PYTHON PROGRAMMING [18CS601]

LECTURE NOTES

B.TECH III YEAR – VI SEMESTER

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12 (B) of UGC ACT 1956
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A+' Certified)

SYLLABUS

PYTHON PROGRAMMING

B. Tech VI Semester : Computer Science & Engineering								
Course code	Category	Hours/week			Credits	Maximum Marks		
18CS601	Core	L	T	P	C	CIA	SEE	TOTAL
		3	0	0	3	30	70	100
Contact Classes:60	Tutorial Classes: 0		Practical Classes: 0			Total Classes:60		

OBJECTIVES:

The course should enable the students to:

1. The course is designed to provide Basic knowledge of Python.
2. Python programming is intended for software engineers, system analysts, program managers.
3. User support personnel who wish to learn the Python programming language.

UNIT-I INTRODUCTION TO PYTHON & EXPRESSIONS Classes:12

Introduction to Python: Why to Learn Python, Difference between C and Python, Python Environment variables, Procedure for execution of Python program Values, Data Types and Expressions: Python Identifiers, Reserved words, Lines and Indentation, Command Line arguments, Data types: Python Numbers, Python Strings, Python Lists, Python Dictionary, Data type conversion. Variables, Statement in Python, Data type convertions

UNIT-II FUNCTIONS AND MODULES Classes:12

Functions and Modules: Introduction to functions, Types of Functions, Flow of Execution, Parameters and arguments, Pass by reference and pass by value, Function arguments, Return Statement, Composition, Recursion, Python Modules.

UNIT-III OPERATORS , CONTROL FLOW AND STRINGS Classes:12

Operators and Control Flow: Operators, Conditional Statements, Iteration, Loop Control statements. Strings: Strings, String Slices, Immutability, String operations, String Methods, String Modules.

UNIT-IV LIST AND TUPLES Classes: 12

List and Tuples: Introduction, List operations, List Slices, List Methods, Loops, Mutability, Aliasing, Cloning Lists, Parameters, Tuples, Tuple assignment, Tuple as return value.

UNIT-V DICTIONARIES AND FILES Classes:12

Dictionaries and Files: Dictionaries, Operations on Dictionaries, Dictionary methods, Difference between List, Tuples and Dictionaries, Introduction to Files, File Types, Exception-Error handling.

Text Book:

1. Think Python,by Allen B. Downey ,second edition ,O'Reilly,Sebastopol,Califomia.

References

1. Michael H.Goldwasser, David Letscher, -Object Oriented Programming in Pythonll, Prentice Hall, 1 st Edition, 2007.

Web References:

1. Online Version www.greenteapress.com/thinkpython2.pdf.
2. How to think like a computer Scientist, by Brad Miller and David Ranum. Online Version www.interactivepython.org/runstone/static/thinkscpy/index.html.
3. https://realpython.com/python3-o bject-oriented-programming/
4. https://python. swaroopch.com/ oop.html
5. https://python-textbok.readthedocs.io/en/1.0/0bject_ Oriented_Programming.html
6. https://www.programiz.com/python-programming/

Outcomes:

At the end of the course students able to

1. Implement a given algorithm as a computer program in Python.
2. Adapt and combine standard algorithms to solve a given problem
3. Adequately use standard programming constructs: repetition, selection, functions, Composition, modules, aggregated data (arrays, lists, etc.). Explain what a given program in Python does. Identify and repair coding errors in a program.
4. Use library software for (e.g.) building a graphical user interface, web application, or mathematical software.

LESSON PLAN

	Loops, Mutability,	
	Aliasing,	
	Cloning Lists,	
	Parameters,	
	Tuples,	
	Tuple assignment,	
	Tuple as return value.	
V	Dictionaries and Files	
	Dictionaries,	
	Operations on Dictionaries,	
	Dictionary methods,	
	Difference between List,	
	Tuples and Dictionaries,	
	Introduction to Files,Error handling.	
	File Types,	
	Exception-Error handling	

UNIT-I

INTRODUCTION TO PYTHON & EXPRESSIONS

Introduction to Python: Why to Learn Python, Difference between C and Python, Python Environment variables, Procedure for execution of Python program Values, Data Types and Expressions: Python Identifiers, Reserved words, Lines and Indentation, Command Line arguments, Data types: Python Numbers, Python Strings, Python Lists, Python Dictionary, Data type conversion. Variables, Statement in Python, Data type conversions

INTRODUCTION TO PYTHON

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991. It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.

- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

FEATURES OF PYTHON

- **Easy to Learn and Use:** Python is easy to learn and use. It is developer-friendly and high level programming language.

- **Expressive Language:** Python language is more expressive means that it is more understandable and readable.

- **Interpreted Language:** Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.

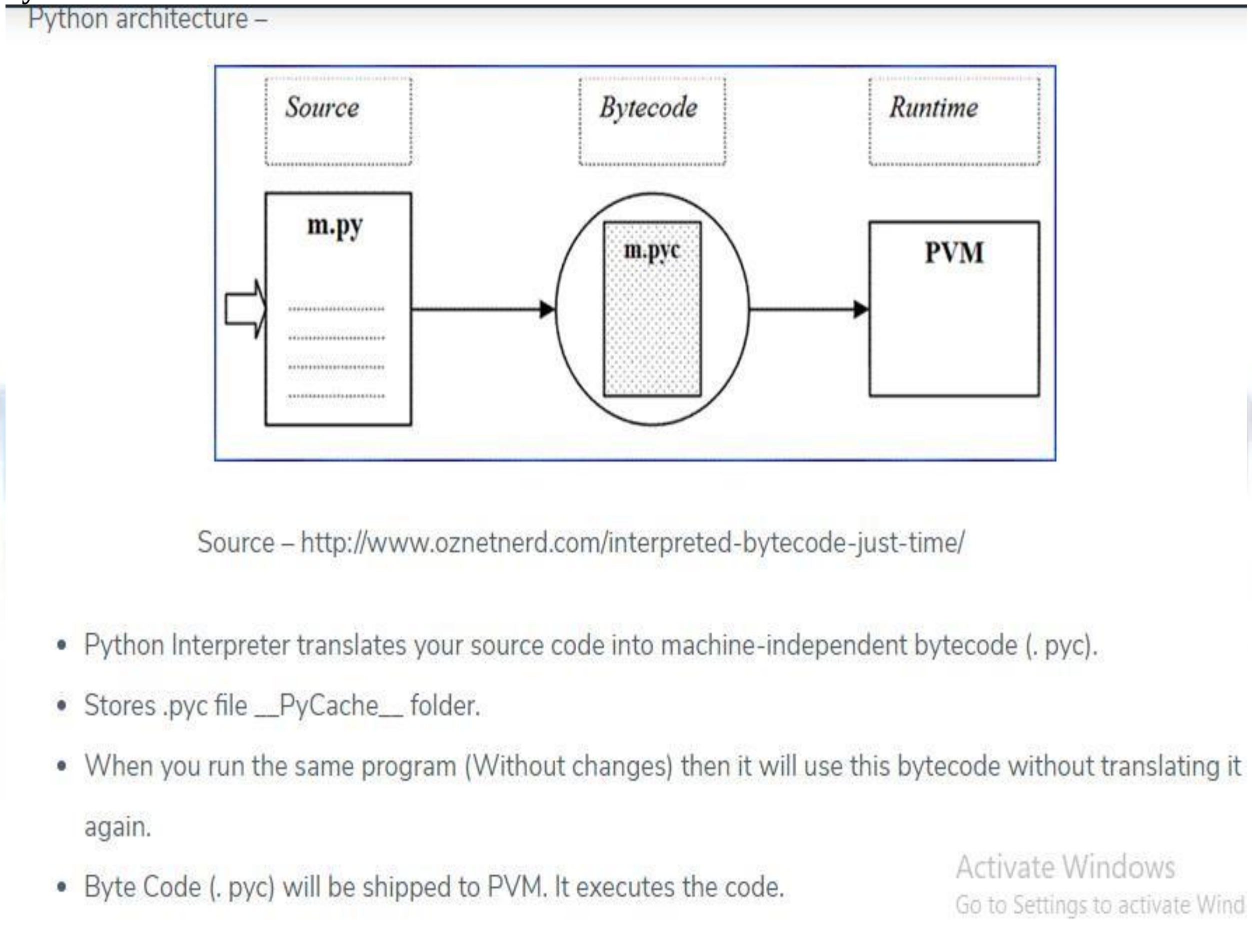
- **Cross-platform Language:** Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc. So, we can say that Python is a portable language.

- **Free and Open Source:** Python language is freely available at official web address. The source-code is also available. Therefore it is open source.

- **Object-Oriented Language:** Python supports object oriented language and concepts of classes and objects come into existence.
- **Extensible: It** implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our python code.
- **Large Standard Library:** Python has a large and broad library and provides rich set of module and functions for rapid application development.
- **GUI Programming Support:** Graphical user interfaces can be developed using Python.

- **Integrated:** It can be easily integrated with languages like C, C++, JAVA etc.
- **Platform independent:** Write once and run any where
- **Dynamically typed Language:** We cannot required to specify the data type explicitly. Dynamically typed language provides more flexibility to the programmer. We are not fix the type of the variable.

Python Architecture:



Python architecture –

Source – http://www.oznetnerd.com/interpreted-bytecode-just-time/

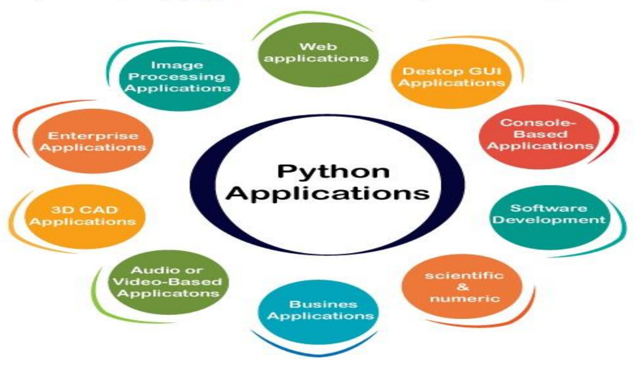
- Python Interpreter translates your source code into machine-independent bytecode (. pyc).
- Stores .pyc file __PyCache__ folder.
- When you run the same program (Without changes) then it will use this bytecode without translating it again.
- Byte Code (. pyc) will be shipped to PVM. It executes the code.

Activate Windows
Go to Settings to activate Wind

Applications of python

Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application.

Here, we are specifying application areas where Python can be applied.



1) Web Applications

➢ We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feedparser, etc.

➢ One of Python web-framework named Django is used on **Instagram**. Python provides many useful frameworks, and these are given below:

Django and Pyramid framework(Use for heavy applications)
Flask and Bottle (Micro-framework)
Plone and Django CMS (Advance Content management)

2) Desktop GUI Applications

➢ The GUI stands for the Graphical User Interface, which provides a smooth interaction to any application. Python provides a **Tk GUI library** to develop a user interface. Some popular GUI libraries are given below.

Tkinter or Tk
wxWidgetM
Kivy (used for writing multitouch applications)
PyQt or Pyside

3) Console-based Application

➢ Console-based applications run from the command-line or shell. These applications are computer program which are used commands to execute.

➢ This kind of application was more popular in the old generation of computers.

➢ Python can develop this kind of application very effectively. It is famous for having REPL, which means **the Read-Eval-Print Loop** that makes it the most suitable language for the command-line applications.

➢ Python provides many free library or module which helps to build the command-line apps.

➢ The necessary **IO** libraries are used to read and write.

➢ It helps to parse argument and create console help text out-of-the-box. There are also advance libraries that can develop independent console apps.

4) Software Development

➢ Python is useful for the software development process. It works as a support language and can be used to build control and management, testing, etc.

➢ **SCons** is used to build control.

➢ **Buildbot** and **Apache** Gumps are used for automated continuous compilation and testing.

➢ **Round** or **Trac** for bug tracking and project management

5) Scientific and Numeric

➢ This is the era of Artificial intelligence where the machine can perform the task the same as the human.

➢ Python language is the most suitable language for Artificial intelligence or machine learning.

➢ It consists of many scientific and mathematical libraries, which makes easy to solve complex calculations.

➢ Implementing machine learning algorithms require complex mathematical calculation.

➢ Python has many libraries for scientific and numeric such as Numpy, Pandas, Scipy, Scikit-learn, etc.

➢ If you have some basic knowledge of Python, you need to import libraries on the top of the code. Few popular frameworks of machine libraries are given below.

SciPy
Scikit-learn
NumPy
Pandas
Matplotlib

6) Business Applications

➢ Business Applications differ from standard applications. E-commerce and ERP are an example of a business application. This kind of application requires extensively, scalability and readability, and Python provides all these features.

- Oddo is an example of the all-in-one Python-based application which offers a range of business applications. Python provides a Tryton platform which is used to develop the business application.

7) Audio or Video-based Applications

- Python is flexible to perform multiple tasks and can be used to create multimedia applications. Some multimedia applications which are made by using Python are **TimPlayer, cplay,** etc. The few multimedia libraries are given below.

Gstreamer
Pyglet
QT Phonon

8) 3D CAD Applications

- The CAD (Computer-aided design) is used to design engineering related architecture. It is used to develop the 3D representation of a part of a system. Python can create a 3D CAD application by using the following functionalities.

Fandango (Popular)
CAMVOX
HeeksCNC
AnyCAD
RCAM

9) Enterprise Applications

- Python can be used to create applications that can be used within an Enterprise or an Organization. Some real-time applications are OpenERP, Tryton, Picalo, etc.

10) Image Processing Application

- Python contains many libraries that are used to work with the image. The image can be manipulated according to our requirements. Some libraries of image processing are given below.

OpenCV
Pillow
SimpleITK

- In this topic, we have described all types of applications where Python plays an essential role in the development of these applications. In the next tutorial, we will learn more concepts about Python.

Limitations of Python

- Performance wise python is not up to the mark.
- Not that much frequently used for developing mobile applications.
- Python Syntax compared to other programming languages
- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

C - Language	**Python**
➢ An Imperative programming model is basically followed by C.	➢ An object-oriented programming model is basically followed by Python.
➢ Variables are declared in C.	➢ Python has no declaration.
➢ C doesn’t have native OOP.	➢ Python has OOP which is a part of language.
➢ Pointers are available in C language.	➢ No pointers functionality is available in Python.
➢ C is a compiled language.	➢ Python is an interpreted language.
➢ There is a limited number of built-in functions available in C.	➢ There is a large library of built-in functions in Python.
➢ Implementation of data structures requires its functions to be explicitly implemented.	➢ It is easy to implement data structures in Python with built-in insert, append functions.
➢ C is compiled direct to machine code which is executed directly by the CPU	➢ Python is firstly compiled to a byte-code and then it is interpreted by a large C program.
➢ Declaring of variable type in C is necessary condition.	➢ There is no need to declare a type of variable in Python.
➢ C does not have complex data structures.	➢ Python has some complex data structures.
➢ C is statically typed.	➢ Python is dynamically typed.
➢ Syntax of C is harder than python because of which programmers prefer to use python instead of C	➢ It is easy to learn, write and read Python programs than C.
➢ C programs are saved with .c extension.	➢ Python programs are saved by .py extension.
➢ An assignment is allowed in a line.	➢ Assignment gives an error in line. For example, a=5 gives an error in python.

➢ In C language testing and debugging is harder.	➢ In Python, testing and debugging is not harder than C.
➢ C is complex than Python.	➢ Python is much easier than C.
➢ The basic if statement in c is represented as: ➢ if ()	➢ The basic if statement in Python is represented as: ➢ if:

➢ The basic if-else statement in Python is represented as:
if ( )
else
	➢ The basic if-else statement is represented as:
	if :
	else:

➢ C language is fast.	➢ Python programming language is slow

Python Environment Variables

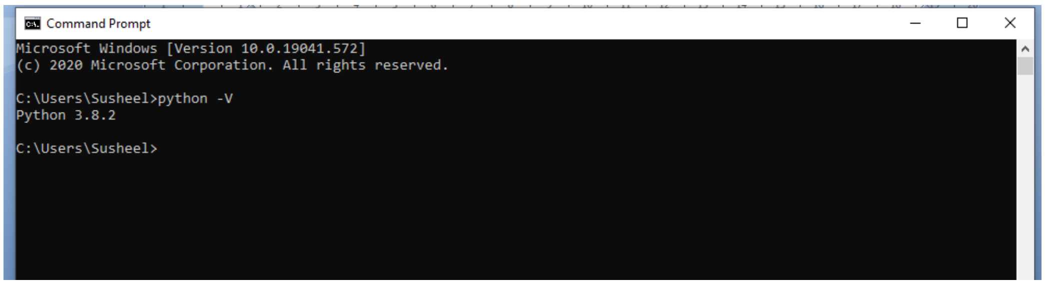
Here are important environment variables, which can be recognized by Python

Sr.No	Variable & Description
1	**PYTHONPATH** It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes preset by the Python installer.
2	**PYTHONSTARTUP** It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as .pythonrc.py in Unix and it contains commands that load utilities or modify PYTHONPATH.
3	**PYTHONCASEOK** It is used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it.
4	**PYTHONHOME** It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.

Running Python

➢ **Python** is a well known high-level programming language. The Python script is basically a file containing code written in Python. The file containing python script has the extension '.py' or can also have the extension '.pyw' if it is being run on a windows machine. To run a python script, we need a python interpreter that needs to be downloaded and installed.

➢ we first have to check whether a **python interpreter** is installed on the system or not. So in windows, open **'cmd' (Command Prompt)** and type the following command.



```
Microsoft Windows [Version 10.0.19041.572]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Susheel>python -V
Python 3.8.2

C:\Users\Susheel>
```

Here are the ways with which we can run a Python script.

- Interactive Mode
- Command Line
- IDE (PyCharm)

- **Interactive Mode/Interactive Interpreter**

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter **python** the command line.

