daily basis. The amount and variety of data available these days can overwhelm any data engineer and that is why it is considered vital to make data accessibility easy and convenient for brand owners and managers.

5. Uncertainty Of Data Management Landscape

With the rise of Big Data, new technologies and companies are being developed every day. However, a big challenge faced by the companies in the Big Data analytics is to find out which technology will be best suited to them without the introduction of new problems and potential risks.

6. Data Storage And Quality

Business organizations are growing at a rapid pace. With the tremendous growth of the companies and large business organizations, increases the amount of data produced. The storage of this massive amount of data is becoming a real challenge for everyone. Popular data storage options like data lakes/ warehouses are commonly used to gather and store large quantities of unstructured and structured data in its native format. The real problem arises when a data lakes/ warehouse try to combine unstructured and inconsistent data from diverse sources, it encounters errors. Missing data, inconsistent data, logic conflicts, and duplicates data all result in data quality challenges.

7. Security And Privacy Of Data

Once business enterprises discover how to use Big Data, it brings them a wide range of possibilities and opportunities. However, it also involves the potential risks associated with big data when it comes to the privacy and the security of the data. The Big Data tools used for analysis and storage utilizes the data disparate sources. This eventually leads to a high risk of exposure of the data, making it vulnerable. Thus, the rise of voluminous amount of data increases privacy and security concerns.

Terminologies Used In Big Data Environments

- **As-a-service infrastructure**

Data-as-a-service, software-as-a-service, platform-as-a-service – all refer to the idea that rather than selling data, licences to use data, or platforms for running Big Data technology, it can be provided “as a service”, rather than as a product. This reduces the upfront capital investment

necessary for customers to begin putting their data, or platforms, to work for them, as the provider bears all of the costs of setting up and hosting the infrastructure. As a customer, as-a-service infrastructure can greatly reduce the initial cost and setup time of getting Big Data initiatives up and running.

- **Data science**

Data science is the professional field that deals with turning data into value such as new insights or predictive models. It brings together expertise from fields including statistics, mathematics, computer science, communication as well as domain expertise such as business knowledge. Data scientist has recently been voted the No 1 job in the U.S., based on current demand and salary and career opportunities.

- **Data mining**

Data mining is the process of discovering insights from data. In terms of Big Data, because it is so large, this is generally done by computational methods in an automated way using methods such as decision trees, clustering analysis and, most recently, machine learning. This can be thought of as using the brute mathematical power of computers to spot patterns in data which would not be visible to the human eye due to the complexity of the dataset.

- **Hadoop**

Hadoop is a framework for Big Data computing which has been released into the public domain as open source software, and so can freely be used by anyone. It consists of a number of modules all tailored for a different vital step of the Big Data process – from file storage (Hadoop File System – HDFS) to database (HBase) to carrying out data operations (Hadoop MapReduce – see below). It has become so popular due to its power and flexibility that it has developed its own industry of retailers (selling tailored versions), support service providers and consultants.

- **Predictive modelling**

At its simplest, this is predicting what will happen next based on data about what has happened previously. In the Big Data age, because there is more data around than ever before, predictions are becoming more and more accurate. Predictive modelling is a core component of most Big Data initiatives, which are formulated to help us choose the course of action which will lead to the most desirable outcome. The speed of modern computers and the volume of data available means that predictions can be made based on a huge number of variables, allowing an ever-increasing number of variables to be assessed for the probability that it will lead to success.

- **MapReduce**

MapReduce is a computing procedure for working with large datasets, which was devised due to difficulty of reading and analysing really Big Data using conventional computing methodologies. As its name suggest, it consists of two procedures – mapping (sorting information into the format needed for analysis – i.e. sorting a list of people according to their age) and reducing (performing an operation, such checking the age of everyone in the dataset to see who is over 21).

- **NoSQL**

NoSQL refers to a database format designed to hold more than data which is simply arranged into tables, rows, and columns, as is the case in a conventional relational database. This database format has proven very popular in Big Data applications because Big Data is often messy, unstructured and does not easily fit into traditional database frameworks.

- **Python**

Python is a programming language which has become very popular in the Big Data space due to its ability to work verywell with large, unstructured datasets (see Part II for the difference between structured and unstructured data). It is considered to be easier to learn for a data science beginner than other languages such as R (see also Part II) and more flexible.

- **R Programming**

R is another programming language commonly used in Big Data, and can be thought of as more specialised than Python, being geared towards statistics. Its strength lies in its powerful handling of structured data. Like Python, it has an active community of users who are constantly expanding and adding to its capabilities by creating new libraries and extensions.

- **Recommendation engine**

A recommendation engine is basically an algorithm, or collection of algorithms, designed to match an entity (for example, a customer) with something they are looking for. Recommendation engines used by the likes of Netflix or Amazon heavily rely on Big Data technology to gain an overview of their customers and, using predictive modelling, match them with products to buy or content to consume. The economic incentives offered by recommendation engines has been a driving force behind a lot of commercial Big Data initiatives and developments over the last decade.

- **Real-time**

Real-time means “as it happens” and in Big Data refers to a system or process which is able to give data-driven insights based on what is happening at the present moment. Recent years have seen a large push for the development of systems capable of processing and offering insights in real-time (or near-real-time), and advances in computing power as well as development of techniques such as machine learning have made it a reality in many applications today.

- **Reporting**

The crucial “last step” of many Big Data initiative involves getting the right information to the people who need it to make decisions, at the right time. When this step is automated, analytics is applied to the insights themselves to ensure that they are communicated in a way that they will be understood and easy to act on. This will usually involve creating multiple reports based on the same data or insights but each intended for a different audience (for example, in-depth technical analysis for engineers, and an overview of the impact on the bottom line for c-level executives).

- **Spark**

Spark is another open source framework like Hadoop but more recently developed and more suited to handling cutting-edge Big Data tasks involving real time analytics and machine learning. Unlike Hadoop it does not include its own filesystem, though it is designed to work with Hadoop's HDFS or a number of other options. However, for certain data related processes it is able to calculate at over 100 times the speed of Hadoop, thanks to its in-memory processing capability. This means it is becoming an increasingly popular choice for projects involving deep learning, neural networks and other compute-intensive tasks.

- **Structured Data**

Structured data is simply data that can be arranged neatly into charts and tables consisting of rows, columns or multi-dimensioned matrixes. This is traditionally the way that computers have stored data, and information in this format can easily and simply be processed and mined for insights. Data gathered from machines is often a good example of structured data, where various data points – speed, temperature, rate of failure, RPM etc. – can be neatly recorded and tabulated for analysis.

- **Unstructured Data**

Unstructured data is any data which cannot easily be put into conventional charts and tables. This can include video data, pictures, recorded sounds, text written in human languages and a great deal more. This data has traditionally been far harder to draw insight from using computers which were generally designed to read and analyze structured information. However, since it has become apparent that a huge amount of value can be locked away in this unstructured data, great efforts have been made to create applications which are capable of understanding unstructured data – for example visual recognition and natural language processing.

- **Visualization**

Humans find it very hard to understand and draw insights from large amounts of text or numerical data – we can do it, but it takes time, and our concentration and attention is limited. For this reason effort has been made to develop computer applications capable of rendering information in a visual form – charts and graphics which highlight the most important insights which have resulted from our Big Data projects. A subfield of reporting (see above), visualizing is now often an automated process, with visualizations customized by algorithm to be understandable to the people who need to act or take decisions based on them.

Basic availability, Soft state and Eventual consistency

Basic availability implies continuous system availability despite network failures **and** tolerance to temporary **inconsistency**.

Soft state refers to **state** change without input which is required for **eventual consistency**.

Eventual consistency means that if no further updates are made to a given updated database item for long enough period of time, all users will see the same value for the updated item.

Top Analytics Tools

* **R** is a language for statistical computing and graphics. It also used for big data analysis. It provides a wide variety of statistical tests.

Features:

- Effective data handling and storage facility,
- It provides a suite of operators for calculations on arrays, in particular, matrices,
- It provides coherent, integrated collection of big data tools for data analysis
- It provides graphical facilities for data analysis which display either on-screen or on hardcopy

* **Apache Spark** is a powerful open source big data analytics tool. It offers over 80 high-level operators that make it easy to build parallel apps. It is used at a wide range of organizations to process large datasets.

Features:

- It helps to run an application in Hadoop cluster, up to 100 times faster in memory, and ten times faster on disk
- It offers lightning Fast Processing
- Support for Sophisticated Analytics
- Ability to Integrate with Hadoop and Existing Hadoop Data

* **Plotly** is an analytics tool that lets users create charts and dashboards to share online.

Features:

- Easily turn any data into eye-catching and informative graphics
- It provides audited industries with fine-grained information on data provenance
- Plotly offers unlimited public file hosting through its free community plan

* **Lumify** is a big data fusion, analysis, and visualization platform. It helps users to discover connections and explore relationships in their data via a suite of analytic options.

Features:

- It provides both 2D and 3D graph visualizations with a variety of automatic layouts

- It provides a variety of options for analyzing the links between entities on the graph
- It comes with specific ingest processing and interface elements for textual content, images, and videos
- Its spaces feature allows you to organize work into a set of projects, or workspaces
- It is built on proven, scalable big data technologies

* **IBM SPSS Modeler** is a predictive big data analytics platform. It offers predictive models and delivers to individuals, groups, systems and the enterprise. It has a range of advanced algorithms and analysis techniques.

Features:

- Discover insights and solve problems faster by analyzing structured and unstructured data
- Use an intuitive interface for everyone to learn
- You can select from on-premises, cloud and hybrid deployment options
- Quickly choose the best performing algorithm based on model performance

* **MongoDB** is a NoSQL, document-oriented database written in C, C++, and JavaScript. It is free to use and is an open source tool that supports multiple operating systems including Windows Vista (and later versions), OS X (10.7 and later versions), Linux, Solaris, and FreeBSD.

Its main features include Aggregation, Adhoc-queries, Uses BSON format, Sharding, Indexing, Replication, Server-side execution of javascript, Schemaless, Capped collection, MongoDB management service (MMS), load balancing and file storage.

Features:

- Easy to learn.
- Provides support for multiple technologies and platforms.
- No hiccups in installation and maintenance.
- Reliable and low cost.

NoSQL

NoSQL is a non-relational DMS, that does not require a fixed schema, avoids joins, and is easy to scale. NoSQL database is used for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example companies like Twitter, Facebook, Google that collect terabytes of user data every single day.

SQL

Structured Query language (SQL) **pronounced as "S-Q-L" or sometimes as "See-Quel"** is the standard language for dealing with Relational Databases. A relational database defines relationships in the form of tables.

SQL programming can be effectively used to insert, search, update, delete database records.

Comparison of SQL and NoSQL

Parameter	SQL	NOSQL
Definition	SQL databases are primarily called RDBMS or Relational Databases	NoSQL databases are primarily called as Non-relational or distributed database
Design for	Traditional RDBMS uses SQL syntax and queries to analyze and get the data for further insights. They are used for OLAP systems.	NoSQL database system consists of various kind of database technologies. These databases were developed in response to the demands presented for the development of the modern application.
Query Language	Structured query language (SQL)	No declarative query language
Type	SQL databases are table based databases	NoSQL databases can be document based, key-value pairs, graph databases
Schema	SQL databases have a predefined schema	NoSQL databases use dynamic schema for unstructured data.
Ability to scale	SQL databases are vertically scalable	NoSQL databases are horizontally scalable
Examples	Oracle, Postgres, and MS-SQL.	MongoDB, Redis, , Neo4j, Cassandra, Hbase.
Best suited for	An ideal choice for the complex query intensive environment.	It is not good fit complex queries.
Hierarchical data storage	SQL databases are not suitable for hierarchical data storage.	More suitable for the hierarchical data store as it supports key-value pair method.
Variations	One type with minor variations.	Many different types which include key-value stores, document databases, and graph databases.

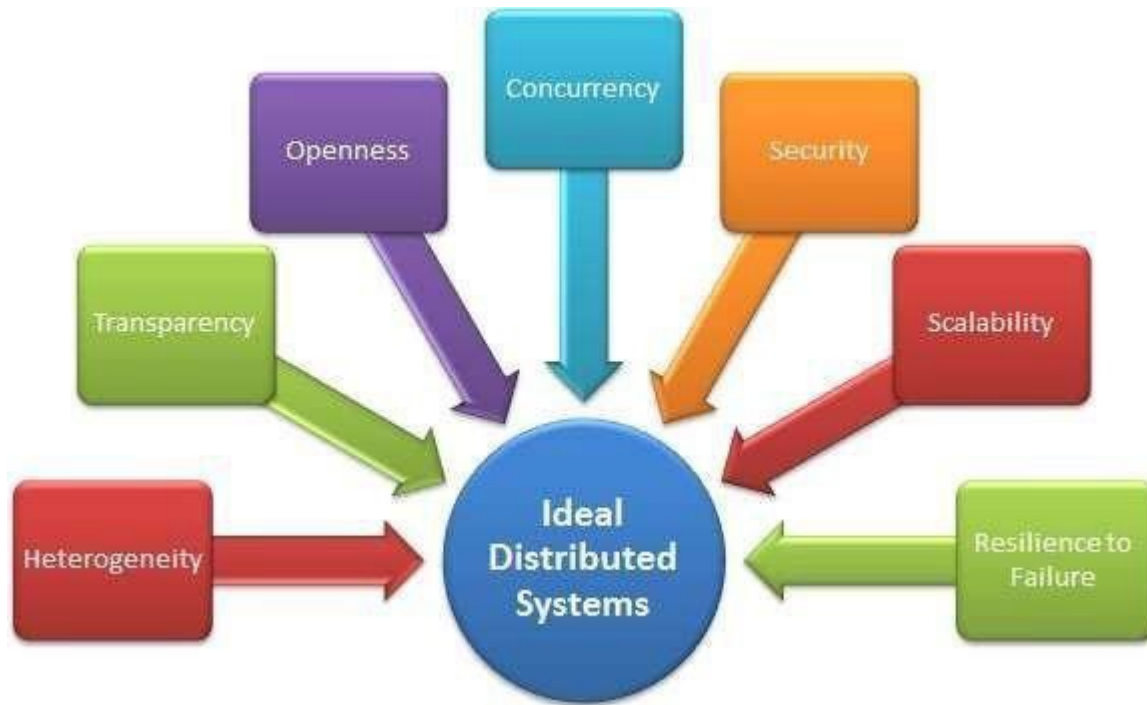
Development Year	It was developed in the 1970s to deal with issues with flat file storage	Developed in the late 2000s to overcome issues and limitations of SQL databases.
Open-source	A mix of open-source like Postgres & MySQL, and commercial like Oracle Database.	Open-source
Consistency	It should be configured for strong consistency.	It depends on DBMS as some offers strong consistency like MongoDB, whereas others offer only offers eventual consistency, like Cassandra.
Best Used for	RDBMS database is the right option for solving ACID problems.	NoSQL is a best used for solving data availability problems
Importance	It should be used when data validity is super important	Use when it's more important to have fast data than correct data
Best option	When you need to support dynamic queries	Use when you need to scale based on changing requirements
Hardware	Specialized DB hardware (Oracle Exadata, etc.)	Commodity hardware
Network	Highly available network (Infiniband, Fabric Path, etc.)	Commodity network (Ethernet, etc.)
Storage Type	Highly Available Storage (SAN, RAID, etc.)	Commodity drives storage (standard HDDs, JBOD)
Best features	Cross-platform support, Secure and free	Easy to use, High performance, and Flexible tool.
Top Companies Using	Hootsuite, CircleCI, Gauges	Airbnb, Uber, Kickstarter
Average salary	The average salary for any professional SQL Developer is \$84,328 per year in the U.S.A.	The average salary for "NoSQL developer" ranges from approximately \$72,174 per year
ACID vs. BASE Model	ACID(Atomicity, Consistency, Isolation, and Durability) is a standard for RDBMS	Base (Basically Available, Soft state, Eventually Consistent) is a model of many NoSQL systems

RDBMS Versus Hadoop

Criteria	Hadoop	RDBMS
Schema	Based on 'Schema on Read'.	Based on 'Schema on Write'.
Data Type	Structured, Semi-Structured and Unstructured data.	Structured Data.
Speed	Writes are Fast.	Reads are Fast.
Cost	Open source framework, free of cost.	Licensed software, Paid.
Application	Data discovery, Storage and processing of Unstructured data.	OLTP and complex ACID transaction.

Distributed Computing Challenges

Designing a distributed system does not come as easy and straight forward. A number of challenges need to be overcome in order to get the ideal system. The major challenges in distributed systems are listed below:



1. Heterogeneity:

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity (that is, variety and difference) applies to all of the following:

- Hardware devices: computers, tablets, mobile phones, embedded devices, etc.
- Operating System: Ms Windows, Linux, Mac, Unix, etc.
- Network: Local network, the Internet, wireless network, satellite links, etc.
- Programming languages: Java, C/C++, Python, PHP, etc.
- Different roles of software developers, designers, system managers

Different programming languages use different representations for characters and data structures such as arrays and records. These differences must be addressed if programs written in different languages are to be able to communicate with one another. Programs written by different developers cannot communicate with one another unless they use common standards, for example, for network communication and the representation of primitive data items and data structures in messages. For this to happen, standards need to be agreed and adopted – as have the Internet protocols.

Middleware: The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. Most middleware is implemented over the Internet protocols, which themselves mask the differences of the underlying networks, but all middleware deals with the differences in operating systems and hardware

Heterogeneity and mobile code: The term mobile code is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.

2. Transparency:

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. In other words, distributed systems designers must hide the complexity of the systems as much as they can. Some terms of transparency in distributed systems are:

Access Hide differences in data representation and how a resource is accessed

Location Hide where a resource is located

Migration Hide that a resource may move to another location

Relocation Hide that a resource may be moved to another location while in use

Replication Hide that a resource may be copied in several places

Concurrency Hide that a resource may be shared by several competitive users

Failure Hide the failure and recovery of a resource

Persistence Hide whether a (software) resource is in memory or a disk

3. Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be

made available for use by a variety of client programs. If the well-defined interfaces for a system are published, it is easier for developers to add new features or replace sub-systems in the future. Example: Twitter and Facebook have API that allows developers to develop their own software interactively.

4. Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time. For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems.

5. Security

Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components: **confidentiality** (protection against disclosure to unauthorized individuals), **integrity** (protection against alteration or corruption), **availability** for the authorized (protection against interference with the means to access the resources).

6. Scalability

Distributed systems must be scalable as the number of user increases. The scalability is defined by B. Clifford Neuman as

A system is said to be scalable if it can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity

Scalability has 3 dimensions:

- Size
 - Number of users and resources to be processed. Problem associated is overloading
- Geography
 - Distance between users and resources. Problem associated is communication reliability
- Administration
 - As the size of distributed systems increases, many of the system needs to be controlled. Problem associated is administrative mess

7. Failure Handling

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation. The handling of failures is particularly difficult.

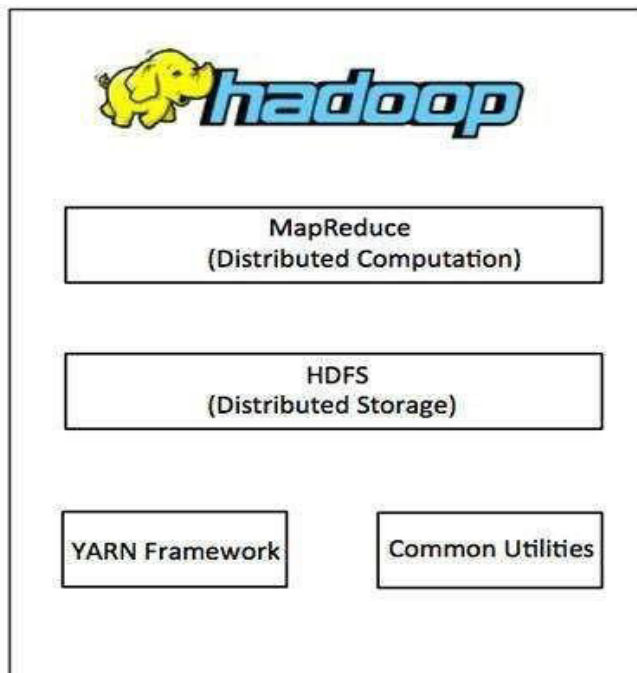
Hadoop Overview

Hadoop is an Apache open source framework written in java that allows distributed processing of large datasets across clusters of computers using simple programming models. The Hadoop framework application works in an environment that provides distributed *storage* and *computation* across clusters of computers. Hadoop is designed to scale up from single server to thousands of machines, each offering local computation and storage.

Hadoop Architecture

At its core, Hadoop has two major layers namely –

- Processing/Computation layer (MapReduce), and
- Storage layer (Hadoop Distributed File System).



MapReduce

MapReduce is a parallel programming model for writing distributed applications devised at Google for efficient processing of large amounts of data (multi-terabyte data-sets), on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. The MapReduce program runs on Hadoop which is an Apache open-source framework.

Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is based on the Google File System (GFS) and provides a distributed file system that is designed to run on commodity hardware. It has many

similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. It is highly fault-tolerant and is designed to be deployed on low-cost hardware. It provides high throughput access to application data and is suitable for applications having large datasets.

Apart from the above-mentioned two core components, Hadoop framework also includes the following two modules –

- **Hadoop Common** – These are Java libraries and utilities required by other Hadoop modules.
- **Hadoop YARN** – This is a framework for job scheduling and cluster resource management.

How Does Hadoop Work?

It is quite expensive to build bigger servers with heavy configurations that handle large scale processing, but as an alternative, you can tie together many commodity computers with single-CPU, as a single functional distributed system and practically, the clustered machines can read the dataset in parallel and provide a much higher throughput. Moreover, it is cheaper than one high-end server. So this is the first motivational factor behind using Hadoop that it runs across clustered and low-cost machines.

Hadoop runs code across a cluster of computers. This process includes the following core tasks that Hadoop performs –

- Data is initially divided into directories and files. Files are divided into uniform sized blocks of 128M and 64M (preferably 128M).
- These files are then distributed across various cluster nodes for further processing.
- HDFS, being on top of the local file system, supervises the processing.
- Blocks are replicated for handling hardware failure.
- Checking that the code was executed successfully.
- Performing the sort that takes place between the map and reduce stages.
- Sending the sorted data to a certain computer.
- Writing the debugging logs for each job.

Advantages of Hadoop

- Hadoop framework allows the user to quickly write and test distributed systems. It is efficient, and it automatic distributes the data and work across the machines and in turn, utilizes the underlying parallelism of the CPU cores.

- Hadoop does not rely on hardware to provide fault-tolerance and high availability (FTHA), rather Hadoop library itself has been designed to detect and handle failures at the application layer.
- Servers can be added or removed from the cluster dynamically and Hadoop continues to operate without interruption.
- Another big advantage of Hadoop is that apart from being open source, it is compatible on all the platforms since it is Java based.

Processing Data with Hadoop - Managing Resources and Applications with Hadoop YARN

Yarn divides the task on resource management and job scheduling/monitoring into separate daemons. There is one ResourceManager and per-application ApplicationMaster. An application can be either a job or a DAG of jobs.

The ResourceManager have two components – Scheduler and ApplicationManager.

