



Python

Dr. Panem Charanarur

Overview

- History
- Installing & Running Python

Brief History of Python

- Invented in the Netherlands, early 90s by Guido van Rossum
- Named after Monty Python
- Open sourced from the beginning
- Considered a scripting language, but is much more
- Scalable, object oriented and functional from the beginning
- Used by Google from the beginning
- Increasingly popular

Python's Benevolent Dictator For Life

“Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered.”

- Guido van Rossum



http://docs.python.org/

The image shows a screenshot of a web browser displaying the Python 2.6.1 documentation website. The browser's address bar shows the URL `http://docs.python.org/`. The page title is "Overview — Python v2.6.1 documentation". The page layout includes a left sidebar with navigation links and a main content area with a welcome message and a list of documentation sections.

Overview — Python v2.6.1 documentation

Python v2.6.1 documentation » modules | index

Download

Download these documents

Other resources

- FAQs
- Introductions
- Guido's Essays
- New-style Classes
- PEP Index
- Beginner's Guide
- Topic Guides
- Book List
- Audio/Visual Talks
- Other Doc Collections

Previous versions

Quick search

Enter search terms or a module, class or function name.

Go

Python v2.6.1 documentation

Welcome! This is the documentation for Python 2.6.1, last updated Jan 29, 2009.

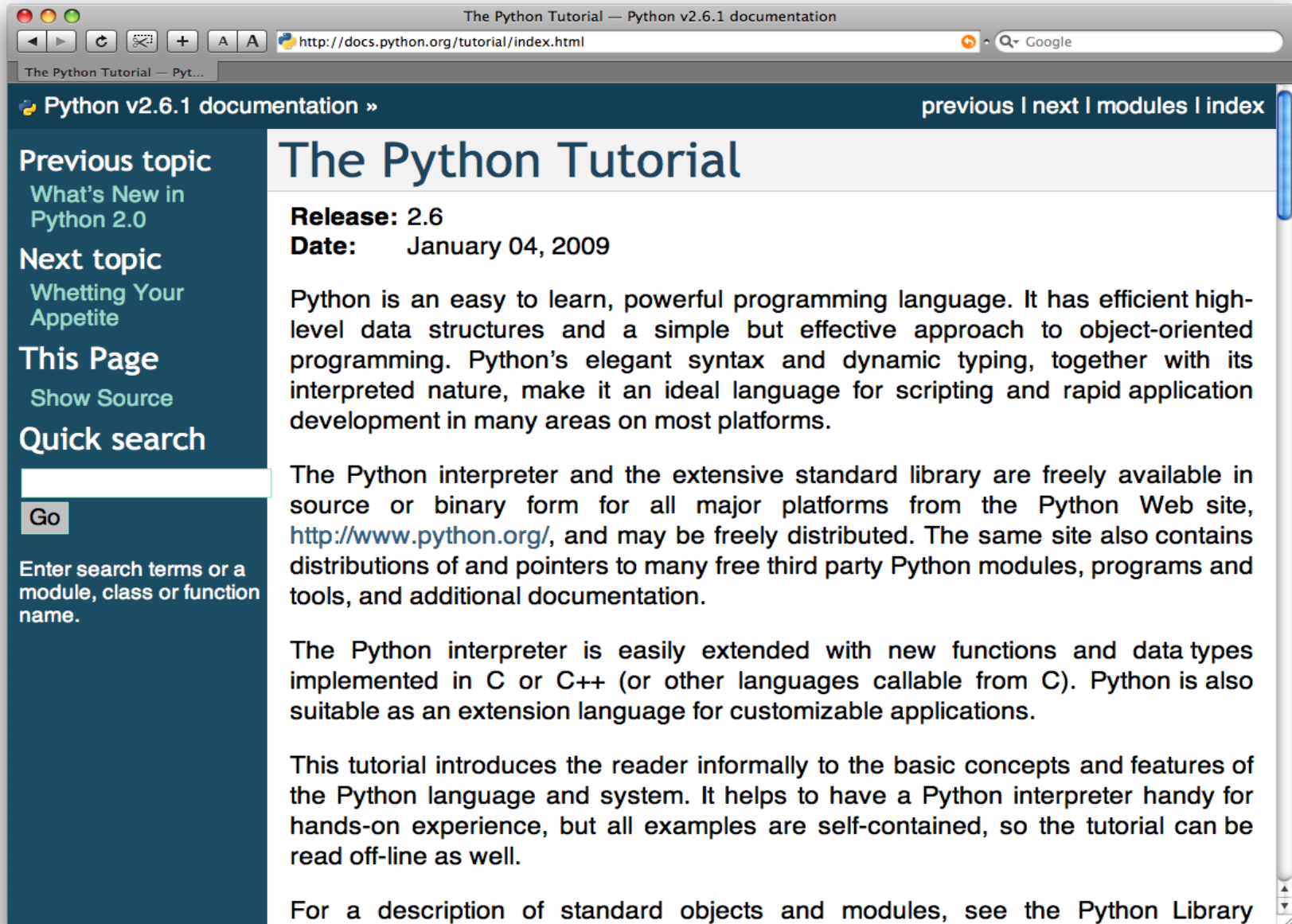
Parts of the documentation:

- What's new in Python 2.6?**
or all "What's new" documents since 2.0
- Tutorial**
start here
- Using Python**
how to use Python on different platforms
- Language Reference**
describes syntax and language elements
- Library Reference**
keep this under your pillow
- Python HOWTOs**
in-depth documents on specific topics
- Extending and Embedding**
tutorial for C/C++ programmers
- Python/C API**
reference for C/C++ programmers
- Installing Python Modules**
information for installers & sys-admins
- Distributing Python Modules**
sharing modules with others
- Documenting Python**
guide for documentation authors

Indices and tables:

- Global Module Index**
quick access to all modules
- General Index**
all functions, classes, terms
- Glossary**
the most important terms explained
- Search page**
search this documentation
- Complete Table of Contents**
lists all sections and subsections

The Python tutorial is good!