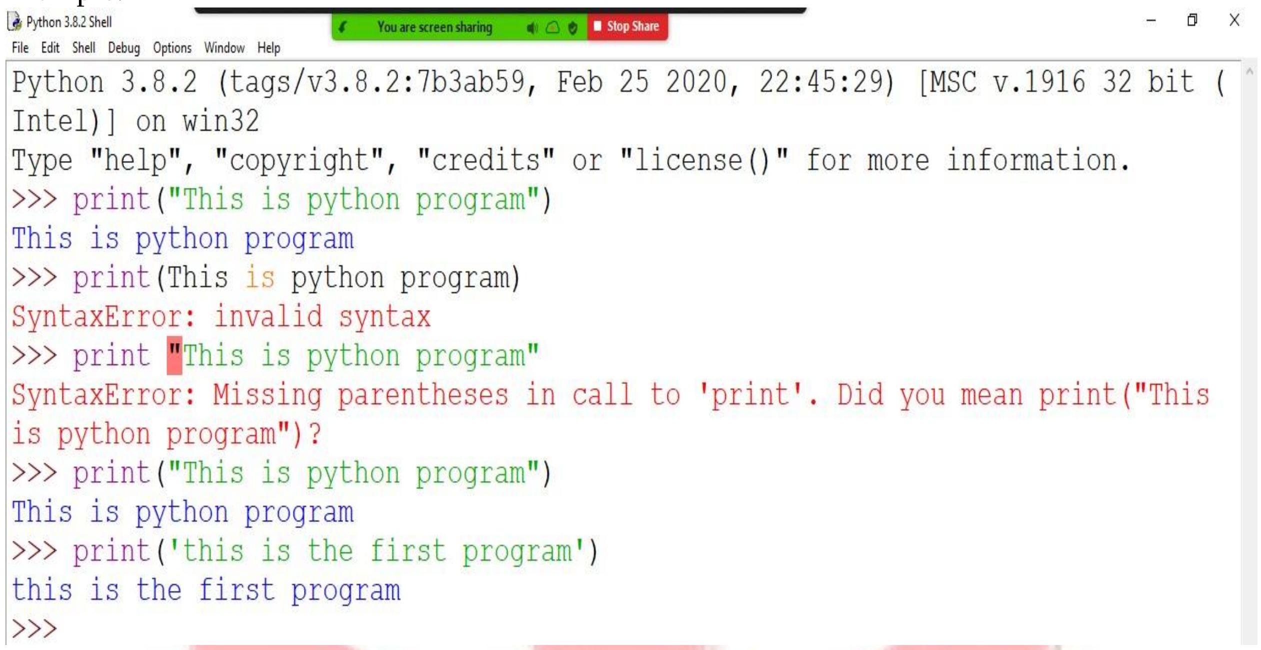
Start coding right away in the interactive interpreter.

```
$python # Unix/Linux
or
python% # Unix/Linux
or
C:> python # Windows/DOS
```

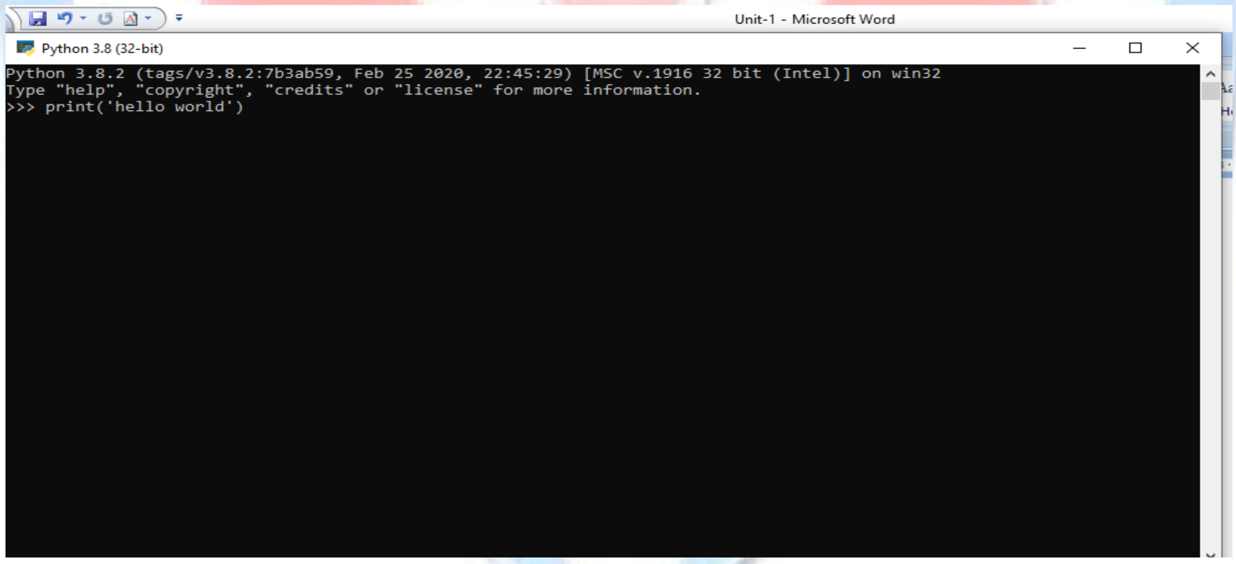
Here is the list of all the available command line options

Sr.No.	Option & Description
1	**-d** It provides debug output.
2	**-O** It generates optimized bytecode (resulting in .pyo files).
3	**-S** Do not run import site to look for Python paths on startup.
4	**-v** verbose output (detailed trace on import statements).
5	**-X** disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6.
6	**-c cmd** run Python script sent in as cmd string
7	**file** run Python script from given file

Example:



```
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (
Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("This is python program")
This is python program
>>> print(This is python program)
SyntaxError: invalid syntax
>>> print "This is python program"
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("This
is python program")?
>>> print("This is python program")
This is python program
>>> print('this is the first program')
this is the first program
>>>
```



```
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello world')
```

➢ Command Line

To run a Python script store in a '.py' file in command line, we have to write 'python' keyword before the file name in the command prompt.

$python script.py # Unix/Linux

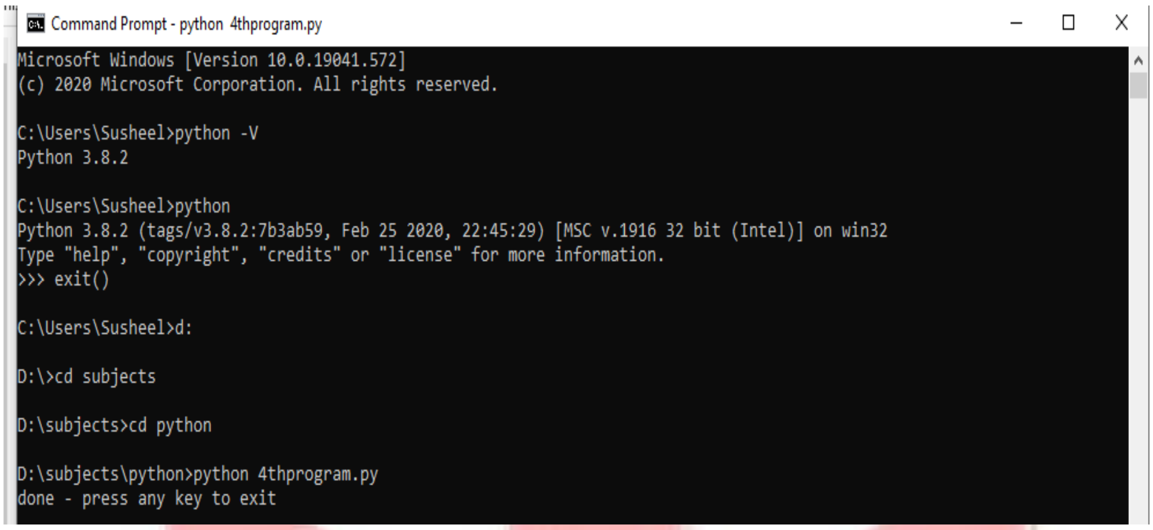
or

python% script.py # Unix/Linux

or

C: >python script.py # Windows/DOS

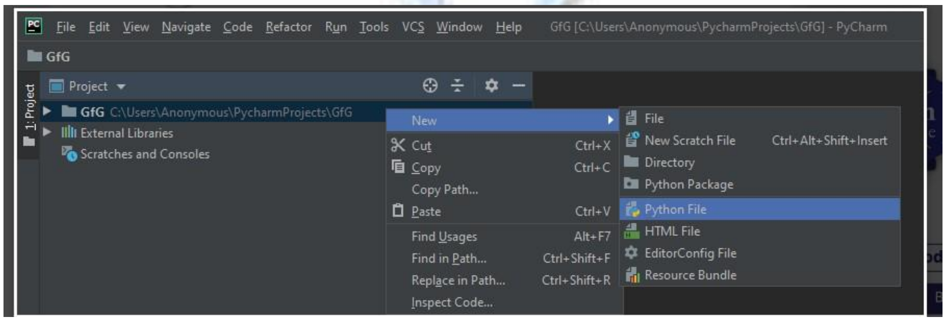
Example:

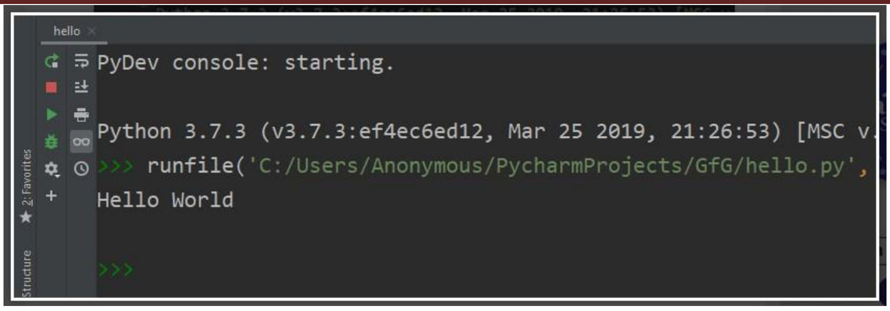


```
Command Prompt - python 4thprogram.py
Microsoft Windows [Version 10.0.19041.572]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Susheel>python -V
Python 3.8.2

C:\Users\Susheel>python
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

C:\Users\Susheel>d:

D:\>cd subjects

D:\subjects>cd python

D:\subjects\python>python 4thprogram.py
done - press any key to exit
```

- **IDE (PyCharm)**
- To run Python script on a **IDE (Integrated Development Environment) like PyCharm** you will have to do the following:
- Create a new project.
- Give a name to that project as 'GfG' and click on Create.
- Select the root directory with the project name we specified in the last step. **Right click** on it, go in **New** and click on '**Python file**' option. Then give the name of the file as '**hello**' (you can specify any name as per your project requirement). This will create a '**hello.py**' file in the project root directory.

  Note: You don't have to specify the extension as it will take it automatically.


File Edit View Navigate Code Refactor Run Tools VCS Window Help
GfG [C:\Users\Anonymous\PycharmProjects\GfG] - PyCharm
GfG
1: Project
Project
GfG C:\Users\Anonymous\PycharmProjects\GfG
External Libraries
Scratches and Consoles
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path...
Paste Ctrl+V
Find Usages Alt+F7
Find in Path... Ctrl+Shift+F
Replace in Path... Ctrl+Shift+R
Inspect Code...
File
New Scratch File Ctrl+Alt+Shift+Insert
Directory
Python Package
Python File
HTML File
EditorConfig File
Resource Bundle



```
PyDev console: starting.

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v
>>> runfile('C:/Users/Anonymous/PycharmProjects/GfG/hello.py',
Hello World

>>>
```

Types of Errors

- Programming errors are called bugs and the process of tracking them down is called debugging.
- Three kinds of errors can occur in a program: syntax errors,runtime errors, and semantic errors.

1) Syntax Errors

- Python can only execute a program if the syntax is correct; otherwise, the interpreter displays an error message. Syntax refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so (1+2) is 3 legal, (1+2)+ is syntax error

2) Runtime Errors:

- The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.

3) Semantic Errors

- The third type of error is the semantic error. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.
- The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

Values and Types

➢ A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1,2 and “Hello World”

➢ These values belong to different types: 2 is an integer, and ‘Hello world’ is a string, so-called because it contains a “string” of letters. We means the interpreter can identify strings because they are enclosed in quotation marks.

- >>> 1,00,000
- (1,0,0)

➢ Well, that’s not what we expected at all! Python interprets 1,00,000 as a comma separated sequence of integers. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn’t do the “right” thing.

Identifiers

➢ Identifiers are used for identification Purpose. An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

Rules for Identifiers

➢ Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore _. Names like myClass, var_1 and print_this_to_screen all are valid example.

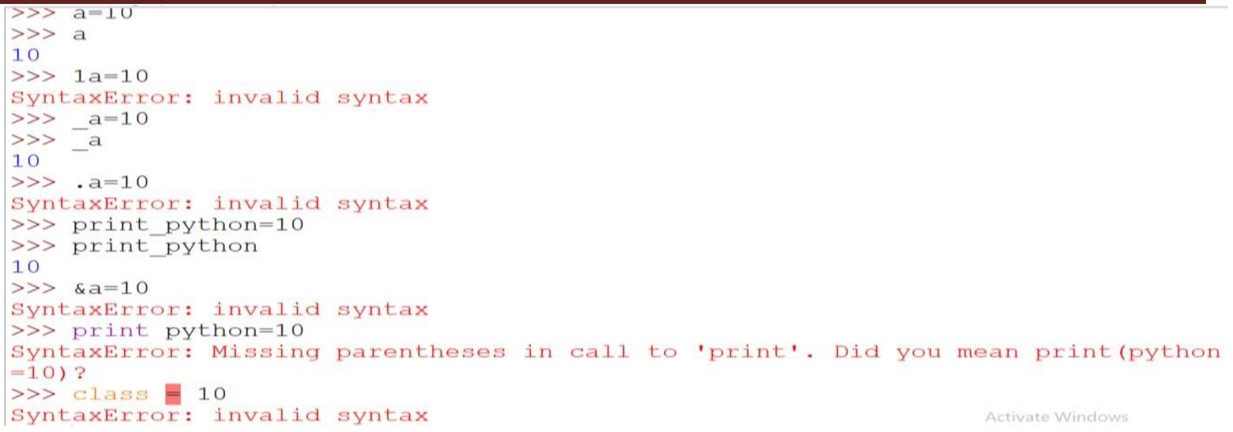
➢ An identifier cannot start with a digit. 1variable, 1a, n%4, n 9 are invalid, but variable1 is perfectly fine.

➢ Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, *).

➢ Keywords cannot be used as identifiers.

➢ Identifier names are case sensitive(differentiating between capital and lower-case letters) for example my name, and MyName is not the same.

➢ There is no length limit for python identifier but it is not recommended because redability of the program is going to be down.



```
>>> a=10
>>> a
10
>>> 1a=10
SyntaxError: invalid syntax
>>> _a=10
>>> _a
10
>>> .a=10
SyntaxError: invalid syntax
>>> print_python=10
>>> print_python
10
>>> &a=10
SyntaxError: invalid syntax
>>> print python=10
SyntaxError: Missing parentheses in call to 'print'. Did you mean print(python
=10)?
>>> class = 10
SyntaxError: invalid syntax
```

Keywords/Reserved Words

➢ Python Keywords are special reserved (fixed) words which convey a special meaning to the compiler/interpreter.

➢ Each keyword has a special meaning and a specific operation.

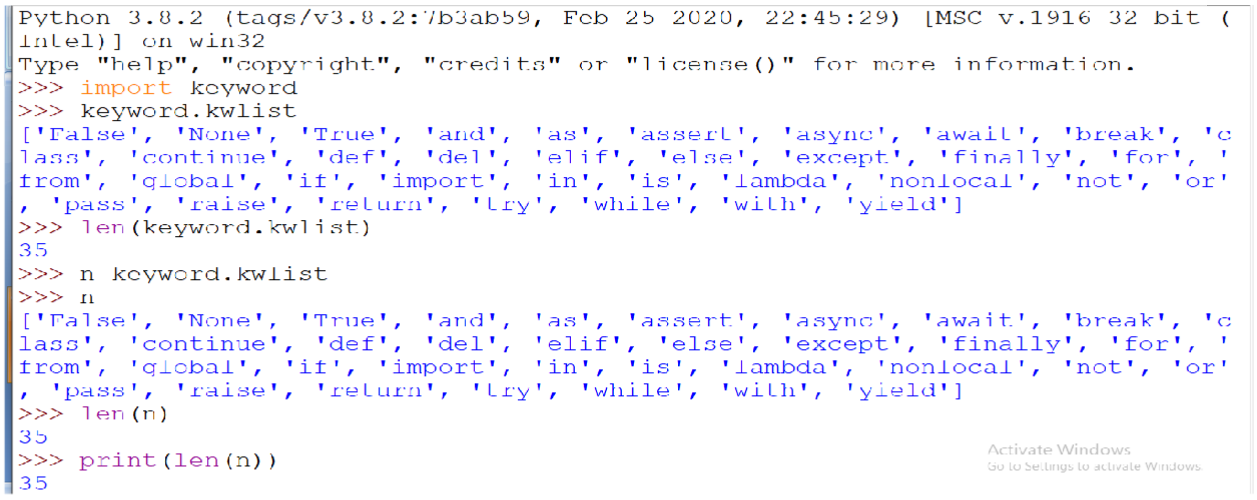
➢ These keywords can't be used as variable. Keywords are case sensitive

➢ There are 35 keywords in Python 3.7, 3.8

➢ All the keywords except True, False and None are in lowercase and they must be written as it is.

Example

import keyword
keyword.kwlist



```
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (
Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'c
lass', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', '
from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or'
, 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>> len(keyword.kwlist)
35
>>> n keyword.kwlist
>>> n
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'c
lass', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', '
from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or'
, 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>> len(n)
35
>>> print(len(n))
35
```

Python Comments

1) Single Line Comments

Comments are very important while writing a program. It describes what's going on inside a program so that a person looking at the source code does not have a hard time figuring it out. In Python, we use the hash (#) symbol to start writing a comment.

Eg:#This is a comment

```
#print out Hello
  print('Hello')
```

Output:

```
Hello
```

2) Multi-line comments

If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line.

Another way of doing this is to use triple quotes, either ''' or """.

These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well.

Example:

```
"""This is a comment written in more than
just one line-multiline comment"""
print("Hello,World")
```

Output

```
Hello World
```

Python Indentation

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation. A code block (body of a function, loop etc.) starts with indentation.

Indentation in Python refers to the (spaces and tabs) that are used at the beginning of a statement. The statements with the same indentation belong to the same group called a suite.

Eg: if True:

```
    print('Hello')
```

Command Line Arguments

argv is not Array it is a List. It is available sys Module.

The Argument which are passing at the time of execution are called Command Line Arguments.

Eg: python test.py arg1 arg2 arg3

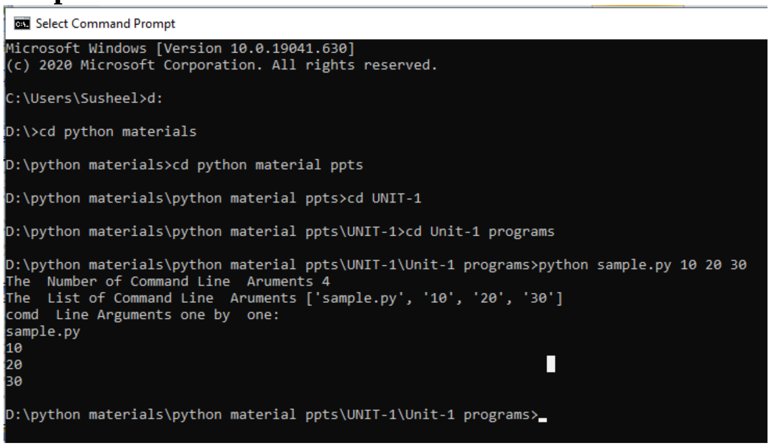
Within the Python Program this Command Line Arguments are available in argv. Which is present in SYS Module.

Note: argv[0] represents Name of Program. But not first Command Line Argument. argv[1] represent First Command Line Argument.