The **scheduler** is a pure scheduler i.e. it does not track the status of running application. It only allocates resources to various competing applications. Also, it does not restart the job after failure due to hardware or application failure. The scheduler allocates the resources based on an abstract notion of a container. A container is nothing but a fraction of resources like CPU, memory, disk, network etc.

Following are the tasks of ApplicationManager:-

- Accepts submission of jobs by client.
- Negotiates first container for specific ApplicationMaster.
- Restarts the container after application failure.

Below are the responsibilities of ApplicationMaster

- Negotiates containers from Scheduler
- Tracking container status and monitoring its progress.

Yarn supports the concept of Resource Reservation via Reservation System. In this, a user can fix a number of resources for execution of a particular job over time and temporal constraints. The Reservation System makes sure that the resources are available to the job until its completion. It also performs admission control for reservation.

Yarn can scale beyond a few thousand nodes via Yarn Federation. YARN Federation allows to wire multiple sub-cluster into the single massive cluster. We can use many independent clusters together for a single large job. It can be used to achieve a large scale system.

Let us summarize how **Hadoop** works step by step:

- Input data is broken into blocks of size **128 Mb** and then blocks are moved to different nodes.
- Once all the blocks of the data are stored on data-nodes, the user can process the data.
- Resource Manager then schedules the program (submitted by the user) on individual nodes.
- Once all the nodes process the data, the output is written back to HDFS.

Interacting with Hadoop Ecosystem

Hadoop Ecosystem Hadoop has an ecosystem that has evolved from its three core components processing, resource management, and storage. In this topic, you will learn the components of the Hadoop ecosystem and how they perform their roles during Big Data processing. The Hadoop ecosystem is continuously growing to meet the needs of Big Data. It comprises the following twelve components:

- HDFS(Hadoop Distributed file system)
- HBase
- Sqoop
- Flume
- Spark
- Hadoop MapReduce
- Pig
- Impala
- Hive
- Cloudera Search
- Oozie
- Hue.

Let us understand the role of each component of the Hadoop ecosystem.

Components of Hadoop Ecosystem

Let us start with the first component HDFS of Hadoop Ecosystem.

HDFS (HADOOP DISTRIBUTED FILE SYSTEM)

- HDFS is a storage layer for Hadoop.
- HDFS is suitable for distributed storage and processing, that is, while the data is being stored, it first gets distributed and then it is processed.
- HDFS provides Streaming access to file system data.
- HDFS provides file permission and authentication.
- HDFS uses a command line interface to interact with Hadoop.

So what stores data in HDFS? It is the HBase which stores data in HDFS.

HBase

- HBase is a NoSQL database or non-relational database .
- HBase is important and mainly used when you need random, real-time, read, or write access to your Big Data.
- It provides support to a high volume of data and high throughput.
- In an HBase, a table can have thousands of columns.

INTRODUCTION TO MONGODB AND MAPREDUCE PROGRAMMING

MongoDB is a cross-platform, document-oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Collection

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

The following table shows the relationship of RDBMS terminology with MongoDB.

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents

Primary Key	Primary Key (Default key _id provided by MongoDB itself)
Database Server and Client	
mysqld/Oracle	mongod
mysql/sqlplus	mongo

Sample Document

Following example shows the document structure of a blog site, which is simply a comma separated key value pair.

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100,
  comments: [
    {
      user: 'user1',
      message: 'My first comment',
      dateCreated: new Date(2011,1,20,2,15),
      like: 0
    }
  ]
}
```

```
    },  
    {  
      user:'user2',  
      message: 'My second comments',  
      dateCreated: new Date(2011,1,25,7,45),  
      like: 5  
    }  
  ]  
}
```

_id is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide _id while inserting the document. If you don't provide then MongoDB provides a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental VALUE.

Any relational database has a typical schema design that shows number of tables and the relationship between these tables. While in MongoDB, there is no concept of relationship.

Advantages of MongoDB over RDBMS

- **Schema less** – MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- Structure of a single object is clear.
- No complex joins.
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Tuning.
- **Ease of scale-out** – MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.
- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

Why Use MongoDB?

- **Document Oriented Storage** – Data is stored in the form of JSON style documents.
- Index on any attribute
- Replication and high availability
- Auto-Sharding
- Rich queries
- Fast in-place updates
- Professional support by MongoDB

Where to Use MongoDB?

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

MongoDB supports many datatypes. Some of them are –

- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – timestamp. This can be handy for recording when a document has been modified or added.
- **Object** – This datatype is used for embedded documents.
- **Null** – This type is used to store a Null value.
- **Symbol** – This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** – This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** – This datatype is used to store the document's ID.

- **Binary data** – This datatype is used to store binary data.
- **Code** – This datatype is used to store JavaScript code into the document.
- **Regular expression** – This datatype is used to store regular expression.

The find() Method

To query data from MongoDB collection, you need to use MongoDB's **find()** method.

Syntax

The basic syntax of **find()** method is as follows –

```
>db.COLLECTION_NAME.find()
```

find() method will display all the documents in a non-structured way.

Example

Assume we have created a collection named mycol as –

```
> use sampleDB
switched to db sampleDB
> db.createCollection("mycol")
{ "ok" : 1 }
>
```

And inserted 3 documents in it using the insert() method as shown below –

```
> db.mycol.insert([
  {
    title: "MongoDB Overview",
    description: "MongoDB is no SQL database",
    by: "tutorials point",
    url: "http://www.tutorialspoint.com",
    tags: ["mongodb", "database", "NoSQL"],
    likes: 100
  },
  {
    title: "NoSQL Database",
    description: "NoSQL database doesn't have tables",
    by: "tutorials point",
    url: "http://www.tutorialspoint.com",
    tags: ["mongodb", "database", "NoSQL"],
    likes: 20,
    comments: [
```

```

        {
            user:"user1",
            message: "My first comment",
            dateCreated: new Date(2013,11,10,2,35),
            like: 0
        }
    ]
}
])

```

Following method retrieves all the documents in the collection –

```

> db.mycol.find()
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534c"), "title" : "MongoDB Overview", "description" : "MongoDB is no SQL database", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 100 }
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534d"), "title" : "NoSQL Database", "description" : "NoSQL database doesn't have tables", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 20, "comments" : [ { "user" : "user1", "message" : "My first comment", "dateCreated" : ISODate("2013-12-09T21:05:00Z"), "like" : 0 } ] }
>

```

The pretty() Method

To display the results in a formatted way, you can use pretty() method.

Syntax

```
>db.COLLECTION_NAME.find().pretty()
```

Example

Following example retrieves all the documents from the collection named mycol and arranges them in an easy-to-read format.

```

> db.mycol.find().pretty()
{
    "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
    "title" : "MongoDB Overview",
    "description" : "MongoDB is no SQL database",
    "by" : "tutorials point",
    "url" : "http://www.tutorialspoint.com",
    "tags" : [
        "mongodb",
        "database",
        "NoSQL"
    ],
    "likes" : 100
}

```

```

}
{
  "_id" : ObjectId("5dd4e2cc0821d3b44607534d"),
  "title" : "NoSQL Database",
  "description" : "NoSQL database doesn't have tables",
  "by" : "tutorials point",
  "url" : "http://www.tutorialspoint.com",
  "tags" : [
    "mongodb",
    "database",
    "NoSQL"
  ],
  "likes" : 20,
  "comments" : [
    {
      "user" : "user1",
      "message" : "My first comment",
      "dateCreated" : ISODate("2013-12-09T21:05:00Z"),
      "like" : 0
    }
  ]
}

```

The findOne() method

Apart from the find() method, there is **findOne()** method, that returns only one document.

Syntax

```
>db.COLLECTIONNAME.findOne()
```

Example

Following example retrieves the document with title MongoDB Overview.

```

> db.mycol.findOne({title: "MongoDB Overview"})
{
  "_id" : ObjectId("5dd6542170fb13eec3963bf0"),
  "title" : "MongoDB Overview",
  "description" : "MongoDB is no SQL database",
  "by" : "tutorials point",
  "url" : "http://www.tutorialspoint.com",
  "tags" : [
    "mongodb",
    "database",
    "NoSQL"
  ],
  "likes" : 100
}

```

}

RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:{\$eq:<value>}}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{\$lt:<value>}}	db.mycol.find({"likes":{\$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{\$lte:<value>}}	db.mycol.find({"likes":{\$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{\$gt:<value>}}	db.mycol.find({"likes":{\$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{\$gte:<value>}}	db.mycol.find({"likes":{\$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{\$ne:<value>}}	db.mycol.find({"likes":{\$ne:50}}).pretty()	where likes != 50
Values in an array	{<key>:{\$in:[<value1>, <value2>,.....<valueN>]}}	db.mycol.find({"name":{\$in:["Raj", "Ram", "Raghu"]}}).pretty()	Where name matches any of the value in :["Raj", "Ram", "Raghu"]

Values not in an array	{<key>:{\$nin:<value>}}	db.mycol.find({"name":{\$nin:["Ramu", "Raghav"]} }).pretty()	Where name values is not in the array :["Ramu", "Raghav"] or, doesn't exist at all
------------------------	-------------------------	--	--

AND in MongoDB

Syntax

To query documents based on the AND condition, you need to use \$and keyword. Following is the basic syntax of AND –

```
>db.mycol.find( { $and: [ {<key1>:<value1>}, { <key2>:<value2>} ] } )
```

Example

Following example will show all the tutorials written by 'tutorials point' and whose title is 'MongoDB Overview'.

```
> db.mycol.find( { $and: [ {"by": "tutorials point"}, {"title": "MongoDB Overview"} ] } ).pretty()
{
  "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
  "title" : "MongoDB Overview",
  "description" : "MongoDB is no SQL database",
  "by" : "tutorials point",
  "url" : "http://www.tutorialspoint.com",
  "tags" : [
    "mongodb",
    "database",
    "NoSQL"
  ],
  "likes" : 100
}
```

For the above given example, equivalent where clause will be '**where by = 'tutorials point' AND title = 'MongoDB Overview'**'. You can pass any number of key, value pairs in find clause.

OR in MongoDB

Syntax

To query documents based on the OR condition, you need to use **\$or** keyword. Following is the basic syntax of **OR** –

```
>db.mycol.find(
  {
    $or: [
      {key1: value1}, {key2:value2}
    ]
  }
).pretty()
```

Example

Following example will show all the tutorials written by 'tutorials point' or whose title is 'MongoDB Overview'.

```
>db.mycol.find({$or:[{"by":"tutorials point"}, {"title": "MongoDB Overview"}]}).pretty()
{
  "_id": ObjectId(7df78ad8902c),
  "title": "MongoDB Overview",
  "description": "MongoDB is no sql database",
  "by": "tutorials point",
  "url": "http://www.tutorialspoint.com",
  "tags": ["mongodb", "database", "NoSQL"],
  "likes": "100"
}
>
```

Using AND and OR Together

Example

The following example will show the documents that have likes greater than 10 and whose title is either 'MongoDB Overview' or by is 'tutorials point'. Equivalent SQL where clause is **'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')**

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"}, {"title": "MongoDB Overview"}]}).pretty()
{
  "_id": ObjectId(7df78ad8902c),
  "title": "MongoDB Overview",
  "description": "MongoDB is no sql database",
  "by": "tutorials point",
  "url": "http://www.tutorialspoint.com",
  "tags": ["mongodb", "database", "NoSQL"],
  "likes": "100"
}
>
```

NOR in MongoDB

Syntax

To query documents based on the NOT condition, you need to use \$not keyword. Following is the basic syntax of **NOT** –

```
>db.COLLECTION_NAME.find(  
    {  
        $not: [  
            {key1: value1}, {key2:value2}  
        ]  
    }  
)
```

Example

Assume we have inserted 3 documents in the collection **empDetails** as shown below –

```
db.empDetails.insertMany(  
    [  
        {  
            First_Name: "Radhika",  
            Last_Name: "Sharma",  
            Age: "26",  
            e_mail: "radhika_sharma.123@gmail.com",  
            phone: "9000012345"  
        },  
        {  
            First_Name: "Rachel",  
            Last_Name: "Christopher",  
            Age: "27",  
            e_mail: "Rachel_Christopher.123@gmail.com",  
            phone: "9000054321"  
        },  
        {  
            First_Name: "Fathima",  
            Last_Name: "Sheik",  
            Age: "24",  
            e_mail: "Fathima_Sheik.123@gmail.com",  
            phone: "9000054321"  
        }  
    ]  
)
```

Following example will retrieve the document(s) whose first name is not "Radhika" and last name is not "Christopher"

```
> db.empDetails.find(  
    {
```

```

        $nor:[
            40
            {"First_Name": "Radhika"},
            {"Last_Name": "Christopher"}
        ]
    }
).pretty()
{
  "_id" : ObjectId("5dd631f270fb13eec3963bef"),
  "First_Name" : "Fathima",
  "Last_Name" : "Sheik",
  "Age" : "24",
  "e_mail" : "Fathima_Sheik.123@gmail.com",
  "phone" : "9000054321"
}

```

NOT in MongoDB

Syntax

To query documents based on the NOT condition, you need to use \$not keyword following is the basic syntax of **NOT** –

```

>db.COLLECTION_NAME.find(
    {
        $NOT: [
            {key1: value1}, {key2:value2}
        ]
    }
).pretty()

```

Example

Following example will retrieve the document(s) whose age is not greater than 25

```

> db.empDetails.find( { "Age": { $not: { $gt: "25" } } } )
{
  "_id" : ObjectId("5dd6636870fb13eec3963bf7"),
  "First_Name" : "Fathima",
  "Last_Name" : "Sheik",
  "Age" : "24",
  "e_mail" : "Fathima_Sheik.123@gmail.com",
  "phone" : "9000054321"
}

```

CASSANDRA

Cassandra is defined as an open-source NoSQL data storage system that leverages a distributed architecture to enable high availability, scalability, and reliability, managed by the Apache non-profit organization.

Cassandra features

1. Open-source availability

Nothing is more exciting than getting a handy product for free. This is probably one of the significant factors behind Cassandra's far-reaching popularity and acceptance. Cassandra is among the open-source products hosted by Apache and is free for anyone who wants to utilize it.

2. Distributed footprint

Another feature of Cassandra is that it is well distributed and meant to run over multiple nodes as opposed to a central system. All the nodes are equal in significance, and without a master node, no bottleneck slows the process down. This is very important because the companies that utilize Cassandra need to constantly run on accurate data and can not tolerate data loss. The equal and wide distribution of Cassandra data across nodes means that losing one node does not significantly affect the system's general performance.

3. Scalability

Cassandra has elastic scalability. This means that it can be scaled up or down without much difficulty or resistance. Cassandra's scalability once again is due to the nodal architecture. It is intended to grow horizontally

as your needs as a developer or company grow. Scaling-up in Cassandra is very easy and not limited to location.

Adding or removing extra nodes can adjust your database system to suit your dynamic needs.

Another exciting point about scaling in Cassandra is that there is no slow down, pause or hitch in the system during the process. This means end-users would not feel the effect of whatever happened, ensuring smooth service to all individuals connected to the network.

4. Cassandra Query Language

Cassandra is not a relational database and does not use the standard query language or SQL. It uses the Cassandra query language (CQL). This would have posed a problem for admins as they would have to master a whole new language – but the good thing about Cassandra Query language is that it is very similar to SQL. It is structured to operate with rows and columns, i.e., table-based data.

However, it does lack the flexibility that comes with the fixed schema of SQL. CQL combines the tabular database management system and the key value. It operates using the data type operations, definition operation, data definition operation, triggers operation, security operations, arithmetic operations, etc.

5. Fault tolerance

Cassandra is fault-tolerant primarily because of its data replicative ability. Data replication denotes the ability of the system to store the same information at multiple locations or nodes. This makes it highly available and tolerant of faults in the system. Failure of a single node or data center does not bring the system to a halt as data

has been replicated and stored across other nodes in the cluster. Data replication leads to a high level of [backup](#) and recovery.

6. Schema free

SQL is a fixed schema database language making it rigid and fixed. However, Cassandra is a schema-optional data model and allows the operator to create as many rows and columns as is deemed necessary.

7. Tunable consistency

Cassandra has two types of consistency – the eventual consistency and the setting consistency. The strong consistency is a type that broadcasts any update or information to every node where the concerned data is located. In eventual consistency, the client has to approve immediately after a cluster receives a write.

Cassandra's tunable consistency is a feature that allows the developer to decide to use any of the two types depending on the function being carried out. The developer can use either or both kinds of consistency at any time.

8. Fast writes

Cassandra is known to have a very high throughput, not hindered by its size. Its ability to write quickly is a function of its data handling process. The initial step taken is to write to the commit log. This is for durability to preserve data in case of damage or node downtime. Writing to the commit log is a speedy and efficient process using this tool.

The next step is to write to the “Memtable” or memory. After writing to Memtable, a node acknowledges the successful writing of data. The Memtable is found in the database memory, and writing to in-memory is much faster than writing to a disk. All of these account for the speed Cassandra writes.

9. Peer-to-peer architecture

Cassandra is built on a peer-to-peer architectural model where all nodes are equal. This is unlike some database models with a “slave to master” relationship. That is where one unit directs the functioning of the other units, and the other unit only communicates with the central unit or master. In Cassandra, different units can communicate with each other as peers in a process called gossiping. This peer-to-peer communication eliminates a single point of failure and is a prominent defining feature of Cassandra.

Cassandra - CQL Datatypes

CQL provides a rich set of built-in data types, including collection types. Along with these data types, users can also create their own custom data types. The following table provides a list of built-in data types available in CQL.

Data Type	Constants	Description
ascii	strings	Represents ASCII character string
bigint	bigint	Represents 64-bit signed long
blob	blobs	Represents arbitrary bytes
Boolean	booleans	Represents true or false
counter	integers	Represents counter column
decimal	integers, floats	Represents variable-precision decimal
double	integers	Represents 64-bit IEEE-754 floating point
float	integers, floats	Represents 32-bit IEEE-754 floating point
inet	strings	Represents an IP address, IPv4 or IPv6
int	integers	Represents 32-bit signed int
text	strings	Represents UTF8 encoded string
timestamp	integers, strings	Represents a timestamp
timeuuid	uuids	Represents type 1 UUID
uuid	uuids	Represents type 1 or type 4
		UUID
varchar	strings	Represents UTF8 encoded string
varint	integers	Represents arbitrary-precision integer

Collection Types

Cassandra Query Language also provides a collection data types. The following table provides a list of Collections available in CQL.

Collection	Description
list	A list is a collection of one or more ordered elements.
map	A map is a collection of key-value pairs.
set	A set is a collection of one or more elements.

User-defined datatypes

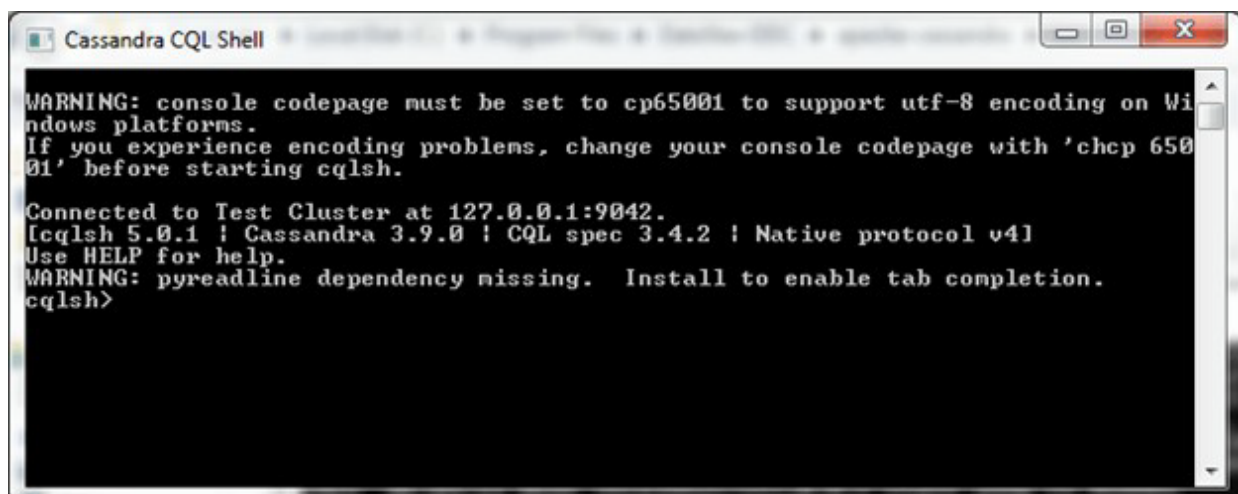
Cqlsh provides users a facility of creating their own data types. Given below are the commands used while dealing with user defined datatypes.

- **CREATE TYPE** – Creates a user-defined datatype.
- **ALTER TYPE** – Modifies a user-defined datatype.
- **DROP TYPE** – Drops a user-defined datatype.
- **DESCRIBE TYPE** – Describes a user-defined datatype.
- **DESCRIBE TYPES** – Describes user-defined datatypes

Cassandra CQLsh

Cassandra CQLsh stands for Cassandra CQL shell. CQLsh specifies how to use Cassandra commands. After installation, Cassandra provides a prompt Cassandra query language shell (cqlsh). It facilitates users to communicate with it.

Cassandra commands are executed on CQLsh. It looks like this:

A screenshot of a Windows command prompt window titled "Cassandra CQL Shell". The window has a black background with white text. The text inside shows a warning about console codepage, connection details to a test cluster at 127.0.0.1:9042, and version information for cqlsh 5.0.1, Cassandra 3.9.0, and CQL spec 3.4.2. It also shows a warning about pyreadline dependency and the cqlsh prompt.

```
Cassandra CQL Shell
WARNING: console codepage must be set to cp65001 to support utf-8 encoding on Windows platforms.
If you experience encoding problems, change your console codepage with 'chcp 65001' before starting cqlsh.

Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.9.0 | CQL spec 3.4.2 | Native protocol v4]
Use HELP for help.
WARNING: pyreadline dependency missing. Install to enable tab completion.
cqlsh>
```

CQLsh provides a lot of options which you can see in the following table:

Options	Usage
help	This command is used to show help topics about the options of CQLsh commands.
version	it is used to see the version of the CQLsh you are using.
color	it is used for colored output.
debug	It shows additional debugging information.
execute	It is used to direct the shell to accept and execute a CQL command.
file= name"	By using this option, cassandra executes the command in the given file and exits.
no-color	It directs cassandra not to use colored output.
u "username"	Using this option, you can authenticate a user. The default user name is: cassandra.
p "password"	Using this option, you can authenticate a user with a password. The default password is: cassandra.

Cassandra Create Keyspace

Cassandra Query Language (CQL) facilitates developers to communicate with Cassandra. The syntax of Cassandra query language is very similar to SQL.

What is Keyspace?

A keyspace is an object that is used to hold column families, user defined types. A keyspace is like RDBMS database which contains column families, indexes, user defined types, data center awareness, strategy used in keyspace, replication factor, etc.

In Cassandra, "Create Keyspace" command is used to create keyspace.

syntax

1. **CREATE** KEYSPACE <identifier> **WITH** <properties>

Or

Create keyspace KeyspaceName **with** replicaton={'class':strategy **name**,
'replication_factor': **No of** replications **on** different nodes}

Cassandra Alter Keyspace

The "ALTER keyspace" command is used to alter the replication factor, strategy name and durable writes properties in created keyspace in Cassandra.