The screenshot shows a web browser window with the title "The Python Tutorial — Python v2.6.1 documentation". The address bar shows the URL "http://docs.python.org/tutorial/index.html". The page has a dark blue header with "Python v2.6.1 documentation »" on the left and navigation links "previous | next | modules | index" on the right. A left sidebar contains links for "Previous topic" (What's New in Python 2.0), "Next topic" (Whetting Your Appetite), "This Page" (Show Source), and a "Quick search" section with a text input field and a "Go" button. The main content area is titled "The Python Tutorial" and contains three paragraphs of text.

The Python v2.6.1 documentation » previous | next | modules | index

Previous topic
What's New in Python 2.0

Next topic
Whetting Your Appetite

This Page
Show Source

Quick search

Go

Enter search terms or a module, class or function name.

The Python Tutorial

Release: 2.6
Date: January 04, 2009

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, <http://www.python.org/>, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off-line as well.

For a description of standard objects and modules, see the Python Library

Warmup

Windows:

- Open anaconda prompt
- Type `conda -V`
- If you get an error, install Anaconda:
<https://docs.anaconda.com/anaconda/install/windows/>
 - #8 is important: **DO NOT** add to your path
- If no error, consider upgrading conda:
`conda update conda`
- Clone <https://github.com/Harvard-IACS/2019-CS109B>
(or pull the latest if you've already cloned)

Mac:

- Open a terminal
- Type `conda -V`
- If you get an error, install Anaconda:
<https://docs.anaconda.com/anaconda/install/mac-os/>
- If no error, consider upgrading conda:
`conda update conda`
- Clone <https://github.com/Harvard-IACS/2019-CS109B>
(or pull the latest if you've already cloned)

Goals

- **Set up the tools you'll need for CS109b**
 - In a way that won't mess up your other classes
- **Teach a workflow that will *keep* your installs tidy**
- **User-level understanding of why 'environments' are helpful**
- ***Stretch*: Ability to produce conda environments for future projects**



Basic Python Commands

- 1. Comments:** # symbol is being used for comments in python. For [multiline comments](#), you have to use `"""` symbols or enclosing the comment in the `"""` symbol.
 - **Example:**
`print "Hello World" # this is the comment section.`
 - **Example:**
`""" This is Hello world project. """`
- 2. Type function:** These Python Commands are used to check the type of variable and used inbuilt functions to check.
 - **Example:**
`type (20), its type is int.`
`>>> type (20) < type 'int' >`
 - **Example:**
`type (-1 +j), its type is complex`
`>>> type (-1+j)`
`< type 'complex' >`
- 3. Strings:** It is mainly enclosed in double-quotes.
 - **Example:**
`type ("hello World"), type is string`
`>>> type ("hello World")`
`< type 'str' >`

4. Lists: Lists are mainly enclosed in square bracket. []

- **Example:**

```
type ( [ 1, 2 ] ), type is list
```

```
>>> type ( [ 1, 2, 3 ] )
```

```
< type 'List' >
```

5.Tuple: Tuple are mainly enclosed in parenthesis. ()

- **Example:**

```
type ( 1, 2, 3), a type is a tuple.
```

```
>>> type ( ( 1, 2, 3 ) )
```

```
<type 'tuple' >
```

6. Range: This function is used to create the list of integers. Range(10)

- **Example:**

```
>>> range ( 10 )
```

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

- **Example:**

range(1,10)

```
>>> range (1,10)
```

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

7. Boolean values: This data type helps in retrieving the data in True or false form.

- **Example:**

```
>>> True
```

```
True
```

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

```
< type 'bool' >
```

- **Example:**

```
>>> False
```

```
False
```

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

```
< type 'bool' >
```

8. Operator: Different operator is used for the different functions like division, multiply, addition and subtraction.

- **Example:**

```
>>> 16/2          16/2=8
8
```

- **Example:**

```
>>> 2 * * 1/2    2 **, -, +
1
```

9. Variable and assignment: The assignment statement has variable = expression. Single '=' is used for assignment, and double '=' is used to test for equality.

- **Example:**

```
>>> X= 235
>>> print X
235
>>> Z= 2* X
>>> print Z
470
```

10. Comparison operators: To compare the two values and the [result of the comparison](#) is always boolean value.

- **Example:**

```
>>> 2 < 3
True
```

- 11. Conditional/ decisions:** It is used to make out the decision between two or more values like if-else
- **Example:**
if x=0:
Print "Hello, world."
Else:
Print "Hello, world in Else."
- 12. For Loop:** This Python command is used when iteration and action consist of the same elements.
- **Example:**
for x in [1, 2, 3, 4, 5, 6]:
Print x;
- 13. While loop:** [While loop](#) will never be executed if the condition evaluates to false for the first time.
- **Example:**
x =0
while x<10:
Print x,
X= x+2
- 14. Else in loop:** Loop have optional else for execution.
- **Example:**
for x in [1, 3, 5, 7, 9, 11]:
Print x
Else:
Print "In Else"

15. Break, continue statement: [break statement is used](#) to exit out the loop when particular output is achieved; continue is used to continue with the next iteration of a loop.

