

**SIDDARTHA INSTITUTE OF SCIENCE AND
TECHNOLOGY: PUTTUR**
(AUTONOMOUS)
II B.Tech. – I Sem.

(20CS0511) PYTHON PROGRAMMING

COURSE OBJECTIVES

The objectives of this course:

- *Introduce Scripting Language*
- *Exposure to various problem solving approaches of computer science*
- *Introduce function-oriented programming paradigm*
- *Exposure to solve the problems using object oriented concepts, exceptional handling*
- *Exposure to solve the problems using Files, Regular Expressions and, Standard Libraries*

COURSE OUTCOMES

- On successful completion of this course, the student will be able to
- *Solve the problems using control structures, input and output statements.*
- *Summarize the features of lists, tuples, dictionaries, strings and files*
- *Experience the usage of standard libraries, objects, and modules*
- *Solve the problems using Object Oriented Programming Concepts*
- *Build the software for real time applications using python*
- *Install various Python packages*

UNIT – I

- **Introduction:** History of Python- Python features- Applications-Programming Using the REPL-Running Python Scripts-Variables – Assignment- Keywords- Input-Output- Indentation.
- **Data Types:** Single-Value data types - int, float, Complex and Boolean.
- Multi-Valued Data types - Lists, Tuples, Sets, Dictionaries, Strings- indexing and slicing.

Python Programming

- **Python History and Versions**
- Python laid its foundation in the late 1980s.
- The implementation of Python was started in December 1989 by **Guido Van Rossum** at CWI in Netherland.
- In February 1991, **Guido Van Rossum** published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.

- Python 2.0 added new features such as list comprehensions, garbage collection systems.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language, which was capable of Exception Handling and interfacing with the Amoeba Operating System.

- **Python Version List**
- Python programming language is being updated regularly with new features and supports. There are lots of update in Python versions, started from 1994 to current release.
- A list of Python versions with its released date is given below.

Python Version

Python 1.0

Python 1.5

Python 1.6

Python 2.0

Python 2.1

Python 2.2

Python 2.3

Python 2.4

Released Date

January 1994

December 31, 1997

September 5, 2000

October 16, 2000

April 17, 2001

December 21, 2001

July 29, 2003

November 30, 2004

Python Version

Python 2.5

Python 2.6

Python 2.7

Python 3.0

Python 3.1

Python 3.2

Released Date

September 19, 2006

October 1, 2008

July 3, 2010

December 3, 2008

June 27, 2009

February 20, 2011

Python Version

Python 3.3

Python 3.4

Python 3.5

Python 3.6

Python 3.7

Python 3.8

Released Date

September 29, 2012

March 16, 2014

September 13, 2015

December 23, 2016

June 27, 2018

October 14, 2019

- **Python Features**

Easy to Learn and Use

Its syntax is straightforward and much the same as the English language. There is no semicolon or curly-bracket.

Expressive Language

Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type **print("Hello World")**.

Interpreted Language

Python is an interpreted language; it means the Python program is executed one line at a time.

Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc.

Free and Open Source

Python is freely available for everyone. It is freely available on its official website www.python.org

Object-Oriented Language

Python supports object-oriented concepts of classes and objects. It supports inheritance, polymorphism, and encapsulation.

Large Standard Library

It provides a vast range of libraries for the various fields such as machine learning, web developer, and also for the scripting.

There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras, and Pytorch, etc. Django, flask, pyramids.

Applications of Python

Python is a general-purpose, popular programming language and it is used in almost every technical field. The various areas of Python use are given below.

Data Science

Date Mining

Desktop Applications

Console-based Applications

Mobile Applications

Software Development

Artificial Intelligence
Web Applications
Enterprise Applications
3D CAD Applications
Machine Learning
Computer Vision or Image Processing Applications.
Speech Recognitions

REPL is the language shell, the **Python Interactive Shell**. The REPL acronym is short for Read, Eval, Print and Loop.