Syntax:

ALTER KEYSPACE <identifier> **WITH** <properties>

Or

ALTER KEYSPACE "KeySpace Name"
WITH replication = {'class': 'Strategy name', 'replication_factor' : 'No.Of replicas'};

or

Alter Keyspace KeyspaceName **with** replication={'class':'StrategyName',
'replication_factor': **no of** replications **on** different nodes}
with DURABLE_WRITES=**true/false**

Main points while altering Keyspace in Cassandra

- **Keyspace Name:** Keyspace name cannot be altered in Cassandra.
- **Strategy Name:** Strategy name can be altered by using a new strategy name.
- **Replication Factor:** Replication factor can be altered by using a new replication factor.
- **DURABLE_WRITES:** DURABLE_WRITES value can be altered by specifying its value true/false.

By default, it is true. If set to false, no updates will be written to the commit log and vice versa.

Example:

Let's take an example to demonstrate "Alter Keyspace". This will alter the keyspace strategy from 'SimpleStrategy' to 'NetworkTopologyStrategy' and replication factor from 3 to 1 for DataCenter1.

ALTER KEYSPACE javatpoint
WITH replication = {'class':'NetworkTopologyStrategy', 'replication_factor' : 1};

Cassandra Drop Keyspace

In Cassandra, "DROP Keyspace" command is used to drop keyspaces with all the data, column families, user defined types and indexes from Cassandra.

Cassandra takes a snapshot of keyspace before dropping the keyspace. If keyspace does not exist in the Cassandra, Cassandra will return an error unless IF EXISTS is used.

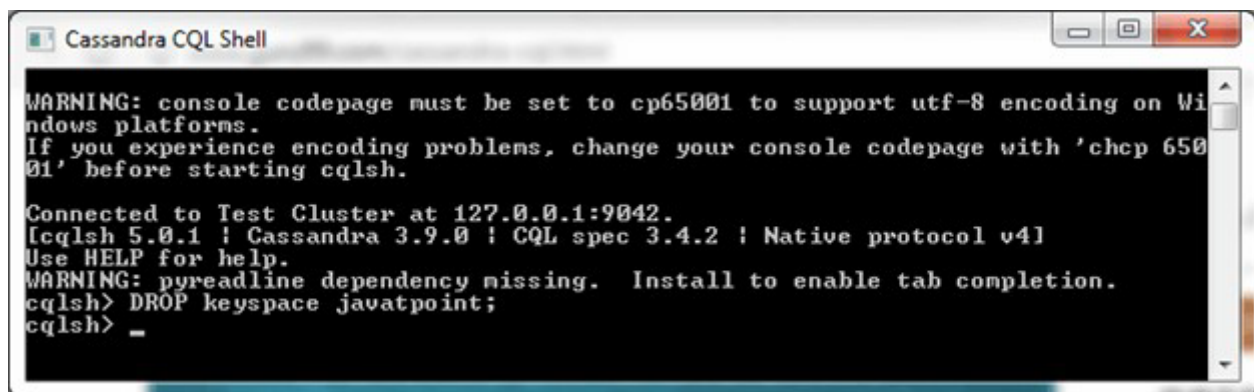
Syntax:

1. **DROP** keyspace KeyspaceName ;

Example:

let's take an example to drop the keyspace named "javatpoint".

DROP keyspace javatpoint;

A screenshot of a terminal window titled "Cassandra CQL Shell". The window has a black background with white text. It shows a warning about console codepage settings, connection details to a test cluster at 127.0.0.1:9042, and the successful execution of the command "DROP keyspace javatpoint;". The prompt "cqlsh>" is visible at the bottom.

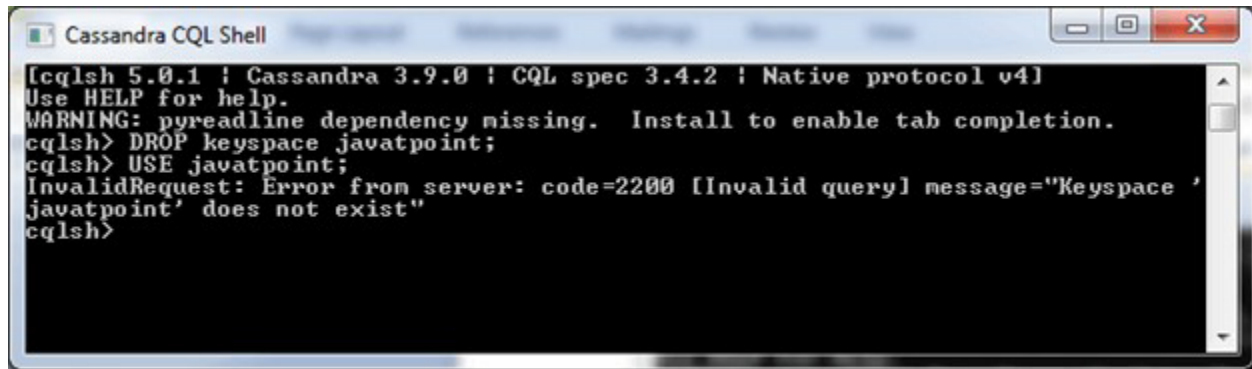
```
Cassandra CQL Shell
WARNING: console codepage must be set to cp65001 to support utf-8 encoding on Windows platforms.
If you experience encoding problems, change your console codepage with 'chcp 65001' before starting cqlsh.

Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.9.0 | CQL spec 3.4.2 | Native protocol v4]
Use HELP for help.
WARNING: pyreadline dependency missing. Install to enable tab completion.
cqlsh> DROP keyspace javatpoint;
cqlsh> _
```

Verification:

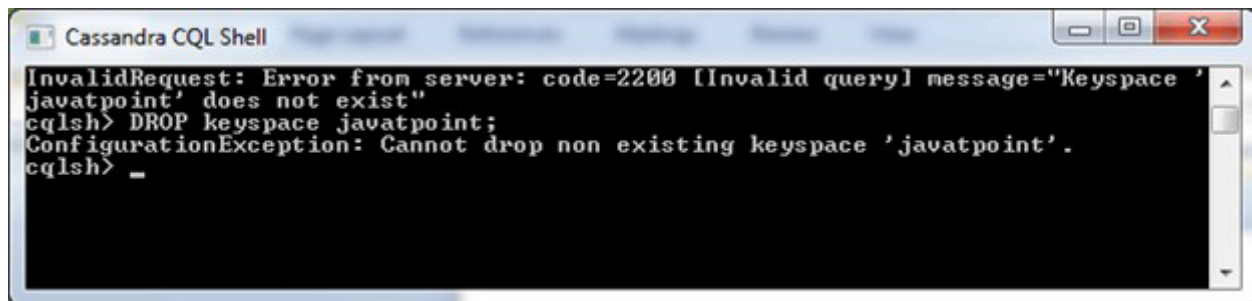
After the execution of the above command the keyspace "javatpoint" is dropped from Cassandra with all the data and schema.

You can verify it by using the "USE" command.



```
Cassandra CQL Shell
[cqlsh 5.0.1 | Cassandra 3.9.0 | CQL spec 3.4.2 | Native protocol v4]
Use HELP for help.
WARNING: pyreadline dependency missing. Install to enable tab completion.
cqlsh> DROP keyspace javatpoint;
cqlsh> USE javatpoint;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Keyspace '
javatpoint' does not exist"
cqlsh>
```

Now you can see that "javatpoint" keyspace is dropped. If you use "DROP" command again, you will get the following message.



```
Cassandra CQL Shell
InvalidRequest: Error from server: code=2200 [Invalid query] message="Keyspace '
javatpoint' does not exist"
cqlsh> DROP keyspace javatpoint;
ConfigurationException: Cannot drop non existing keyspace 'javatpoint'.
cqlsh> _
```

What is Cassandra CRUD Operation?

Cassandra CRUD Operation stands for Create, Update, Read and Delete or Drop. These operations are used to manipulate data in Cassandra. Apart from this, CRUD operations in Cassandra, a user can also verify the command or the data.

a. Create Operation

A user can insert data into the table using Cassandra CRUD operation. The data is stored in the columns of a row in the table. Using INSERT command with proper what, a user can perform this operation.

[Read about Important Features of Cassandra](#)

A Syntax of Create Operation-

```
INSERT INTO <table name>
```

```
(<column1>,<column2> ...)
```

```
VALUES (<value1>,<value2>...)
```

USING<option>

Let's create a table data to illustrate the operation. Example consist of a table with information about students in college. The following table will give the details about the students.

Table.1 Cassandra Crud Operation – Create Operation

EN	NAME	BRANCH	PHONE	CITY
001	Ayush	Electrical Engineering	9999999999	Boston
002	Aarav	Computer Engineering	8888888888	New York City
003	Kabir	Applied Physics	7777777777	Philadelphi

EXAMPLE 1: Creating a table and inserting the data into a table:
INPUT:

```
cqlsh:keyspace1> INSERT INTO student(en, name, branch, phone, city)
VALUES(001, 'Ayush', 'Electrical Engineering', 9999999999, 'Boston');
cqlsh:keyspace1> INSERT INTO student(en, name, branch, phone, city)
VALUES(002, 'Aarav', 'Computer Engineering', 8888888888, 'New York City');
cqlsh:keyspace1> INSERT INTO student(en, name, branch, phone, city)
VALUES(003, 'Kabir', 'Applied Physics', 7777777777, 'Philadelphia');
```

Table.2 Cassandra Crud Operation – OUTPUT After Verification (READ operation)

EN	NAME	BRANCH	PHONE	CITY
001	Ayush	Electrical Engineering	9999999999	Boston
002	Aarav	Computer Engineering	8888888888	New York City
003	Kabir	Applied Physics	7777777777	Philadelphia

EN	NAME	BRANCH	PHONE	CITY	b.Update Operation
001	Ayush	Electrical Engineering	9999999999	Boston	
002	Aarav	Computer Engineering	8888888888	San Fransisco	
003	Kabir	Applied Physics	7777777777	Philadelphia	operation in the Cassandra CRUD

operation is the UPDATE operation. A user can use UPDATE command for the operation. This operation uses three keywords while updating the table.

- **Where:** This keyword will specify the location where data is to be updated.
- **Set:** This keyword will specify the updated value.
- **Must:** This keyword includes the columns composing the primary key.

Furthermore, at the time of updating the rows, if a row is unavailable, then Cassandra has a feature to create a fresh row for the same.

[Do you know How Cassandra Stores Data](#)

A Syntax of Update Operation-

```
UPDATE <table name>
SET <column name>=<new value>
<column name>=<value>...
WHERE <condition>
```

EXAMPLE 2: Let’s change few details in the table ‘student’. In this example, we will update Aarav’s city from ‘New York City’ to ‘San Fransisco’.

INPUT:

```
cqlsh:keyspace1> UPDATE student SET city='San Fransisco'
WHERE en=002;
```

Table.3 Cassandra Crud Operation – OUTPUT After Verification

EN	NAME	BRANCH	PHONE	CITY
02	Aarav	Computer Engineering	8888888888	San Fransisco

c. Read Operation

This is the third Cassandra CRUD Operation – Read Operation. A user has a choice to read either the whole table or a single column. To read data from a table, a user can use SELECT clause. This command is also used for verifying the table after every operation.

[Have a look at Cassandra Shell Commands](#)

SYNTAX to read the whole table-

```
SELECT * FROM <table name>;
```

EXAMPLE 3: To read the whole table ‘student’.

INPUT:

```
cqlsh:keyspace1> SELECT * FROM student;
```

Table.4 Cassandra Crud Operation – OUTPUT After Verification

SYNTAX to read selected columns-

```
SELECT <column name1>,<column name2> .....FROM <table name>;
```

EXAMPLE 4: To read columns of name and city from table ‘student’.

INPUT:\

```
cqlsh:keyspace1> SELECT name, city FROM student;
```

Table.5 Cassandra Crud Operation – OUTPUT After Verification

NAME	CITY
Ayush	Boston
Aarav	San Fransisco
Kabir	Philadelphia

d. Delete Operation

Delete operation is the last Cassandra CRUD Operation, allows a user to delete data from a table. The user can use DELETE command for this operation.

A Syntax of Delete Operation-

DELETE <identifier> FROM <table name> WHERE <condition>;

EXAMPLE 5: In the ‘student’ table let us delete the ‘phone’ or phone number from 003 row.

```
cqlsh:keyspace1> DELETE phone FROM student WHERE en=003;
```

Table.6 Cassandra Crud Operation – OUTPUT After Verification

EN	NAME	BRANCH	PHONE	CITY
001	Ayush	Electrical Engineering	9999999999	Boston
002	Aarav	Computer Engineering	8888888888	San Fransisco
003	Kabir	Applied Physics	null	Philadelphia

SYNTAX for deleting the entire row-

DELETE FROM <identifier> WHERE <condition>;

EXAMPLE 6: In the ‘student’ table, let us delete the entire third row.

```
cqlsh:keyspace1> DELETE FROM student WHERE en=003;
```

[Let’s Explore Best Books To Learn Cassandra](#)

Table.7 Cassandra Crud Operation – OUTPUT After Verification

EN	NAME	BRANCH	PHONE	CITY
001	Ayush	Electrical Engineering	9999999999	Boston
002	Aarav	Computer Engineering	8888888888	San Fransisco

collections

what are Cassandra Collections?

Cassandra collections are a good way for handling tasks. Multiple elements can be stored in collections. There are limitations in Cassandra collections.

- Cassandra collection cannot store data more than 64KB.
- Keep a collection small to prevent the overhead of querying collection because entire collection needs to be traversed.
- If you store more than 64 KB data in the collection, only 64 KB will be able to query, it will result in loss of data.

Types of Cassandra Collections

There are mainly three types of collections that Cassandra supports:

1. Set
2. List
3. Map

Cassandra Set Collection

A Set stores group of elements that returns sorted elements when querying.

Syntax

Here is the syntax of the Set collection that store multiple email addresses for the teacher.

```
Create table University.Teacher
(
  id int,
  Name text,
  Email set<text>,
  Primary key(id)
);
```

Example

Here is the snapshot where table “Teacher” is created with “Email” column as a collection.

```
cqlsh> Create table University.Teacher
... (
... id int,
... Name text,
... Email set<text>,
... Primary key(id)
... );
cqlsh>
```

collection of
Emails

here is the snapshot where data is being inserted in the collection.

```
cqlsh> insert into University.Teacher(id,Name,Email) values(1,'Guru99',{'abc@gmail.com','xyz@hotmail.com'});
cqlsh>
```

insertion in
collection

```
insert into University.Teacher(id,Name,Email)
values(1, 'Guru99', {'abc@gmail.com', 'xyz@hotmail.com'});
```

Cassandra List Collection

When the order of elements matters, the list is used.

Example

Here is the snapshot where column courses of list type is added in table “Teacher.”

```
cqlsh> alter table University.Teacher add courses list<text>;
cqlsh>
```

Here is the snapshot where data is being inserted in column "coursenames".

```
cqlsh> insert into University.Teacher(id,Name,Email,coursenames) values(2,'Hamilton',{'hamilton@yahoo.com'},['Data Science']);
cqlsh>
```

```
insert into University.Teacher(id,Name,Email)
values(2,'Hamilton',{'hamilton@hotmail.com'},[Data Science]);
```

Here is the snapshot that shows the current database state after insertion.

```
id | coursenames          | email                      | name
---+-----+-----+-----
 2 | ['Data Science'] | {'hamilton@yahoo.com'} | Hamilton
(1 rows)
```

Cassandra Map Collection

The map is a collection type that is used to store key value pairs. As its name implies that it maps one thing to another.

For example, if you want to save course name with its prerequisite course name, map collection can be used.

Example

Here is the snapshot where map type is created for course name and its prerequisite course name.

```
cqlsh> Create table University.Course
... (id int,
... prereq map<text, text>,
... primary key(id)
... );
cqlsh>
```

map collection
type

Here is the snapshot where data is being inserted in map collection type

```
cqlsh> insert into University.Course(id,prereq) values(1,{'DataScience':'Database', 'Neural Network':'Artificial Intelligence'});  
cqlsh>
```

course name mapped
to its prereq course

```
insert into University.Course(id,prereq) values(1,{'DataScience':'Database', 'Neural  
Network':'Artificial Intelligence'});
```

Creating a counter table

A counter is a special column used to store an integer that is changed in increments.

Counters are useful for many data models. Some examples:

- To keep track of the number of web page views received on a company website
- To keep track of the number of games played online or the number of players who have joined an online game

The table shown below uses id as the primary key and keeps track of the popularity of a cyclist based on thumbs up/thumbs down clicks in the popularity field of a counter table.

id	popularity
6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	62

Tracking count in a distributed database presents an interesting challenge. In Cassandra, at any given moment, the counter value may be stored in the Memtable, commit log, and/or one or more SSTables. Replication between nodes can cause consistency issues in certain edge cases. Cassandra counters were redesigned in Cassandra 2.1 to alleviate some of the difficulties. Read ["What's New in Cassandra 2.1: Better Implementation of Counters"](#) to discover the improvements made in the counters.

Because counters are implemented differently from other columns, counter columns can only be created in dedicated tables. A counter column must have the datatype [counter data type](#). This data type cannot be assigned to a column that serves as the primary key or partition key. To implement a counter column, create a table that only includes:

- The primary key (can be one or more columns)
- The counter column

Many [counter-related settings](#) can be set in the `cassandra.yaml` file.

A counter column cannot be indexed or deleted.. To load data into a counter column, or to increase or decrease the value of the counter, use the UPDATE command. Cassandra rejects USING TIMESTAMP or USING TTL when updating a counter column.

To create a table having one or more counter columns:

- Use CREATE TABLE to define the counter and non-counter columns. Use all non-counter columns as part of the PRIMARY KEY definition.

- [Using a counter](#)

A counter is a special column for storing a number that is changed in increments.

Time To Live (TTL) command and how to determine the expire time limit of an existing column.

In [Cassandra](#) Time to Live (TTL) is play an important role while if we want to set the time limit of a column and we want to automatically delete after a point of time then at the time using TTL keyword is very useful to define the time limit for a particular column.

1. In Cassandra Both the INSERT and UPDATE commands support setting a time for data in a column to expire.
2. It is used to set the time limit for a specific period of time. By USING TTL clause we can set the TTL value at the time of insertion.
3. We can use TTL function to get the time remaining for a specific selected query.
4. At the point of insertion, we can set expire limit of inserted data by using TTL clause. Let us consider if we want to set the expire limit to two days then we need to define its TTL value.
5. By using TTL we can set the expiration period to two days and the value of TTL will be 172800 seconds. Let's understand with an example.

Table : student_Registration

To create the table used the following CQL query.

```
CREATE TABLE student_Registration(
```

```
Id int PRIMARY KEY,
```

```
Name text,
```

```
Event text
```

```
);
```

Insertion using TTL :

To insert data by using TTL then used the following CQL query.

```
INSERT INTO student_Registration (Id, Name, Event)
```

```
VALUES (101, 'Ashish', 'Ninza') USING TTL 172800;
```

```
INSERT INTO student_Registration (Id, Name, Event)
```

```
VALUES (102, 'Ashish', 'Code') USING TTL 172800;
```

```
INSERT INTO student_Registration (Id, Name, Event)
```

```
VALUES (103, 'Aksh', 'Ninza') USING TTL 172800;
```

Output:

Id	Name	Event
101	Ashish	Ninza
102	Ashish	Code
103	Aksh	Ninza

Now, to determine the remaining time to expire for a specific column used the following CQL query.

```
SELECT TTL (Name)
```

```
from student_Registration
```

```
WHERE Id = 101;
```

Output:

ttl(Name)

172700

It will decrease as you will check again for its TTL value just because of TTL time limit. Now, used the following CQL query to check again.

```
SELECT TTL (Name)
from student_Registration
WHERE Id = 101;
```

Output:

ttl(Name)

172500

Updating using TTL:

Now, if we want to extend the time limit then we can extend with the help of UPDATE command and USING TTL keyword. Let's have a look. To extend time limit with 3 days and also to update the name to 'rana' then used the following CQL query.

```
UPDATE student_Registration
USING TTL 259200
SET Name = 'Rana'
WHERE Id= 102
```

Output:

Id	Name	Event
101	Ashish	Ninza
102	Rana	Code
103	Aksh	Ninza

```
SELECT TTL (Name)
from student_Registration
```

WHERE Id = 102;

Output:

ttl(Name)

259100

Deleting a column using TTL:

To delete the specific existing column used the following CQL query.

UPDATE student_Registration

USING TTL 0

SET Name = 'Ashish'

WHERE Id = 102;

Cassandra Alter Table

ALTER TABLE command is used to alter the table after creating it. You can use the ALTER command to perform two types of operations:

- Add a column
- Drop a column

Syntax:

ALTER (**TABLE** | COLUMNFAMILY) <tablename> <instruction>

Adding a Column

You can add a column in the table by using the ALTER command. While adding column, you have to aware that the column name is not conflicting with the existing column names and that the table is not defined with compact storage option.

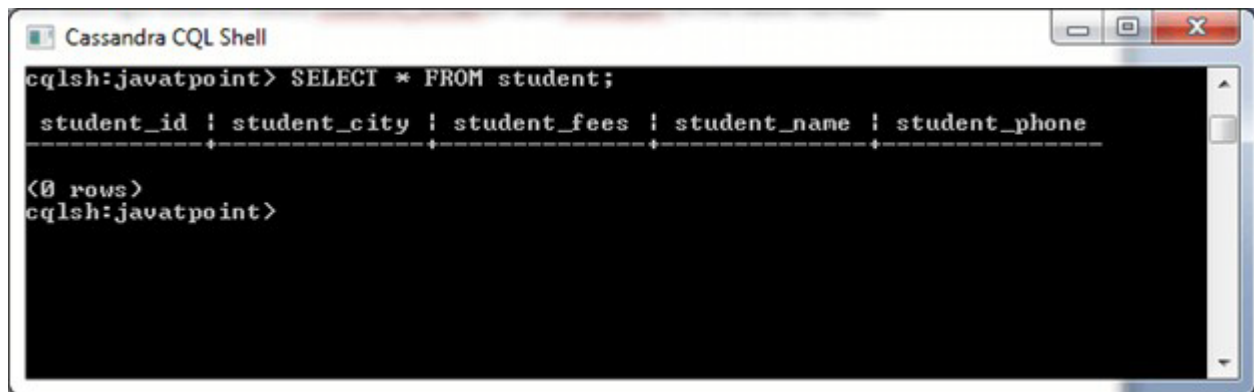
Syntax:

1. **ALTER TABLE** table name
2. **ADD** new **column** datatype;

Example:

Let's take an example to demonstrate the ALTER command on the already created table named "student". Here we are adding a column called student_email of text datatype to the table named student.