Program-1

```
from sys import argv
print ("The Number of Command Line Aruments", len(argv))
print ("The List of Command Line Aruments", argv)
print ("comd Line Arguments one by one:")
for x in argv:
   print(x)
```

Output:

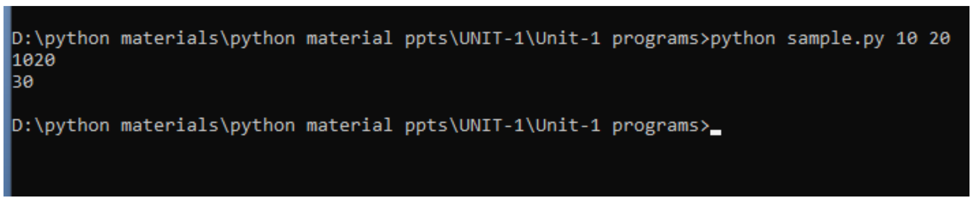


```
Select Command Prompt
Microsoft Windows [Version 10.0.19041.630]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Susheel>d:

D:\>cd python materials

D:\python materials>cd python material ppts

D:\python materials\python material ppts>cd UNIT-1

D:\python materials\python material ppts\UNIT-1>cd Unit-1 programs

D:\python materials\python material ppts\UNIT-1\Unit-1 programs>python sample.py 10 20 30
The  Number of Command Line  Aruments 4
The  List of Command Line  Aruments ['sample.py', '10', '20', '30']
comd  Line Arguments one by  one:
sample.py
10
20
30

D:\python materials\python material ppts\UNIT-1\Unit-1 programs>
```

Program-2

```
from sys import argv
print(argv[1]+argv[2])
print(int(argv[1])+int(argv[2]))
```

Output

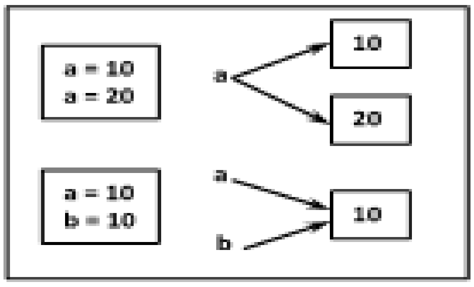


```
D:\python materials\python material ppts\UNIT-1\Unit-1 programs>python sample.py 10 20
1020
30

D:\python materials\python material ppts\UNIT-1\Unit-1 programs>
```

Data Types

- Data type represent the type of data present inside the variable.
- In Python we cannot required to specify the type explicitly. Based on value provided, the type will be assigned automatically. Hence python is dynamically typed language.
- Python contains following data types:

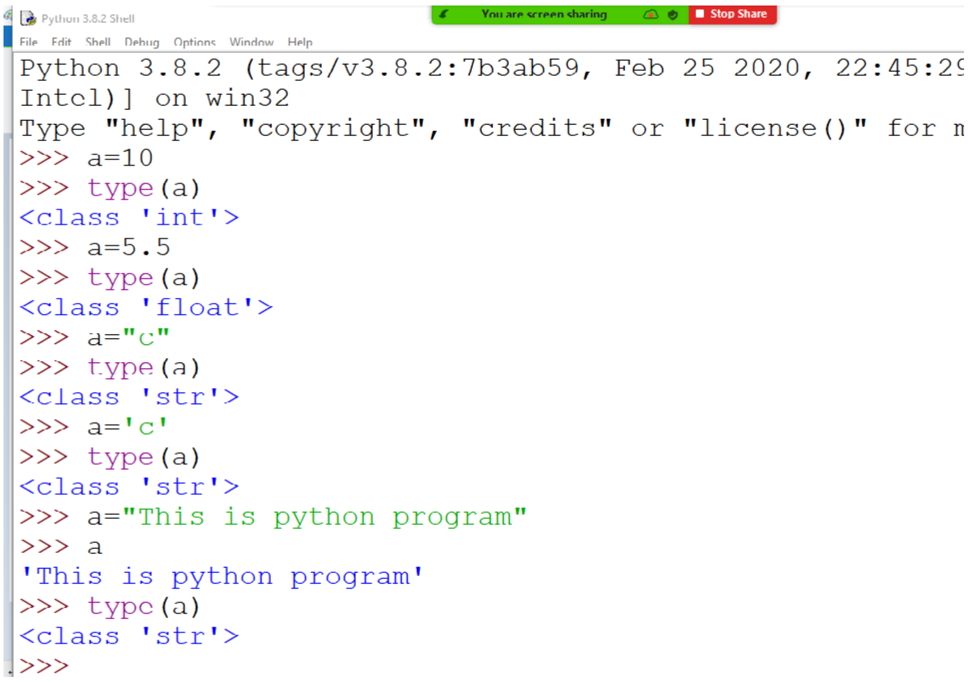
  int, float, complex, bool, range, str, list, tuple, set, dict, None



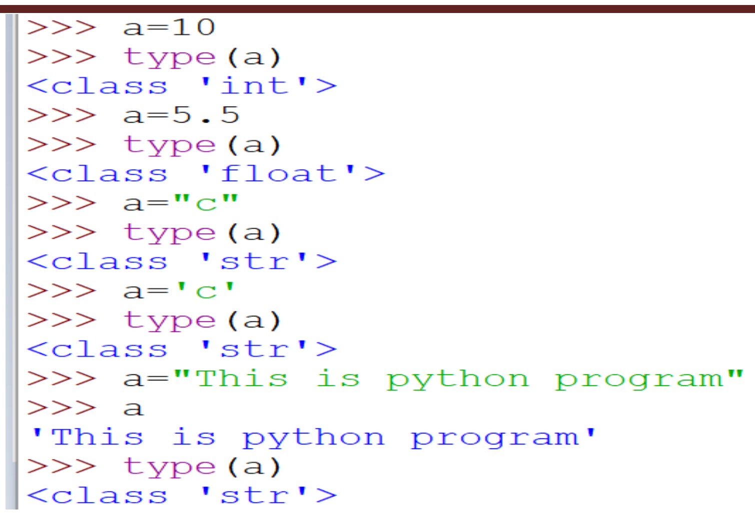
- Python contains several in-built functions.
- type()- to check the type of the variable. Eg:type(a)
- id()- to get the address of the object.
- print()- to print the values.

Note: In python everything is treated as object only.

Examples



```
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:2
Intel)] on win32
Type "help", "copyright", "credits" or "license()" for m
>>> a=10
>>> type(a)
<class 'int'>
>>> a=5.5
>>> type(a)
<class 'float'>
>>> a="c"
>>> type(a)
<class 'str'>
>>> a='c'
>>> type(a)
<class 'str'>
>>> a="This is python program"
>>> a
'This is python program'
>>> type(a)
<class 'str'>
>>>
```



```
>>> a=10
>>> type(a)
<class 'int'>
>>> a=5.5
>>> type(a)
<class 'float'>
>>> a="c"
>>> type(a)
<class 'str'>
>>> a='c'
>>> type(a)
<class 'str'>
>>> a="This is python program"
>>> a
'This is python program'
>>> type(a)
<class 'str'>
```

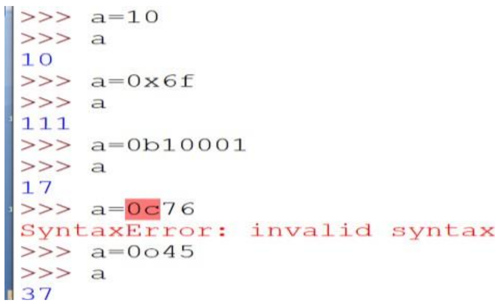
1) Int datatype

- We can use int datatype to represent the whole numbers(integral numbers). Integral Numbers means number without decimal points.
- In python2 we have long datatype to represent very large integral values. Eg: 908090800L, -0x1929292L
- In python3 there is no long type explicitly and we can represent long values also by using int type only. Size is not fixed in python.
- **Syntax:** Variable Name= Integer Value (or) Decimal (or) Binary (or) Octal (or) Hexa Decimal
- We can represent the int values in the following ways.

  1. Decimal
  2. Binary
  3. Octal
  4. Hexa

- **Decimal Form(Base-10):** It is the default number system in python. The allowed digits are 0 to 9.
- Eg: a=10
- **Binary Form(Base-2):** It allowed the digits are 0 and 1. Literal value should be prefixed with 0b or 0B. Eg: 1.x=0B1111 2. x=0B1011
- **Octal Form(Base-8):** The allowed digits are 0 to 7. Literal value should be prefixed with 0o or 0O. Eg: 0o123.
- **Hexadecimal Form(Base-16):** The allowed digits are 0-9, a-f(both lower case and upper case letters). Literal value should be prefixed with 0x or 0X. Eg: 0xaf

Examples

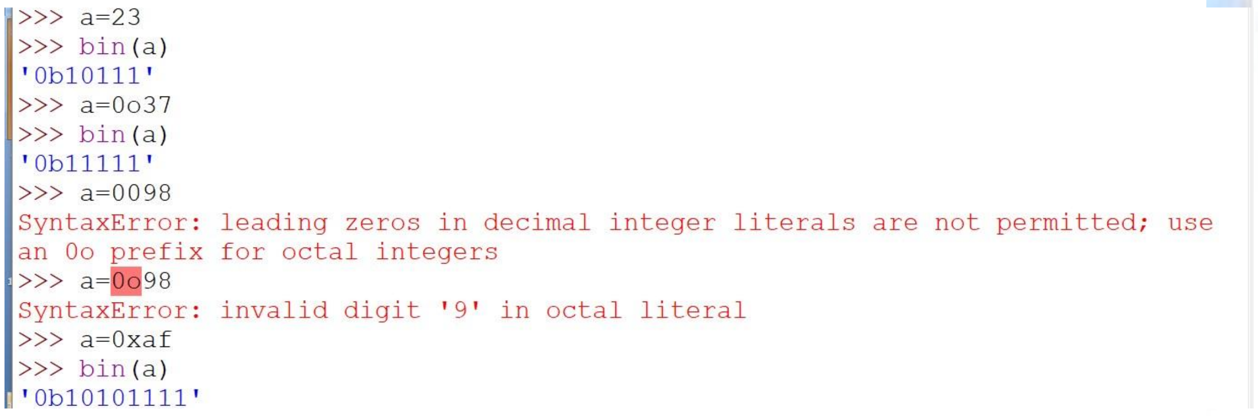

```
>>> a=10
>>> a
10
>>> a=0x6f
>>> a
111
>>> a=0b10001
>>> a
17
>>> a=0c76
SyntaxError: invalid syntax
>>> a=0o45
>>> a
37
```

Base Conversions

Python provide inbuilt functions for base conversions.

- **bin():** We can use bin() to convert from any base to binary. Eg: bin(0o723)
- **oct():** We can use oct() to convert from any base to octal. Eg: oct(10)
- **hex():** We can use hex() to convert from any base to hexadecimal. Eg: hex(100)

Examples


```
>>> a=23
>>> bin(a)
'0b10111'
>>> a=0o37
>>> bin(a)
'0b11111'
>>> a=0098
SyntaxError: leading zeros in decimal integer literals are not permitted; use
an 0o prefix for octal integers
>>> a=0o98
SyntaxError: invalid digit '9' in octal literal
>>> a=0xaf
>>> bin(a)
'0b10101111'
```

2) Float Datatype:

We use float type to represent the floating point values (decimal values).

Eg: f=10.5

We can also represent floating point values by using exponential forms(scientic notations)

Eg:a=1.2e3 a=6E+5 a=3.12E4

>>> a

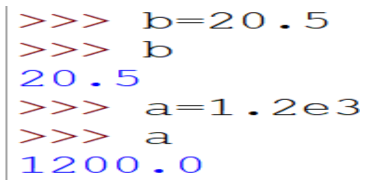
1200.0

instead of e we can use E. The main advantage of exponential form is we can represent big values in less memory.

Syntax: Variable Name= Float Values

Note: We can represent int values in decimal, binary octal and hexa decimal forms. But we can represent the float values by using decimal form.

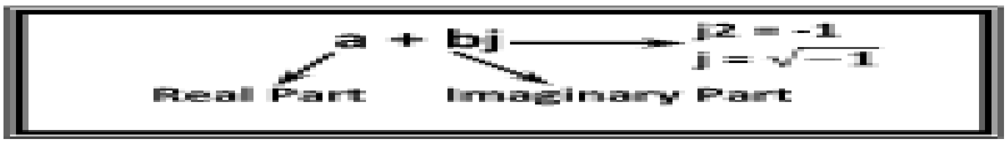
Examples



```
>>> b=20.5
>>> b
20.5
>>> a=1.2e3
>>> a
1200.0
```

3) Complex Datatype:

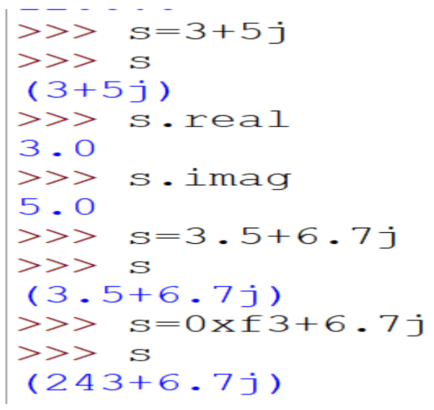
A complex number is of the form



a and b contain intergers or floating point values

- A complex number is of two parts one is real part and other one is imaginary part.
- Eg: 3+5j, 10+5.5j, 0.5+0.1j
- In the real part if we use int value that we can specify either by decimal, octal, binary or hexadecimal form.
- But imaginary part should be specified only by using decimal form.
- Complex datatype has some inbuilt attributes to retrieve the imaginary part
- and real part.   Eg: c=5+3j, c.real c.imag
- We can use complex datatype generally in scientific applications and EE Appli.
- **Syntax:** Variable Name= Complex Value

Examples



```
>>> s=3+5j
>>> s
(3+5j)
>>> s.real
3.0
>>> s.imag
5.0
>>> s=3.5+6.7j
>>> s
(3.5+6.7j)
>>> s=0xf3+6.7j
>>> s
(243+6.7j)
```

4) Bool Datatype

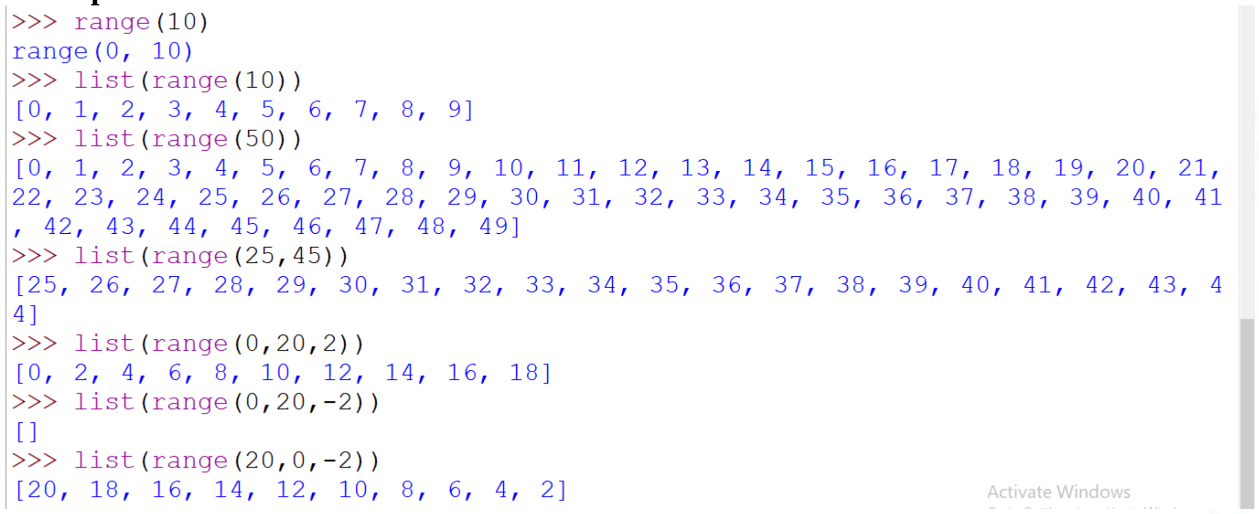
- We can use this datatype to represent boolean values. The boolean values are True and False.
- Internally python True as 1 and False as 0.
- **Syntax:** Variable Name=Boolean value

Eg: a=True
    type(a)-> True.

5) Range Datatype

- Range Data Type represents a sequence of numbers. The elements present in range Data type are not modifiable. i.e. range Data type is immutable.
- Range object does not support item assignment
- We can access elements present in the range data type by using index
- **Syntax:** list(range(values))
  list(range(begin,end))
  list(range(begin,end,step))
- Where step represents the increment or decrement the values in a range

Examples

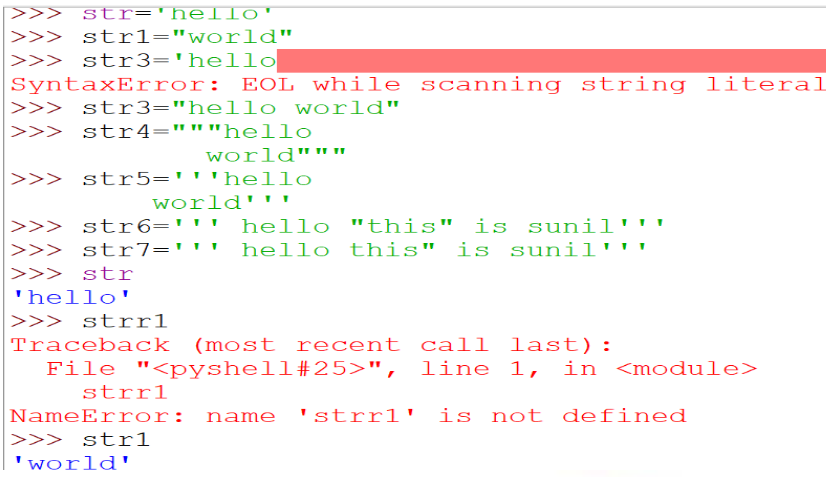


```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(50))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41
, 42, 43, 44, 45, 46, 47, 48, 49]
>>> list(range(25,45))
[25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 4
4]
>>> list(range(0,20,2))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> list(range(0,20,-2))
[]
>>> list(range(20,0,-2))
[20, 18, 16, 14, 12, 10, 8, 6, 4, 2]
```

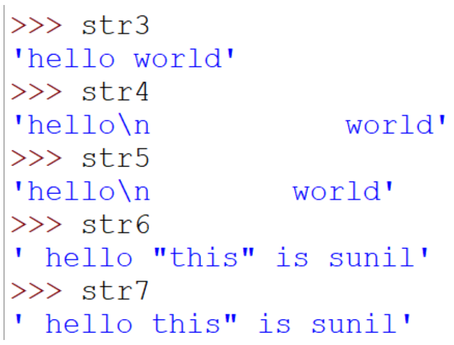
6) Str Datatype

- str represents String data type.
- A String is a sequence of characters enclosed within single quotes or double quotes.
- By using single quotes or double quotes we cannot represent multi line string literals.
- For this requirement we should go for triple single quotes(''') or triple double quotes(""")
- We can also use triple quotes to use single quote or double quote in our String.
- We can embed one string in another string
- In Python, we can represent char values also by using str type and explicitly char type is not available.
- **Syntax:** Variable name= String Value

Examples



```
>>> str='hello'
>>> str1="world"
>>> str3='hello
SyntaxError: EOL while scanning string literal
>>> str3="hello world"
>>> str4="""hello
           world"""
>>> str5='''hello
        world'''
>>> str6=''' hello "this" is sunil'''
>>> str7=''' hello this" is sunil'''
>>> str
'hello'
>>> strr1
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    strr1
NameError: name 'strr1' is not defined
>>> str1
'world'
```



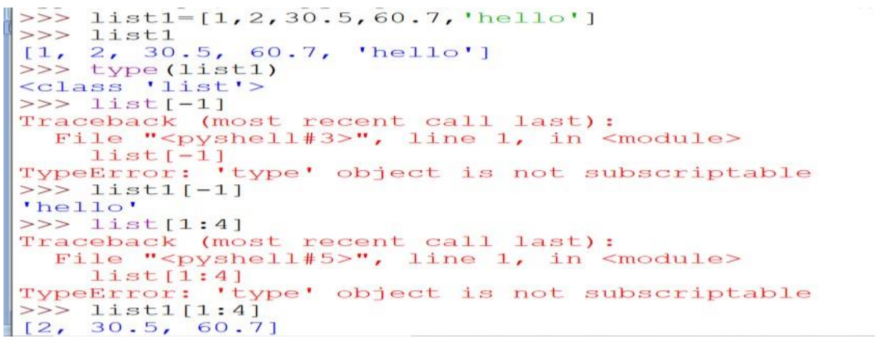
```
>>> str3
'hello world'
>>> str4
'hello\n           world'
>>> str5
'hello\n        world'
>>> str6
' hello "this" is sunil'
>>> str7
' hello this" is sunil'
```

7) List Datatype

 If we want to represent a group of values as a single entity where insertion order required to preserve and duplicates are allowed then we should go for list data type.