- **Example:**

```
if x==0:  
    Print "X is 0"  
    Break  
Else:  
    Print "X is greater than 0."
```

16. Lists: It is the finite number of items, and by assigning a value to list the list value will get changed.

- **Example:**

```
>>> a = [1, "JAY", 34] >>> a [0] 1  
>>> a [0] = 101  
>>> a  
[101, "JAY", 34]
```

17. Length of list: To know the length of list.

- **Example:**

```
>>> X = [1, 4, 67,9] >>> len (X)  
4
```

18. Sublists: It will give you the values between the mentioned start index and the end index.

- **Example:**

```
x [start : end] >>> X [1, 3, 4,6, 7, 8, 9, 0, 2] >>> X [2:5] [4, 6, 7]
```

19. Joining two list: + operator is being used to concatenate 2 lists.

- **Example:**

```
>>> [2, 1, 5] + [0, 4,6,7] [2, 1, 5, 0, 4, 6,7]
```

20. Strings: It is used to check the index to know the character written in string.

- **Example:**

```
>>> x= "hello, world" ""  
>>> x [2] 'l'  
>>> x [5] 'o'
```

21. List methods: The different methods available to in list to perform the function.

- **Example:**

```
X [1, 2,3,4,5]  
>>> X.append (7)  
>>> x  
[1, 2, 3, 4, 5, 7]  
>>>X.insert (0, 0)  
>>> x  
[0, 1, 2, 3, 4, 5]  
>>> X.remove (2)  
>>> x  
[0, 1, 3, 4, 5]  
>>> X.pop (1)  
>>> x  
[2,3,4,5]
```

Python Conditions and If statements:

- Python supports the usual logical conditions from mathematics:
- Equals: $a == b$
- Not Equals: $a != b$
- Less than: $a < b$
- Less than or equal to: $a <= b$
- Greater than: $a > b$
- Greater than or equal to: $a >= b$
- These conditions can be used in several ways, most commonly in "if statements" and loops.
- An "if statement" is written by using the if keyword.

- Example If statement:

```
a = 33
```

```
b = 200
```

```
if b > a:
```

```
    print("b is greater than a")
```

- Elif: The **elif** keyword is python's way of saying "if the previous conditions were not true, then try this condition".

- Example

```
a = 33
```

```
b = 33
```

```
if b > a:
```

```
    print("b is greater than a")
```

```
elif a == b:
```

```
    print("a and b are equal")
```

- Else:The else keyword catches anything which isn't caught by the preceding conditions.

- Example

```
a = 200
```

```
b = 33
```

```
if b > a:
```

```
    print("b is greater than a")
```

```
elif a == b:
```

```
    print("a and b are equal")
```

```
else:
```

```
    print("a is greater than b")
```

- And:The and keyword is a logical operator, and is used to combine conditional statements
- Example:Test if a is greater than b, AND if c is greater than a:

```
a = 200
```

```
b = 33
```

```
c = 500
```

```
if a > b and c > a:
```

```
    print("Both conditions are True")
```

- Python For Loops: A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
- Example

Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)
```

- The break Statement: With the break statement we can stop the loop before it has looped through all the items:
- Example

Exit the loop when x is "banana":

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)  
    if x == "banana":  
        break
```

- The continue Statement: With the continue statement we can stop the current iteration of the loop, and continue with the next
- Example

Do not print banana:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    if x == "banana":  
        continue  
    print(x)
```

- The range() Function: To loop through a set of code a specified number of times, we can use the range() function,
- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.
- Example

Using the range() function:

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

- Python Lists: Lists are used to store multiple items in a single variable.
- Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are [Tuple](#), [Set](#), and [Dictionary](#), all with different qualities and usage.
- Lists are created using square brackets:
- Example

Create a List:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist)
```

- List Length: To determine how many items a list has, use the len() function:
- Example

Print the number of items in the list:

```
thislist = ["apple", "banana", "cherry"]  
print(len(thislist))
```

- List Items - Data Types: List items can be of any data type:
- Example: String, int and boolean data types:

```
list1 = ["apple", "banana", "cherry"]  
list2 = [1, 5, 7, 9, 3]  
list3 = [True, False, False]
```

- Dictionary: Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values
- Example

Create and print a dictionary:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict)
```

- Dictionary Length: To determine how many items a dictionary has, use the len() function:
- Example

```
print(len(thisdict))
```

- Dictionary Items - Data Types
- Example String, int, boolean, and list data types:

```
thisdict = {  
    "brand": "Ford",  
    "electric": False,  
    "year": 1964,  
    "colors": ["red", "white", "blue"]  
}
```

Network: The standard library of Python has full support for network protocols, encoding, and decoding of data and other networking concepts, and it is simpler to write network programs in Python

- There are two levels of network service access in Python. These are:
- Low-Level Access
- High-Level Access
- In the first case, programmers can use and access the basic socket support for the operating system using Python's libraries, and programmers can implement both connection-less and connection-oriented protocols for programming.
- Sockets use protocols for determining the connection type for port-to-port communication between client and server machines. The protocols are used for:
- Domain Name Servers (DNS)
- IP addressing
- E-mail
- FTP (File Transfer Protocol) etc.

- Python has a socket method that let programmers' set-up different types of socket virtually. The syntax for the socket method is:
- Syntax:

g = socket.socket (socket_family, type_of_socket, protocol=value)

For example, if we want to establish a TCP socket, we can write the following code snippet:

- Example:

```
# imports everything from 'socket'
```

```
from socket import *
```

```
# use socket.socket() - function
```

```
tcp1=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

you defined the socket, you can use several methods to manage the connections. Some of the important server socket methods are:

- **listen()**: is used to establish and start TCP listener.
- **bind()**: is used to bind-address (host-name, port number) to the socket.
- **accept()**: is used to TCP client connection until the connection arrives.
- **connect()**: is used to initiate TCP server connection.
- **send()**: is used to send TCP messages.
- **recv()**: is used to receive TCP messages.
- **sendto()**: is used to send UDP messages
- **close()**: is used to close a socket.