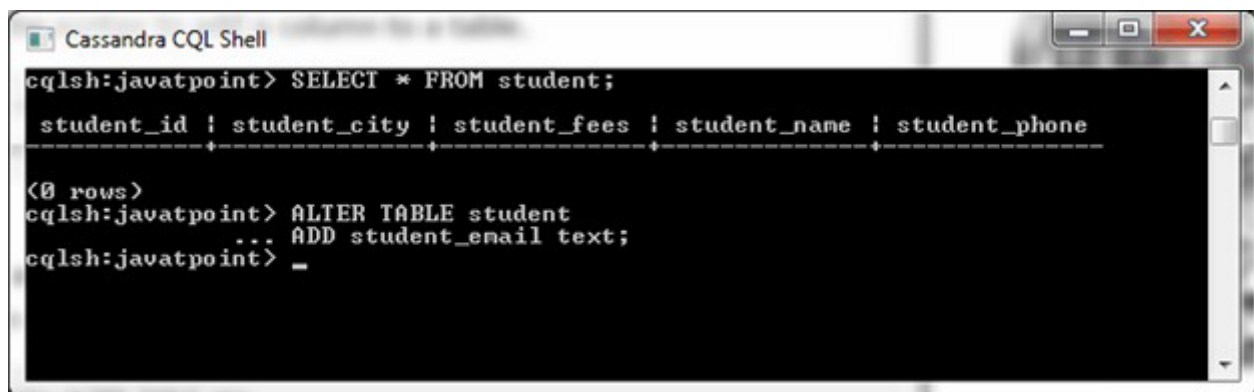
Prior table:



```
Cassandra CQL Shell
cqlsh:javatpoint> SELECT * FROM student;
 student_id | student_city | student_fees | student_name | student_phone
-----+-----+-----+-----+-----
<0 rows>
cqlsh:javatpoint>
```

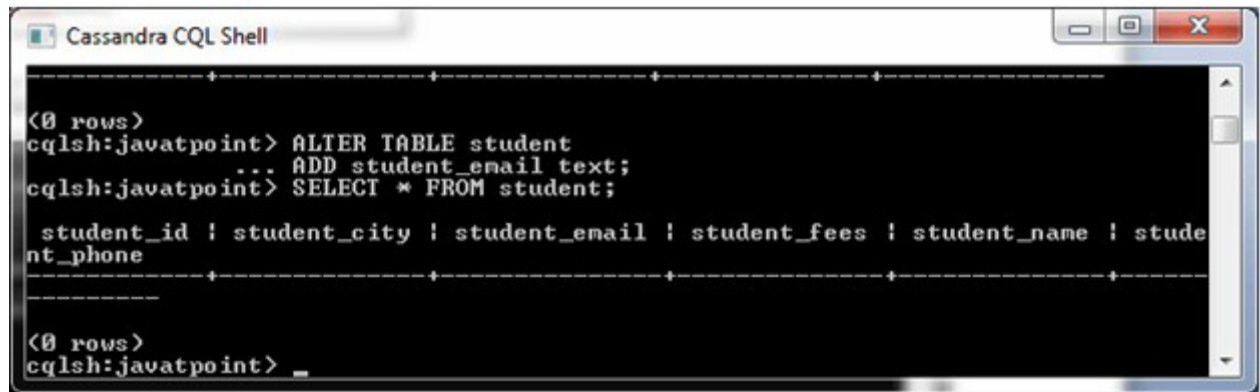
After using the following command:

```
ALTER TABLE student
ADD student_email text;
```



```
Cassandra CQL Shell
cqlsh:javatpoint> SELECT * FROM student;
 student_id | student_city | student_fees | student_name | student_phone
-----+-----+-----+-----+-----
<0 rows>
cqlsh:javatpoint> ALTER TABLE student
... ADD student_email text;
cqlsh:javatpoint> _
```

A new column is added. You can check it by using the SELECT command.



```
Cassandra CQL Shell

<0 rows>
cqlsh:javatpoint> ALTER TABLE student
... ADD student_email text;
cqlsh:javatpoint> SELECT * FROM student;

 student_id | student_city | student_email | student_fees | student_name | student_phone
-----
<0 rows>
cqlsh:javatpoint> _
```

Dropping a Column

You can also drop an existing column from a table by using ALTER command. You should check that the table is not defined with compact storage option before dropping a column from a table.

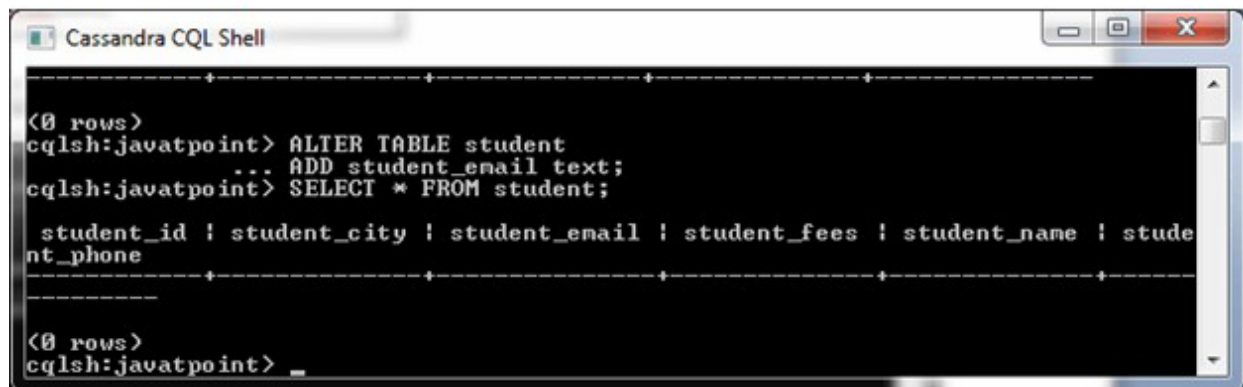
Syntax:

1. **ALTER table name**
2. **DROP column name;**

Example:

Let's take an example to drop a column named student_email from a table named student.

Prior table:



```
Cassandra CQL Shell

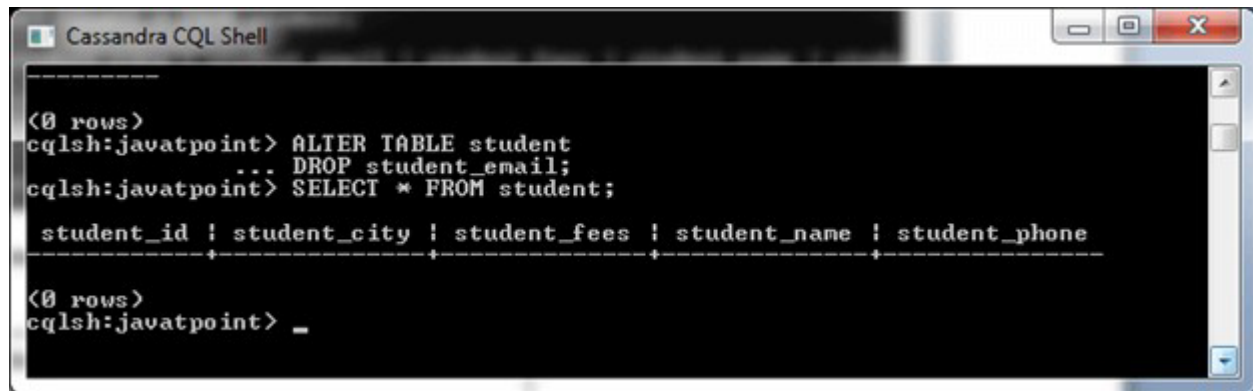
<0 rows>
cqlsh:javatpoint> ALTER TABLE student
... ADD student_email text;
cqlsh:javatpoint> SELECT * FROM student;

 student_id | student_city | student_email | student_fees | student_name | student_phone
-----
<0 rows>
cqlsh:javatpoint> _
```

After using the following command:

ALTER TABLE student

DROP student_email;



```
Cassandra CQL Shell

<0 rows>
cqlsh:javatpoint> ALTER TABLE student
... DROP student_email;
cqlsh:javatpoint> SELECT * FROM student;

 student_id | student_city | student_fees | student_name | student_phone
-----+-----+-----+-----+-----
<0 rows>
cqlsh:javatpoint> _
```

Now you can see that a column named "student_email" is dropped now.

If you want to drop the multiple columns, separate the columns name by ","

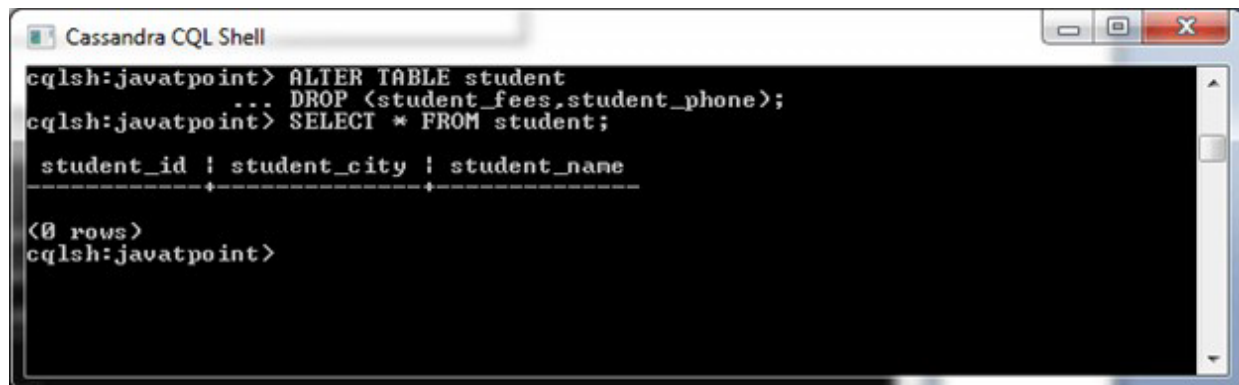
See this example:

Here we will drop two columns student_fees and student_phone.

ALTER TABLE student

DROP (student_fees, student_phone);

Output:



```
Cassandra CQL Shell

cqlsh:javatpoint> ALTER TABLE student
... DROP (student_fees,student_phone);
cqlsh:javatpoint> SELECT * FROM student;

 student_id | student_city | student_name
-----+-----+-----
<0 rows>
cqlsh:javatpoint>
```

Export and Import data in Cassandra

First, we are going to create table namely as Data in which id, firstname, lastname are the fields for sample exercise.

Let's have a look.

Table name: Data

```
CREATE TABLE Data (  
    id UUID PRIMARY KEY,  
    firstname text,  
    lastname text  
);
```

Now, we are going to insert some data to export and import data for sample exercise. let's have a look.

```
INSERT INTO Data (id, firstname, lastname )  
VALUES (3b6441dd-3f90-4c93-8f61-abcfa3a510e1, 'Ashish', 'Rana');
```

```
INSERT INTO Data (id, firstname, lastname)  
VALUES (3b6442dd-bc0d-4157-a80f-abcfa3a510e2, 'Amit', 'Gupta');
```

```
INSERT INTO Data (id, firstname, lastname)  
VALUES (3b6443dd-d358-4d99-b900-abcfa3a510e3, 'Ashish', 'Gupta');
```

```
INSERT INTO Data (id, firstname, lastname)  
VALUES (3b6444dd-4860-49d6-9a4b-abcfa3a510e4, 'Dhruv', 'Gupta');
```

```
INSERT INTO Data (id, firstname, lastname)  
VALUES (3b6445dd-e68e-48d9-a5f8-abcfa3a510e5, 'Harsh', 'Vardhan');
```

```
INSERT INTO Data (id, firstname, lastname)  
VALUES (3b6446dd-eb95-4bb4-8685-abcfa3a510e6, 'Shivang', 'Rana');
```

Now, we are going to Export Data used the following cqlsh query given below. let's have a look.

```
cqlsh>COPY Data(id, firstname, lastname)  
TO 'AshishRana\Desktop\Data.csv' WITH HEADER = TRUE;
```

The CSV file is created:

Using 7 child processes

Starting copy of Data with columns [id, firstname, lastname].

Processed: 6 rows; Rate: 20 rows/s; Avg. rate: 30 rows/s

6 rows exported to 1 files in 0.213 seconds.

Now, we are going to delete data from table 'Data' to import again from CSV file which is already has been created.

```
truncate Data;
```

Now, here we are going to import data again. To import Data used the following cqlsh query given below.

```
COPY Data (id, firstname, lastname)
```

```
FROM 'AshishRana\Desktop\Data.csv'
```

```
WITH HEADER = TRUE;
```

The rows are imported:

Using 7 child processes

Starting copy of Data with columns [id, firstname, lastname].

Processed: 6 rows; Rate: 10 rows/s; Avg. rate: 14 rows/s

6 rows imported from 1 files in 0.423 seconds (0 skipped).

To verify the results whether it is successfully imported or not. let's have a look.

```
SELECT *
```

```
FROM Data;
```

Output:

id	firstname	lastname
3b6446dd-eb95-4bb4-8685-abcfa3a510e6	Shivang	Rana
3b6444dd-4860-49d6-9a4b-abcfa3a510e4	Dhruv	Gupta
3b6445dd-e68e-48d9-a5f8-abcfa3a510e5	Harsh	Vardhan
3b6441dd-3f90-4c93-8f61-abcfa3a510e1	Ashish	Rana
3b6442dd-bc0d-4157-a80f-abcfa3a510e2	Amit	Gupta
3b6443dd-d358-4d99-b900-abcfa3a510e3	Ashish	Gupta

To copy a specific rows of a table used the following cqlsh query given below.
First, export data from table and then truncate after these two steps follow these steps given below.

COPY Data FROM STDIN;

After executing above cqlsh query the line prompt changes to [copy] let's have a look.

Using 7 child processes

Starting copy of cluster1.

Data with columns [id, firstname, lastname].

[Use . on a line by itself to end input]

[copy]

Now, insert the row value of table which you want to import.

[copy] 3b6441dd-3f90-4c93-8f61-abcfa3a510e1, 'Ashish', 'Rana'

[copy] . // keep it in mind at the end insert the period

After successfully executed above given cqlsh query will give you the following results given below. let's have a look.

Processed: 1 rows; Rate: 0 rows/s; Avg. rate: 0 rows/s

1 rows imported from 1 files in 36.991 seconds (0 skipped).

Now, let verify the results.

SELECT *

FROM Data;

Output:

id	firstname	lastname
3b6441dd-3f90-4c93-8f61-abcfa3a510e1	Ashish	Rana
1 rows		

Querying system table

Columns in System Tables

Columns in System Tables

Table name	Column name	Comment
available_ranges	keyspace_name, ranges	
batches	id, mutations, version	
batchlog	id, data, version, written_at	
built_views	keyspace_name, view_name	Information on materialized views
compaction_history	id, bytes_in, bytes_out, columnfamily_name, compacted_at, keyspace_name, rows_merged	Information on compaction history
hints	target_id, hint_id, message_version, mutation	
"IndexInfo"	table_name, index_name	Information on indexes
local	key, bootstrapped, broadcast_address, cluster_name, cql_version, data_center, gossip_generation, host_id, listen_address, native_protocol_version, partitioner, rack, release_version, rpc_address, schema_version, thrift_version, tokens, truncated_at map	Information on a node has about itself and a superset of gossip .
paxos	row_key, cf_id, in_progress_ballot, most_recent_commit, most_recent_commit_at, most_recent_commit_version, proposal, proposal_ballot, proposal_version	Information on lightweight Paxos transactions
peers	peer, data_center, host_id, preferred_ip, rack, release_version, rpc_address, schema_version, tokens	Each node records what other nodes tell it about themselves over the gossip.
peer_events	peer, hints_dropped	
range_xfers	token_bytes, requested_at	
size_estimates	keyspace_name, table_name, range_start, range_end, mean_partition_size, partitions_count	Information on partitions
sstable_activity	keyspace_name, columnfamily_name, generation, rate_120m, rate_15m	
views_builds_in_progress	keyspace_name, view_name, generation_number, last_token	

Columns in System_Schema Tables

Table name	Column name	Comment
aggregates	keyspace_name, aggregate_name, argument_types, final_func, initcond, return_type, state_func, state_type	Information about user-defined aggregates
columns	keyspace_name, table_name, column_name, clustering_order, column_name_bytes, kind, position, type	Information about table columns
Table name	Column name	Comment
aggregates	keyspace_name, aggregate_name, argument_types, final_func, initcond, return_type, state_func, state_type	Information about user-defined aggregates
columns	keyspace_name, table_name, column_name, clustering_order, column_name_bytes, kind, position, type	Information about table columns
dropped_columns	keyspace_name, table_name, column_name, dropped_time, type	Information about dropped columns
functions	keyspace_name, function_name, argument_types, argument_names, body, called_on_null_input, language, return_type	Information on user-defined functions
indexes	keyspace_name, table_name, index_name, kind, options	Information about indexes
keyspaces	keyspace_name, durable_writes, replication	Information on keyspace durable writes and replication
tables	keyspace_name, table_name, bloom_filter_fp_chance, caching, comment, compaction, compression, crc_check_chance, dlocal_read_repair_chance, default_time_to_live, extensions, flags, gc_grace_seconds, id, max_index_interval, memtable_flush_period_in_ms, min_index_interval, read_repair_chance, speculative_retry	Information on columns and column indexes. Used internally for compound primary keys.
triggers	keyspace_name, table_name, trigger_name, options	Information on triggers
types	keyspace_name, type_name, field_names, field_types	Information about user-defined types
views	keyspace_name, view_name, base_table_id, base_table_name, bloom_filter_fp_chance, caching, comment, compaction, compression, crc_check_chance, dlocal_read_repair_chance, default_time_to_live, extensions,	Information about materialized views

	flags,gc_grace_seconds,include_all_columns, max_index_interval,memtable_flush_period_in_ms, min_index_interval,read_repair_chance, speculative_retry,where_clause	
--	--	--

INTRODUCTION TO MAP REDUCE PROGRAMMING AND HIVE

MapReduce:

MapReduce addresses the challenges of distributed programming by providing an abstraction that isolates the developer from system-level details (e.g., locking of data structures, data starvation issues in the processing pipeline, etc.). The programming model specifies simple and well-defined interfaces between a small number of components, and therefore is easy for the programmer to reason about. MapReduce maintains a separation of what computations are to be performed and how those computations are actually carried out on a cluster of machines. The first is under the control of the programmer, while the second is exclusively the responsibility of the execution framework or “runtime”. The advantage is that the execution framework only needs to be designed once and verified for correctness—thereafter, as long as the developer expresses computations in the programming model, code is guaranteed to behave as expected. The upshot is that the developer is freed from having to worry about system-level details (e.g., no more debugging race conditions and addressing lock contention) and can instead focus on algorithm or application design.

ich often has multiple cores). Why is MapReduce important? In practical terms, it provides a very effective tool for tackling large-data problems. But beyond that, MapReduce is important in how it has changed the way we organize computations at a massive scale. MapReduce represents the first widely-adopted step away from the von Neumann model that has served as the foundation of computer science over the last half plus century. Valiant called this a bridging model [148], a conceptual bridge between the physical implementation of a machine and the software that is to be executed on that machine. Until recently, the von Neumann model has served us well: Hardware designers focused on efficient implementations of the von Neumann model and didn’t have to think much about the actual software that would run on the machines. Similarly, the software industry developed software targeted at the model without worrying about the hardware details. The result was extraordinary growth: chip designers churned out successive generations of increasingly powerful processors, and software engineers were able to develop applications in high-level languages that exploited those processors.

MapReduce can be viewed as the first breakthrough in the quest for new abstractions that allow us to organize computations, not over individual machines, but over entire clusters. As Barroso puts it, the datacenter is the computer. MapReduce is certainly not the first model of parallel computation that has been proposed. The most prevalent model in theoretical computer science, which dates back several decades, is the PRAM. MAPPERS AND REDUCERS Key-value pairs form the basic data structure in MapReduce. Keys and values may be primitives such as integers, floating point values, strings, and raw bytes, or they may be arbitrarily complex structures (lists, tuples, associative arrays, etc.). Programmers typically need to define their own custom data types, although a number of libraries such as Protocol Buffers,⁵ Thrift,⁶ and Avro⁷ simplify the task. Part of the design of MapReduce algorithms involves imposing the key-value structure on arbitrary datasets. For a collection of web pages, keys may be URLs and values may be the actual HTML content. For a graph, keys may represent node ids and values may contain the adjacency lists of those nodes (see Chapter 5 for more details). In some algorithms, input keys are not particularly

meaningful and are simply ignored during processing, while in other cases input keys are used to uniquely identify a datum (such as a record id). In Chapter 3, we discuss the role of complex keys and values in the design of various algorithms. In MapReduce, the programmer defines a mapper and a reducer with the following signatures: $\text{map}: (k1, v1) \rightarrow [(k2, v2)]$ $\text{reduce}: (k2, [v2]) \rightarrow [(k3, v3)]$ The convention $[\dots]$ is used throughout this book to denote a list. The input to a MapReduce job starts as data stored on the underlying distributed file system (see Section 2.5). The mapper is applied to every input key-value pair (split across an arbitrary number of files) to generate an arbitrary number of intermediate key-value pairs. The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs.⁸ Implicit between the map and reduce phases is a distributed “group by” operation on intermediate keys. Intermediate data arrive at each reducer in order, sorted by the key. However, no ordering relationship is guaranteed for keys across different reducers. Output key-value pairs from each reducer are written persistently back onto the distributed file system (whereas intermediate key-value pairs are transient and not preserved). The output ends up in r files on the distributed file system, where r is the number of reducers. For the most part, there is no need to consolidate reducer output, since the r files often serve as input to yet another MapReduce job. Figure 2.2 illustrates this two-stage processing structure. A simple word count algorithm in MapReduce is shown in Figure 2.3. This algorithm counts the number of occurrences of every word in a text collection, which may be the first step in, for example, building a unigram language model (i.e., probability

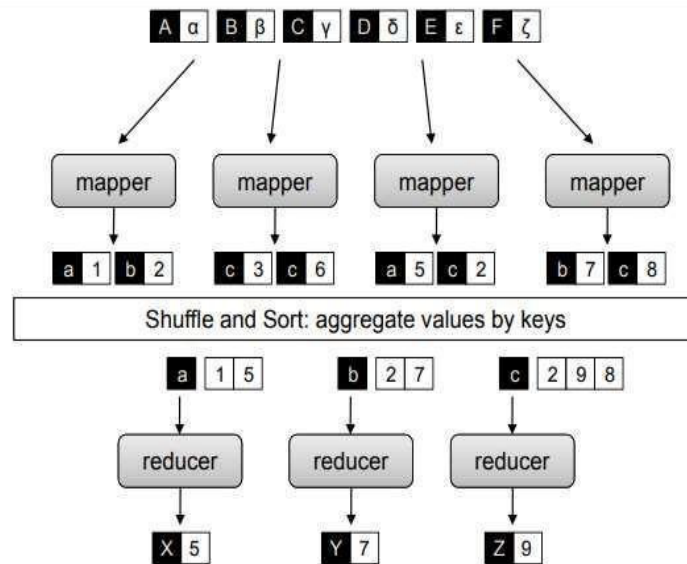


Figure 2.2: Simplified view of MapReduce. Mappers are applied to all input key-value pairs, which generate an arbitrary number of intermediate key-value pairs. Reducers are applied to all values associated with the same key. Between the map and reduce phases lies a barrier that involves a large distributed sort and group by.