1. Insertion order is preserved
2. Mutable objects are allowed
3. Heterogeneous objects are allowed
4. Duplicates are allowed
5. Growable in nature
6. Values should be enclosed within square brackets []

Examples

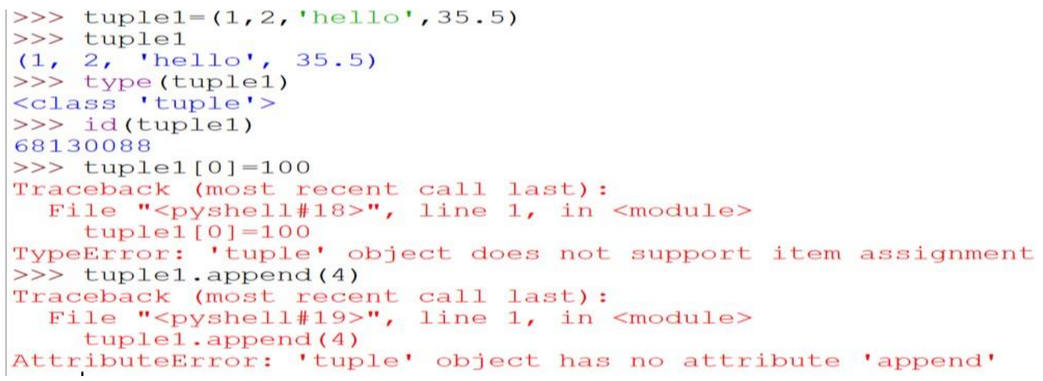


```
>>> list1=[1,2,30.5,60.7,'hello']
>>> list1
[1, 2, 30.5, 60.7, 'hello']
>>> type(list1)
<class 'list'>
>>> list[-1]
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    list[-1]
TypeError: 'type' object is not subscriptable
>>> list1[-1]
'hello'
>>> list[1:4]
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    list[1:4]
TypeError: 'type' object is not subscriptable
>>> list1[1:4]
[2, 30.5, 60.7]
```

8) Tuple Datatype

 Tuple data type is exactly same as list data type except that it is immutable. i.e we cannot change values. Tuple elements can be represented within parenthesis ().

Examples



```
>>> tuple1=(1,2,'hello',35.5)
>>> tuple1
(1, 2, 'hello', 35.5)
>>> type(tuple1)
<class 'tuple'>
>>> id(tuple1)
68130088
>>> tuple1[0]=100
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    tuple1[0]=100
TypeError: 'tuple' object does not support item assignment
>>> tuple1.append(4)
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    tuple1.append(4)
AttributeError: 'tuple' object has no attribute 'append'
```

9) Set Datatype

 If we want to represent a group of values without duplicates where order is not important then we should go for set Data Type. Set elements are represented with in curly braces{}

1. Insertion order is not preserved
2. Duplicates are not allowed
3. Heterogeneous objects are allowed
4. Index concept is not applicable
5. It is mutable collection
6. Growable in nature

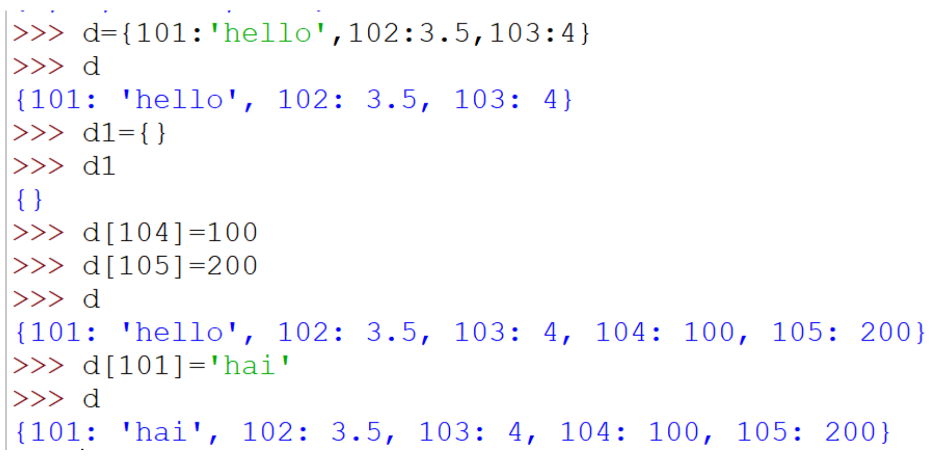
Examples

```
>>> s={1,2,3.6,'mam'}
>>> s
{3.6, 1, 2, 'mam'}
>>> s[0]=100
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    s[0]=100
TypeError: 'set' object does not support item assignment
>>> s.append(100)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    s.append(100)
AttributeError: 'set' object has no attribute 'append'
>>> s.add(100)
>>> s
{1, 2, 100, 'mam', 3.6}
```

10) Dict Datatype

 If we want to represent a group of values as key-value pairs then we should go for dict data type. Duplicate keys are not allowed but values can be duplicated. If we are trying to insert an entry with duplicate key then old value will be replaced with new value. dict is mutable and the order won't be preserved.

Examples

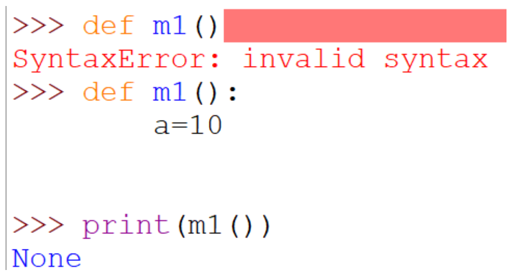


```
>>> d={101:'hello',102:3.5,103:4}
>>> d
{101: 'hello', 102: 3.5, 103: 4}
>>> d1={}
>>> d1
{}
>>> d[104]=100
>>> d[105]=200
>>> d
{101: 'hello', 102: 3.5, 103: 4, 104: 100, 105: 200}
>>> d[101]='hai'
>>> d
{101: 'hai', 102: 3.5, 103: 4, 104: 100, 105: 200}
```

11) None Datatype

 None means Nothing or No value associated. If the value is not available, then to handle such type of cases None introduced. It is something like null value in Java.

Examples



```
>>> def m1()
SyntaxError: invalid syntax
>>> def m1():
        a=10


>>> print(m1())
None
```

Type Casting

- We can convert one type value to another type. This conversion is called Typecasting or Type coersion.
- The following are various inbuilt functions for type casting.

1. int()
2. float()
3. complex()
4. bool()
5. str()

1) int()

We can use this function to convert values from other types to int

Example

```
>>> int(123.987)
123
>>> int(10+5j)
TypeError: can't convert complex to int
>>> int(True)
1
```

```
>>>int(False)
0
 >>>int("10")
 10
>>>int("10.5")
ValueError: invalid literal for int() with base 10: '10.5'
>>>int("ten")
ValueError: invalid literal for int() with base 10: 'ten'
>>> int("0B1111")
ValueError: invalid literal for int() with base 10: '0B1111'
```

2) float():

We can use float() function to convert other type values to float type.

Example

```
>>>float(10)
10.0
>>>float(10+5j)
TypeError: can't convert complex to float
>>>float(True)
1.0
>>>float(False)
0.0
>>>float("10")
10.0
 >>>float("10.5")
10.5
>>>float("ten")
ValueError: could not convert string to float: 'ten'
>>>float("0B1111")
ValueError: could not convert string to float: '0B1111'
```

3) complex()

We can use complex() function to convert other types to complex type.

Form-1: complex(x)

We can use this function to convert x into complex number with real part x and imaginary part 0.

Form-2: complex(x,y)

We can use this method to convert x and y into compl

Example-1

```
  complex(10)==>10+0j
  complex(10.5)===>10.5+0j
  complex(True)==>1+0j
  complex(False)==>0j
  complex("10")==>10+0j
  complex("10.5")==>10.5+0j
  complex("ten")
```

```
ValueError: complex() arg is a malformed string
```

Example-2

```
complex(10,-2)==>10-2j
complex(True,False)==>1+0j
```

4) bool()

We can use this function to convert other type values to bool type.
The bool() method is used to return or convert a value to a Boolean value i.e., True or False.
Here are few cases, in which Python’s bool() method returns false. Except these all other values return True.

If a False value is passed.
If None is passed.
If an empty sequence is passed, such as (), [], ”, etc
If Zero is passed in any numeric type, such as 0, 0.0 etc
If an empty mapping is passed, such as {}.

Example

```
bool(0)==>False
bool(1)==>True
bool(10)===>True
bool(10.5)===>True
bool(0.178)==>True
bool(0.0)==>False
bool(10-2j)==>True
bool(0+1.5j)==>True
bool(0+0j)==>False
bool("True")==>True
bool("False")==>True
bool("")==>False
```

5) str():
We can use this method to convert other type values to str type

Example

```
>>>str(10)
'10'
>>>str(10.5)
 '10.5'
>>>str(10+5j)
 '(10+5j)'
>>>str(True)
 'True'
```

UNIT-II

FUNCTIONS AND

Functions and Modules: Introduction to functions, Types of Functions, Flow of Execution, Parameters and arguments, Pass by reference and pass by value, Function arguments, Return Statement, Composition, Recursion, Python Modules.

FUNCTIONS

- Function is a named sequence of statements that performs a computation.
- When we define a function, we used to specify the name and the sequence of statements. Later, we can “call” the function by name.
- Function is a block of organized and reusable program code that performs a single, specific and well defined task.
- Programmers can break up a program into functions, each of which can be written more or less independently of others.
- Every function interfaces to the outside world in terms of how information is transferred to it and how results are generated by the function are transmitted back from it.
- This interface is basically called as function name.
- As soon as when a program encounters a function call, the control is passed to the first statement in the function.
- After completing the execution of all the statements in the function then the program control is passed to the statement following the one that called the function.

TYPES OF FUNCTIONS:

Functions are classified into 3 types. They are

1. Built in fuctions
2. Anonymous functions
3. User defined functions

1. Built in functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

Built-in Functions

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	import()
complex()	hasattr()	max()	round()	

2. Anonymous Functions

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n
```

```
mydoubler = myfunc(2)

print(mydoubler(11))
```

Or, use the same function definition to make a function that always triples the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

Or, use the same function definition to make both functions, in the same program:

Example

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Use lambda functions when an anonymous function is required for a short period of time.

3. User Defining Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def functionname( parameters ):
   "function_docstring"
```

```
  function_suite
  return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

FLOW OF EXECUTION:

- Order in which the statements are executed.
- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order from top to bottom.
- 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.
- A function call is like a detour in the flow of execution.
- Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.
- One function can call another. While in the middle of one function, the program might have to execute the statements in another function.

PARAMETERS & ARGUMENTS:

- Some of the built-in functions we have seen require arguments.
- For example, when we call math.sin we need to pass a number as an argument.
- Some functions take more than one argument, math.pow takes two, the base and the exponent.

```
>>> import math

>>> math.sin(20)

0.9129452507276277

>>> math.pow(2,3)

8.0
```

- Inside the function, the arguments are assigned to variables called parameters.
- Here is an example of a user-defined function that takes an argument:

```
import math

def print_twice(bruce):

    print (bruce)

    print (bruce)

print_twice('Spam')

print_twice(17)

print_twice(math.pi)

print_twice('Spam'*4)
```

- We can also use a variable as an argument,

```
>>> a = ‘hello'

>>> print_twice(a)

hello

hello
```

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

```
Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
```

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]; # This would assig new reference in mylist
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result −

```
Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
```

Function Arguments

You can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function printme(), you definitely need to pass one argument, otherwise it gives a syntax error as follows −

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme()
```

When the above code is executed, it produces the following result −

```
Traceback (most recent call last):
   File "test.py", line 11, in <module>
      printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function in the following ways −

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme( str = "My string")
```

When the above code is executed, it produces the following result −

```
My string
```

The following example gives more clear picture. Note that the order of parameters does not matter.

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result −

```
Name: miki
Age 50
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

When the above code is executed, it produces the following result −

```
Name: miki
Age 50
Name: miki
Age 35
```

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

```
#!/usr/bin/python

# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result −

```
Output is:
10
Output is:
70
60
50
```

RETURN STATEMENT:

- Every function has the implicit return statement as the last instruction in the function body
- This implicit return statement returns nothing to its caller.
- We can change this default value by explicitly using the return statement by which we can return some value back to the caller.
- Syntax:
- return [expression]
- The expression is written in brackets because it is optional. If the expression is present, it is evaluated and the resultant value is returned to the calling function

- The return statement is used for 2 things,

  Return value to the caller

  To end and exit a function and go back to its caller

Example:

```
def cube(x):
    z=x*x*x
    return(z)
num=10
result=cube(num)
print("Cube of",num,"=",result)
def display(str):
    print(str)
x=display("Hello World")
print(x)
```

Composition

While I don't consider myself a functional programming expert, all those hours spent in Haskell, Lisp and Scheme definitively changed my way of programming. So, after seeing a lot of unnecessarily complex implementations of **function composition** in Python on the Web, I decided to write this article to present a simple yet powerful solution that covers all use cases. If you are familiar with function composition, you may want to go to the **solution**.

Composing two functions

Function composition is a way of combining functions such that the result of each function is passed as the argument of the next function. For example, the composition of two functions f and g is denoted f(g(x)). x is the argument of g, the result of g is passed as the argument of f and the result of the composition is the result of f.

Let’s define compose2, a function that takes two functions as arguments (f and g) and returns a function representing their composition:

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

Example:

```
>>> def double(x):
...     return x * 2
...
>>> def inc(x):
...     return x + 1
...
>>> inc_and_double = compose2(double, inc)
>>> inc_and_double(10)
22
```

Composing n functions

Now that we know how to compose two functions, it would be interesting to generalize it to accept n functions. Since the solution is based on compose2, let’s first look at the composition of three functions using compose2.

```
>>> def dec(x):
...     return x - 1
...
>>> inc_double_and_dec = compose2(compose2(dec, double), inc)
>>> inc_double_and_dec(10)
21
```

Do you see the pattern? First, we compose the first two functions, then we compose the newly created function with the next one and so on.

Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, tri_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Example

Recursion Example

```
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable_name_. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>>
```

```
>>> import fibo
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>>
```

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,  89]
>>> fibo.__name__
```

```
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>>
```

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

UNIT-III

OPERATORS , CONTROL FLOW AND STRINGS

Operators and Control Flow: Operators, Conditional Statements, Iteration, Loop Control statements. Strings: Strings, String Slices, Immutability, String operations, String Methods, String Modules.

Operators in python

Operator is a symbol that performs certain operations on operands.

Python provides the following set of operators

1. Arithmetic Operators
2. Relational Operators or Comparison Operators
3. Logical operators
4. Bitwise oeprators
5. Assignment operators
6. Ternary Operator
7. Special operators

1) Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.

+	==>Addition
-	==>Subtraction
*	==>Multiplication
/	==>Division operator
%	==>Modulo operator
//	==>Floor Division operator
**	==>Exponent operator or power operator

Program

```
a=10
b=2
print('a+b=',a+b)
print('a-b=',a-b)
print('a*b=',a*b)
print('a/b=',a/b)
print('a//b=',a//b)
print('a%b=',a%b)
print('a**b=',a**b)
```

Output

```
a+b= 12
              = 8
a*b= 20
a/b= 5.0
a//b= 5
a%b= 0
a**b= 100
```

Note

- / operator always performs floating point arithmetic. Hence it will always returns float value. But Floor division (//) can perform both floating point and integral arithmetic. If arguments are int type then result is int type. If atleast one argument is float type then result is float type.
- We can use +,* operators for str type also. If we want to use + operator for str type then compulsory both arguments should be str type only otherwise we will get error.
- If we use * operator for str type then compulsory one argument should be int and other argument should be str type.

2) Relational Operators

Relational operators compare the values. It either returns **True** or **False** according to the condition.
> (greater than), >= (greaterthan or equal to), < (lessthan), <= (lessthan or equal to), == (double equal), != (not equal)

Program

```
a=10
b=20
print("a > b is ",a>b)
print("a >= b is ",a>=b)
print("a < b is ",a<b)
print("a <= b is ",a<=b)
print("a == b is ",a==b)
print("a != b is ",a!=b)
```

Output

```
a > b is  False
a >= b is False
a < b is True
a <= b is True
a == b is False
a != b is True
```

Note

Chaining of relational operators is possible. In the chaining, if all comparisons returns True then only result is True. If atleast one comparison returns False then the result is False

Eg:

1) 10<20 ==>True
2) 10<20<30 ==>True
3) 10<20<30<40 ==>True
4) 10<20<30<40>50 ==>False

3) Logical Operators

They allow a program to make a decision based on multiple conditions. Each operand is considered a condition that can be evaluated to a true or false value.

and, or ,not

For boolean types behavior:

- and ==>If both arguments are True then only result is True
- or ==>If atleast one arugemnt is True then result is True
- not ==>complement

For non boolean type's behavior

- 0 means False
- non-zero means True
- empty string is always treated as False

Program