- **Selection:**
- Python's `select()` function is a direct interface to the underlying operating system implementation.
- It monitors sockets, open files, and pipes (anything with a `fileno()` method that returns a valid file descriptor) until they become readable or writable, or a communication error occurs. `select()` makes it easier to monitor multiple connections at the same time, and is more efficient than writing a polling loop in Python using socket timeouts, because the monitoring happens in the operating system network layer, instead of the interpreter.
- **Note:** Using Python's file objects with `select()` works for Unix, but is not supported under Windows.
- **Example:**

```
import select
import socket
import sys
import Queue

# Create a TCP/IP socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(0)

# Bind the socket to the port
server_address = ('localhost', 10000)
print >>sys.stderr, 'starting up on %s port %s' %
server_address
server.bind(server_address)

# Listen for incoming connections
server.listen(5)
```

- **Functions:** In Python, a defined function's declaration begins with the keyword `def` and followed by the function name.
- The function may take arguments(s) as input within the opening and closing parentheses, just after the function name followed by a colon.
- After defining the function name and arguments(s) a block of program statement(s) start at the next line and these statement(s) must be indented.
- Here is the syntax of a user defined function.
- **Syntax:**

```
def function_name(argument1, argument2, ...) :  
statement_1    statement_2    ....
```

Example:

```
def nsquare(x, y):  
    return (x*x + 2*x*y + y*y)  
print("The square of the sum of 2 and 3 is : ", nsquare(2, 3))
```

- **Python Iterators:** An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`

Example: Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self
    def __next__(self):
        x = self.a
        self.a += 1
        return x

myclass = MyNumbers()
myiter = iter(myclass)
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Example: Stop after 20 iterations:

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self
    def __next__(self):
        if self.a <= 20:
            x = self.a
            self.a += 1
            return x
        else:
            raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)
for x in myiter:
    print(x)
```

Python Classes:

To create a class, use the keyword **class**.

Example 1: Create a class named MyClass, with a property named x:

```
class MyClass:
```

```
    x = 5
```

Example 2: Create a class named Person, use the `__init__()` function to assign values for name and age:

```
class Person:
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
p1 = Person("John", 36)
```

```
print(p1.name)
```

```
print(p1.age)
```

Python Objects:

Example 1: Create an object named p1, and print the value of x:

```
p1 = MyClass()
```

```
print(p1.x)
```

Example 2: Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
    def myfunc(self):
```

```
        print("Hello my name is " + self.name)
```

```
p1 = Person("John", 36)
```

```
p1.myfunc()
```

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Example

Create a class named Person, with firstname and lastname properties, and a printname method:
class Person:

```
def __init__(self, fname, lname):  
    self.firstname = fname  
    self.lastname = lname  
def printname(self):  
    print(self.firstname, self.lastname)
```

#Use the Person class to create an object, and then execute the printname method:

```
x = Person("John", "Doe")  
x.printname()
```

Composition (Has-A Relation)

Here, by using the class name or by creating the object we can access the members of one class inside another class. It enables creating complex types by combining objects of different classes. It means that a class Composite can contain an object of another class Component. This type of relationship is known as **Has-A Relation**.

Example:

```
class Component:  
    # composite class constructor  
    def __init__(self):  
        print('Component class object created...')  
    # composite class instance method  
    def m1(self):  
        print('Component class m1() method executed...')  
class Composite:  
    # composite class constructor  
    def __init__(self):  
        # creating object of component class  
        self.obj1 = Component()  
        print('Composite class object also created...')  
    # composite class instance method  
    def m2(self):  
        print('Composite class m2() method executed...')  
        # calling m1() method of component class  
        self.obj1.m1()  
# creating object of composite class  
obj2 = Composite()  
# calling m2() method of composite class  
obj2.m2()
```

Python Comments

Comments can be used to explain Python code.

Comments can be used to make the code more readable.

Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a #, and Python will ignore them:

Example

```
#This is a comment
```

```
print("Hello, World!")
```

Python Numbers

There are three numeric types in Python:

int

float

complex

Variables of numeric types are created when you assign a value to them:

Example

```
x = 1 # int
```

```
y = 2.8 # float
```

```
z = 1j # complex
```

```
print(type(x))
```

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

```
print(type(z))
```


Python Math

Python has a set of built-in math functions, including an extensive math module, that allows you to perform mathematical tasks on numbers.

Built-in Math Functions

The min() and max() functions can be used to find the lowest or highest value in an iterable:

Example

```
x = min(5, 10, 25)
y = max(5, 10, 25)
print(x)
print(y)
```

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

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)

Example

#Legal variable names:	#Illegal variable names:
myvar = "John"	2myvar = "John"
my_var = "John"	my-var = "John"
_my_var = "John"	my var = "John"
myVar = "John"	
MYVAR = "John"	
myvar2 = "John"	

- **Introduction of Anaconda:**
- Anaconda is a data science platform for data scientists, IT professionals, and business leaders. It is a distribution of Python, R, etc. With more than 300 packages for data science, it quickly became one of the best platforms for any project
- <https://docs.anaconda.com/anaconda/install/windows/>

File Handling

- Python also support file handling like other languages which include different file operations like opening a file, reading, writing and closing the file but before going into depth first we need to understand why we do file handling what is the need of file handling.

File Operations

Opening a File

- python has inbuilt function to open a file. This function returns a file object, also called handle. we call it handle as it is used to read or modify a file accordingly.

f=open('file_name.txt')

- **Read and Write (perform operations)**
- Python File Mode
- 'r' open file for read (default)
- 'w' write , it create a new file if it does not exist or truncate the file if it exists.'
- 'x' open a file for exclusive creation but if file already exists then the operation will fail.
- 'a' open for appending at the end of file without truncating it and if no file exist already then it create a new one.
- 't' open file in text mode(default)'b' open file in binary mode. we can use binary mode for storing matrix or list of data.
- '+' open a file for updating (reading and writing)

Close file

- closing a file is done by mostly using close() method. python has a garbage collector to clean up an referenced object but, we must not rely on it to close the file.

```
f=open('file_name.txt')
```

```
f.close()
```

Python - sys Module

- The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. You will learn some of the important features of this module here.

sys.argv

- sys.argv** returns a list of command line arguments passed to a Python script. The item at index 0 in this list is always the name of the script. The rest of the arguments are stored at the subsequent indices.

```
import sys
print("You entered: ",sys.argv[1], sys.argv[2], sys.argv[3])
```

sys.exit

- This causes the script to exit back to either the Python console or the command prompt. This is generally used to safely exit from the program in case of generation of an exception.

sys.maxsize

- Returns the largest integer a variable can take.

```
import sys
sys.maxsize
```

sys.path

- This is an environment variable that is a search path for all Python modules.

```
import sys
sys.path
```

sys.version

- This attribute displays a string containing the version number of the current Python interpreter.

```
import sys
sys.version
```

Python - OS Module

- ✓ It is possible to automatically perform many operating system tasks. The OS module in Python provides functions for creating and removing a directory (folder), fetching its contents, changing and identifying the current directory, etc.
- ✓ You first need to import the os module to interact with the underlying operating system. So, import it using the import os statement before using its functions.