The Python interactive interpreter can be used to easily check Python commands. To start the Python interpreter, type the command `python` without any parameter and hit the “return” key.

The process is:

Read: take user input.

Eval: evaluate the input.

Print: shows the output to the user.

Loop: repeat.

To start the Python language shell (the interactive shell), first open a terminal or command prompt. Then type the command `python` and press enter.

Python then outputs some information like this (including the Python version):

```
$ python
```

```
Python 3.7.5 (default, Nov 20 2019, 09:21:52)  
[GCC 9.2.1 20191008] on linux
```

```
Type "help", "copyright", "credits" or "license" for  
more information.
```

```
>>> 128 / 8
```

```
16.0
```

```
>>> 256 * 4
```

```
1024
```

```
>>>
```

to quit the Python interactive shell (REPL).

- One way is by typing `exit()`.
- `>>> exit()`

- A keyboard shortcut can be used to exit the shell too: `Ctrl-D`.

Running Python Scripts

Python interpreter is responsible for executing the Python scripts.

Various ways to run Python scripts.

The operating system command-line or terminal.

The Python interactive mode.

The IDE or Text editor

The file manager of system.

 sample - Notepad

File Edit Format View Help

```
# Python run script example
```

```
print("Welcome to JavaTpoint")
```

- **Using the Python command line**
- Open the command line to run a Python script. We need to type the *python*, followed by the file name to execute the file. Now, hit the enter key, and if there is no the error in file, we will see the output as follows.

C:\WINDOWS\system32\cmd.exe

- □ X

```
C:\Users\DEVANSH SHARMA\Desktop>python sample.py
Welcome to JavaTpoint
```

```
C:\Users\DEVANSH SHARMA\Desktop>■
```

- **The IDE or Text Editor**
- The IDE stands for Integrated Development Environment. There are various IDEs but **Pycharm** is Python's most popular and useful text editor among them. It is recommended for developing large and more complex applications. Here are using the **Pycharm** to run python script.
- Create a new project and then create a new Python file using the .py extension.

The screenshot shows the PyCharm IDE interface. The top menu bar includes File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, and Help. The title bar indicates the project is "Practice Python" and the file is "first.py". The toolbar on the right has icons for saving, running, and other operations. The left sidebar is titled "Project" and lists "Practice Python" (C:\Users\DEVANSH SHARMA\PycharmProjects\Practice Python) with a "first.py" file selected. Below it are "External Libraries" and "Scratches and Consoles". The main code editor window shows two lines of Python code:

```
# Using Pycharm to run code
print("Hello JavaTpoint")
```

Variables

Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.

In Python, we don't need to specify the type of variable

Variable names can be a group of both the letters and digits, but they have to begin with a letter or an underscore.

Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. **The rules to name an identifier are given below.**

The first character of the variable must be an alphabet or underscore (_).

All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore, or digit (0-9).

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

Identifier name must not be similar to any keyword defined in the language.

Identifier names are case sensitive; for example, my name, and MyName is not the same.

Examples of valid identifiers: a123, _n, n_9, etc.

Examples of invalid identifiers: 1a, n%4, n 9, etc.

- **Declaring Variable and Assigning Values**
- Python allows us to create a variable at the required time.
- When we assign any value to the variable, that variable is declared automatically.
- The equal (=) operator is used to assign value to a variable.
- Ex: `a = 50`



- In the above image, the variable **a** refers to an integer object.

- Suppose we assign values as follows
- $a = 50$
- $b = a$



- $a = 50$
- $b = 100$



Object Identity

In Python, every created object identifies uniquely in Python.
The built-in **id()** function, is used to identify the object identifier.

Ex:

```
a = 50  
b = a  
print(id(a))  
print(id(b))  
# Reassigned variable a
```

```
a = 500  
print(id(a))
```

Output:

```
140734982691168  
140734982691168  
2822056960944
```

Multiple Assignments

1. Assigning multiple values to multiple variables

We can assign the more than one variable simultaneously on the same line. For example -

```
a, b = 5, 4
```

```
print(a,b)
```

Output:

```
5 4
```

Values are printed in the given order.