```
a=10
b=20
c=0
print("a and b is ",a and b)
print("a or b is ",a or b)
print("not b is ",not b)
print("c and b is ",c and b)
print("c and b is ",c or b)
print("not c is", not c)
```

Output:

```
a and b is 20
a or b is 10
```

```
not b is False
c and b is 0
c and b is 20
not c is True
```

4) Bitwise Operators

- Bitwise operators acts on bits and performs bit by bit operation.We can apply these operators bitwise. These operators are applicable only for int and boolean types. By mistake if we are trying to apply for any other type then we will get Error.
- & (bitwise and),| (bitwise or), ^(bitwise xor), ~ (bitwise complement),<< (bitwise left shift), >> (bitwise right shit)

- & ==> If both bits are 1 then only result is 1 otherwise result is 0
- | ==> If atleast one bit is 1 then result is 1 otherwise result is 0
- ^ ==>If bits are different then only result is 1 otherwise result is 0
- ~ ==>Bitwise complement operator 1==>0 & 0==>1
- << ==>Bitwise Left shift
- >> ==>Bitwise Right Shift

Note:
The most significant bit acts as sign bit. 0 value represents +ve number where as 1 represents -ve value. Positive numbers will be represented directly in the memory where as -ve numbers will be represented indirectly in 2's complement form.

Program

```
a=10
b=4
print("a & b =", a & b)
print("a | b =", a | b)
print("~a =", ~a)
print("a ^ b =", a ^ b)
print("a<<b=",a<<b)
print("a>>b=",a>>b)
```

Output:

```
a & b = 0
a | b = 14
~a = -11
a ^ b = 14
a<<b= 160
a>>b= 0
```

5) Assignment Operators

Assignment operators are used to assign values to the variables. We can combine assignment operator with some other operator to form compound assignment operator.

The following is the list of all possible compound assignment operators in Python

+= , -= ,*= ,/= , %= ,//= ,**= ,&= ,|= ,^= , >>= ,<<=

Program:

```
a = 21
b = 10
c = 0
c += a
print("Line 2 - Value of c is ", c )
c *= a
print("Line 3 - Value of c is ", c)
c /= a
print("Line 4 - Value of c is ", c)
c = 2
c %= a
print("Line 5 - Value of c is ", c)
c **= a
print("Line 6 - Value of c is ", c)
c //= a
print("Line 7 - Value of c is ", c)
```

Output:

```
Line 2 - Value of c is  21
Line 3 - Value of c is  441
Line 4 - Value of c is  21.0
Line 5 - Value of c is  2
Line 6 - Value of c is  2097152
Line 7 - Value of c is  99864
```

6) Ternary Operator

Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being true or false.

Syntax:

x = firstValue if condition else secondValue

If condition is True then firstValue will be considered else secondValue will be considered.

Program

```
a,b=10,20
x=30 if a<b else 40
print(x)
```

Output:

```
30
```

Program

```
a=int(input("Enter First Number:"))
b=int(input("Enter Second Number:"))
min=a if a<b else b
print("Minimum Value:",min)
```

Output:

```
Enter First Number:4
Enter Second Number:5
Minimum Value: 4
```

7) Special Operator

Python defines the following 2 special operators

1. Identity Operators
2. Membership operators

1. Identity Operator

We can use identity operators for address comparison. There are two identity operators are available

1. is
2. is not

r1 is r2 -- returns True if both r1 and r2 are pointing to the same object

r1 is not r2 -- returns True if both r1 and r2 are not pointing to the same object

Program

```
a="durga"
b="durga"
print(id(a))
print(id(b))
print(a is b)
print(a is not b)
```

Output:

```
65621440
65621440
True
False
```

2. Membership Operator

- We can use Membership operators to check whether the given object present in the given collection.(It may be String, List, Set, Tuple or Dict) .

- in- Returns True if the given object present in the specified location
- not in- Returns if the given object is not present in the specified location

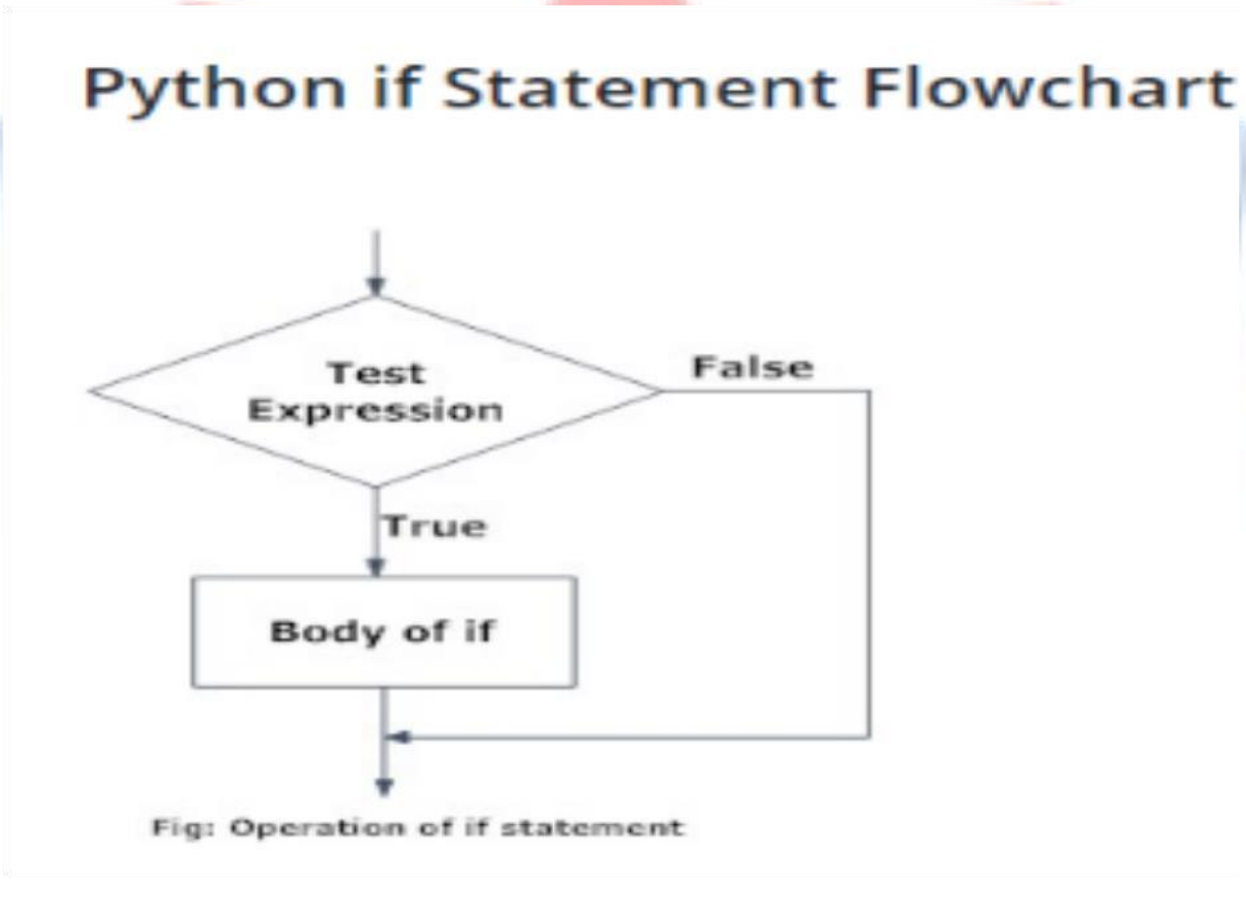
Output:

```
True
False,
True
True
```

Conditional Recursion (If Statement)

- Decision making is required when we want to execute a code only if a certain condition is satisfied.
- **Syntax:** iftest expression: statement(s)

Flowchart



Example

```
num = 3
if num > 0:
print(num,"isapositivenumber.") print("This is always printed.")
```

Output

```
3 is a positive number This is always printed
```

If-Else(Alternative Execution)

The if..else statement evaluates test expression and will execute body of if only when test condition is True.

If the condition is False, body of else is executed. Indentation is used to separate the blocks.

Syntax

```
if test expression:
        Body of if
        else:
        Body of else
```

FlowChart

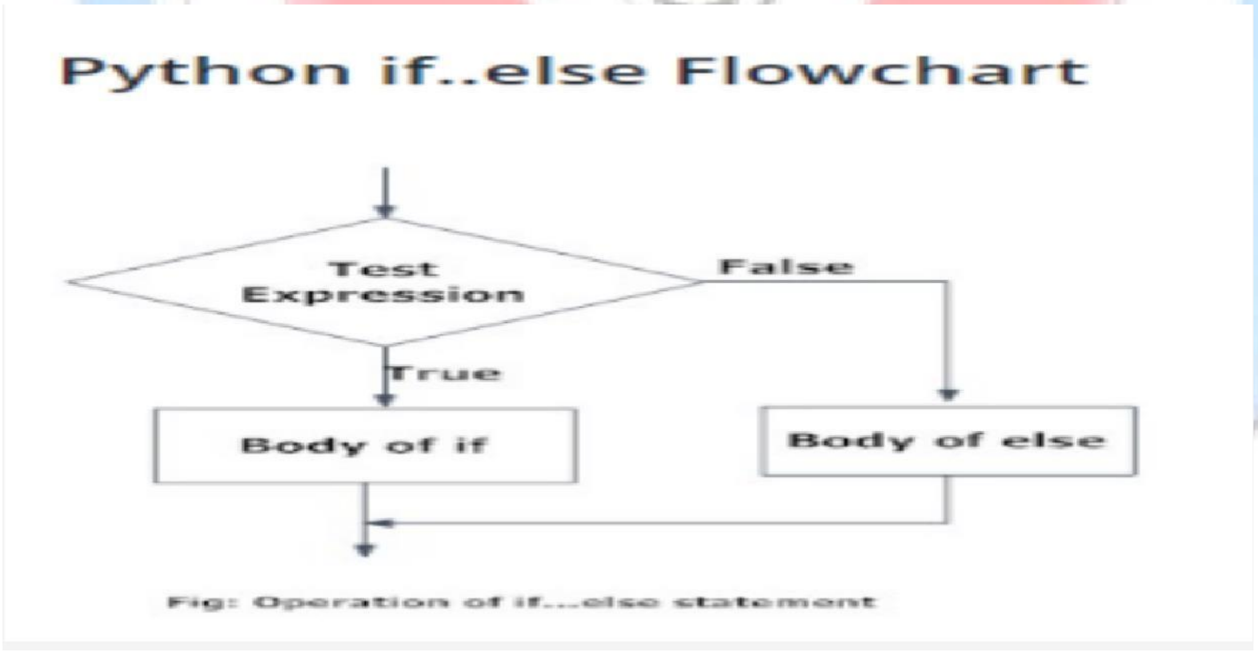


Fig: Operation of if...else statement

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
   print("b is not greater than a")
```

Output

```
        b is not greater than a
```

if...elif...else(Chained Conditionls)

The elif is short for else if. It allows us to check for multiple expressions.
Sometimes there are more than two possibilities and we need more than two branches.
One way to express a computation like that is a chained conditional.

If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, body of else is executed.
Only one block among the several if...elif...else blocks is executed according to the condition.
The if block can have only one else block. But it can have multiple elif blocks Syntax:

```
if test expression:
Body of if
elif test expression:
Body of elif
else: Body of else
```

Flowchart

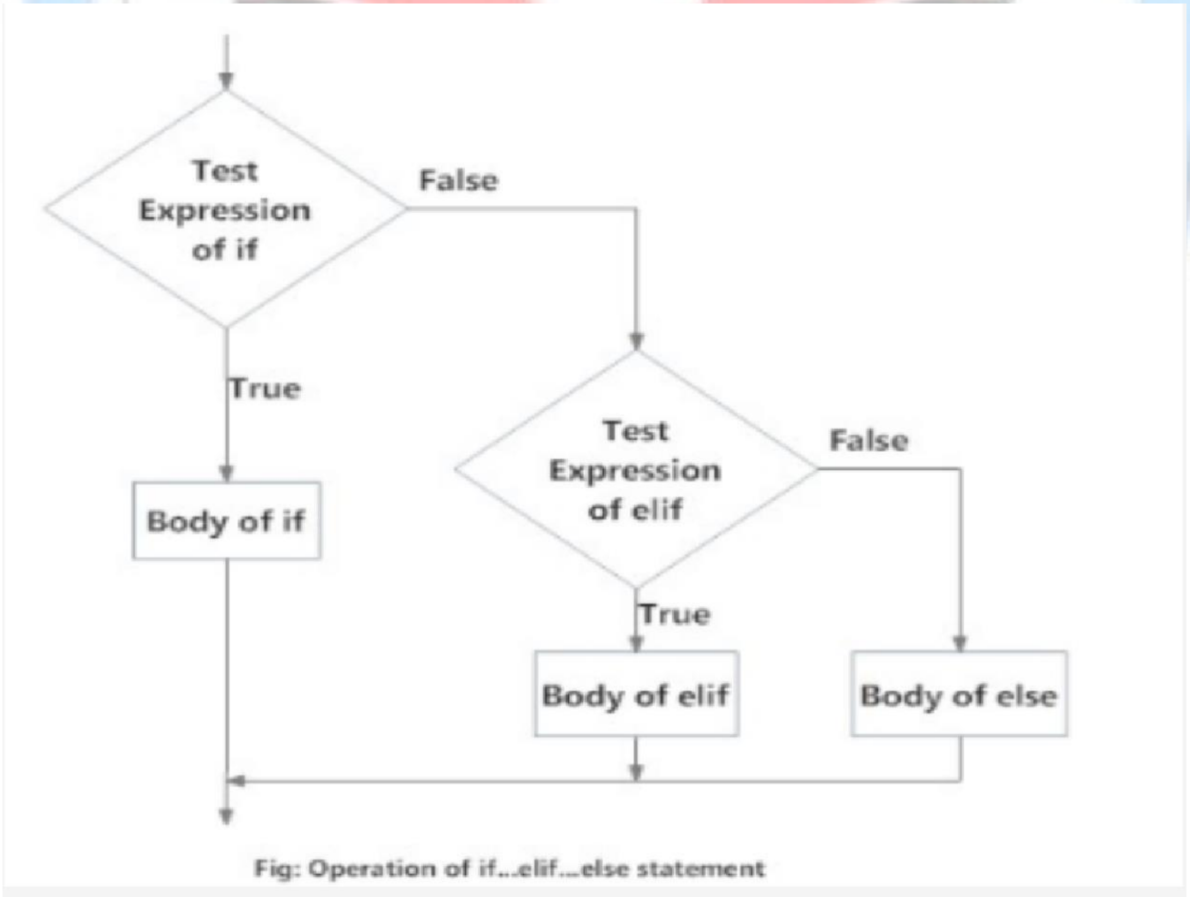


Fig: Operation of if...elif...else statement

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

Output

```
i is 20
```

Nested Conditionals

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting.

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly.

Syntax:

```
 if expression1:
 statement(s)
 if expression2:
  statement(s)
 elif expression3:
 statement(s)
 elif expression4:
statement(s) else: statement(s)
else:
 statement(s)
```

Example

```
num = float(input("Enter a number: ")) if num >= 0:
    if num == 0:
          print("Zero")
    else:
          print("Positive number") else:
    print("Negative number")
```

Output

Enter a number: 5 Positive number Recursion

It is legal for one function to call another; it is also legal for a function to call itself.
A function that calls itself is recursive; the process is called recursion.

Example

```
def countdown(n):
   if n<=0:
         print("blatsoff")
   else:
         print(n)
         countdown(n-1) countdown(3)
```

Output

```
3
2
1
Blastoff
```

The execution of countdown begins with n=3, and since n is greater than 0, it outputs the value 3, and then calls itself...

The execution of countdown begins with n=2, and since n is greater than 0, it outputs the value 2, and then calls itself...

The execution of countdown begins with n=1, and since n is greater than 0, it outputs the value 1, and then calls itself...

The execution of countdown begins with n=0, and since n is not greater than 0, it outputs the word, “Blastoff!” and then returns.

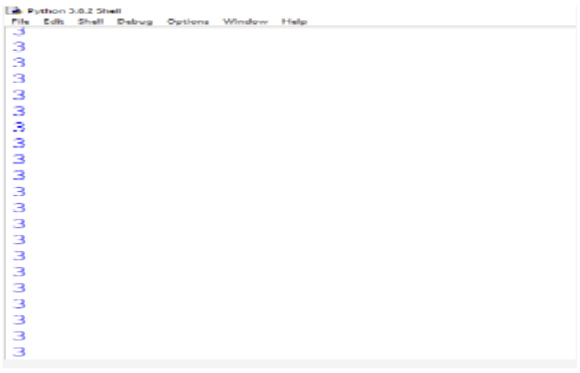
Infinite Recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as infinite recursion

Example

```
def print_n(s,n):
   if n<=0:
         return 0
   print(s)
   print_n(s,n+1)
print(print_n(3,3))
```

Output



```
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
```

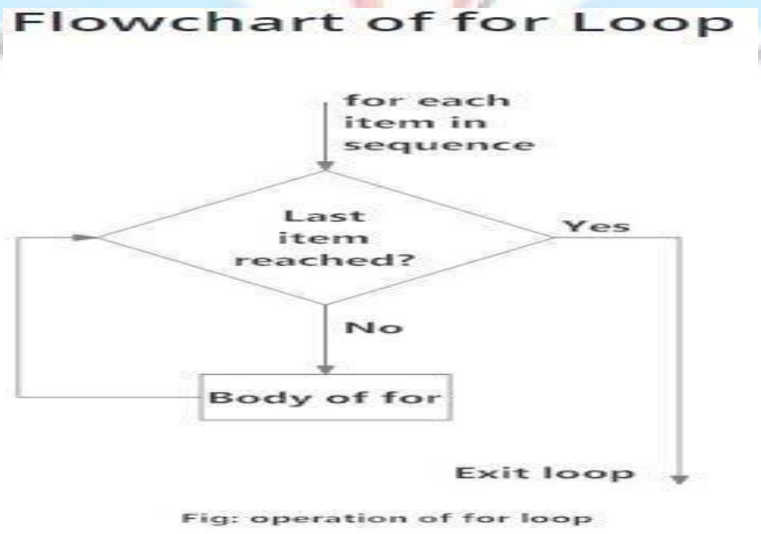
Iterative Statements

- ➢ If we want to execute a group of statements multiple times then we should go for Iterative statements. Python supports 2 types of iterative statements.

1. For Loop
2. While Loop
3. Infinite Loop
4. Nested Loop

1. For Loop

- ➢ The for loop in Python is used to iterate over a sequence (list,tuple,string etc) or other iterable objects. Iterating over a sequence is called traversal.



Syntax:

- ➢ for variable_name in sequence: body
- ➢ where sequence can be string or any collection. Body will be executed for every element present in the sequence.

Program

```
def word(word): previous=word[0] for c in word:
if c<previous: return False
previous=c return True
print(word('money'))
```

Output

```
False
```

2. While Loop:

- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

Syntax:

```
while test_expression:
Body of while
```

- In while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.
- In Python, the body of the while loop is determined through indentation.