MAPREDUCE BASICS

distribution over words in a collection). Input key-values pairs take the form of (docid, doc) pairs stored on the distributed file system, where the former is a unique identifier for the document, and the latter is the text of the document itself. The mapper takes an input key-value pair, tokenizes the document, and emits an intermediate key-value pair for every word: the word itself serves as the key, and the integer one serves as the value (denoting that we've seen the word once). The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Therefore, in our word count algorithm, we simply need to sum up all counts (ones) associated with each word. The reducer does exactly this, and emits final keyvalue pairs with the word as the key, and the count as the value. Final output is written to the distributed file system, one file per reducer. Words within each file will be sorted by alphabetical order, and each file will contain roughly the same number of words. The partitioner, which we discuss later in Section 2.4, controls the assignment of words to reducers. The output can be examined by the programmer or used as input to another MapReduce program.

There are some differences between the Hadoop implementation of MapReduce and Google's implementation.⁹ In Hadoop, the reducer is presented with a key and an iterator over all values associated with the particular key. The values are arbitrarily ordered. Google's implementation allows the programmer to specify a secondary sort key for ordering the values (if desired)—in which case values associated with each key would be presented to the developer's reduce code in sorted order. Later in Section 3.4 we discuss how to overcome this limitation in Hadoop to perform secondary sorting. Another difference: in Google's implementation the programmer is not allowed to change the key in the reducer. That is, the reducer output key must be exactly the same as the reducer input key. In Hadoop, there is no such restriction, and the reducer can emit an arbitrary number of output key-value pairs (with different keys).

To provide a bit more implementation detail: pseudo-code provided in this book roughly mirrors how MapReduce programs are written in Hadoop. Mappers and reducers are objects that implement the Map and Reduce methods, respectively. In Hadoop, a mapper object is initialized for each map task (associated with a particular sequence of key-value pairs called an input split) and the Map method is called on each key-value pair by the execution framework. In configuring a MapReduce job, the programmer provides a hint on the number of map tasks to run, but the execution framework (see next section) makes the final determination based on the physical layout of the data (more details in Section 2.5 and Section 2.6). The situation is similar for the reduce phase: a reducer object is initialized for each reduce task, and the Reduce method is called once per intermediate key. In contrast with the number of map tasks, the programmer can precisely specify the number of reduce tasks. We will return to discuss the details of Hadoop job execution in Section 2.6, which is dependent on an understanding of the distributed file system (covered in Section 2.5). To reiterate: although the presentation of algorithms in this book closely mirrors the way they would be implemented in Hadoop, our focus is on algorithm design and conceptual

understanding—not actual Hadoop programming. For that, we would recommend Tom White’s book [154]. What are the restrictions on mappers and reducers? Mappers and reducers can express arbitrary computations over their inputs. However, one must generally be careful about use of external resources since multiple mappers or reducers may be contending for those resources. For example, it may be unwise for a mapper to query an external SQL database, since that would introduce a scalability bottleneck on the number of map tasks that could be run in parallel (since they might all be simultaneously querying the database).¹⁰ In general, mappers can emit an arbitrary number of intermediate key-value pairs, and they need not be of the same type as the input key-value pairs. Similarly, reducers can emit an arbitrary number of final key-value pairs, and they can differ in type from the intermediate key-value pairs. Although not permitted in functional programming, mappers and reducers can have side effects. This is a powerful and useful feature: for example, preserving state across multiple inputs is central to the design of many MapReduce algorithms (see Chapter 3). Such algorithms can be understood as having side effects that only change state that is internal to the mapper or reducer. While the correctness of such algorithms may be more difficult to guarantee (since the function’s behavior depends not only on the current input but on previous inputs), most potential synchronization problems are avoided since internal state is private only to individual mappers and reducers. In other cases (see Section 4.4 and Section 6.5), it may be useful for mappers or reducers to have external side effects, such as writing files to the distributed file system. Since many mappers and reducers are run in parallel, and the distributed file system is a shared global resource, special care must be taken to ensure that such operations avoid synchronization conflicts. One strategy is to write a temporary file that is renamed upon successful completion of the mapper or reducer .

In addition to the “canonical” MapReduce processing flow, other variations are also possible. MapReduce programs can contain no reducers, in which case mapper output is directly written to disk (one file per mapper). For embarrassingly parallel problems, e.g., parse a large text collection or independently analyze a large number of images, this would be a common pattern. The converse—a MapReduce program with no mappers—is not possible, although in some cases it is useful for the mapper to implement the identity function and simply pass input key-value pairs to the reducers. This has the effect of sorting and regrouping the input for reduce-side processing. Similarly, in some cases it is useful for the reducer to implement the identity function, in which case the program simply sorts and groups mapper output. Finally, running identity mappers and reducers has the effect of regrouping and resorting the input data (which is sometimes useful).

Although in the most common case, input to a MapReduce job comes from data stored on the distributed file system and output is written back to the distributed file system, any other system that satisfies the proper abstractions can serve as a data source or sink. With Google’s MapReduce implementation, BigTable [34], a sparse, distributed, persistent multidimensional sorted map, is frequently used as a source of input and as a store of MapReduce output. HBase is an open-source BigTable clone and has similar capabilities. Also, Hadoop has been integrated with existing MPP (massively parallel processing) relational databases, which allows a programmer to write MapReduce jobs over database rows and dump output into a new database table. Finally, in some

cases MapReduce jobs may not consume any input at all (e.g., computing π) or may only consume a small amount of data (e.g., input parameters to many instances of processor-intensive simulations running in parallel).

PARTITIONERS AND COMBINERS

We have thus far presented a simplified view of MapReduce. There are two additional elements that complete the programming model: partitioners and combiners. Partitioners are responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. In other words, the partitioner specifies the task to which an intermediate key-value pair must be copied. Within each reducer, keys are processed in sorted order (which is how the “group by” is implemented). The simplest partitioner involves computing the hash value of the key and then taking the mod of that value with the number of reducers. This assigns approximately the same number of keys to each reducer (dependent on the quality of the hash function). Note, however, that the partitioner only considers the key and ignores the value—therefore, a roughly-even partitioning of the key space may nevertheless yield large differences in the number of key-values pairs sent to each reducer (since different keys may have different numbers of associated values). This imbalance in the amount of data associated with each key is relatively common in many text processing applications due to the Zipfian distribution of word occurrences.

Combiners are an optimization in MapReduce that allow for local aggregation before the shuffle and sort phase. We can motivate the need for combiners by considering the word count algorithm in Figure 2.3, which emits a key-value pair for each word in the collection. Furthermore, all these key-value pairs need to be copied across the network, and so the amount of intermediate data will be larger than the input collection itself. This is clearly inefficient. One solution is to perform local aggregation on the output of each mapper, i.e., to compute a local count for a word over all the documents processed by the mapper. With this modification (assuming the maximum amount of local aggregation possible), the number of intermediate key-value pairs will be at most the number of unique words in the collection times the number of mappers (and typically far smaller because each mapper may not encounter every word).

smaller because each mapper may not encounter every word). The combiner in MapReduce supports such an optimization. One can think of combiners as “mini-reducers” that take place on the output of the mappers, prior to the shuffle and sort phase. Each combiner operates in isolation and therefore does not have access to intermediate output from other mappers. The combiner is provided keys and values associated with each key (the same types as the mapper output keys and values). Critically, one cannot assume that a combiner will have the opportunity to process all values associated with the same key. The combiner can emit any number of key-value pairs, but the keys and values must be of the same type as the mapper output (same as the reducer input).¹² In cases where an operation is both associative and commutative (e.g., addition or multiplication), reducers can directly serve as combiners. In general, however, reducers and combiners are not interchangeable.

In many cases, proper use of combiners can spell the difference between an impractical algorithm and an efficient algorithm. This topic will be discussed in Section 3.1, which focuses on various techniques for local aggregation. It suffices to say for now that a combiner can significantly reduce the amount of data that needs to be copied over the network, resulting in much faster algorithms. The complete MapReduce model is shown in Figure 2.4. Output of the mappers are processed by the combiners, which perform local aggregation to cut down on the number of intermediate key-value pairs. The partitioner determines which reducer will be responsible for processing a particular key, and the execution framework uses this information to copy the data to the right location during the shuffle and sort phase.¹³ Therefore, a complete MapReduce job consists of code for the mapper, reducer, combiner, and partitioner, along with job configuration parameters. The execution framework handles everything else.

30 CHAPTER 2. MAPREDUCE BASICS

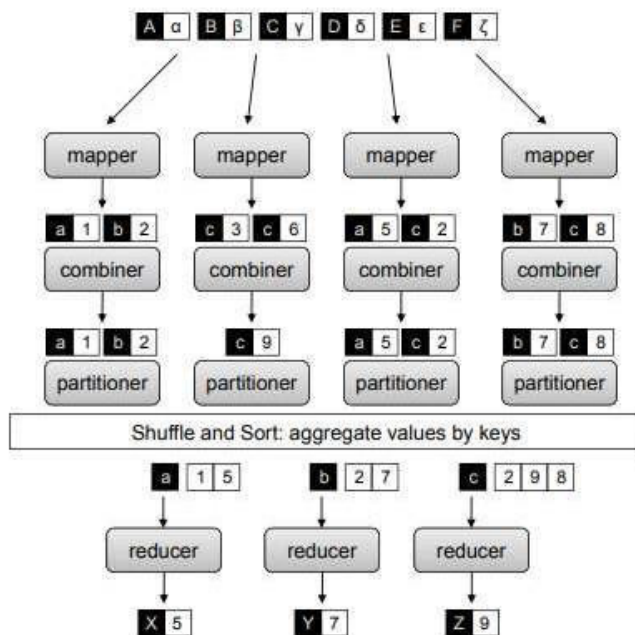


Figure 2.4: Complete view of MapReduce, illustrating combiners and partitioners in addition to mappers and reducers. Combiners can be viewed as “mini-reducers” in the map phase. Partitioners determine which reducer is responsible for a particular key.

SECONDARY SORTING

MapReduce sorts intermediate key-value pairs by the keys during the shuffle and sort phase, which is very convenient if computations inside the reducer rely on sort order (e.g., the order inversion design pattern described in the previous section). However, what if in addition to sorting by key, we also need to sort by value? Google’s MapReduce implementation provides built-in

functionality for (optional) secondary sorting, which guarantees that values arrive in sorted order. Hadoop, unfortunately, does not have this capability built in.

Consider the example of sensor data from a scientific experiment: there are m sensors each taking readings on continuous basis, where m is potentially a large number. A dump of the sensor data might look something like the following, where rx after each timestamp represents the actual sensor readings (unimportant for this discussion, but may be a series of values, one or more complex records, or even raw bytes of images).

(t1, m1, r80521)

(t1, m2, r14209)

(t1, m3, r76042) ...

(t2, m1, r21823)

(t2, m2, r66508)

(t2, m3, r98347)

Suppose we wish to reconstruct the activity at each individual sensor over time. A MapReduce program to accomplish this might map over the raw data and emit the sensor id as the intermediate key, with the rest of each record as the value:

$m1 \rightarrow (t1, r80521)$

This would bring all readings from the same sensor together in the reducer. However, since MapReduce makes no guarantees about the ordering of values associated with the same key, the sensor readings will not likely be in temporal order. The most obvious solution is to buffer all the readings in memory and then sort by timestamp before additional processing. However, it should be apparent by now that any in-memory buffering of data introduces a potential scalability bottleneck. What if we are working with a high frequency sensor or sensor readings over a long period of time? What if the sensor readings themselves are large complex objects? This approach may not scale in these cases—the reducer would run out of memory trying to buffer all values associated with the same key.

This is a common problem, since in many applications we wish to first group together data one way(e.g., by sensor id), and then sort within the groupings another way(e.g., by time). Fortunately, there is a general purpose solution, which we call the “value-to-key conversion” design pattern. The basic idea is to move part of the value into the intermediate key to form a composite key, and let the MapReduce execution framework handle the sorting. In the above example, instead of emitting the sensor id as the key, we would emit the sensor id and the timestamp as a composite key: $(m1, t1) \rightarrow (r80521)$

The sensor reading itself now occupies the value. We must define the intermediate key sort order to first sort by the sensor id (the left element in the pair) and then by the timestamp (the right element in the pair). We must also implement a custom partitioner so that all pairs associated with the same sensor are shuffled to the same reducer. Properly orchestrated, the key-value pairs will be presented to the reducer in the correct sorted order: $(m1, t1) \rightarrow [(r80521)]$ $(m1, t2) \rightarrow [(r21823)]$ $(m1, t3) \rightarrow [(r146925)] \dots$

However, note that sensor readings are now split across multiple keys. The reducer will need to preserve state and keep track of when readings associated with the current sensor end and the next sensor begin.⁹ The basic tradeoff between the two approaches discussed above (buffer and inmemory sort vs. value-to-key conversion) is where sorting is performed. One can explicitly implement secondary sorting in the reducer, which is likely to be faster but suffers from a scalability bottleneck.¹⁰ With value-to-key conversion, sorting is offloaded to the MapReduce execution framework. Note that this approach can be arbitrarily extended to tertiary, quaternary, etc. sorting. This pattern results in many more keys for the framework to sort, but distributed sorting is a task that the MapReduce runtime excels at since it lies at the heart of the programming model.

INDEX COMPRESSION

We return to the question of how postings are actually compressed and stored on disk. This chapter devotes a substantial amount of space to this topic because index compression is one of the main differences between a “toy” indexer and one that works on real-world collections. Otherwise, MapReduce inverted indexing algorithms are pretty straightforward.

Let us consider the canonical case where each posting consists of a document id and the term frequency. A naïve implementation might represent the first as a 32-bit integer⁹ and the second as a 16-bit integer. Thus, a postings list might be encoded as follows: $[(5, 2), (7, 3), (12, 1), (49, 1), (51, 2), \dots]$

where each posting is represented by a pair in parentheses. Note that all brackets, parentheses, and commas are only included to enhance readability; in reality the postings would be represented as a long stream of integers. This naïve implementation would require six bytes per posting. Using this scheme, the entire inverted index would be about as large as the collection itself. Fortunately, we can do significantly better. The first trick is to encode differences between document ids as opposed to the document ids themselves. Since the postings are sorted by document ids, the differences (called d-gaps) must be positive integers greater than zero. The above postings list, represented with d-gaps, would be: $[(5, 2), (2, 3), (5, 1), (37, 1), (2, 2)]$

Of course, we must actually encode the first document id. We haven’t lost any information, since the original document ids can be easily reconstructed from the d-gaps. However, it’s not obvious that we’ve reduced the space requirements either, since the largest possible d-gap is one less than

the number of documents in the collection. This is where the second trick comes in, which is to represent the d-gaps in a way such that it takes less space for smaller numbers. Similarly, we want to apply the same techniques to compress the term frequencies, since for the most part they are also small values. But to understand how this is done, we need to take a slight detour into compression techniques, particularly for coding integers.

Compression, in general, can be characterized as either lossless or lossy: it's fairly obvious that lossless compression is required in this context. To start, it is important to understand that all compression techniques represent a time–space tradeoff. That is, we reduce the amount of space on disk necessary to store data, but at the cost of extra processor cycles that must be spent coding and decoding data. Therefore, it is possible that compression reduces size but also slows processing. However, if the two factors are properly balanced (i.e., decoding speed can keep up with disk bandwidth), we can achieve the best of both worlds: smaller and faster.

POSTINGS COMPRESSION

Having completed our slight detour into integer compression techniques, we can now return to the scalable inverted indexing algorithm shown in Figure 4.4 and discuss how postings lists can be properly compressed. As we can see from the previous section, there is a wide range of choices that represent different tradeoffs between compression ratio and decoding speed. Actual performance also depends on characteristics of the collection, which, among other factors, determine the distribution of d-gaps. Buttcher et al. [30] recently compared the performance of various compression techniques on coding document ids. In terms of the amount of compression that can be obtained (measured in bits per docid), Golomb and Rice codes performed the best, followed by γ codes, Simple-9, varInt, and group varInt (the least space efficient). In terms of raw decoding speed, the order was almost the reverse: group varInt was the fastest, followed by varInt.¹⁴ Simple-9 was substantially slower, and the bit-aligned codes were even slower than that. Within the bit-aligned codes, Rice codes were the fastest, followed by γ , with Golomb codes being the slowest (about ten times slower than group varInt).

Let us discuss what modifications are necessary to our inverted indexing algorithm if we were to adopt Golomb compression for d-gaps and represent term frequencies with γ codes. Note that this represents a space-efficient encoding, at the cost of slower decoding compared to alternatives. Whether or not this is actually a worthwhile tradeoff in practice is not important here: use of Golomb codes serves a pedagogical purpose, to illustrate how one might set compression parameters.

Coding term frequencies with γ codes is easy since they are parameterless. Compressing d-gaps with Golomb codes, however, is a bit tricky, since two parameters are required: the size of the document collection and the number of postings for a particular postings list (i.e., the document frequency, or df). The first is easy to obtain and can be passed into the reducer as a constant. The df of a term, however, is not known until all the postings have been processed—and unfortunately,

the parameter must be known before any posting is coded. At first glance, this seems like a chicken-and-egg problem. A two-pass solution that involves first buffering the postings (in memory) would suffer from the memory bottleneck we've been trying to avoid in the first place.

To get around this problem, we need to somehow inform the reducer of a term's df before any of its postings arrive. This can be solved with the order inversion design pattern introduced in Section 3.3 to compute relative frequencies. The solution is to have the mapper emit special keys of the form `ht, *i` to communicate partial document frequencies. That is, inside the mapper, in addition to emitting intermediate key-value pairs of the following form:

`(tuple ht, docid,tf f)`

we also emit special intermediate key-value pairs like this:

`(tuple ht, *i, df e)`

to keep track of document frequencies associated with each term. In practice, we can accomplish this by applying the in-mapper combining design pattern (see Section 3.1). The mapper holds an in-memory associative array that keeps track of how many documents a term has been observed in (i.e., the local document frequency of the term for the subset of documents processed by the mapper). Once the mapper has processed all input records, special keys of the form `ht, *i` are emitted with the partial df as the value.

To ensure that these special keys arrive first, we define the sort order of the tuple so that the special symbol `*` precedes all documents (part of the order inversion design pattern). Thus, for each term, the reducer will first encounter the `ht, *i` key, associated with a list of values representing partial df values originating from each mapper. Summing all these partial contributions will yield the term's df, which can then be used to set the Golomb compression parameter `b`. This allows the postings to be incrementally compressed as they are encountered in the reducer—memory bottlenecks are eliminated since we do not need to buffer postings in memory.

Once again, the order inversion design pattern comes to the rescue. Recall that the pattern is useful when a reducer needs to access the result of a computation (e.g., an aggregate statistic) before it encounters the data necessary to produce that computation. For computing relative frequencies, that bit of information was the marginal. In this case, it's the document frequency.

PARALLEL BREADTH-FIRST SEARCH

One of the most common and well-studied problems in graph theory is the single-source shortest path problem, where the task is to find shortest paths from a source node to all other nodes in the graph (or alternatively, edges can be associated with costs or weights, in which case the task is to compute lowest-cost or lowest-weight paths). Such problems are a staple in undergraduate

algorithm courses, where students are taught the solution using Dijkstra's algorithm. However, this famous algorithm assumes sequential processing—how would we solve this problem in parallel, and more specifically, with MapReduce?

```
Dijkstra(G, w, s)
2:  $d[s] \leftarrow 0$ 
3: for all vertex  $v \in V$  do
4:  $d[v] \leftarrow \infty$ 
5:  $Q \leftarrow \{V\}$ 
6: while  $Q \neq \emptyset$  do
7:  $u \leftarrow \text{ExtractMin}(Q)$ 
8: for all vertex  $v \in u.\text{AdjacencyList}$  do
9: if  $d[v] > d[u] + w(u, v)$  then
10:  $d[v] \leftarrow d[u] + w(u, v)$ 
```

Figure 5.2: Pseudo-code for Dijkstra's algorithm, which is based on maintaining a global priority queue of nodes with priorities equal to their distances from the source node. At each iteration, the algorithm expands the node with the shortest distance and updates distances to all reachable nodes. As a refresher and also to serve as a point of comparison, Dijkstra's algorithm is shown in Figure 5.2, adapted from Cormen, Leiserson, and Rivest's classic algorithms textbook [41] (often simply known as CLR). The input to the algorithm is a directed, connected graph $G = (V, E)$ represented with adjacency lists, w containing edge distances such that $w(u, v) \geq 0$, and the source node s . The algorithm begins by first setting distances to all vertices $d[v]$, $v \in V$ to ∞ , except for the source node, whose distance to itself is zero. The algorithm maintains Q , a global priority queue of vertices with priorities equal to their distance values d

Dijkstra's algorithm operates by iteratively selecting the node with the lowest current distance from the priority queue (initially, this is the source node). At each iteration, the algorithm "expands" that node by traversing the adjacency list of the selected node to see if any of those nodes can be reached with a path of a shorter distance. The algorithm terminates when the priority queue Q is empty, or equivalently, when all nodes have been considered. Note that the algorithm as presented in Figure 5.2 only computes the shortest distances. The actual paths can be recovered by storing "backpointers" for every node indicating a fragment of the shortest path.

A sample trace of the algorithm running on a simple graph is shown in Figure 5.3 (example also adapted from CLR). We start out in (a) with n_1 having a distance of zero (since it's the source) and all other nodes having a distance of ∞ . In the first iteration (a), n_1 is selected as the node to expand (indicated by the thicker border). After the expansion, we see in (b) that n_2 and n_3 can be reached at a distance of 10 and 5, respectively. Also, we see in (b) that n_3 is the next node selected for expansion. Nodes we have already considered for expansion are shown in black. Expanding n_3 , we see in (c) that the distance to n_2 has decreased because we've found a shorter path. The nodes that will be expanded next, in order, are n_5 , n_2 , and n_4 . The algorithm terminates with the end state shown in (f), where we've discovered the shortest distance to all nodes.

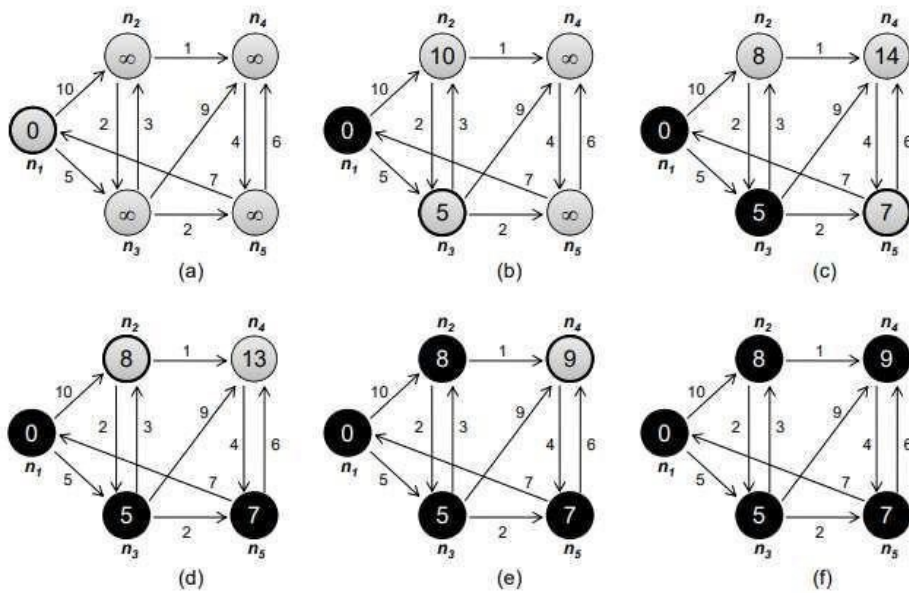


Figure 5.3: Example of Dijkstra's algorithm applied to a simple graph with five nodes, with n_1 as the source and edge distances as indicated. Parts (a)–(e) show the running of the algorithm at each iteration, with the current distance inside the node. Nodes with thicker borders are those being expanded; nodes that have already been expanded are shown in black.