Getting Current Working Directory

The getcwd() function confirms returns the current working directory.

Example: Get Current Working Directory

```
import os  
os.getcwd()
```

Creating a Directory

We can create a new directory using the `os.mkdir()` function, as shown below.

Example: Create a Physical Directory

```
import os
os.mkdir("C:\MyPythonProject")
```

Changing the Current Working Directory

We must first change the current working directory to a newly created one before doing any operations in it. This is done using the `chdir()` function. The following change current working directory to `C:\MyPythonProject`.

Example: Change Working Directory

```
import os

os.chdir("C:\MyPythonProject") # changing current working directory

os.getcwd()
```


Removing a Directory

The `rmdir()` function in the `OS` module removes the specified directory either with an absolute or relative path. Note that, for a directory to be removed, it should be empty.

Example: Remove Directory

```
import os  
os.rmdir("C:\\MyPythonProject")
```

List Files

The `listdir()` function returns the list of all files and directories in the specified directory.

Example: List Directories

```
import os  
os.listdir("c:\\python37")
```

Geolocation Acquisition in Python?

we will discuss on how to get Geolocation when you enter any location name and its gives all the useful information such as postal code, city, state, country etc. with the latitudes and the longitudes (the specific coordinates) and vice-versa in which we provide the coordinates to get the location name.

This can be done using the GeoPy library in python. This library isn't built into python and hence needs to be installed explicitly.

Method 1: Getting coordinates from location name

With provided location, it is possible using geopy to extract the coordinates meaning its latitude and longitude. Therefore, it can be used to express the location in terms of coordinates.

Approach

Import module

Import Nominatim from geopy- Nominatim is a free service or tool or can be called an API with no keys that provide you with the data after providing it with name and address and vice versa.

On calling the Nomination tool which accepts an argument of **user_agent** you can give any name as it considers it to be the name of the app to which it is providing its services.

The **geocode()** function accepts the location name and returns a geodataframe that has all the details and since it's a dataframe we can get the address, latitude and longitude by simply calling it with the given syntax

Syntax:

variablename.address

variablename.latitude

variablename.longitude.

```
# importing geopy library
from geopy.geocoders import Nominatim

# calling the Nominatim tool
loc = Nominatim(user_agent="GetLoc")

# entering the location name
getLoc = loc.geocode("Gosainganj Lucknow")

# printing address
print(getLoc.address)

# printing latitude and longitude
print("Latitude = ", getLoc.latitude, "\n")
print("Longitude = ", getLoc.longitude)
```

Method 2: Getting location name from latitude and longitude

In this method all the things are same as the above, the only difference is instead of using the geocode function we will now use the **reverse()** method which accepts the coordinates (latitude and longitude) as the argument, this method gives the address after providing it with the coordinates.

Syntax:

`reverse(latitude,longitude)`

Approach

Import module

Call nominatim tool

Pass latitude and longitude to reverse()

Print location name

```
# importing modules
from geopy.geocoders import Nominatim

# calling the nominatim tool
geoLoc = Nominatim(user_agent="GetLoc")

# passing the coordinates
locname = geoLoc.reverse("26.7674446, 81.109758")

# printing the address/location name
print(locname.address)
```

What Are Whitelists and Blacklists?

As an example of how whitelists and blacklists work, let's consider a system administrator working for Google. Google is [*the most popular website globally*](#) — nearly everyone on the internet uses Google — but with its popularity comes nefarious users who may try to attack it, bring down its servers, or compromise user data. **A system administrator will need to *blacklist* IP addresses acting nefariously while allowing all other valid incoming traffic.**

Now, let's suppose that this same system administrator needs to configure a development server for Google's own internal use and testing. **This system admin will need to block all incoming IP addresses *except* for the *whitelisted* IP addresses of Google's developers.**

The concept of whitelisting and blacklisting characters for OCR purposes is the same. **A whitelist specifies a list of characters that the OCR engine is *only* allowed to recognize** — if a character is not on the whitelist, it *cannot* be included in the output OCR results.

The opposite of a whitelist is a blacklist. **A blacklist specifies the characters that, *under no circumstances*, can be included in the output.**

Packet Analysis:

With Python code, you can iterate over the packets in a pcap, extract relevant data, and process that data in ways that make sense to you.

You can use code to go over the pcap and locate a specific sequence of packets (i.e. locate the needle in the haystack) for later analysis in a GUI tool like Wireshark.

Or you can create customized graphical plots that can help you visualize the packet information. Further, since this is all code, you can do this repeatedly with multiple pcaps

Note:

<https://vnetman.github.io/pcap/python/pyshark/scapy/libpcap/2018/10/25/analyzing-packet-captures-with-python-part-1.html>

Packet Reassembly:

Network protocols often need to transport large chunks of data which are complete in themselves, e.g. when transferring a file. The underlying protocol might not be able to handle that chunk size (e.g. limitation of the network packet size), or is stream-based like TCP, which doesn't know data chunks at all.

In that case the network protocol has to handle the chunk boundaries itself and (if required) spread the data over multiple packets. It obviously also needs a mechanism to determine the chunk boundaries on the receiving side.

Wireshark calls this mechanism reassembly, although a specific protocol specification might use a different term for this (e.g. desegmentation, defragmentation, etc).