Flowchart of while Loop

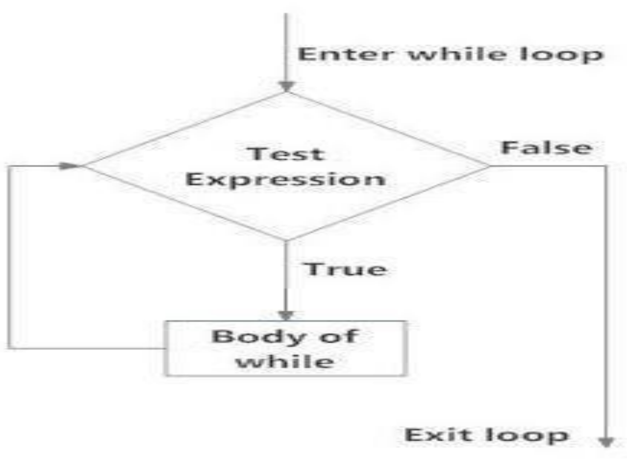


Fig: operation of while loop

Program

```
def word1(word):
i=0
while i<len(word)-1:
if word[i+1]<word[i]: return False
i=i+1 return True
print(word1('money'))
```

Output

```
False
```

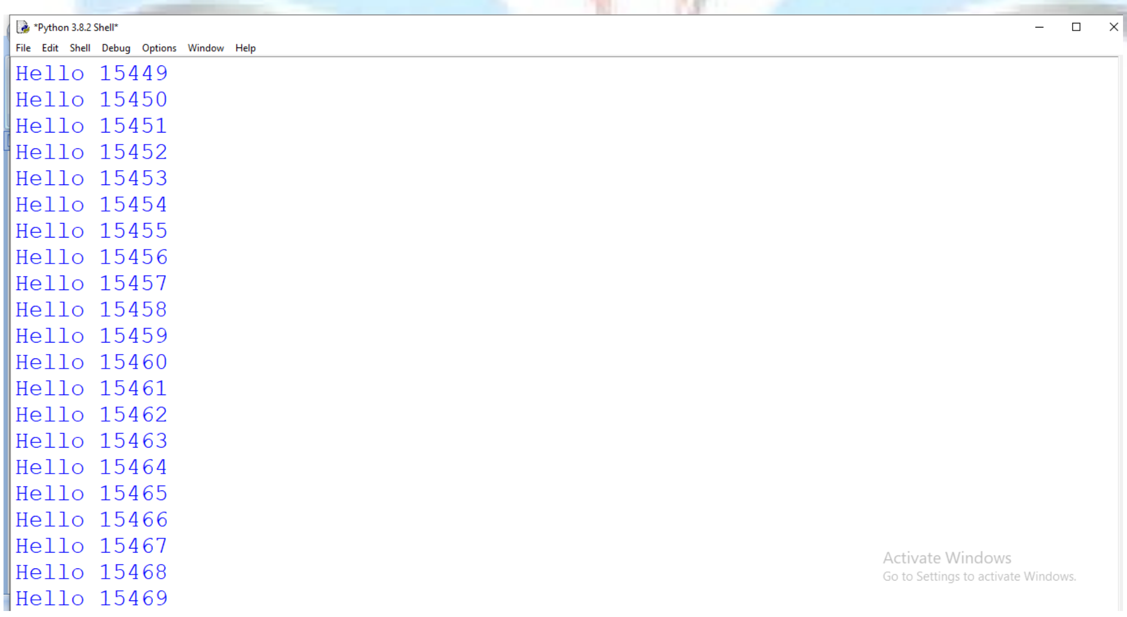
Infinite Loop

An Infinite Loop in Python is a continuous repetitive conditional loop that gets executed until an external factor interfere in the execution flow, like insufficient CPU memory, a failed feature/ error code that stopped the execution, or a new feature in the other legacy systems that needs code integration.

Program

```
i=0;
while True : i=i+1; print("Hello",i)
```

Output


Python 3.8.2 Shell
File Edit Shell Debug Options Window Help
Activate Windows
Go to Settings to activate Windows.

```
Hello 15449
Hello 15450
Hello 15451
Hello 15452
Hello 15453
Hello 15454
Hello 15455
Hello 15456
Hello 15457
Hello 15458
Hello 15459
Hello 15460
Hello 15461
Hello 15462
Hello 15463
Hello 15464
Hello 15465
Hello 15466
Hello 15467
Hello 15468
Hello 15469
```

Nested Loops

Python programming language allows to use one loop inside another loop

3. Nested For Loop

Python allows to use for loop inside the another for loop

Syntax:

```
for iterating_var in sequence:
for iterating_var in sequence:
statements(s)
statements(s)
```

program

```
for i in range(4):
for j in range(4):
print("i=",i," j=",j)
```

Output

```
i= 0    j= 0
i= 0    j= 1
i= 0    j= 2
i= 0    j= 3
i= 1    j= 0
i= 1    j= 1
i= 1    j= 2
i= 1    j= 3
i= 2    j= 0
i= 2    j= 1
i= 2    j= 2
i= 2    j= 3
i= 3    j= 0
i= 3    j= 1
i= 3    j= 2
i= 3    j= 3
```

1. Nested While Loop

Python allows to use while loop inside the another while loop

Syntax:

```
        while expression: while expression:
        statement(s) statement(s)
```

Program

```
i = 2
while(i < 100):
j = 2
while(j <= (i/j)):
if not(i%j): break j = j + 1
if (j > i/j) : print(i, " is prime") i = i + 1
print("Good bye!")
```

Output

```
2 is prime
3 is prime
5  is prime
7  is prime
11  is prime
13  is prime
17  is prime
19  is prime
23  is prime
29  is prime
31  is prime
37  is prime
41  is prime
43  is prime
47  is prime
53  is prime
59  is prime
61  is prime
67  is prime
71  is prime
73  is prime
79  is prime
83  is prime
89  is  prime
97  is  prime
Good bye!
```

Strings:

String is a sequence

A string is a sequence of characters. You can access the characters one at a time with the

bracket operator:

fruit=‘banana’ letter=fruit[1]

- The second statement selects character number 1 from fruitt and assigns it to letter.
- The expression in brackets is called an index. The index indicates which character in the sequence you want (hence the name).

Print (letter)

Output: a

- For most people, the first letter of ‘banana’ is b, not a. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.
  fruit=‘banana’
  letter=fruit[0]
  Print (letter)

Output: b

- So b is the 0th letter (“zero-eth”) of ‘banana’, a is the 1th letter (“one-eth”), and n is the 2th (“two- eth”) letter.
- You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

letter= fruit[1.5]
TypeError: String indices must be integers not float

Program

```
>>> fruit='banana'
>>> fruit 'banana'
>>> letter=fruit[1]
>>> letter 'a'
>>> print(letter) a
>>> letter=fruit[0]
>>> letter 'b'
>>> print(letter) b
>>> letter=fruit[1.5]
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module> letter=fruit[1.5]
TypeError: string indices must be integers
```

Len Function

- len is a built-in function that returns the number of characters in a string. fruit=‘banana’
  len(fruit)
  Output:6

- To get the last letter of a string, you might be tempted to try something like this:
  length=len(fruit)

  last=fruit(length)

  Index Error: String index out of range
- The reason for the index errror is that there is no letter in banana with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length.
  last=fruit(length-1) last
  Output:a

Program

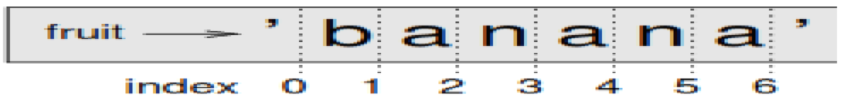
```
>>> fruit='banana'
>>>length=len(fruit)
>>>last=fruit[length]
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module> last=fruit[length]
IndexError: string index out of range
>>> last=fruit[length-1]
>>> las
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module> las
NameError: name 'las' is not defined
>>> last 'a'
```

Traversal with a loops

- A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. The ways to write a traversal is with a for loop and while loop.
- This loop traverses the string and displays each letter on a line by itself.
- The loop condition is index<len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed.
- The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.
- Each time through the loop, the next character in the string is assigned to the variable

char. The loop continues until no characters are left.

- The following example shows how to use concatenation (string addition) and a for loop to generate an abecedarian series (that is, in alphabetical order).
- In Robert McCloskey's book Make Way for Ducklings, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order.



Program-1

```
fruit='banana' for char in fruit:
print(char)
```

Output

```
b a n a n a
```

Program-2

```
fruit='banana' index=0
while(index<len(fruit)): letter=fruit[index]
print(letter) index=index+1
```

Output

```
ba n a n a
```

Program-3

```
prefixies='JKLMNOPQ' suffix='ack'
for letter in prefixies: print(letter+suffix)
```

Output

```
Jack Kack Lack Mack Nack Oack Pack Qack
```

String Slices

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

- The operator [n:m] returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing between the characters.
- If you omit the first index (before the colon), the slice starts at the beginning of the string.
- If you omit the second index, the slice goes to the end of the string.
- If the first index is greater than or equals to the second the result is an empty string, represented by two quotation marks.
- An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Program

```
s='monthy python'
print(s[0:5])
print(s[6:12])
fruit='banana'
print(fruit[:3])
print(fruit[3:])
print(fruit[3:3])
```

Output

```
month pytho ban ana
```

Strings are Immutable:

- It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string.
- greeting=‘Hello World’
- Greeting[0]=‘j’
- TypeError: String assignment doesnot support item assignment
- The “object” in this case is the string and the “item” is the character you tried to assign. For now, an object is the same thing as a value, but we will refine that definition later. An item is one of the values in a sequence.
- The reason for the error is that strings are immutable, which means you can’t change an existing string. The best you can do is create a new string that is a variation on the original.

Program

```
greeting='Hello World'
new_greeting='j'+greeting[1:]
print(new_greeting)
```

Output

```
jello World
```

Searching

- In a sense, find is the opposite of the [] operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns -1
- This is the first example we have seen of a return statement inside a loop. If word[index]==lettter, the function breaks out of the loop and returns immediately.
- If the character doesn't appear in the string, the program exits the loop normally and returns -1
- This pattern of computation—traversing a sequence and returning when we find what we are looking for—is called a search.

Program

```
def find(word,letter):
index=0
while(index<len(word)):
if(word[index]==letter):return index
index=index+1
return -1
 print(find('hello','l'))
```

Output

```
2
```

Looping and Counting

- This program demonstrates another pattern of computation called a counter. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result—the total number of a's.

Program

```
word='banana' count=0
for letter in word: if letter=='a':
count=count+1
print(letter,count)
```

Output

```
a 3
```

String Methods

- A method is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method upper takes a string and returns a new string with all uppercase letters.
- This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no argument.
- A method call is called an invocation; in this case, we would say that we are invoking upper on the word.

1. find()

Returns index of first occurrence of the given substring.

If it is not available then we will get -1

Syntax:
s.find(substring)
It will always search from begin index to end-1 index
Syntax: s.find(substring,begin,end)

Program-1

```
1) s="Learning Python is very easy“
2)      print(s.find("Python"))
3)      print(s.find("Java"))
4)      print(s.find("r"))
5)      print(s.rfind("r"))
```

Output

```
9
-1
3
21
```

Program-2

```
s="durgaravipavanshiva"
 print(s.find('a'))
print(s.find('a',7,15))
print(s.find('z',7,15))
```

Output

```
4
10
-1
```

2. rfind()

- rfind() method returns the highest index of the substring if found in given string.
- If not found then it returns -1.

Syntax: string.rfind(value, start, end)

Program

```
txt = "Hello, welcome to my world."
 x = txt.rfind("e", 5, 10)
print(x)
```

Output 8

3. index() method

index() method is exactly same as find() method except that if the specified substring is not available then we will get ValueError.
Syntax: string.index(elmnt)

Program

```
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
 index = vowels.index('e')
 print('The index of e:', index)
```

Output

```
The index of e: 1
```

4. **rindex():**

- The rindex() method returns the highest index of the substring inside the string (if found). If the substring is not found, it raises an exception

Syntax: string.rindex(value, start, end)

Program-1

```
quote = 'Let it be, let it be, let it be'
result = quote.rindex('let it')
print("Substring 'let it':", result)
```

Output

```
Substring 'let it': 22
```

Program-2

```
txt = "Hello, welcome to my world."
 x = txt.rindex("e", 5, 10)
print(x)
```

Output 8

Counting substring in the given String:

- We can find the number of occurrences of substring present in the given string by using count() method.
- **s.count(substring)** ==> It will search through out the string
- **s.count(substring, begin, end)** ===> It will search from begin index to end-1 index

program

```
s="abcabcabcabcadda" print(s.count('a'))
print(s.count('ab'))
print(s.count('a',3,7))
```

Output

```
6
4
2
```

Replacing a string with another string

- The replace() method returns a copy of the string where all occurrences of a substring is replaced with another substring.
  Syntax: string.replace(oldvalue, newvalue, count)
- Once we creates string object, we cannot change the content. This non changeable behaviour is nothing but immutability. If we are trying to change the content by using any method, then with those changes a new object will be created and changes won't be happened in existing object.
- Hence with replace() method also a new object got created but existing object won't be changed.

Program-1

```
s="Learning Python is very difficult"
 s1=s.replace("difficult","easy")
 print(s1)
```

Output

```
Learning Python is very easy
```

Program-2

```
txt = "one one was a race horse, two two was one too."
x = txt.replace("one", "three", 2)
print(x)
```

Output

```
Three three was a race horse, two two was one too
```

Splitting of Strings

split() method returns a list of strings after breaking the given string by the specified separator.
Syntax: str.split(separator, maxsplit)

Program

```
text = 'Hello python students' # Splits at space print(text.split())
word = 'Hello, python, students' # Splits at ',' print(word.split(','))
word = 'Hello:python:students' # Splitting at ':' print(word.split(':'))
word = 'CatBatSatFatOr' # Splitting at 3
print([word[i:i+3] for i in range(0, len(word), 3)])
```

Output

```
 ['Hello', 'python', 'students']
['Hello', ' python', ' students']
['Hello', 'python', 'students'] ['Cat', 'Bat', 'Sat', 'Fat', 'Or']
```

Joining of Strings

- The join() string method returns a string by joining all the elements of an iterable, separated by a string separator.
- The join() method provides a flexible way to create strings from iterable objects. It joins each element of an iterable (such as list, string, and tuple) by a string separator (the string on which the join() method is called) and returns the concatenated string.

  Syntax: string.join(iterable)

Program

```
# .join() with lists numList = ['1', '2', '3', '4']
separator = ', '
print(separator.join(numList))
 # .join() with tuples numTuple = ('1', '2', '3', '4')
print(separator.join(numTuple))
s1 = 'abc'
s2 = '123'
# each element of s2 is separated by s1 # '1'+ 'abc'+ '2'+ 'abc'+ '3'
print('s1.join(s2):', s1.join(s2))
# each element of s1 is separated by s2 # 'a'+ '123'+ 'b'+ '123'+ 'b'
print('s2.join(s1):', s2.join(s1))
```

Output

```
1, 2, 3, 4
1, 2, 3, 4
s1.join(s2): 1abc2abc3 s2.join(s1): a123b123c
```

Changing case of a String:

We can change case of a string by using the following 5 methods.

1. **upper()**===>To convert all characters to upper case

   Syntax: string.upper()

2. **lower()** ===>To convert all characters to lower case

   Syntax: string.lower()
3. **swapcase()**===>converts all lower case characters to upper case and all upper case characters to lower case

   Syntax: string.swapcase()
4. **title()** ===>To convert all character to title case. i.e first character in every word should be upper case and all remaining characters should be in lower case.

   Syntax: string.title()
5. **capitalize()** ==>Only first character will be converted to upper case and all remaining characters can be converted to lower case

   Syntax: string.capitalize()

Program

```
s='learning Python is very Easy' print(s.upper())
print(s.lower())
print(s.swapcase())
print(s.title())
print(s.capitalize())
```

Output

```
LEARNING PYTHON IS VERY EASY
learning python is very easy LEARNING pYTHON IS VERY eASY
Learning Python Is Very Easy Learning python is very easy
```

To check type of characters present in a string

Python contains the following methods for this purpose.

1) isalnum(): Returns True if all characters are alphanumeric(a to z , A to Z ,0 to9)

Syntax: string.isalnum()

2) isalpha(): Returns True if all characters are only alphabet symbols(a to z,A to Z)

Syntax: string.isalpha()

3) isdigit(): Returns True if all characters are digits only(0 to 9). **Syntax:** string.isdigit()

4) islower(): Returns True if all characters are lower case alphabet symbols. **Syntax:** string.islower()

5) isupper(): Returns True if all characters are upper case alphabet symbols **Syntax:** string.isupper()

6) istitle(): Returns True if string is in title case. **Syntax:** string.istitle()

7) isspace(): Returns True if string contains only spaces **Syntax:** string.isspace()

program

```
print('Durga786'.isalnum())
print('durga786'.isalpha())
print('durga'.isalpha())
print('durga'.isdigit())
print('786786'.isdigit())
print('abc'.islower())
print('Abc'.islower())
print('abc123'.islower())
print('ABC'.isupper())
print('Learning python is Easy'.istitle())
print('Learning Python Is Easy'.istitle())
print('  '.isspace())
```

Output

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

In Operator:

- The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second.
- With well-chosen variable names, Python sometimes reads like English. You could read this loop, “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.”