The key to Dijkstra's algorithm is the priority queue that maintains a globally sorted list of nodes by current distance. This is not possible in MapReduce, as the programming model does not provide a mechanism for exchanging global data. Instead, we adopt a brute force approach known as parallel breadth-first search. First, as a simplification let us assume that all edges have unit distance (modeling, for example, hyperlinks on the web). This makes the algorithm easier to understand, but we'll relax this restriction later.

The intuition behind the algorithm is this: the distance of all nodes connected directly to the source node is one; the distance of all nodes directly connected to those is two; and so on. Imagine water rippling away from a rock dropped into a pond—that's a good image of how parallel breadth-first search works. However, what if there are multiple paths to the same node? Suppose we wish to compute the shortest distance to node n . The shortest path must go through one of the nodes in M that contains an outgoing edge to n : we need to examine all $m \in M$ to find m_s , the node with the shortest distance. The shortest distance to n is the distance to m_s plus one.

Pseudo-code for the implementation of the parallel breadth-first search algorithm is provided in Figure 5.4. As with Dijkstra's algorithm, we assume a connected, directed graph represented as adjacency lists. Distance to each node is directly stored alongside the adjacency list of that node, and initialized to ∞ for all nodes except for the source node. In the pseudo-code, we use n to denote the node id (an integer) and N to denote the node's corresponding data structure (adjacency list and current distance). The algorithm works by mapping over all nodes and emitting a key-value pair for each neighbor on the node's adjacency list. The key contains the node id of the neighbor, and the value is the current distance to the node plus one. This says: if we can reach node n with a distance d , then we must be able to reach all the nodes that are connected to n with distance $d + 1$.

After shuffle and sort, reducers will receive keys corresponding to the destination node ids and distances corresponding to all paths leading to that node. The reducer will select the shortest of these distances and then update the distance in the node data structure.

h iteration corresponds to a MapReduce job. The first time we run the algorithm, we “discover” all nodes that are connected to the source. The second iteration, we discover all nodes connected to those, and so on. Each iteration of the algorithm expands the “search frontier” by one hop, and, eventually, all nodes will be discovered with their shortest distances (assuming a fully-connected graph). Before we discuss termination of the algorithm, there is one more detail required to make the parallel breadth-first search algorithm work. We need to “pass along” the graph structure from one iteration to the next. This is accomplished by emitting the node data structure itself, with the node id as a key (Figure 5.4, line 4 in the mapper). In the reducer, we must distinguish the node data structure from distance values (Figure 5.4, lines 5–6 in the reducer), and update the minimum distance in the node data structure before emitting it as the final value. The final output is now ready to serve as input to the next iteration.

So how many iterations are necessary to compute the shortest distance to all nodes? The answer is the diameter of the graph, or the greatest distance between any pair of nodes. This number is surprisingly small for many real-world problems: the saying “six degrees of separation” suggests that everyone on the planet is connected to everyone else by at most six steps (the people a person knows are one step away, people that they know are two steps away, etc.). If this is indeed true, then parallel breadthfirst search on the global social network would take at most six MapReduce iterations.

```
class Mapper
2: method Map(nid n, node N)
3: d  $\leftarrow$  N.Distance
4: Emit(nid n, N) . Pass along graph structure
5: for all nodeid m  $\in$  N.AdjacencyList do
6: Emit(nid m, d + 1) . Emit distances to reachable nodes
1: class Reducer
2: method Reduce(nid m, [d1, d2, . . .])
3: dmin  $\leftarrow \infty$ 
4: M  $\leftarrow \emptyset$ 
5: for all d  $\in$  counts [d1, d2, . . .] do
6: if IsNode(d) then
7: M  $\leftarrow$  d . Recover graph structure
8: else if d < dmin then . Look for shorter distance
9: dmin  $\leftarrow$  d
10: M.Distance  $\leftarrow$  dmin . Update shortest distance
11: Emit(nid m, node M)
```

Figure 5.4: Pseudo-code for parallel breath-first search in MapReduce: the mappers emit distances to reachable nodes, while the reducers select the minimum of those distances for each destination node. Each iteration (one MapReduce job) of the algorithm expands the “search frontier” by one hop.

For more serious academic studies of “small world” phenomena in networks, we refer the reader to a number of publications [61, 62, 152, 2]. In practical terms, we iterate the algorithm until there are no more node distances that are ∞ . Since the graph is connected, all nodes are reachable, and since all edge distances are one, all discovered nodes are guaranteed to have the shortest distances (i.e., there is not a shorter path that goes through a node that hasn’t been discovered).

The actual checking of the termination condition must occur outside of MapReduce. Typically, execution of an iterative MapReduce algorithm requires a nonMapReduce “driver” program, which submits a MapReduce job to iterate the algorithm, checks to see if a termination condition has been met, and if not, repeats. Hadoop provides a lightweight API for constructs called “counters”, which, as the name suggests, can be used for counting events that occur during execution, e.g., number of corrupt records, number of times a certain condition is met, or anything that the programmer desires. Counters can be defined to count the number of nodes that have distances of ∞ : at the end of the job, the driver program can access the final counter value and check to see if another iteration is necessary.

5.2. PARALLEL BREADTH-FIRST SEARCH 99

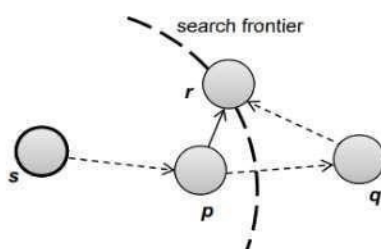


Figure 5.5: In the single source shortest path problem with arbitrary edge distances, the shortest path from source s to node r may go outside the current search frontier, in which case we will not find the shortest distance to r until the search frontier expands to cover q .

Finally, as with Dijkstra’s algorithm in the form presented earlier, the parallel breadth-first search algorithm only finds the shortest distances, not the actual shortest paths. However, the path can be straightforwardly recovered. Storing “backpointers” at each node, as with Dijkstra’s algorithm, will work, but may not be efficient since the graph needs to be traversed again to reconstruct the path segments. A simpler approach is to emit paths along with distances in the mapper, so that each node will have its shortest path easily accessible at all times. The additional space requirements for shuffling these data from mappers to reducers are relatively modest, since for the most part paths (i.e., sequence of node ids) are relatively short.

Up until now, we have been assuming that all edges are unit distance. Let us relax that restriction and see what changes are required in the parallel breadth-first search algorithm. The adjacency lists, which were previously lists of node ids, must now encode the edge distances as well. In line

6 of the mapper code in Figure 5.4, instead of emitting $d + 1$ as the value, we must now emit $d + w$ where w is the edge distance. No other changes to the algorithm are required, but the termination behavior is very different. To illustrate, consider the graph fragment in Figure 5.5, where s is the source node, and in this iteration, we just “discovered” node r for the very first time. Assume for the sake of argument that we’ve already discovered the shortest distance to node p , and that the shortest distance to r so far goes through p . This, however, does not guarantee that we’ve discovered the shortest distance to r , since there may exist a path going through q that we haven’t encountered yet (because it lies outside the search frontier).⁶ However, as the search frontier expands, we’ll eventually cover q and all other nodes along the path from p to q to r —which means that with sufficient iterations, we will discover the shortest distance to r . But how do we know that we’ve found the shortest distance to p ? Well, if the shortest path to p lies within the search frontier, we would have already discovered it. And if it doesn’t, the above argument applies. Similarly, we can repeat the same argument for all nodes on the path from s to p . The conclusion is that, with sufficient iterations, we’ll eventually discover all the shortest distances.

So exactly how many iterations does “eventually” mean? In the worst case, we might need as many iterations as there are nodes in the graph minus one. In fact, it is not difficult to construct graphs that will elicit this worse-case behavior: Figure 5.6 provides an example, with n_1 as the source. The parallel breadth-first search algorithm would not discover that the shortest path from n_1 to n_6 goes through n_3 , n_4 , and n_5 until the fifth iteration. Three more iterations are necessary to cover the rest of the graph. Fortunately, for most real-world graphs, such extreme cases are rare, and the number of iterations necessary to discover all shortest distances is quite close to the diameter of the graph, as in the unit edge distance case.

In practical terms, how do we know when to stop iterating in the case of arbitrary edge distances? The algorithm can terminate when shortest distances at every node no longer change. Once again, we can use counters to keep track of such events. Every time we encounter a shorter distance in the reducer, we increment a counter. At the end of each MapReduce iteration, the driver program reads the counter value and determines if another iteration is necessary.

Compared to Dijkstra’s algorithm on a single processor, parallel breadth-first search in MapReduce can be characterized as a brute force approach that “wastes” a lot of time performing computations whose results are discarded. At each iteration, the algorithm attempts to recompute distances to all nodes, but in reality only useful work is done along the search frontier: inside the search frontier, the algorithm is simply repeating previous computations.⁷ Outside the search frontier, the algorithm hasn’t discovered any paths to nodes there yet, so no meaningful work is done. Dijkstra’s algorithm, on the other hand, is far more efficient. Every time a node is explored, we’re guaranteed to have already found the shortest path to it. However, this is made possible by maintaining a global data structure (a priority queue) that holds nodes sorted by distance—this is not possible in MapReduce because the programming model does not provide support for global data that is mutable and accessible by the mappers and reducers. These inefficiencies represent the cost of parallelization.

The parallel breadth-first search algorithm is instructive in that it represents the prototypical structure of a large class of graph algorithms in MapReduce. They share in the following characteristics:

The graph structure is represented with adjacency lists, which is part of some larger node data structure that may contain additional information (variables to store intermediate output, features of the nodes). In many cases, features are attached to edges as well (e.g., edge weights).

The graph structure is represented with adjacency lists, which is part of some larger node data structure that may contain additional information (variables to store intermediate output, features of the nodes). In many cases, features are attached to edges as well (e.g., edge weights).

In addition to computations, the graph itself is also passed from the mapper to the reducer. In the reducer, the data structure corresponding to each node is updated and written back to disk.

Graph algorithms in MapReduce are generally iterative, where the output of the previous iteration serves as input to the next iteration. The process is controlled by a non-MapReduce driver program that checks for termination.

For parallel breadth-first search, the mapper computation is the current distance plus edge distance (emitting distances to neighbors), while the reducer computation is the Min function (selecting the shortest path). As we will see in the next section, the MapReduce algorithm for PageRank works in much the same way



INTRODUCTION TO HIVE AND PIG

The term 'Big Data' is used for collections of large datasets that include huge volume, high velocity, and a variety of data that is increasing day by day. Using traditional data management systems, it is difficult to process Big Data. Therefore, the Apache Software Foundation introduced a framework called Hadoop to solve Big Data management and processing challenges.

Hadoop

Hadoop is an open-source framework to store and process Big Data in a distributed environment. It contains two modules, one is MapReduce and another is Hadoop Distributed File System (HDFS).

- **MapReduce:** It is a parallel programming model for processing large amounts of structured, semi-structured, and unstructured data on large clusters of commodity hardware.
- **HDFS:** Hadoop Distributed File System is a part of Hadoop framework, used to store and process the datasets. It provides a fault-tolerant file system to run on commodity hardware.

The Hadoop ecosystem contains different sub-projects (tools) such as Sqoop, Pig, and Hive that are used to help Hadoop modules.

- **Sqoop:** It is used to import and export data to and from between HDFS and RDBMS.
- **Pig:** It is a procedural language platform used to develop a script for MapReduce operations.
- **Hive:** It is a platform used to develop SQL type scripts to do MapReduce operations.

Note: There are various ways to execute MapReduce operations:

- The traditional approach using Java MapReduce program for structured, semi-structured, and unstructured data.
- The scripting approach for MapReduce to process structured and semi structured data using Pig.
- The Hive Query Language (HiveQL or HQL) for MapReduce to process structured data using Hive.

What is Hive

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

Hive is not

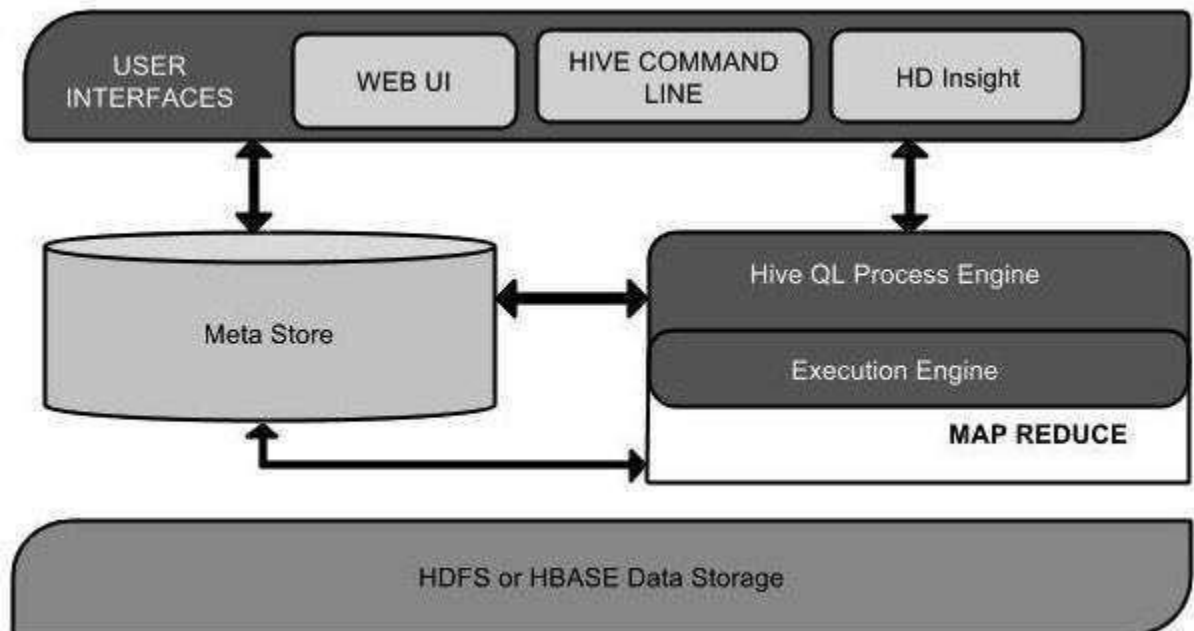
- A relational database
- A design for OnLine Transaction Processing (OLTP)
- A language for real-time queries and row-level updates

Features of Hive

- It stores schema in a database and processed data into HDFS.
- It is designed for OLAP.
- It provides SQL type language for querying called HiveQL or HQL.
- It is familiar, fast, scalable, and extensible.

Architecture of Hive

The following component diagram depicts the architecture of Hive:

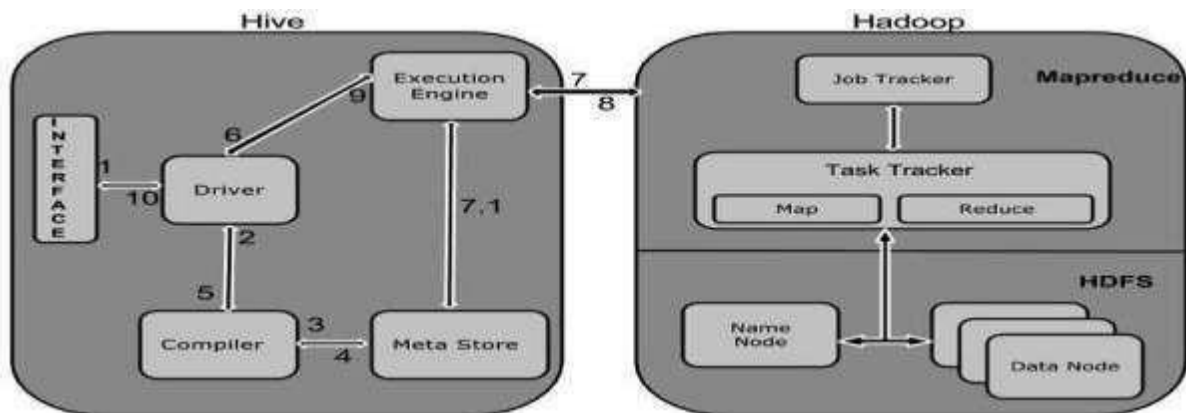


This component diagram contains different units. The following table describes each unit:

Unit Name	Operation
User Interface	Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server).
Meta Store	Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.
HiveQL Process Engine	HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it.
Execution Engine	The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce.
HDFS or HBASE	Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Working of Hive

The following diagram depicts the workflow between Hive and Hadoop.



The following table defines how Hive interacts with Hadoop framework:

Step No.	Operation
1	Execute Query The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute.
2	Get Plan The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query.
3	Get Metadata The compiler sends metadata request to Metastore (any database).
4	Send Metadata Metastore sends metadata as a response to the compiler.
5	Send Plan The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete.
6	Execute Plan The driver sends the execute plan to the execution engine.
7	Execute Job Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Here, the query executes MapReduce job.
7.1	Metadata Ops

	Meanwhile in execution, the execution engine can execute metadata operations with Metastore.
8	Fetch Result The execution engine receives the results from Data nodes.
9	Send Results The execution engine sends those resultant values to the driver.
10	Send Results The driver sends the results to Hive Interfaces.

File Formats in Hive

- File Format specifies how records are encoded in files
- Record Format implies how a stream of bytes for a given record are encoded
- The default file format is **TEXTFILE** – each record is a line in the file
- Hive uses different control **characters as delimiters** in textfiles
 - ^A (octal 001) , ^B(octal 002), ^C(octal 003), \n
- The term **field** is used when overriding the default delimiter
 - **FIELDS TERMINATED BY ‘\001’**
- Supports text files – csv, tsv
- TextFile can contain JSON or XML documents.

ommonly used File Formats –

1. **TextFile format**
 - Suitable for sharing data with other tools
 - Can be viewed/edited manually
2. **SequenceFile**
 - Flat files that stores binary key ,value pair
 - SequenceFile offers a Reader ,Writer, and Sorter classes for reading ,writing, and sorting respectively
 - Supports – Uncompressed, Record compressed (only value is compressed) and Block compressed (both key,value compressed) formats
3. **RCFile**
 - RCFile stores columns of a table in a record columnar way
4. **ORC**
5. **AVRO**

Hive Commands

Hive supports Data definition Language(DDL), Data Manipulation Language(DML) and User defined functions.

Hive DDL Commands

create database

drop database

create table

drop table

alter table

create index

create view

Hive DML Commands

Select

Where

Group By

Order By

Load Data

Join:

- Inner Join
- Left Outer Join
- Right Outer Join
- Full Outer Join

Hive DDL Commands

Create Database Statement

A database in Hive is a namespace or a collection of tables.



1. hive> CREATE SCHEMA userdb;
2. hive> SHOW DATABASES;

Drop database

1. hive> DROP DATABASE IF EXISTS userdb;

Creating Hive Tables

Create a table called Sonoo with two columns, the first being an integer and the other a string.

1. hive> CREATE TABLE Sonoo(foo INT, bar STRING);

Create a table called HIVE_TABLE with two columns and a partition column called ds. The partition column is a virtual column. It is not part of the data itself but is derived from the partition that a particular dataset is loaded into. By default, tables are assumed to be of text input format and the delimiters are assumed to be ^A(ctrl-a).

1. hive> CREATE TABLE HIVE_TABLE (foo INT, bar STRING) PARTITIONED BY (ds STRING);

Browse the table

1. hive> Show tables;

Altering and Dropping Tables

1. hive> ALTER TABLE Sonoo RENAME TO Kafka;
2. hive> ALTER TABLE Kafka ADD COLUMNS (col INT);
3. hive> ALTER TABLE HIVE_TABLE ADD COLUMNS (col1 INT COMMENT 'a comment');
4. hive> ALTER TABLE HIVE_TABLE REPLACE COLUMNS (col2 INT, weight STRING, baz INT COMMENT 'baz replaces new_col1');

Hive DML Commands

To understand the Hive DML commands, let's see the employee and employee_department table first.

Employee			Employee Department	
EMP ID	Emp Name	Address	Emp ID	Department
1	Rose	US	1	IT
2	Fred	US	2	IT
3	Jess	In	3	Eng
4	Frey	Th	4	Admin

LOAD DATA

1. hive> LOAD DATA LOCAL INPATH './usr/Desktop/kv1.txt' OVERWRITE INTO TABLE Employee;

SELECTS and FILTERS

1. hive> SELECT E.EMP_ID FROM Employee E WHERE E.Address='US';

GROUP BY

1. hive> SELECT E.EMP_ID FROM Employee E GROUP BY E.Address;

Adding a Partition

We can add partitions to a table by altering the table. Let us assume we have a table called **employee** with fields such as Id, Name, Salary, Designation, Dept, and yoj.

Syntax:

```
ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION partition_spec
[LOCATION 'location1'] partition_spec [LOCATION 'location2'] ...;
```

partition_spec:

```
:(p_column = p_col_value, p_column = p_col_value, ...)
```

The following query is used to add a partition to the employee table.

```
hive> ALTER TABLE employee
> ADD PARTITION (year='2012')
> location '/2012/part2012';
```

Renaming a Partition

The syntax of this command is as follows.

```
ALTER TABLE table_name PARTITION partition_spec RENAME TO PARTITION partition_spec;
```

The following query is used to rename a partition:

```
hive> ALTER TABLE employee PARTITION (year='1203')  
> RENAME TO PARTITION (Yoj='1203');
```

Dropping a Partition

The following syntax is used to drop a partition:

```
ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec, PARTITION partition_spec,...;
```

The following query is used to drop a partition:

```
hive> ALTER TABLE employee DROP [IF EXISTS]  
> PARTITION (year='1203');
```

Hive Query Language

The Hive Query Language (HiveQL) is a query language for Hive to process and analyze structured data in a Metastore. This chapter explains how to use the SELECT statement with WHERE clause.

SELECT statement is used to retrieve the data from a table. WHERE clause works similar to a condition. It filters the data using the condition and gives you a finite result. The built-in operators and functions generate an expression, which fulfils the condition.