Payload Extraction:

Forensic Python

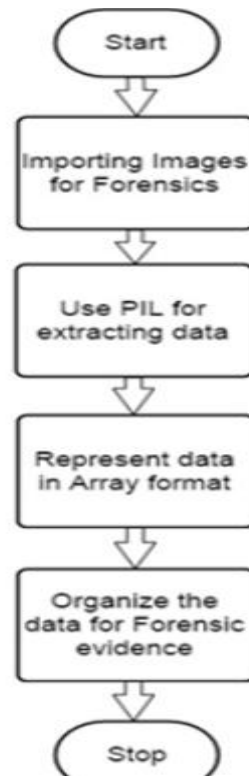
- ✓ Extracting valuable information from the resources available is a vital part of digital forensics.
- ✓ Getting access to all the information available is essential for an investigation process as it helps in retrieving appropriate evidence.
- ✓ Resources that contain data can be either simple data structures such as databases or complex data structures such as a JPEG image.
- ✓ Simple data structures can be easily accessed using simple desktop tools, while extracting information from complex data structures require sophisticated programming tools.

Python Imaging Library

The Python Imaging Library (PIL) adds image processing capabilities to your Python interpreter. This library supports many file formats, and provides powerful image processing and graphics capabilities.

Pillow is a fork of the Python Imaging Library (PIL). PIL is a library that offers several standard procedures for manipulating images. It's a powerful library but hasn't been updated since 2009 and doesn't support Python 3.

Pillow builds on this, adding more features and support for Python 3. It supports a range of image file formats such as PNG, JPEG, PPM, GIF, TIFF, and BMP. We'll see how to perform various operations on images such as cropping, resizing, adding text to images, rotating, greyscaling, etc., using this library.



Example

Now, let's have a programming example to understand how it actually works.

Step 1 – Suppose we have the following image from where we need to extract information

Step 2 – When we open this image using PIL, it will first note the necessary points required for extracting evidence, which includes various pixel values. Here is the code to open the image and record its pixel values

Step 3 – Our code will produce the following output, after extracting the pixel values of the image.

The output delivered represents the pixel values of RGB combination, which gives a better picture of what data is needed for evidence. The data fetched is represented in the form of an array.

The Image Object

```
from PIL import Image
image = Image.open('demo_image.jpg')
image.show()
print(im.format)
print(im.mode)
print(im.size)
im.save('new_image.png')
new_image = im.resize((400, 400))
new_image.save('image_400.jpg')
print(im.size)
print(new_image.size)
```

Python MySQL

Python can be used in database applications.
One of the most popular databases is MySQL.

Why Python with SQL?

For Data Analysts and Data Scientists, Python has many advantages. A huge range of open-source libraries make it an incredibly useful tool for any Data Analyst

Create Connection

Start by creating a connection to the database.
Use the username and password from your MySQL database:

```
import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword"
)

print(mydb)
```

Creating a Database

To create a database in MySQL, use the "CREATE DATABASE" statement:

Example

create a database named "mydatabase":

```
import mysql.connector
```

```
mydb = mysql.connector.connect(  
    host="localhost",  
    user="yourusername",  
    password="yourpassword"  
)
```

```
mycursor = mydb.cursor()
```

```
mycursor.execute("CREATE DATABASE mydatabase")
```

Insert Into Table

To fill a table in MySQL, use the "INSERT INTO" statement.

Example

Insert a record in the "customers" table:

```
import mysql.connector
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)
mycursor = mydb.cursor()

sql = "INSERT INTO customers (name, address)
VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)
mydb.commit()
print(mycursor.rowcount, "record inserted.")
```

Insert Into Table

To fill a table in MySQL, use the "INSERT INTO" statement.

Example

Insert a record in the "customers" table:

```
import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)

mycursor = mydb.cursor()

sql = "INSERT INTO customers (name,
address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)

mydb.commit()

print(mycursor.rowcount, "record inserted.")
```


Select With a Filter

When selecting records from a table, you can filter the selection by using the "WHERE" statement:

Example

Select record(s) where the address is "Park Lane 38": result:

```
import mysql.connector
```

```
mydb = mysql.connector.connect(  
    host="localhost",  
    user="yourusername",  
    password="yourpassword",  
    database="mydatabase"  
)
```

```
mycursor = mydb.cursor()
```

```
sql = "SELECT * FROM customers WHERE  
address ='Park Lane 38'"
```

```
mycursor.execute(sql)
```

```
myresult = mycursor.fetchall()
```

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

Delete Record

You can delete records from an existing table by using the "DELETE FROM" statement:

Example

Delete any record where the address is "Mountain 21":

```
import mysql.connector
```

```
mydb = mysql.connector.connect(  
    host="localhost",  
    user="yourusername",  
    password="yourpassword",  
    database="mydatabase"  
)
```

```
mycursor = mydb.cursor()
```

```
sql = "DELETE FROM customers WHERE  
address = 'Mountain 21'"
```

```
mycursor.execute(sql)
```

```
mydb.commit()
```

```
print(mycursor.rowcount, "record(s) deleted")
```

HTTP Communications with Python Built in Libraries

The http or Hyper Text Transfer Protocol works on client server model. Usually the web browser is the client and the computer hosting the website is the server. IN python we use the requests module for creating the http requests.

It is a very powerful module which can handle many aspects of http communication beyond the simple request and response data. It can handle authentication, compression/decompression, chunked requests etc.

```
# api testing using python
# request module to work with api's
import requests
# json module to parse and get json
import json
# This function are return the
# json response from given url
def getReq(url):
    # handle the exceptions
    try:
        # make a get request using requests
        # module and store the result.
        res = requests.get(url)

        # return the result after
        # formatting in json.
        return json.dumps(res.json(), indent=4)
    except Exception as ee:
        return f"Message : {ee}"

# This function are return the json response of url and json
# data you send the server
def postReq(url, data):
    # handle the exceptions
    try:
        # make a post request with
        # the json data
        res = requests.post(url, json=data)

        # return the response after
        # formatting in json.
        return json.dumps(res.json(), indent=4)
    except Exception as er:
        return f"Message : {er}"

# Driver Code
if __name__ == '__main__':

    # always run loop to make
    # menu driven program
    while True:

        # handle the exceptions
        try:
            choice = int(input("1.Get Request\n2.Post
            Request\n3.Exit\nEnter Choice : "))

            # get user choice and perform tasks.
            if choice == 1:

                # take a url as a input.
                url_inp = input("Enter a valid get url : ")

                # print the result of the url.
                print(getReq(url_inp))

            elif choice == 2:

                # take a url as a input
                url_inp = input("Enter a valid get url : ")

                # take a formal data as input in dictionary.
                data_inp = {
                    "name": input("Name : "),
                    "email": input("Email : "),
                    "work": input("Work : "),
                    "age": input("Age : ")
                }

                # print the result.
                print(postReq(url_inp, data_inp))

            elif choice == 3:

                # if user want to exit.
                exit(0)

        except Exception as e:
            print("Error : ", e)
```

1. Grequests

GRquests allows you to use Requests with Gevent to make asynchronous HTTP Requests easily.