Program

```
def in_both(word1,word2): for letter in word1:
if letter in word2: print(letter)
in_both('apples','oranes')
```

Output

```
a e s
```

String Comparison

- Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters.
- A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

Program

```
word="orange"
if word=='banana': print('All right, bananas')
if word<'banana':
print('your word,'+word+',comes before banana') else:
print('your word,'+word+',comes after banana')
```

Output

```
your word,orange,comes after banana
```

STRING MODULES:

- String Module consist of a number of useful constants, classes and functions.
- These functions are used to manipulate strings.
- To see the content of string module, we can use dir() with the module name as the argument as shown below,

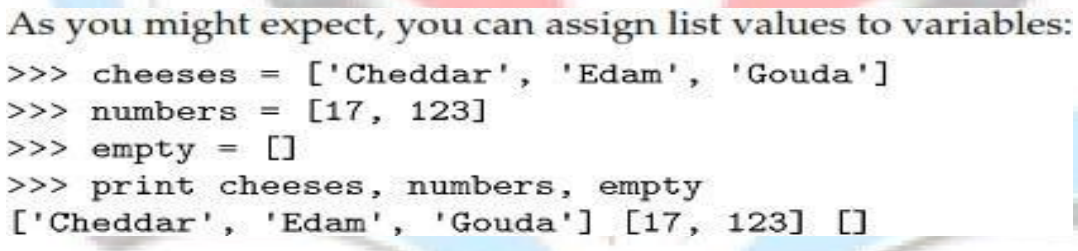
```
>>> dir(str)
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'_format_', '_ge_', '_getattribute_', '_getitem_', '_getnewargs_', '_gt_', '_hash_',
'__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul_',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'_setattr_', '_sizeof_', '_str_', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

List and Tuples: Introduction, List operations, List Slices, List Methods, Loops, Mutability, Aliasing, Cloning Lists, Parameters, Tuples, Tuple assignment, Tuple as return value.

List is a sequence:

- Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called elements or sometimes items.
- There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):
  - [10,20,30,40] ['a','b','c','d']
- The first example is a list of four integers. The second is a list of three strings.
- The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list.
  - Eg:['spam',30,40.5,[1,2]]
- A list within another list is nested.
- A list that contains no elements is called an empty list; you can create one with empty brackets [ ].



As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

Note: In the above example put the brackets to print function

Lists are mutable

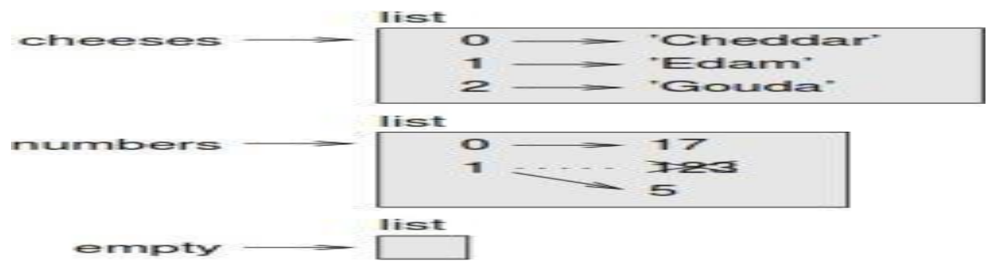
The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0.

Eg:

```
Cheeses=['Cheddar','Edam','Gouda']
 print(cheeses[0])
```

Output:

cheddar



Eg:

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
numbers=[17,123]

numbers[1]=5

print(numbers)
```

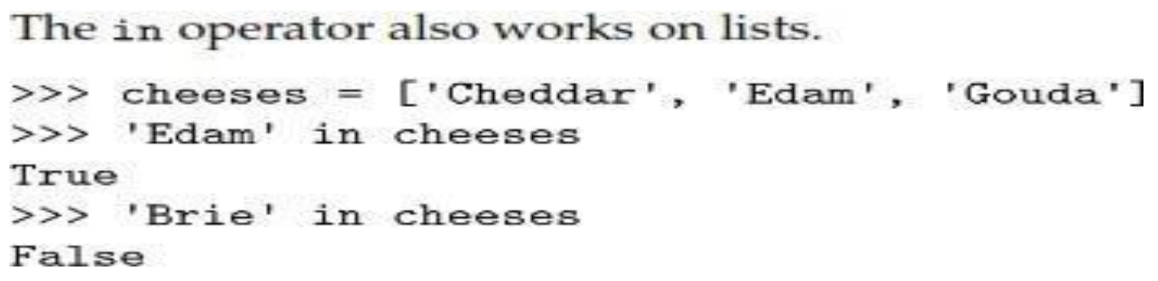
Output:

```
[17,5]
```

- The one-eth element of numbers, which used to be 123, is now 5.
- We can think of a list as a relationship between indices and elements. This relationship is called a mapping; each index "maps to" one of the elements.
- Lists are represented by boxes with the word "list" outside and the elements of the list inside. cheeses refers to a list with three elements indexed 0, 1 and 2.
- Numbers contains two elements; the diagram shows that the value of the second element has been reassigned from 123 to 5. empty refers to a list with no elements.
- List indices work the same way as string indices:
- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, we can get an index error
- If an index has a negative value, it counts backward from the end of the list.

In Operator

- The ‘in’ operator is used to check if a value exists in a sequence or not. Evaluates to true if it finds a variable in the specified sequence and false otherwise


The in operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

Traversing a list:

- The most common way to traverse the 0elements of a list is with a for loop. The syntax is the same as for strings.
  Eg: for cheese in cheeses: print(cheese)
- This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions range and len.
  for i in range(len(numbers)): numbers[i]=[i]*2
- This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to n- 1, where n is the length of the list.
- Each time through the loop i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.
- A for loop over an empty list never executes the body.

  Eg: for x in []:

  print(“this never happens”)
- Although a list can contain another list, the nested list still counts as a single element. The length of this list is four.

  [‘spam’,1,[‘sam’,’hari’,’robert’],[1,2,3]]

Program-1

```
cheeses=['cheddar','Edam','Gouda']
for cheese in cheeses:
print(cheese)
```

Output

```
cheddar Edam Gouda
```

Program-2

```
numbers=[12,13,14,15]
for i in range(len(numbers)): numbers[i]=[i]*2
print(numbers)
```

Output

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

List Operations

- The most conventional method to perform the list concatenation, the use of "+" operator can easily add the whole of one list behind the other list and hence perform the concatenation.

Program-1

```
test_list3 = [1, 4, 5, 6, 5]
test_list4 = [3, 5, 7, 2, 5]
# using + operator to concat test_list3 = test_list3 + test_list4
# Printing concatenated list
print ("Concatenated list using + : " + str(test_list3))
```

Output

```
Concatenated list using + : [1, 4, 5, 6, 5, 3, 5, 7, 2, 5]
```

- Sequences datatypes (both mutable and immutable) support a **repetition** operator *. The repetition operator * will make multiple copies of that particular object and combines them together. When * is used with an integer it performs multiplication but with list, tuple or strings it performs a repetition

Program-2

```
l1= [1,2,3]
print(l1*3)
```

Output

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

Nested List

- A list can contain any sort object, even another list (sublist), which in turn can contain sublists themselves, and so on. This is known as nested list.
- A nested list is created by placing a comma-separated sequence of sublists.
- **Eg:** L = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', 'h']
- Access individual items in a nested list using multiple indexes.
- Access a nested list by negative indexing as well.
- Negative indexes count backward from the end of the list. So, L[-1] refers to the last item, L[-2] is the second-last, and so on.

Program-1

```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']

 print(L[2])
print(L[2][2])
print(L[2][2][0])
```

Output

```
['cc', 'dd', ['eee', 'fff']] ['eee', 'fff']
eee
```

Program-2

```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h'] print(L[-3])
print(L[-3][-1])
print(L[-3][-1][-2])
```

Output

```
['cc', 'dd', ['eee', 'fff']] ['eee', 'fff']
eee
```

List Slices

- To access a range of items in a list, you need to slice a list
- The way to do this is to use the simple slicing operator : With this operator you can

specify where to start the slicing, where to end and specify the step.

- If L is a list, the expression L [start : stop : step] returns the portion of the list from index start to index stop, at a step size step. **Syntax:** L[start:stop:step]
- Where start means starting position, stop means ending position and step means increment or decrement.
- Since lists are mutable, it is often useful to make a copy before performing operations thatfold, spindle or mutilate lists.
- A slice operator on the left side of an assignment can update multiple elements.

Program-1

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[2:7])
```

Output

```
['c', 'd', 'e', 'f', 'g']
```

Program-2

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[-7:-2])
```

Output

```
['c', 'd', 'e', 'f', 'g']
```

Program-3

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[2:-5])
```

Output

```
['c', 'd']
```

Program-4

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[2:7:2])
```

Output

```
['c', 'e', 'g']
```

Program-5

```
L = ['a', 'b', 'c', 'd', 'e'] L[1:4] = [1, 2, 3]
print(L)
```

Output

```
['a', 1, 2, 3, 'e']
```

List Methods

1. count()

The count() method returns the number of times the specified element appears in the list.

Syntax: list.count(element)

Program

```
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
count = vowels.
count('i')
print('The count of i is:', count)
count = vowels.count('p')
print('The count of p is:', count)
```

Output

```
The count of i is: 2 The count of p is: 0
```

2. index()

The index() method returns the index of the specified element in the list.
Syntax: list.index(element, start, end)
If the element is not found, a ValueError exception is raised.

Program

```
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
index = vowels.index('e')
print('The index of e:', index) index = vowels.index('i')
print('The index of i:', index)
```

Output

```
The index of e: 1 The index of i: 2
```

3. remove()

This method is used to remove specified item from the list. If the item present multiple times then only first occurrence will be removed.

Syntax: list.remove(element)

If the specified item not present in list then we will get ValueError

Program

```
n=[10,20,10,30]
n.remove(10) print(n)
```

Output

```
[20, 10, 30]
```

4. pop()

It removes and returns the last element of the list. This is only function which manipulates list and returns some element

Syntax: list.pop(index)

If the list is empty then pop() raises IndexError
If the list is out of range when doing pop() raises IndexError

Program

```
n=[10,20,10,30]
n.pop(2)
n
n.pop()
```

Output

```
10
[10,20,30]
30
```

5. reverse()

The reverse() method reverses the elements of the list.

Syntax: list.reverse()

Program

```
systems = ['Windows', 'macOS', 'Linux']
print('Original List:', systems)
systems.reverse()
print('Updated List:', systems)
```

Output

```
Original List: ['Windows', 'macOS', 'Linux'] Updated List: ['Linux', 'macOS', 'Windows']
```

6. sort()

In list by default insertion order is preserved. If want to sort the elements of list according to default natural sorting order then we should go for sort() method.

Syntax: list.sort(key=..., reverse=...)

For numbers ==>default natural sorting order is Ascending Order

For Strings ==> default natural sorting order is Alphabetical Order

To use sort() function, compulsory list should contain only homogeneous elements. otherwise we will get TypeError

Program

```
n=[20,5,15,10,0]
n.sort()
n
s=["Dog","Banana","Cat","Apple"]
s.sort()
s n=[20,10,"A","B"]
n.sort() n=[20,5,15,10,0]
n.sort(reverse=True)
n
```

Output

```
[0, 5, 10, 15, 20]
['Apple', 'Banana', 'Cat', 'Dog'] Traceback (most recent call last):
File "<pyshell#7>", line 1, in <module> n.sort()
TypeError: '<' not supported between instances of 'str' and 'int' [20, 15, 10, 5, 0]
```

7. clear()

The clear() method removes all items from the list.
Syntax:list.clear()

Program

```
n=[20,5,15,10,0]
n.clear()
 n
```

Output

```
[ ]
```

8. append()

The append() method adds an item to the end of the list.

Syntax:list.append(item)

Program

```
animals = ['cat', 'dog', 'rabbit']
animals.append('guinea pig')
print('Updated animals list: ', animals)
```

Output

```
Updated animals list: ['cat', 'dog', 'rabbit', 'guinea pig']
```

9. extend()

The extend() method adds all the elements of an iterable (list, tuple, string etc.) to the end of the list.

Syntax: list.extend(iterable)

Program

```
languages = ['French', 'English']
languages1 = ['Spanish', 'Portuguese']
languages.extend(languages1)
print('Languages List:', languages)
```

Output

```
Languages List: ['French', 'English', 'Spanish', 'Portuguese']
```

Map function

The map() function applies a given function to each item of an iterable (list, tuple etc.) and returns a list of the results.

Syntax: map(function, iterable, ...)

The map() function applies a given to function to each item of an iterable and returns a list of the results.

The returned value from map() (map object) can then be passed to functions like list() (to create a list), set() (to create a set) and so on.

Program

```
def myfunc(a, b):
return a+b
x = map(myfunc, ('apple', 'banana', 'cherry'), ('orange', 'lemon', 'pineapple'))
#convert the map into a list, for readability:
print(list(x))
```

Output

```
['appleorange', 'bananalemon', 'cherrypineapple']
```

Reduce function

- An operation like that combines a sequence of elements into a single value is sometimes called reduces.

  Syntax: reduce(function,sequence[,initial])
- The reduce() function accepts a function and a sequence and returns a single value calculated as follows:
- Initially, the function is called with the first two items from the sequence and the result is returned.

- The function is then called again with the result obtained in step 1 and the next value in the sequence.
- This process keeps repeating until there are items in the sequence.
- In Python2, reduce() was a built-in function. However, in Python 3, it is moved to functools module.

Program

```
def do_sum(x1, x2): return x1 + x2
def my_reduce(func, seq): first = seq[0]
for i in seq[1:]:
first = func(first, i) return first
result=my_reduce(do_sum, [1, 2, 3, 4]) print(result)
```

Output

```
10
```

Filter Functions

- The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.
  Syntax: filter(function, iterable(s))
- filter() method returns an iterator that passed the function check for each element in the iterable.
- filter() method is equivalent to:
- # when function is defined (element for element in iterable if function(element))
- # when function is None (element for element in iterable if element)

Program

```
letters = ['a', 'b', 'd', 'e', 'i', 'j', 'o'] def filterVowels(letter):
vowels = ['a', 'e', 'i', 'o', 'u'] if(letter in vowels):
return True
else:
return False
filteredVowels = filter(filterVowels, letters) print('The filtered vowels are:')
for vowel in filteredVowels: print(vowel)
```

Output

```
The filtered vowels are:
a e i o
```

Deleting Elements

There are several ways to delete elements from a list. The deleting elements of the methods are are pop, remove and del keyword.
The deleting elements(pop, remove) methods are discussed in list methods concept are in above.

➢ **del keyword**

The del keyword is used to delete objects. In Python everything is an object, so the del keyword can also be used to delete variables, lists, or parts of a list etc.
Syntax: del obj_name

Program-1

```
x= ["apple", "banana", "cherry"] del x[0]
print(x)
```

Output

```
['banana', 'cherry']
```

Program-2

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

Output

```
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module> x
NameError: name 'x' is not defined
```

List and Strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, we can use list.

Eg: s='spam'
t=list(s) print(t)

Output: ['s','p','a','m']

- Because list is the name of a built-in function, you should avoid using it as a variable name. I also avoid l because it looks too much like 1. So that’s why I use t.
- The list function breaks a string into individual letters. If you want to break a string into words, we can use the split method.

  Eg: s=“learning python is easy” print(s.split())

  Output: [‘learning’,’python’, ‘is’, ‘easy’]
- An optional argument called a delimiter specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter.

  Eg: s=‘spam-spam-spam’ delimiter=‘-’ s.split(delimiter)

  Output: ['spam', 'spam', 'spam']
- join is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter.

  Eg: t=[‘learning’,’python’,’is’,’very’,’easy’] delimiter=‘’

  delimiter.join(t)

  Output: 'learning-python-is-very-easy'

Objects and Values

- If we execute these assignment statements: a=‘banana’

  b=‘banana’
- We know that a and b both refer to a string, but we don’t know whether they refer to the same string.
- There are two possible states.
- In one case, a and b refer to two different objects that have the same value.
- In the second case, they refer to the same object. To check whether two variables refer to the same object, you can use the is operator
- In this example, Python only created one string object, and both a and b refer to it.

  Eg: a=‘banana’

  b=‘banana’ a is b

Output: True

But when you create two lists, you get two objects

Eg: A=[1,2,3]
B=[1,2,3]
A is B
Output: False

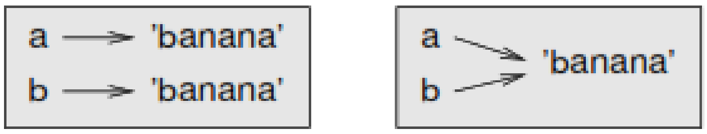
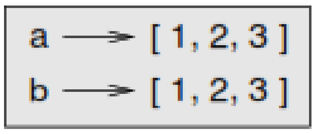


Figure 10.2: State diagram.



- In this case we would say that the two lists are equivalent, because they have the same elements, but not identical, because they are not the same object.
- If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.
- Until now, we have been using “object” and “value” interchangeably, but it is more precise to say that an object has a value.
- If you execute [1,2,3], we get a list object whose value is a sequence of integers. If another list has the same elements, we say it has the same value, but it is not the same object.

Aliasing

If a refers to an object and you assign b=a, then both variables refer to the same object.

Eg: a=[1,2,3]
b=a
b is a

Output: True
The association of a variable with an object is called a reference. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is aliased.

If the aliased object is mutable, changes made with one alias affect the other.

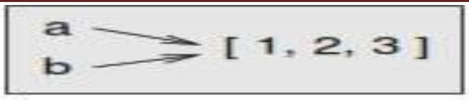
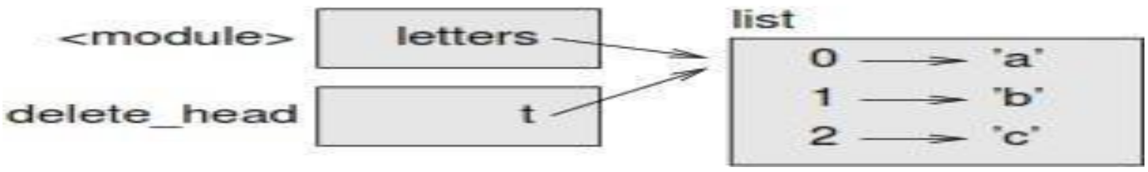


Figure 10.4: State diagram.



```
b[0]=17
print(a)
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.
For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
a=“banana”
b=“banana”
```

It almost never makes a difference whether a and b refer to the same string or not.

List Arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, delete_head removes the first element from a list.

Program

```
def  delete_head(t):
del t[0] letters=[‘a’,’b’,’c’] delete_head(letters)
print(letters)
```

Output

```
['b', 'c']
```

- The parameter t and the variable letters are aliases for the same object.
- Since the list is shared by two frames, I drew it between them.
- It is important to distinguish between operations that modify lists and operations that create new lists.

For example, the append method modifies a list, but the + operator creates a new list.

Program

```
t=[1,2]
t1=t.append(3)
print(t)
t2=t+[4]
print(t2)
```

Output

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

This difference is important when you write functions that are supposed to modify lists. For example, this function does not delete the head of a list.

Eg: def bad_delete_head(t):

t=t[1:] #wrong

The slice operator creates a new list and the assignment makes t refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, tail returns all but the first element of a list:

Eg: def tail(t):

return t[1:]

This function leaves the original list unmodified. Here's how it is used.

Program

```
def tail(t):
return t[1:]
letters=['a','b','c']
rest=tail(letters)
print(rest)
```

Output

```
['b', 'c']
```

Cloning Lists:

If we want to modify a list and also keep a copy of the original list. Then we should create a separate copy of the list. This process is called Cloning.

The Slice Operation is used to clone a list.

Example:

```
list1=[1,2,3,4,5]

list2=list1

print("List 1 as:",list1)

print("List 2 as:",list2)

list3=list1[2:5]

print("List 3 as:",list3)
```

Output:

```
List 1 as: [1, 2, 3, 4, 5]

List 2 as: [1, 2, 3, 4, 5]

List 3 as: [3, 4, 5]
```

Tuples are immutable:

- A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable.
- Syntactically, a tuple is a comma-separated list of values.

Eg

```
>>> t='a','b','c','d','e'
>>> t
('a', 'b', 'c', 'd', 'e')
```

Although it is not necessary, it is common to enclose tuples in parentheses.