Syntax

Given below is the syntax of the SELECT query:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...  
FROM table_reference  
[WHERE where_condition]  
[GROUP BY col_list]  
[HAVING having_condition]  
[CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY col_list]]  
[LIMIT number];
```

Example

Let us take an example for SELECT...WHERE clause. Assume we have the employee table as given below, with fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details who earn a salary of more than Rs 30000.

ID	Name	Salary	Designation	Dept
1201	Gopal	45000	Technical manager	TP
1202	Manisha	45000	Proofreader	PR
1203	Masthanvali	40000	Technical writer	TP
1204	Krian	40000	Hr Admin	HR
1205	Kranthi	30000	Op Admin	Admin

The following query retrieves the employee details using the above scenario:

```
hive> SELECT * FROM employee WHERE salary>30000;
```

On successful execution of the query, you get to see the following response:

ID	Name	Salary	Designation	Dept
1201	Gopal	45000	Technical manager	TP
1202	Manisha	45000	Proofreader	PR
1203	Masthanvali	40000	Technical writer	TP
1204	Krian	40000	Hr Admin	HR

JDBC Program

The JDBC program to apply where clause for the given example is as follows.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;

public class HiveQLWhere {
    private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

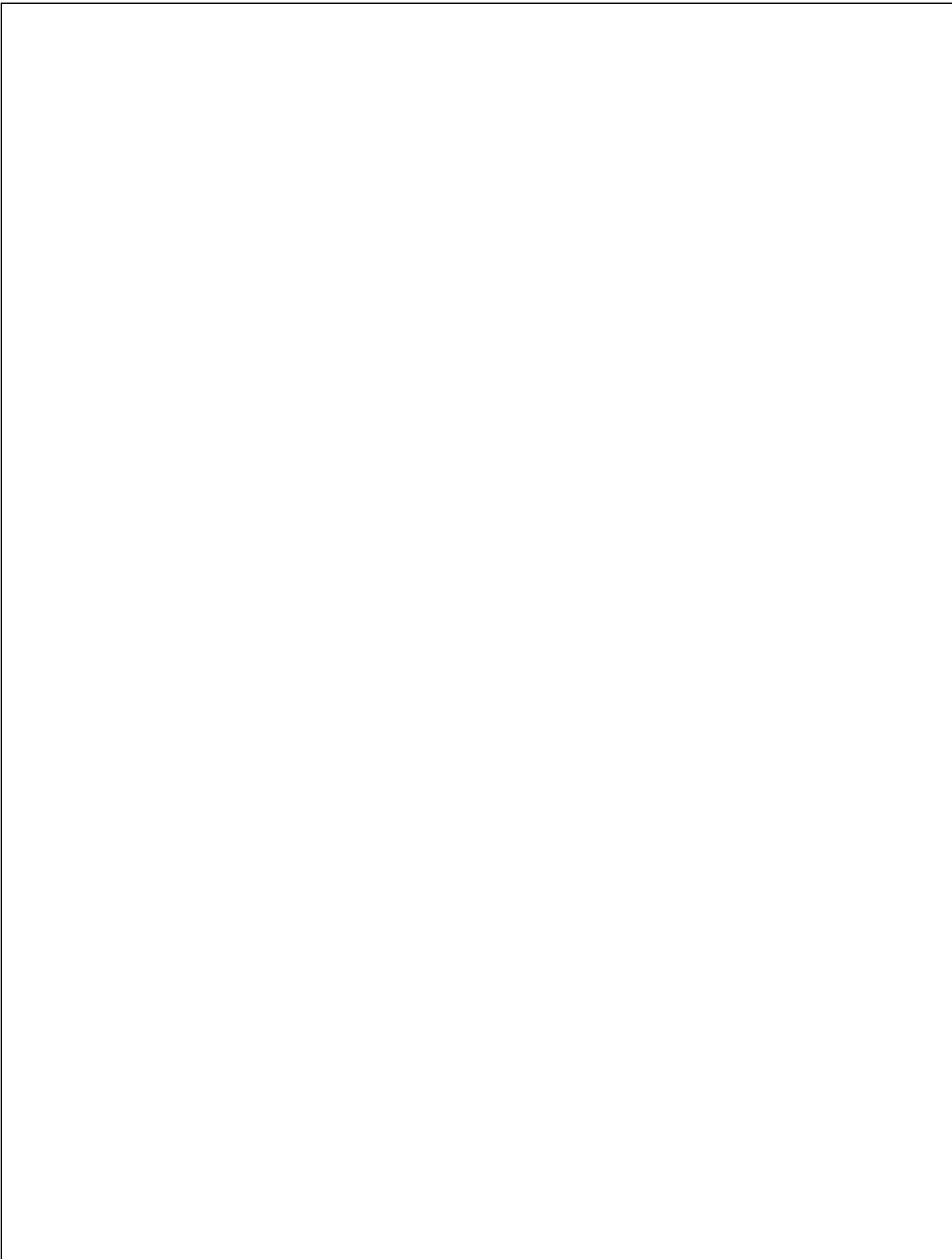
    public static void main(String[] args) throws SQLException {

        // Register driver and create driver instance
        Class.forName(driverName);

        // get connection
        Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "",
        "");

        // create statement
        Statement stmt = con.createStatement();

        // execute statement
```



```

Resultset res = stmt.executeQuery("SELECT * FROM employee WHERE salary>30000;");

System.out.println("Result:");
System.out.println(" ID \t Name \t Salary \t Designation \t Dept ");

while (res.next()) {
    System.out.println(res.getInt(1) + " " + res.getString(2) + " " + res.getDouble(3) + " " +
res.getString(4) + " " + res.getString(5));
}
con.close();
}
}

```

Save the program in a file named HiveQLWhere.java. Use the following commands to compile and execute this program.

```

$ javac HiveQLWhere.java
$ java HiveQLWhere

```

Output:

ID	Name	Salary	Designation	Dept
1201	Gopal	45000	Technical manager	TP
1202	Manisha	45000	Proofreader	PR
1203	Masthanvali	40000	Technical writer	TP
1204	Krian	40000	Hr Admin	HR

The ORDER BY clause is used to retrieve the details based on one column and sort the result set by ascending or descending order.

Syntax

Given below is the syntax of the ORDER BY clause:

```

SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[ORDER BY col_list]
[LIMIT number];

```

Example

Let us take an example for SELECT...ORDER BY clause. Assume employee table as given below, with the fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details in order by using Department name.

ID	Name	Salary	Designation	Dept
1201	Gopal	45000	Technical manager	TP
1202	Manisha	45000	Proofreader	PR
1203	Masthanvali	40000	Technical writer	TP
1204	Krian	40000	Hr Admin	HR
1205	Kranthi	30000	Op Admin	Admin

The following query retrieves the employee details using the above scenario:

```
hive> SELECT Id, Name, Dept FROM employee ORDER BY DEPT;
```

On successful execution of the query, you get to see the following response:

ID	Name	Salary	Designation	Dept
1205	Kranthi	30000	Op Admin	Admin
1204	Krian	40000	Hr Admin	HR
1202	Manisha	45000	Proofreader	PR
1201	Gopal	45000	Technical manager	TP
1203	Masthanvali	40000	Technical writer	TP

JDBC Program

Here is the JDBC program to apply Order By clause for the given example.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;

public class HiveQLOrderBy {
    private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

    public static void main(String[] args) throws SQLException {

        // Register driver and create driver instance
        Class.forName(driverName);

        // get connection
```

```

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "",
""");

// create statement
Statement stmt = con.createStatement();

// execute statement
ResultSet res = stmt.executeQuery("SELECT * FROM employee ORDER BY DEPT;");
System.out.println(" ID \t Name \t Salary \t Designation \t Dept ");

while (res.next()) {
    System.out.println(res.getInt(1) + " " + res.getString(2) + " " + res.getDouble(3) + " " +
res.getString(4) + " " + res.getString(5));
}

con.close();
}
}

```

Save the program in a file named HiveQLOrderBy.java. Use the following commands to compile and execute this program.

```

$ javac HiveQLOrderBy.java
$ java HiveQLOrderBy

```

Output:

ID	Name	Salary	Designation	Dept
1205	Kranthi	30000	Op Admin	Admin
1204	Krian	40000	Hr Admin	HR
1202	Manisha	45000	Proofreader	PR
1201	Gopal	45000	Technical manager	TP
1203	Masthanvali	40000	Technical writer	TP
1204	Krian	40000	Hr Admin	HR

The GROUP BY clause is used to group all the records in a result set using a particular collection column. It is used to query a group of records.

Syntax

The syntax of GROUP BY clause is as follows:

```

SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[ORDER BY col_list]

```

```
[LIMIT number];
```

Example

Let us take an example of SELECT...GROUP BY clause. Assume employee table as given below, with Id, Name, Salary, Designation, and Dept fields. Generate a query to retrieve the number of employees in each department.

+_____+	+_____+	+_____+	+_____+	+_____+
ID	Name	Salary	Designation	Dept
+_____+	+_____+	+_____+	+_____+	+_____+
1201	Gopal	45000	Technical manager	TP
1202	Manisha	45000	Proofreader	PR
1203	Masthanvali	40000	Technical writer	TP
1204	Krian	45000	Proofreader	PR
1205	Kranthi	30000	Op Admin	Admin
+_____+	+_____+	+_____+	+_____+	+_____+

The following query retrieves the employee details using the above scenario.

```
hive> SELECT Dept,count(*) FROM employee GROUP BY DEPT;
```

On successful execution of the query, you get to see the following response:

+_____+	+_____+
Dept	Count(*)
+_____+	+_____+
Admin	1
PR	2
TP	3
+_____+	+_____+

JDBC Program

Given below is the JDBC program to apply the Group By clause for the given example.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;

public class HiveQLGroupBy {
    private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

    public static void main(String[] args) throws SQLException {

        // Register driver and create driver instance
        Class.forName(driverName);

        // get connection
```

```

Connection con = DriverManager.
getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement
Statement stmt = con.createStatement();

// execute statement
ResultSet res = stmt.executeQuery("SELECT Dept,count(*)" + "FROM employee GROUP
BY DEPT; ");
System.out.println(" Dept \t count(*)");

while (res.next()) {
    System.out.println(res.getString(1) + " " + res.getInt(2));
}
con.close();
}
}

```

Save the program in a file named HiveQLGroupBy.java. Use the following commands to compile and execute this program.

```

$ javac HiveQLGroupBy.java
$ java HiveQLGroupBy

```

Output:

Dept	Count(*)
Admin	1
PR	2
TP	3

JOIN is a clause that is used for combining specific fields from two tables by using values common to each one. It is used to combine records from two or more tables in the database.

Syntax

join_table:

```

table_reference JOIN table_factor [join_condition]
| table_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table_reference
join_condition
| table_reference LEFT SEMI JOIN table_reference join_condition
| table_reference CROSS JOIN table_reference [join_condition]

```

Example

We will use the following two tables in this chapter. Consider the following table named CUSTOMERS..

+	+	+	+	+	+
ID	NAME	AGE	ADDRESS	SALARY	
+	+	+	+	+	+
1	Ramesh	32	Ahmedabad	2000.00	
2	Khilan	25	Delhi	1500.00	
3	kaushik	23	Kota	2000.00	
4	Chaitali	25	Mumbai	6500.00	
5	Hardik	27	Bhopal	8500.00	
6	Komal	22	MP	4500.00	
7	Muffy	24	Indore	10000.00	
+	+	+	+	+	+

Consider another table ORDERS as follows:

+	+	+	+	+
OID	DATE	CUSTOMER_ID	AMOUNT	
+	+	+	+	+
102	2009-10-08 00:00:00	3	3000	
100	2009-10-08 00:00:00	3	1500	
101	2009-11-20 00:00:00	2	1560	
103	2008-05-20 00:00:00	4	2060	
+	+	+	+	+

There are different types of joins given as follows:

- JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

JOIN

JOIN clause is used to combine and retrieve the records from multiple tables. JOIN is same as OUTER JOIN in SQL. A JOIN condition is to be raised using the primary keys and foreign keys of the tables.

The following query executes JOIN on the CUSTOMER and ORDER tables, and retrieves the records:

```
hive> SELECT c.ID, c.NAME, c.AGE, o.AMOUNT
FROM CUSTOMERS c JOIN ORDERS o
ON (c.ID = o.CUSTOMER_ID);
```

On successful execution of the query, you get to see the following response:

+	+	+	+	+
ID	NAME	AGE	AMOUNT	
+	+	+	+	+
3	kaushik	23	3000	
3	kaushik	23	1500	
2	Khilan	25	1560	

```
| 4 | Chaitali | 25 | 2060 |
+___+_____+___+_____+
```

LEFT OUTER JOIN

The HiveQL LEFT OUTER JOIN returns all the rows from the left table, even if there are no matches in the right table. This means, if the ON clause matches 0 (zero) records in the right table, the JOIN still returns a row in the result, but with NULL in each column from the right table.

A LEFT JOIN returns all the values from the left table, plus the matched values from the right table, or NULL in case of no matching JOIN predicate.

The following query demonstrates LEFT OUTER JOIN between CUSTOMER and ORDER tables:

```
hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE
FROM CUSTOMERS c
LEFT OUTER JOIN ORDERS o
ON (c.ID = o.CUSTOMER_ID);
```

On successful execution of the query, you get to see the following response:

```
+___+_____+_____+_____+
| ID | NAME   | AMOUNT | DATE           |
+___+_____+_____+_____+
| 1 | Ramesh | NULL   | NULL           |
| 2 | Khilan | 1560   | 2009-11-20 00:00:00 |
| 3 | kaushik | 3000   | 2009-10-08 00:00:00 |
| 3 | kaushik | 1500   | 2009-10-08 00:00:00 |
| 4 | Chaitali | 2060   | 2008-05-20 00:00:00 |
| 5 | Hardik | NULL   | NULL           |
| 6 | Komal  | NULL   | NULL           |
| 7 | Muffy  | NULL   | NULL           |
+___+_____+_____+_____+
```

RIGHT OUTER JOIN

The HiveQL RIGHT OUTER JOIN returns all the rows from the right table, even if there are no matches in the left table. If the ON clause matches 0 (zero) records in the left table, the JOIN still returns a row in the result, but with NULL in each column from the left table.

A RIGHT JOIN returns all the values from the right table, plus the matched values from the left table, or NULL in case of no matching join predicate.

The following query demonstrates RIGHT OUTER JOIN between the CUSTOMER and ORDER tables.

```
notranslate"> hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c
RIGHT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER_ID);
```

On successful execution of the query, you get to see the following response:

```
+___+_____+___+_____+
```

ID	NAME	AMOUNT	DATE
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

FULL OUTER JOIN

The HiveQL FULL OUTER JOIN combines the records of both the left and the right outer tables that fulfil the JOIN condition. The joined table contains either all the records from both the tables, or fills in NULL values for missing matches on either side.

The following query demonstrates FULL OUTER JOIN between CUSTOMER and ORDER tables:

```
hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE
FROM CUSTOMERS c
FULL OUTER JOIN ORDERS o
ON (c.ID = o.CUSTOMER_ID);
```

On successful execution of the query, you get to see the following response:

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

Bucketing

- Bucketing concept is based on (hashing function on the bucketed column) mod (by total number of buckets). The hash_function depends on the type of the bucketing column.

- Records with the same bucketed column will always be stored in the same bucket.

- We use CLUSTERED BY clause to divide the table into buckets.

- Physically, each bucket is just a file in the table directory, and Bucket numbering is 1-based.
- Bucketing can be done along with Partitioning on Hive tables and even without partitioning.
- Bucketed tables will create almost equally distributed data file parts, unless there is skew in data.
- Bucketing is enabled by setting `hive.enforce.bucketing= true;`

Advantages

- Bucketed tables offer efficient sampling than by non-bucketed tables. With sampling, we can try out queries on a fraction of data for testing and debugging purpose when the original data sets are very huge.
- As the data files are equal sized parts, map-side joins will be faster on bucketed tables than non-bucketed tables.
- Bucketing concept also provides the flexibility to keep the records in each bucket to be sorted by one or more columns. This makes map-side joins even more efficient, since the join of each bucket becomes an efficient merge-sort.

Bucketing Vs Partitioning

- Partitioning helps in elimination of data, if used in WHERE clause, where as bucketing helps in organizing data in each partition into multiple files, so that the same set of data is always written in same bucket.
- Bucketing helps a lot in joining of columns.
- Hive Bucket is nothing but another technique of decomposing data or decreasing the data into more manageable parts or equal parts.

Sampling

- TABLESAMPLE() gives more disordered and random records from a table as compared to LIMIT.
- We can sample using the rand() function, which returns a random number.

```
SELECT * from users TABLESAMPLE(BUCKET 3 OUT OF 10 ON rand()) s;
```

```
SELECT * from users TABLESAMPLE(BUCKET 3 OUT OF 10 ON rand()) s;
```

- Here rand() refers to any random column.
- The denominator in the bucket clause represents the number of buckets into which data will be hashed.
- The numerator is the bucket number selected.

```
SELECT * from users TABLESAMPLE(BUCKET 2 OUT OF 4 ON name) s;
```

- If the columns specified in the TABLESAMPLE clause match the columns in the CLUSTERED BY clause, TABLESAMPLE queries only scan the required hash partitions of the table.

```
SELECT * FROM buck_users TABLESAMPLE(BUCKET 1 OUT OF 2 ON id) s LIMIT 1;
```

Joins and Types

Reduce-Side Join

- If datasets are large, reduce side join takes place.

Map-Side Join

- In case one of the dataset is small, map side join takes place. • In map side join, a local job runs to create hash-table from content of HDFS file and sends it to every node.

SET hive.auto.convert.join =true;

Bucket Map Join

- The data must be bucketed on the keys used in the ON clause and the number of buckets for one table must be a multiple of the number of buckets for the other table. • When these conditions are met, Hive can join individual buckets between tables in the map phase, because it does not have to fetch the entire content of one table to match against each bucket in the other table. • set hive.optimize.bucketmapjoin =true; • SET hive.auto.convert.join =true;

SMBM Join

- Sort-Merge-Bucket (SMB) joins can be converted to SMB map joins as well.
- SMB joins are used wherever the tables are sorted and bucketed.
- The join boils down to just merging the already sorted tables, allowing this operation to be faster than an ordinary map-join.
- set hive.enforce.sortmergebucketmapjoin =false;
- set hive.auto.convert.sortmerge.join =true;
- set hive.optimize.bucketmapjoin = true;
- set hive.optimize.bucketmapjoin.sortedmerge = true;

LEFT SEMI JOIN

- A left semi-join returns records from the lefthand table if records are found in the righthand table that satisfy the ON predicates.
 - It's a special, optimized case of the more general inner join.
 - Most SQL dialects support an IN ... EXISTS construct to do the same thing.
 - SELECT and WHERE clauses can't reference columns from the righthand table.
 - Right semi-joins are not supported in Hive.
-

- The reason semi-joins are more efficient than the more general inner join is as follows:
- For a given record in the lefthand table, Hive can stop looking for matching records in the righthand table as soon as any match is found.
- At that point, the selected columns from the lefthand table record can be projected
- A file format is a way in which information is stored or encoded in a computer file.
- In Hive it refers to how records are stored inside the file.
- InputFormat reads key-value pairs from files.
- As we are dealing with structured data, each record has to be its own structure.
- How records are encoded in a file defines a file format.
- These file formats mainly vary between data encoding, compression rate, usage of space and disk I/O.
- Hive does not verify whether the data that you are loading matches the schema for the table or not. •However, it verifies if the file format matches the table definition or not.

SerDe in Hive

- The SerDe interface allows you to instruct Hive as to how a record should be processed.
- A SerDe is a combination of a Serializer and a Deserializer (hence, Ser-De).
- The Deserializer interface takes a string or binary representation of a record, and translates it into a Java object that Hive can manipulate.
- The Serializer, however, will take a Java object that Hive has been working with, and turn it into something that Hive can write to HDFS or another supported system.
- Commonly, Deserializers are used at query time to execute SELECT statements, and Serializers are used when writing data, such as through an INSERT-SELECT statement.

CSVSerDe

- Use ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
- Define following in SERDEPROPERTIES

```
( " separatorChar " = < value_of_separator
, " quoteChar " = < value_of_quote_character ,
" escapeChar " = < value_of_escape_character
```

)

JSONSerDe

• Include `hive-hcatalog-core-0.14.0.jar` • Use `ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'`

RegexSerDe

• It is used in case of pattern matching. • Use `ROW FORMAT SERDE`

`'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'`

• In `SERDEPROPERTIES`, define input pattern and output fields.

For Example

• `input.regex = '(.)/(.)(@(.*)'` • `output.format.string' = '1 s 2 s 3 s';`

USE PARTITIONING AND BUCKETING

• Partitioning a table stores data in sub-directories categorized by table location, which allows Hive to exclude unnecessary data from queries without reading all the data every time a new query is made.

• Hive does support Dynamic Partitioning (DP) where column values are only known at EXECUTION TIME. To enable Dynamic Partitioning :

`SET hive.exec.dynamic.partition=true;`

• Another situation we want to protect against dynamic partition insert is that the user may accidentally specify all partitions to be dynamic partitions without specifying one static partition, while the original intention is to just overwrite the sub-partitions of one root partition.

`SET hive.exec.dynamic.partition.mode=strict;`

To enable bucketing:

`SET hive.enforce.bucketing=true;`

Optimizations in Hive

• Use Denormalisation , Filtering and Projection as early as possible to reduce data before join.

• Join is a costly affair and requires extra map-reduce phase to accomplish query job. With Denormalisation, the data is present in the same table so there is no need for any joins, hence the selects are very fast.

- As join requires data to be shuffled across nodes, use filtering and projection as early as possible to reduce data before join.

TUNE CONFIGURATIONS

- To increase number of mapper, reduce split size :

SET mapred.max.split.size =1000000; (~1 MB)

- Compress map/reduce output

SET mapred.compress.map.output =true;

SET mapred.output.compress =true;

- Parallel execution

- Applies to MapReduce jobs that can run in parallel, for example jobs processing different source tables before a join.

SET hive.exec.parallel =true;

USE ORCFILE

- Hive supports ORCfile , a new table storage format that sports fantastic speed improvements through techniques like predicate push-down, compression and more.
- Using ORCFile for every HIVE table is extremely beneficial to get fast response times for your HIVE queries.

USE TEZ

- With Hadoop2 and Tez , the cost of job submission and scheduling is minimized.
- Also Tez does not restrict the job to be only Map followed by Reduce; this implies that all the query execution can be done in a single job without having to cross job boundaries.
- Let's look at an example. Consider a click-stream event table:

```
CREATE TABLE clicks (  
  timestamp date,  
  sessionID string,  
  url string,  
  source_ip string  
)  
STORED as ORC  
tblproperties ("orc.compress" = "SNAPPY");
```

- Each record represents a click event, and we would like to find the latest URL for each sessionID
 - One might consider the following approach:
-

```
SELECT clicks.sessionID, clicks.url FROM clicks inner join (select sessionID, max(timestamp) as max_ts from clicks group by sessionID) latest ON clicks.sessionID = latest.sessionID and clicks.timestamp = latest.max_ts;
```

- In the above query, we build a sub-query to collect the timestamp of the latest event in each session, and then use an inner join to filter out the rest.

- While the query is a reasonable solution—from a functional point of view—it turns out there's a better way to re-write this query as follows:

```
SELECT ranked_clicks.sessionID , ranked_clicks.url FROM (SELECT sessionID , url , RANK() over (partition by sessionID,order by timestamp desc ) as rank FROM clicks) ranked_clicks WHERE ranked_clicks.rank =1;
```

- Here, we use Hive's OLAP functionality (OVER and RANK) to achieve the same thing, but without a Join.

- Clearly, removing an unnecessary join will almost always result in better performance, and when using big data this is more important than ever.

MAKING MULTIPLE PASS OVER SAME DATA

- Hive has a special syntax for producing multiple aggregations from a single pass through a source of data, rather than rescanning it for each aggregation.

- This change can save considerable processing time for large input data sets.