Command: `pip install grequests`

```
import grequests
urls = [
    'http://www.heroku.com',
    'http://python-tablib.org',
    'http://httpbin.org',
    'http://python-requests.org',
    'https://yeahhub.com/'
]
```

2. urllib2 – Comprehensive HTTP client library.

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.

Python urllib2 module provides methods for accessing Web resources via HTTP. It supports many features, such as HTTP and HTTPS, authentication, caching, redirects, and compression. urllib2 is a comprehensive HTTP client library, urllib2.py supports many features left out of other HTTP libraries.

HTTPS support is only available if the socket module was compiled with SSL support.

Command: pip install urllib2

```
#!/usr/bin/python3
import urllib2
http = urllib2.Http()
content = http.request("https://www.yeahhub.com")[1]
print(content.decode())
```

3. Requests – HTTP Requests for Humans.

Requests allows you to send organic, grass-fed HTTP/1.1 requests, without the need for manual labor. There's no need to manually add query strings to your URLs, or to form-encode your POST data. Keep-alive and HTTP connection pooling are 100% automatic.

Command: `pip install requests`

```
#!/usr/bin/python3
```

```
import requests
```

```
#GET Request
```

```
r = requests.get('https://api.github.com/events')
```

```
#POST Request
```

```
r = requests.post('http://httpbin.org/post', data = {'key':'value'})
```

4. Treq – Python requests like API built on top of Twisted's HTTP client

Treq is an HTTP library inspired by requests but written on top of Twisted's Agents. It provides a simple, higher level API for making HTTP requests when using Twisted.

Command: `pip install treq`

```
#!/usr/bin/env python
from twisted.internet.task import react
from _utils import print_response
import treq
def main(reactor, *args):
    d = treq.get('http://httpbin.org/get')
    d.addCallback(print_response)
    return d
react(main, [])
```

5. Urllib3 – A HTTP library with thread-safe connection pooling, file post support, sanity friendly.

Urllib3 is a powerful, sanity-friendly HTTP client for Python. Much of the Python ecosystem already uses urllib3 and you should too. urllib3 brings many critical features that are missing from the Python standard libraries:

- ✓ Thread safety.
- ✓ Connection pooling.
- ✓ Client-side SSL/TLS verification.
- ✓ File uploads with multipart encoding.
- ✓ Helpers for retrying requests and dealing with HTTP redirects.
- ✓ Support for gzip and deflate encoding.
- ✓ Proxy support for HTTP and SOCKS.
- ✓ 100% test coverage.

Command: `pip install urllib3`

Yeahhub.com

```
from requests.packages import urllib3
```

```
http = urllib3.PoolManager()
```

```
r = http.request('GET', 'https://yeahhub.com')
```

```
print "r.status: ", r.status
```

```
print "r.data", r.data
```

Web Communications with the Requests Module

What is the Request?

When you want to interact with data via a API, this is called a request. A request is made up of the following components:

Endpoint – The URL that delineates what data you are interacting with. Similar to how a web page URL is tied to a specific page, an endpoint URL is tied to a specific resource within an API.

Method– Specifies how you're interacting with the resource located at the provided endpoint. REST APIs can provide methods to enable full Create, Read, Update, and

Delete (CRUD) functionality. Here are common methods most REST APIs provide:

GET – Retrieve data

PUT – Replace data

POST – Create data

DELETE – Delete data

Unit-I

Scripting Part-1

Shell Overview

What is a Shell?

- UNIX shells provide a "command line" interface which allows the user to enter commands which are translated by the shell into something the kernel can comprehend and then is sent off to the kernel for it to act upon.
- The user can pick their shell (just like the applications, desktop manger, window manager, etc. on a UNIX system).

UMBC Shells

- On the UMBC GL network, the default UNIX shell is tcsh - Turbo C SHell. The default can be changed by the user via <http://accounts.umbc.edu/>.
- Shells available on GL include:
 - tcsh - Turbo C SHell
 - csh - C SHell
 - ksh - Korn SHell
 - bash - Bourne Again SHell
 - sh - SHell

Shell Overview

- **Linux Default Shell**

- Most Linux systems (especially home installations) default to the bash shell.

- **Changing Your Shell - On a Home Based System**

- Usually there is a command called **chsh** that stands for change shell. You have to enter your password and then the absolute path to the new shell that you wish to use.

```
(06:08 PM) : chsh
```

```
Changing shell for Eric.
```

```
Password: New shell [/bin/bash]: /bin/csh
```

```
Shell changed.
```

Understanding the File System Tree and its Navigation

A file system is a logical collection of files on a partition or disk. A partition is a container for information and can span an entire hard drive if desired.

Your hard drive can have various partitions which usually contain only one file system, such as one file system housing the **/file system** or another containing the **/home file system**.

One file system per partition allows for the logical maintenance and management of differing file systems.

Everything in Unix is considered to be a file, including physical devices such as DVD-ROMs, USB devices, and floppy drives.