Eg

```
>>> t=('a','b','c','d','e')
>>> t
('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include a final comma

Eg

```
>>> t='a',
>>> t ('a',)
>>> type(t)
<class 'tuple'>
```

A value in parentheses is not a tuple:

Eg

```
>>>t1=('a')
>>>type(t1)
<class 'str'>
```

Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple.

Eg-1

```
>>> t=tuple()
>>> print(t) ()
```

Eg-2

```
>>> a=1,2,3,4,5
>>> t=tuple(a)
>>> t
(1, 2, 3, 4, 5)
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence.

Eg

```
>>> t=tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

Because tuple is the name of a built-in function, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator indexes an element.

Eg-1

```
>>> t=('a','b','c','d','e')
>>> t[0]
'a'
>>> t[5]
Traceback (most recent call last):
File "<pyshell#25>", line 1, in <module>
t[5]
IndexError: tuple index out of range
>>> t[4]
'e'
```

And the slice operator selects a range of elements.

Eg

```
>>> t=('a','b','c','d','e')
>>> t[1:3] ('b', 'c')
>>> t[-1:-3] ()
>>> t[1:7] ('b', 'c', 'd', 'e')
>>> t[-3:-1]('c', 'd')
```

But if you try to modify one of the elements of the tuple, you get an error.

Eg

```
>>> t=('a','b','c','d','e')
>>> t[0]='A'
Traceback (most recent call last):
File "<pyshell#34>", line 1, in <module> t[0]='A'
TypeError: 'tuple' object does not support item assignment
```

We can't modify the elements of a tuple, but you can replace one tuple with another.

Eg

```
x = ("apple", "banana", "cherry")
 y = list(x)
y[1] = "kiwi" x = tuple(y) print(x)
```

Output

```
("apple", "kiwi", "cherry")
```

Tuple Assignment

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap a and b.

Eg

```
temp=a
a=b
b=temp
```

This solution is cumbersome; tuple assignment is more elegant.

Eg

```
a,b=b,a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right have to be thesame.

Eg

```
a,b=1,2,3
```

Output

```
Value Error: to many values to unpack
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, we could write

Eg

```
addr='monthypthon.org' Uname,domain=addr.split('@')
```

The return value from split is a list with two elements; the first element is assigned to uname, the second to domain.

Eg

```
print(uname) print(domain)
```

Output

```
monty python.org
```

Tuple As A Return Value

- Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute x/y and then x%y.

- It is better to compute them both at thesame time.

- The built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple.

Eg

```
t=divmod(7,3) print(t)
```

Output

```
(2,1)
```

Or use tuple assignment to store the elements separately

Eg

```
quot, rem=divmod(7,3) Print(quot)
Print(rem)
```

Output

```
2
```

```
def min_max(t):
return min(t), max(t)
t=divmod(7,3)
print(min_max(t))
```

Output

```
(1, 2)
```

min and max are built-in functions that find the largest and smallest elements of a sequence.

min-max computes both and returns a tuple of twovalues.

UNIT-V

UNIT-V DICTIONARIES AND FILES

Dictionaries and Files: Dictionaries, Operations on Dictionaries, Dictionary methods, Difference between List, Tuples and Dictionaries, Introduction to Files, File Types, Exception-Error handling.

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":"ASCET","model":"college","year":2004}
>>> dict1
```

Output:

```
{'brand': 'ASCET', '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.
fromkeys(seq[, v])	Return a new dictionary with keys from seq and value equal to v (defaults to None).
get(key[,d])	Return the value of key. If key doesnot exit, return d (defaults to None).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.

pop(key[,d])	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.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	If key is in the dictionary, return its value. If not, insert key with a value of d and return d(defaults to None)
update([other])	Update the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Return a new view of the dictionary's values

Dictionary operations:

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

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

Output:

```
'ASCET'
```

To access keys and values and items of dictionary:

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

Output:

```
ASCET
college 2004
```

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

Output:

```
brand model year
```

Program:

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

Output:

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

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

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

Ouput:

```
college
```

```
>>> dict1{'brand': 'ASCET', '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:

Program

```
>>> z={x: x**2 for x in (2,4,6)}
>>> z
```

Output:

```
{2: 4, 4: 16, 6: 36}
```

Program

```
>>> dict11 = {x: x*x for x in range(6)}
>>> dict11
```

Output:

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Difference between List, Tuples and Dictionaries

List	Tuple	Dictionary
List is a non-homogeneous data structure which stores the elements in single row and multiple rows and columns	Tuple is also a non-homogeneous data structure which stores single row and multiple rows and columns	Dictionary is also a non-homogeneous data structure which stores key value pairs
List can be represented by [ ]	Tuple can be represented by ( )	Dictionary can be represented by { }
List allows duplicate elements	Tuple allows duplicate elements	Set will not allow duplicate elements but keys are not duplicated
List can use nested among all Example: [1, 2, 3, 4, 5]	Tuple can use nested among all Example: (1, 2, 3, 4, 5)	Dictonary can use nested among all Example: {1, 2, 3, 4, 5}
List can be created using **list()** function	Tuple can be created using **tuple()** function.	Dictonary can be created using **dict()** function.
List is mutable i.e we can make any changes in list.	Tuple is immutable i.e we can not make any changes in tuple	Dictionary is mutable. But Keys are not duplicated.
List is ordered	Tuple is ordered	Dictionary is ordered
Creating an empty list l=[]	Creating an empty Tuple t=()	Creating an empty dictionary d={}
Used in JSON format Useful for Array operations Used in Databases	Used to insert records in the database through SQL query at a time Ex:(1.'sravan', 34).(2.'geek', 35) Used in parentheses checker	Used to create a data frame with lists Used in JSON

Introduction to Files:

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 bydefault.

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.

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+")

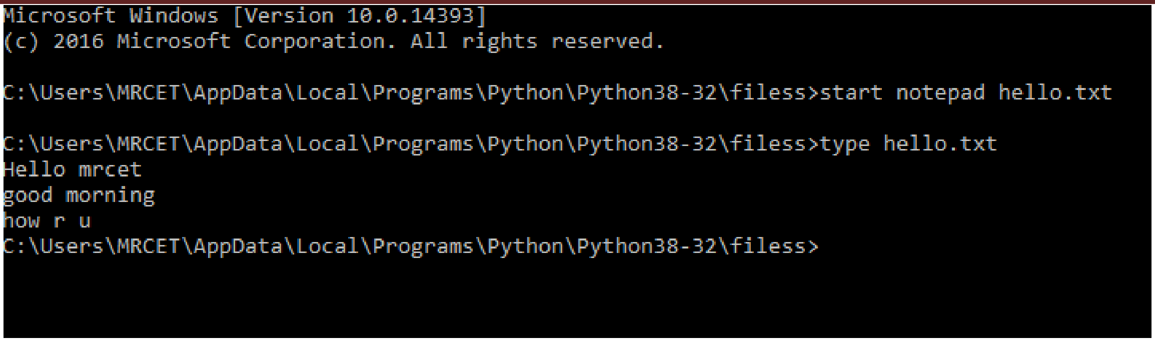
- We declared the variable f to open a file named hello.txt. **Open** takes 2 arguments, the file that we want to open and a string that represents the kinds of permission or operation we want to do on the file
- Here we used "w" letter in our argument, which indicates write and the plus sign that means 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)

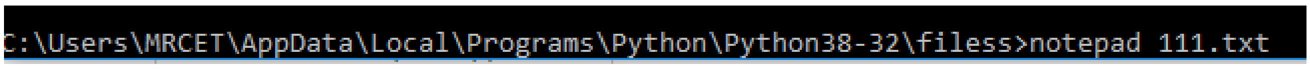
Reading and Writing files:

The following image shows how to create and open a text file in notepad from command prompt



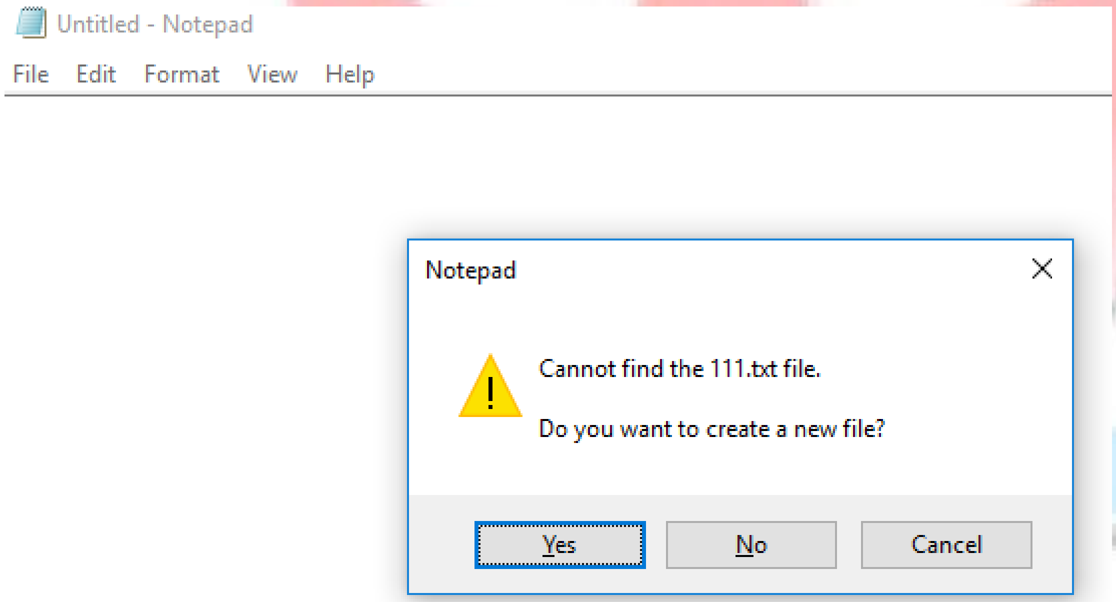
```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

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?



Cannot find the 111.txt file.

Do you want to create a new file?

Click on "yes" to open else "no" to cancel

# Write a python program to open and read a file

a=open("one.txt","r")

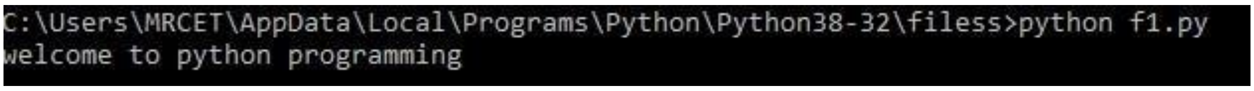
print(a.read())

a.close()

Output:

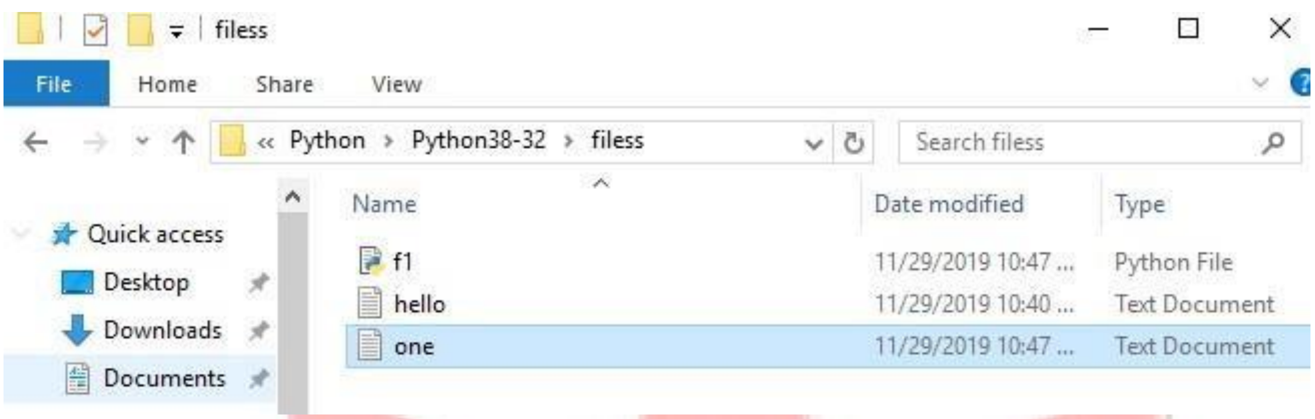
C:/Users/ASCET/AppData/Local/Programs/Python/Python38-32/filess/f1.py welcome to python programming

(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 saved together in a particular file then only the program performs the operations in the given file mode

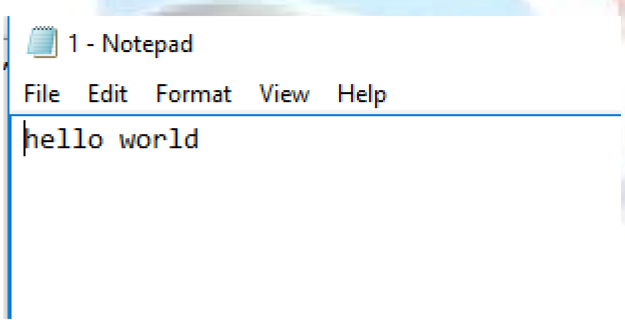


```
filess
Python > Python38-32 > filess
Name        Date modified         Type
f1          11/29/2019 10:47 ...  Python File
hello       11/29/2019 10:40 ...  Text Document
one         11/29/2019 10:47 ...  Text Document
```

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?

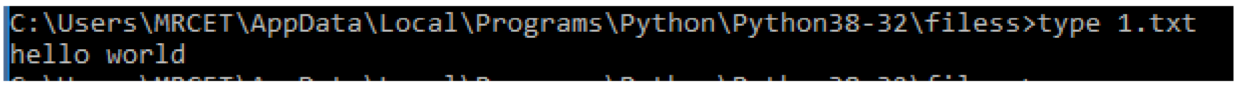
```
f=open("1.txt","a")
f.write("hello world")
f.close()
```

Output:



```
1 - Notepad
hello world
```

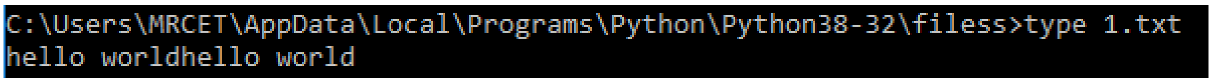
(or)



```
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
```

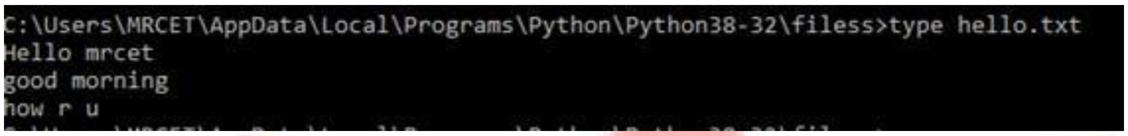
# Write a python program to write the content “hi python programming” for the existing file.

f=open("1.txt",'w')

f.write("hi python programming")

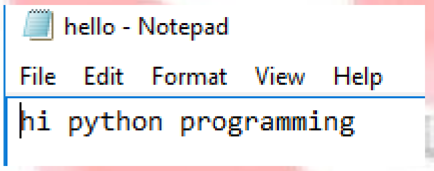
f.close()

Output:



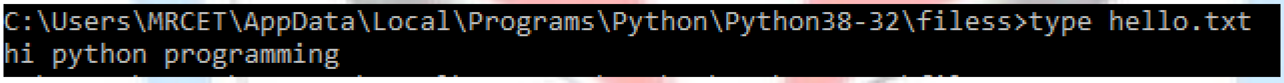
```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type hello.txt
Hello mrcet
good morning
how r u
```

In the above program the hello txt file consist of data like



```
hi python programming
```

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.

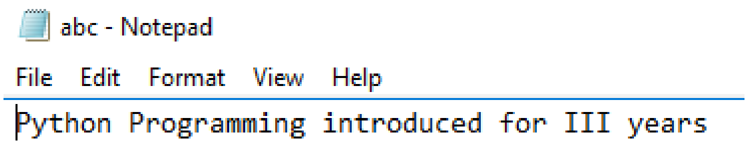
fo=open("abc.txt","w+")

fo.write("Python Programming")

print(fo.read())

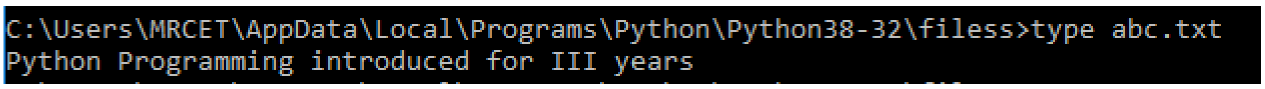
fo.close()

Output:



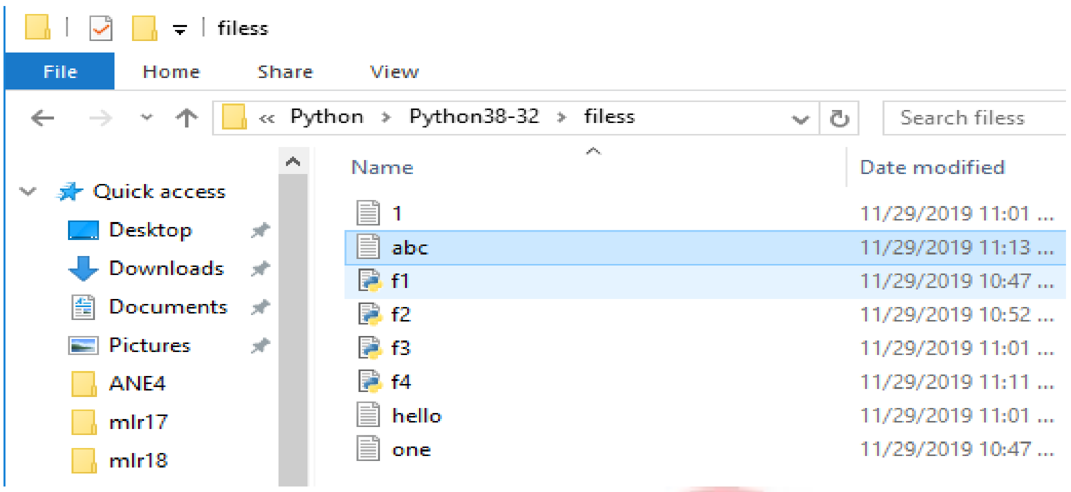
```
Python Programming introduced for III years
```

(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



Name	Date modified
1	11/29/2019 11:01 ...
abc	11/29/2019 11:13 ...
f1	11/29/2019 10:47 ...
f2	11/29/2019 10:52 ...
f3	11/29/2019 11:01 ...
f4	11/29/2019 11:11 ...
hello	11/29/2019 11:01 ...
one	11/29/2019 10:47 ...

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 consistentindentation. 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:

- ArrayIndexOutOfBoundException.
- 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:

- You should be able to understand the mistakes; the error might be done by user, DB connection or server.
- Whenever there is an error execution should not stop.
  Ex: Banking Transaction

1. The aim is execution should not stop even though an error occurs.

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/ASCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex3.py number can not be
divided by zero
bye
```

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/ASCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex5.py

number can not be divided by zero **division by zero**
bye

$\downarrow$

(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)
```