- For example, each of the following two queries creates a table from the same source table, history:

```
INSERT OVERWRITE TABLE sales
```

```
SELECT * FROM history WHERE action='purchased';
```

```
INSERT OVERWRITE TABLE credits
```

```
SELECT * FROM history WHERE action='returned';
```

Optimizations in Hive

- This syntax is correct, but inefficient.

- The following rewrite achieves the same thing, but using a single pass through the source history table:

```
FROM history
```

```
INSERT OVERWRITE sales SELECT * WHERE action='purchased'
```

```
INSERT OVERWRITE credits SELECT * WHERE action='returned';
```

What is Apache Pig

Apache Pig is a high-level data flow platform for executing MapReduce programs of Hadoop. The language used for Pig is Pig Latin.

The Pig scripts get internally converted to Map Reduce jobs and get executed on data stored in HDFS. Apart from that, Pig can also execute its job in Apache Tez or Apache Spark.

Pig can handle any type of data, i.e., structured, semi-structured or unstructured and stores the corresponding results into Hadoop Data File System. Every task which can be achieved using PIG can also be achieved using java used in MapReduce.

Features of Apache Pig

Let's see the various uses of Pig technology.

1) Ease of programming

Writing complex java programs for map reduce is quite tough for non-programmers. Pig makes this process easy. In the Pig, the queries are converted to MapReduce internally.

2) Optimization opportunities

It is how tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.

3) Extensibility

A user-defined function is written in which the user can write their logic to execute over the data set.

4) Flexible

It can easily handle structured as well as unstructured data.

5) In-built operators

It contains various type of operators such as sort, filter and joins.

Differences between Apache MapReduce and PIG

Advantages of Apache Pig

- Less code - The Pig consumes less line of code to perform any operation.
- Reusability - The Pig code is flexible enough to reuse again.



- Nested data types - The Pig provides a useful concept of nested data types like tuple, bag, and map.

Pig Latin

The Pig Latin is a data flow language used by Apache Pig to analyze the data in Hadoop. It is a textual language that abstracts the programming from the Java MapReduce idiom into a notation.

Pig Latin Statements

The Pig Latin statements are used to process the data. It is an operator that accepts a relation as an input and generates another relation as an output.

- It can span multiple lines.
- Each statement must end with a semi-colon.
- It may include expression and schemas.
- By default, these statements are processed using multi-query execution

Pig Latin Conventions

Convention	Description
()	The parenthesis can enclose one or more items. It can also be used to indicate the tuple data type. Example - (10, xyz, (3,6,9))
[]	The straight brackets can enclose one or more items. It can also be used to indicate the map data type. Example - [INNER OUTER]
{ }	The curly brackets enclose two or more items. It can also be used to indicate the bag data type Example - { block nested_block }
...	The horizontal ellipsis points indicate that you can repeat a portion of the code. Example - cat path [path ...]

Latin Data Types

Simple Data Types

Type	Description
int	It defines the signed 32-bit integer. Example - 2
long	It defines the signed 64-bit integer. Example - 2L or 2l
float	It defines 32-bit floating point number. Example - 2.5F or 2.5f or 2.5e2f or 2.5.E2F
double	It defines 64-bit floating point number. Example - 2.5 or 2.5 or 2.5e2f or 2.5.E2F
chararray	It defines character array in Unicode UTF-8 format. Example - javatpoint
bytearray	It defines the byte array.
boolean	It defines the boolean type values. Example - true/false
datetime	It defines the values in datetime order. Example - 1970-01- 01T00:00:00.000+00:00
biginteger	It defines Java BigInteger values. Example - 5000000000000
bigdecimal	It defines Java BigDecimal values. Example - 52.232344535345

Pig Data Types

Apache Pig supports many data types. A list of Apache Pig Data Types with description and examples are given below.

Type	Description	Example
Int	Signed 32 bit integer	2
Long	Signed 64 bit integer	15L or 15l
Float	32 bit floating point	2.5f or 2.5F
Double	32 bit floating point	1.5 or 1.5e2 or 1.5E2
charArray	Character array	hello javatpoint
byteArray	BLOB(Byte array)	
tuple	Ordered set of fields	(12,43)
bag	Collection of tuples	{(12,43),(54,28)}
map	collection of tuples	[open#apache]

Apache Pig Execution Modes

You can run Apache Pig in two modes, namely, **Local Mode** and **HDFS mode**.

Local Mode

In this mode, all the files are installed and run from your local host and local file system. There is no need of Hadoop or HDFS. This mode is generally used for testing purpose.

MapReduce Mode

MapReduce mode is where we load or process the data that exists in the Hadoop File System (HDFS) using Apache Pig. In this mode, whenever we execute the Pig Latin statements to process the data, a MapReduce job is invoked in the back-end to perform a particular operation on the data that exists in the HDFS.

Apache Pig Execution Mechanisms

Apache Pig scripts can be executed in three ways, namely, interactive mode, batch mode, and embedded mode.

- **Interactive Mode** (Grunt shell) – You can run Apache Pig in interactive mode using the Grunt shell. In this shell, you can enter the Pig Latin statements and get the output (using Dump operator).
- **Batch Mode** (Script) – You can run Apache Pig in Batch mode by writing the Pig Latin script in a single file with **.pig** extension.
- **Embedded Mode** (UDF) – Apache Pig provides the provision of defining our own functions (**User Defined Functions**) in programming languages such as Java, and using them in our script.
- Given below in the table are some frequently used Pig Commands.

Command	Function
load	Reads data from the system
Store	Writes data to file system
foreach	Applies expressions to each record and outputs one or more records
filter	Applies predicate and removes records that do not return true

Group/cogroup	Collects records with the same key from one or more inputs
join	Joins two or more inputs based on a key
order	Sorts records based on a key
distinct	Removes duplicate records
union	Merges data sets
split	Splits data into two or more sets based on filter conditions
stream	Sends all records through a user-provided binary
dump	Writes output to stdout

limit	Limits the number of records
-------	------------------------------

Complex Types

Type	Description
tuple	It defines an ordered set of fields. Example - (15,12)
bag	It defines a collection of tuples. Example - {(15,12), (12,15)}
map	It defines a set of key-value pairs. Example - [open#apache]

Pig Latin – Relational Operations

The following table describes the relational operators of Pig Latin.

Operator	Description
Loading and Storing	
LOAD	To Load the data from the file system (local/HDFS) into a relation.
STORE	To save a relation to the file system (local/HDFS).
Filtering	

FILTER	To remove unwanted rows from a relation.
DISTINCT	To remove duplicate rows from a relation.
FOREACH, GENERATE	To generate data transformations based on columns of data.
STREAM	To transform a relation using an external program.
Grouping and Joining	
JOIN	To join two or more relations.
COGROUP	To group the data in two or more relations.
GROUP	To group the data in a single relation.
CROSS	To create the cross product of two or more relations.
Sorting	
ORDER	To arrange a relation in a sorted order based on one or more fields (ascending or descending).
LIMIT	To get a limited number of tuples from a relation.
Combining and Splitting	
UNION	To combine two or more relations into a single relation.
SPLIT	To split a single relation into two or more relations.

Diagnostic Operators

DUMP	To print the contents of a relation on the console.
DESCRIBE	To describe the schema of a relation.
EXPLAIN	To view the logical, physical, or MapReduce execution plans to compute a relation.
ILLUSTRATE	To view the step-by-step execution of a series of statements.

Eval Functions

Given below is the list of **eval** functions provided by Apache Pig.

S.N.	Function & Description
1	AVG() To compute the average of the numerical values within a bag.
2	BagToString() To concatenate the elements of a bag into a string. While concatenating, we can place a delimiter between these values (optional).
3	CONCAT() To concatenate two or more expressions of same type.
4	COUNT() To get the number of elements in a bag, while counting the number of tuples in a bag.
5	COUNT_STAR()

	It is similar to the COUNT() function. It is used to get the number of elements in a bag.
6	DIFF() To compare two bags (fields) in a tuple.
7	IsEmpty() To check if a bag or map is empty.
8	MAX() To calculate the highest value for a column (numeric values or chararrays) in a single-column bag.
9	MIN() To get the minimum (lowest) value (numeric or chararray) for a certain column in a single-column bag.
10	PluckTuple() Using the Pig Latin PluckTuple() function, we can define a string Prefix and filter the columns in a relation that begin with the given prefix.
11	SIZE() To compute the number of elements based on any Pig data type.
12	SUBTRACT() To subtract two bags. It takes two bags as inputs and returns a bag which contains the tuples of the first bag that are not in the second bag.
13	SUM() To get the total of the numeric values of a column in a single-column bag.
14	TOKENIZE() To split a string (which contains a group of words) in a single tuple and return a bag which contains the output of the split operation.

Apache Pig provides extensive support for **User Defined Functions (UDF's)**. Using these UDF's, we can define our own functions and use them. The UDF support is provided in six programming languages, namely, Java, Jython, Python, JavaScript, Ruby and Groovy.

For writing UDF's, complete support is provided in Java and limited support is provided in all the remaining languages. Using Java, you can write UDF's involving all parts of the processing like data load/store, column transformation, and aggregation. Since Apache Pig has been written in Java, the UDF's written using Java language work efficiently compared to other languages.

In Apache Pig, we also have a Java repository for UDF's named **Piggybank**. Using Piggybank, we can access Java UDF's written by other users, and contribute our own UDF's.

Types of UDF's in Java

While writing UDF's using Java, we can create and use the following three types of functions –

- **Filter Functions** – The filter functions are used as conditions in filter statements. These functions accept a Pig value as input and return a Boolean value.
- **Eval Functions** – The Eval functions are used in FOREACH-GENERATE statements. These functions accept a Pig value as input and return a Pig result.
- **Algebraic Functions** – The Algebraic functions act on inner bags in a FOREACHGENERATE statement. These functions are used to perform full MapReduce operations on an inner bag.

Writing UDF's using Java

To write a UDF using Java, we have to integrate the jar file **Pig-0.15.0.jar**. In this section, we discuss how to write a sample UDF using Eclipse. Before proceeding further, make sure you have installed Eclipse and Maven in your system.

Follow the steps given below to write a UDF function –

- Open Eclipse and create a new project (say **myproject**).
- Convert the newly created project into a Maven project.
- Copy the following content in the pom.xml. This file contains the Maven dependencies for Apache Pig and Hadoop-core jar files.

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0http://maven.apache
.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>Pig_Udf</groupId>
  <artifactId>Pig_Udf</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <build>
    <sourceDirectory>src</sourceDirectory>
```

```

<plugins>
  <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.3</version>
    <configuration>
      <source>1.7</source>
      <target>1.7</target>
    </configuration>
  </plugin>
</plugins>
</build>

<dependencies>

  <dependency>
    <groupId>org.apache.pig</groupId>
    <artifactId>pig</artifactId>
    <version>0.15.0</version>
  </dependency>

  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-core</artifactId>
    <version>0.20.2</version>
  </dependency>

</dependencies>

</project>

```

- Save the file and refresh it. In the **Maven Dependencies** section, you can find the downloaded jar files.
- Create a new class file with name **Sample_Eval** and copy the following content in it.

```

import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

public class Sample_Eval extends EvalFunc<String>{

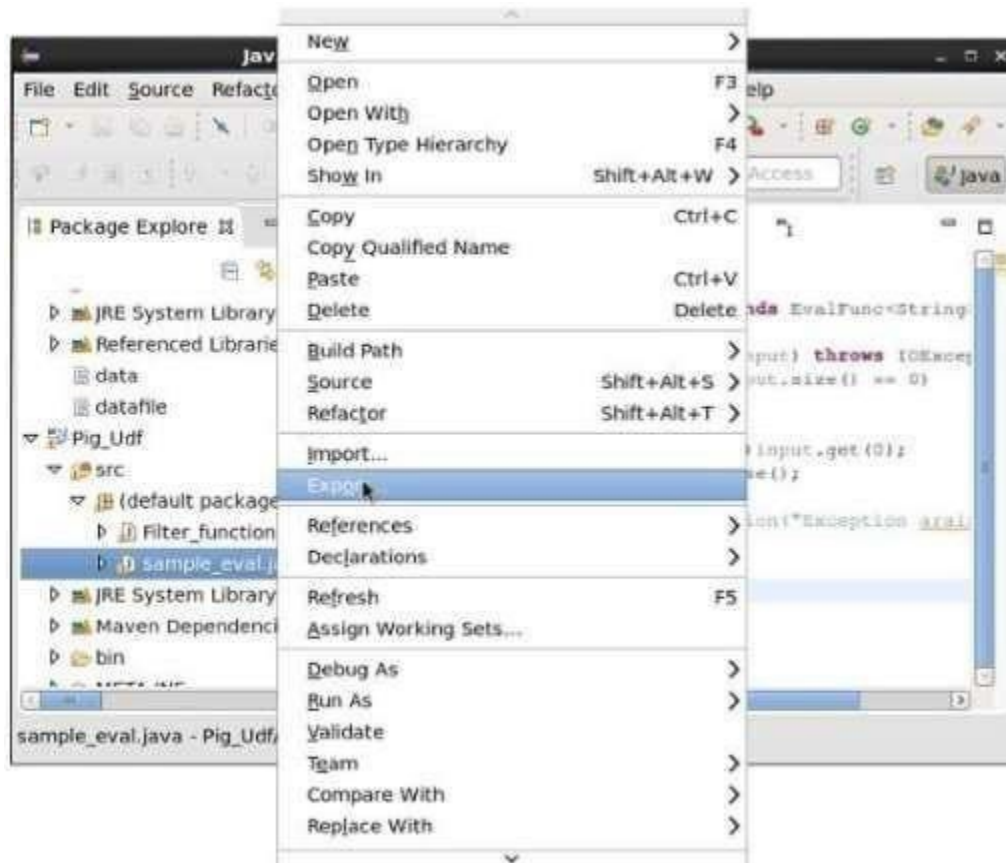
  public String exec(Tuple input) throws IOException {
    if (input == null || input.size() == 0)
      return null;
    String str = (String)input.get(0);
    return str.toUpperCase();
  }
}

```

```
}  
}
```

While writing UDF's, it is mandatory to inherit the EvalFunc class and provide implementation to **exec()** function. Within this function, the code required for the UDF is written. In the above example, we have return the code to convert the contents of the given column to uppercase.

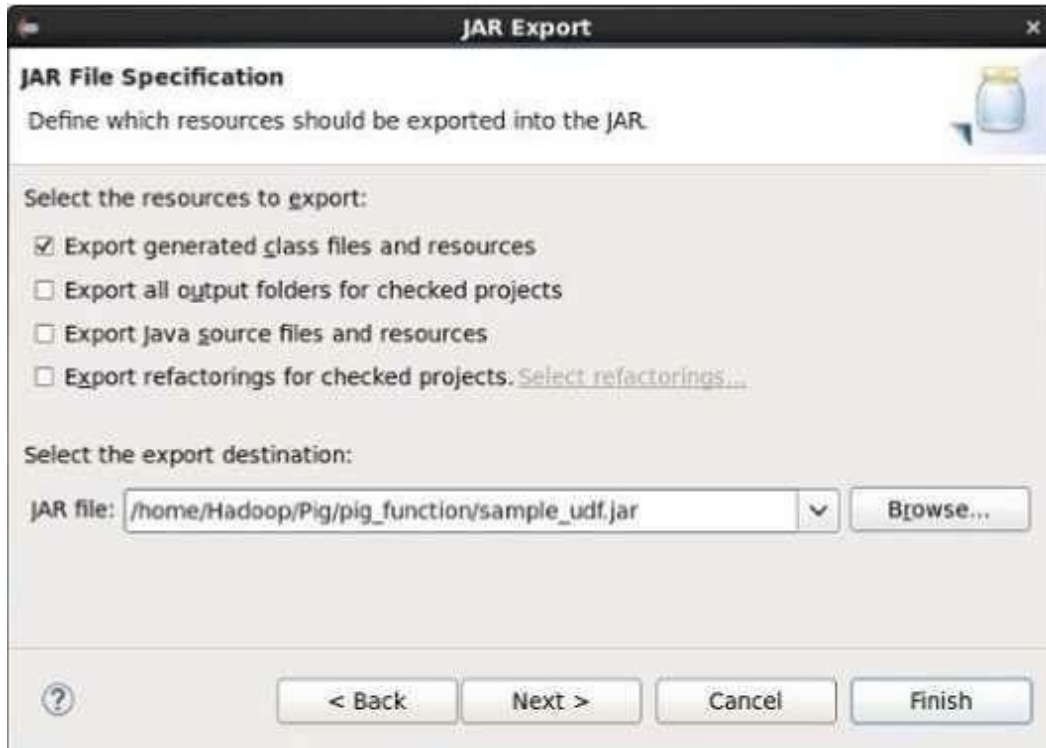
- After compiling the class without errors, right-click on the Sample_Eval.java file. It gives you a menu. Select **export** as shown in the following screenshot.



- On clicking **export**, you will get the following window. Click on **JAR file**.



- Proceed further by clicking **Next>** button. You will get another window where you need to enter the path in the local file system, where you need to store the jar file.



- Finally click the **Finish** button. In the specified folder, a Jar file **sample_udf.jar** is created. This jar file contains the UDF written in Java.

Using the UDF

After writing the UDF and generating the Jar file, follow the steps given below –

Step 1: Registering the Jar file

After writing UDF (in Java) we have to register the Jar file that contain the UDF using the Register operator. By registering the Jar file, users can intimate the location of the UDF to Apache Pig.

Syntax

Given below is the syntax of the Register operator.

REGISTER path;

Example

As an example let us register the sample_udf.jar created earlier in this chapter.

Start Apache Pig in local mode and register the jar file sample_udf.jar as shown below.

```
$cd PIG_HOME/bin  
$./pig -x local
```

```
REGISTER '$PIG_HOME/sample_udf.jar'
```

Note – assume the Jar file in the path – /\$PIG_HOME/sample_udf.jar

Step 2: Defining Alias

After registering the UDF we can define an alias to it using the **Define** operator.

Syntax

Given below is the syntax of the Define operator.

```
DEFINE alias {function | [`command` [input] [output] [ship] [cache] [stderr] ] };
```

Example

Define the alias for sample_eval as shown below.

```
DEFINE sample_eval sample_eval();
```

Step 3: Using the UDF

After defining the alias you can use the UDF same as the built-in functions. Suppose there is a file named emp_data in the HDFS /**Pig_Data**/ directory with the following content.

```
001,Robin,22,newyork  
002,BOB,23,Kolkata  
003,Maya,23,Tokyo  
004,Sara,25,London  
005,David,23,Bhuwaneshwar  
006,Maggy,22,Chennai
```

007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuwaneshwar
012,Kelly,22,Chennai

And assume we have loaded this file into Pig as shown below.

```
grunt> emp_data = LOAD 'hdfs://localhost:9000/pig_data/emp1.txt' USING PigStorage(',')  
as (id:int, name:chararray, age:int, city:chararray);
```

Let us now convert the names of the employees in to upper case using the UDF **sample_eval**.

```
grunt> Upper_case = FOREACH emp_data GENERATE sample_eval(name);
```

Verify the contents of the relation **Upper_case** as shown below.

```
grunt> Dump Upper_case;
```

(ROBIN)
(BOB)
(MAYA)
(SARA)
(DAVID)
(MAGGY)
(ROBERT)
(SYAM)
(MARY)
(SARAN)
(STACY)
(KELLY)

Parameter substitution in Pig

Earlier I have discussed about writing [reusable scripts using Apache Hive](#), now we see how to achieve same functionality using Pig Latin.

Pig Latin has an option called [param](#), using this we can write dynamic scripts .

Assume ,we have a file called numbers with below data.

12

23

34

12

56

34

57

12

```
Numbers = load '/data/numbers' as (number:int);  
specificNumber = filter numbers by number==12;  
Dump specificNumber;
```

Usually we write above code in a file .let us assume we have written it in a file called numbers.pig And we write code from file using

```
Pig -f/path/to/numbers.pig
```

Later if we want to see only numbers equals to 34, then we change second line to

```
specificNumber = filter numbers by number==34;
```

and we re-run the code using same command.

But Its not a good practice to touch the code in production ,so we can make this script dynamic by using -param option of Piglatin.

Whatever values we want to decide at the time of running we make them dynamic .now we want to decide number to be filtered at the time running job,we can write second line like below.

```
specificNumber = filter numbers by number==$dyanumber
```

and we run code like below.

```
Pig -param dyanumber=12 -f numbers.pig
```

Assume we even want to take path at the time of running script, now we write code like below

```
Numbers = load '$path' as (number:int);  
specificNumber = filter numbers by number=='$ dyanumber';
```

```
Dump specificNumber;
```

And run like below

```
Pig -param path=/data/path -param dynanumber =34 -f numbers.pig
```

If you feel this code is missing readability, we can specify all these dynamic values in a file like below

```
##Dyna.params (file name)
```

```
Path = /data/numbers
```

```
dynanumber = 34
```

Then you can run script with param-file option like below.

```
Pig -param-file dyna.params -f numbers.pig
```

Pig Latin provides four different types of diagnostic operators –

- Dump operator
- Describe operator
- Explanation operator
- Illustration operator

Word	Count	Example	Using	Pig	Script:
------	-------	---------	-------	-----	---------

```
lines = LOAD '/user/hadoop/HDFS_File.txt' AS (line:chararray);  
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) as word;  
grouped = GROUP words BY word;  
wordcount = FOREACH grouped GENERATE group, COUNT(words);  
DUMP wordcount;
```

The above pig script, first splits each line into words using the **TOKENIZE** operator. The tokenize function creates a bag of words. Using the **FLATTEN** function, the bag is

converted into a tuple. In the third statement, the words are grouped together so that the count can be computed which is done in fourth statement.

Pig at Yahoo

Pig was initially developed by Yahoo! for its data scientists who were using Hadoop. It was incepted to focus mainly on analysis of large datasets rather than on writing mapper and reduce functions. This allowed users to focus on what they want to do rather than bothering with how its done. On top of this with Pig language you have the facility to write commands in other languages like Java, Python etc. Big applications that can be built on Pig Latin can be custom built for different companies to serve different tasks related to data management. Pig systemizes all the branches of data and relates it in a manner that when the time comes, filtering and searching data is checked efficiently and quickly.

Pig Versus Hive

Pig Vs Hive

Here are some basic difference between Hive and Pig which gives an idea of which to use depending on the type of data and purpose.

Pig	Hive
Used by Programmers and Researchers	Used by Analysts
Used for Programming	Used for Reporting
Procedural data-flow language	Declarative SQLish language
Works on the Client side of a Cluster	Works on the Server side of a Cluster
For Semi-Structured Data	For Structured Data

Why Go for Hive When Pig is There?

The tabular column below gives a comprehensive comparison between the two. The Hive can be used in places where partitions are necessary and when it is essential to define and create cross-language services for numerous languages.



Features	Hive	Pig
Language	SQL-like	PigLatin
Schemas/Types	Yes (explicit)	Yes (implicit)
Partitions	Yes	No
Server	Optional (Thrift)	No
User Defined Functions (UDF)	Yes (Java)	Yes (Java)
Custom Serializer/Deserializer	Yes	Yes
DFS Direct Access	Yes (implicit)	Yes (explicit)
Join/Order/Sort	Yes	Yes
Shell	Yes	Yes
Streaming	Yes	Yes
Web Interface	Yes	No
JDBC/ODBC	Yes (limited)	No