I/O Redirection

The work of any command is either taking input or gives an output or both. So, Linux has some command or special character to redirect these input and output functionalities. For example: suppose we want to run a command called “date” if we run it will print the output to the current terminal screen. But our requirement is different, we don’t want the output to be displayed on the terminal. We want the output to be saved in a file. This could be done very easily with output redirection. Redirection here simply means diverting the output or input.

Similarly, if we have a command that needs input to be performed. Let take a command “head” this needs input to give output. So either we write input in the form of command directly or redirect input from any other place or file. Suppose we have a file called “file.txt” to print the starting some lines of the file we could use the “head”. So let’s see how this all is done on the terminal.

Types of Redirection

1. Overwrite

“>” standard output

“<” standard input

2. Appends

“>>” standard output

“<<” standard input

3. Merge

“p >& q” Merges output from stream p with stream q

“p <& q” Merges input from stream p with stream q

cat > file.txt

cat < file.txt

cat

cat >> file.txt

Expansion

If you are using a Linux system, you might already know how crucial a shell interface is for interacting with your system. On most Linux distributions, Bash is the default shell that we use to run commands and execute scripts. A shell script is a set of commands that, when executed, is used to perform some useful function(s) on Linux. This .sh file, written by a user, contains all the commands used to perform a task so that we do not have to run those commands manually, one by one.

In this tutorial, we will explain two of the most useful bash expansions used in shell scripts:

`$()` – the command substitution: *Command substitution allows the output of a command to replace the command itself*

`${}` – the parameter substitution/variable expansion: A parameter, in Bash, is an entity that is used to store values. A parameter can be referenced by a number, a name, or by a special symbol.

An expansion in Shell is performed on the script after it has been split into tokens. A token is a sequence of characters considered a single unit by the shell. It can either be a word or an operator.

```
#!/bin/sh
echo ***Status Report***
TODAY=$(date)
echo "Today is $TODAY"
USERS=$(who | wc -l)
echo "$USERS users are currently logged in"
UPTIME=$(date ; uptime)
echo "The uptime is $UPTIME"
```

```
${parameter}
name="john doe"
```

```
echo "${name}"
```

```
echo "The name of the person
is $name_"
```

```
echo "The name of the person
is ${name}_"
```

Permissions

The Unix-like operating systems, such as Linux differ from other computing systems in that they are not only *multitasking* but also *multi-user*.

What exactly does this mean? It means that more than one user can be operating the computer at the same time. While a desktop or laptop computer only has one keyboard and monitor, it can still be used by more than one user. For example, if the computer is attached to a network, or the Internet, remote users can log in via ssh (secure shell) and operate the computer. In fact, remote users can execute graphical applications and have the output displayed on a remote computer. The X Window system supports this.

This lesson will cover the following commands:

chmod - modify file access rights

su- temporarily become the superuser

sudo- temporarily become the superuser

chown- change file ownership

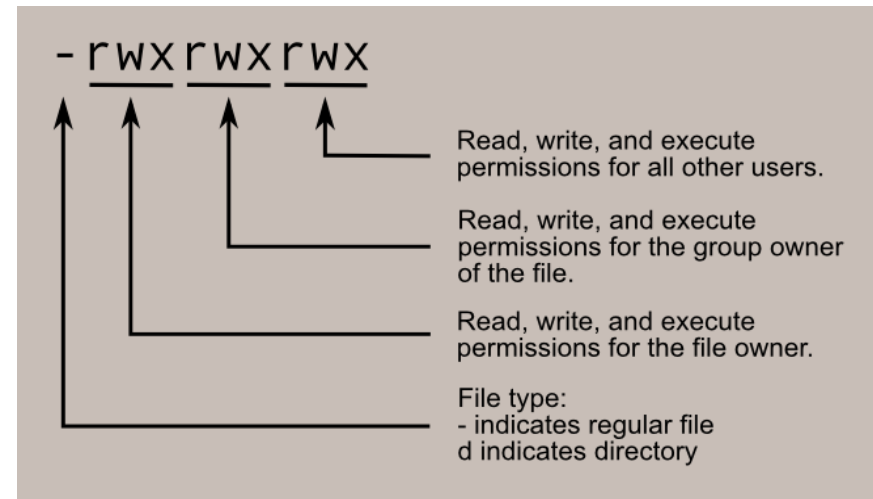
chgrp- change a file's group ownership

To check the file permission, give the following command:

\$ ll file_name

The following command will add the read/write and execute permissions to the file wherein, u is for user, g is for group, and o is for others:

\$ chmod ugo+rw file_name



Job Control

Job control refers to the ability to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point. A user typically employs this facility via an interactive interface supplied jointly by the operating system kernel's terminal driver and Bash.

The shell associates a *job* with each pipeline. It keeps a table of currently executing jobs, which may be listed with the `jobs` command.

A process in Linux means any running program. For example, when we run *ls*, it's run as a process. If we search for a filename, by using *ls -l* and piping the result to *grep*, we're actually running two processes:

```
$ ls -l
```

We list jobs using the *jobs* command.

we can kill a job we don't need by using the *kill* command

Configuration and Environment variable

The configure shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a Makefile in each directory of the package (the top directory, the builtins, doc, and support directories, each directory under lib, and several others).

What is an environment variable?

Environment variables or **ENVs** basically define the behavior of the environment. They can affect the processes ongoing or the programs that are executed in the environment.

Scope of an environment variable

Scope of any variable is the region from which it can be accessed or over which it is defined. An environment variable in Linux can have **global** or **local** scope.

Global

A globally scoped ENV that is defined in a terminal can be accessed from anywhere in that particular environment which exists in the terminal. That means it can be used in all kind of scripts, programs or processes running in the environment bound by that terminal.

Local

A locally scoped ENV that is defined in a terminal cannot be accessed by any program or process running in the terminal. It can only be accessed by the terminal(in which it was defined) itself.

How to access ENVs?

SYNTAX:

\$NAME

\$ echo \$NAME

To set a global ENV

\$ export NAME=Value or \$ set

NAME=Value

\$ echo \$NAME

To set a local ENV

SYNTAX:

\$ NAME=Value