2. Assign a single value to the multiple variables

We can assign the single value to the multiple variables on the same line. Consider the following example.

Example

```
a=b=c="JavaTpoint"
```

```
print(a)
```

```
print(b)
```

```
print(c)
```

Output:

```
JavaTpoint JavaTpoint JavaTpoint
```

Python Keywords

- Python Keywords are special reserved words that 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 a variable. Following is the List of Python Keywords.
- True False None and as assert
def class continue break else finally
elif del except global for if
from import raise try or return
pass nonlocal in not is lambda

- **True** - It represents the Boolean true, if the given condition is true, then it returns "True". Non-zero values are treated as true.
- **False** - It represents the Boolean false; if the given condition is false, then it returns "False". Zero value is treated as false
- **None** - It denotes the null value or void. An empty list or Zero can't be treated as **None**.
- **and** - It is a logical operator. It is used to check the multiple conditions. It returns true if both conditions are true. Consider the following truth table.

assert - This keyword is used as the debugging tool in Python. It checks the correctness of the code. It raises an **AssertionError** if found any error in the code and also prints the message with an error.

Example:

```
a = 10
```

```
b = 0
```

```
print('a is dividing by Zero')
```

```
assert b != 0
```

```
print(a / b)
```

Output:

```
a is dividing by Zero
Runtime Exception: Traceback (most recent call last):
File
```

```
"/home/40545678b342ce3b70beb1224bed345f.py", line 4,
in assert b != 0, "Divide by 0 error"
AssertionError: Divide
by 0 error
```

Python input() Function

Python input() function is used to get input from the user. It prompts for the user input and reads a line. After reading data, it converts it into a string and returns that. It throws an error EOFError if EOF is read.

Signature

`input ([prompt])`

Parameters

prompt: It is a string message which prompts for the user input.

Return

It returns user input after converting into a string.

Python input() Function Example 1

Here, we are using this function get user input and display to the user as well.

```
# Python input() function example
```

```
# Calling function
```

```
val = input("Enter a value: ")
```

```
# Displaying result
```

```
print("You entered:",val)
```

Output:

Enter a value: 45 You entered: 45

Python input() Function Example 2

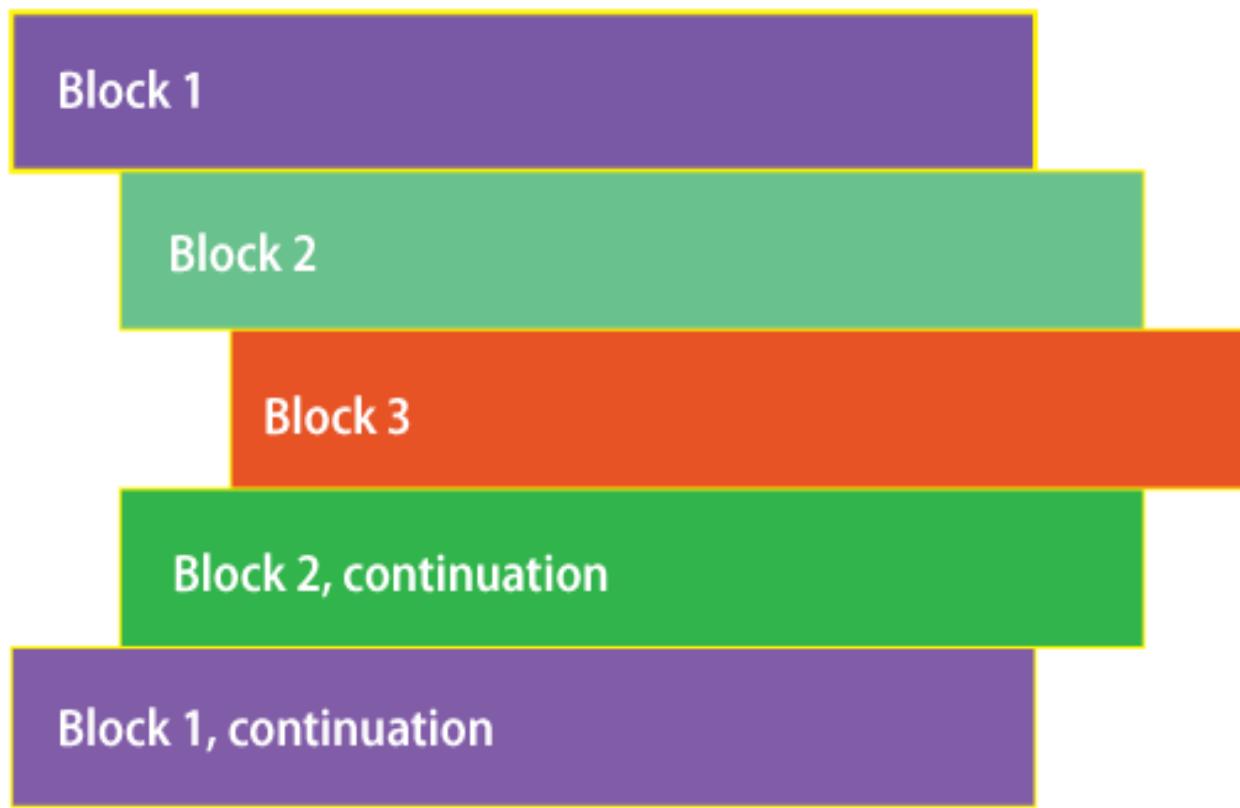
The `input()` method returns string value. So, if we want to perform arithmetic operations, we need to cast the value first. See the example below.

```
# Python input() function example
# Calling function
val = input("Enter an integer: ")
# Displaying result
val = int(val) # casting into string
sqr = (val*val) # getting square
print("Square of the value:",sqr)
```

Output:

Enter an integer: 12 Square of the value: 144

Indentation in Python Programming Language



- Indentation refers to implementing proper **spaces and tabs** at starting any statement, method and block of code in the python programming language.
- Due to these indentations, we can easily identify the beginning point and at the endpoint of any conditional loop, functions, if-else statements and so on.
- The role of all whitespaces matters; if a block of code is starting by applying a white space, it must end on the same indent.

- All statements must have the same number of whitespaces or tabs so that the distance to the left of the screen belongs to the same block of code. If a block has to be more deeply nested, it is indented further to the right.

```
Statement  
if condition  
  if condition:  
    Statement  
  else:  
    Statement  
Statement
```

How the
interpreter
visualises



```
Code block 1 begins  
code block 1 continues  
Code block 2 begins  
  Code block 3 begins  
  Code block 2 continues  
Code block 1 continues
```

- **Advantages of Indentation in Python**
- Indentation is used in python to represent a certain block of code, but in other programming languages, they refer to various brackets. Due to indentation, the code looks more efficient and beautifully structured.
- Indentation rules used in a python programming language are very simple; even a programmer wants their code to be readable.
- Also, indentation increases the efficiency and readability of the code by structuring it beautifully.

- **Disadvantages of Indentation in Python**
- Due to the uses of whitespaces in indentation, sometimes it is a very tough task to fix the indentation error when there are many lines of code.
- The various popular programming languages like C, C++, Java use braces for indentation, so anybody coming from the other side of the developed world finds it hard to adjust to the idea of using whitespaces for the indentation

```
# Python program to find the maximum out of two numbers:  
def max(x,y): # max function will return the maximum am  
ong the two numbers  
if(x>y):  
    return x  
else:  
    return y  
a = int(input("Enter a number: "))  
b = int(input("Enter another number: "))  
print("Finding the Maximum out of a:", a , "and b:", b)  
c=max(a,b) # calling the max function  
print(c,"is maximum") # printing the result
```

The output of the above program:

IndentationError: expected an indented block

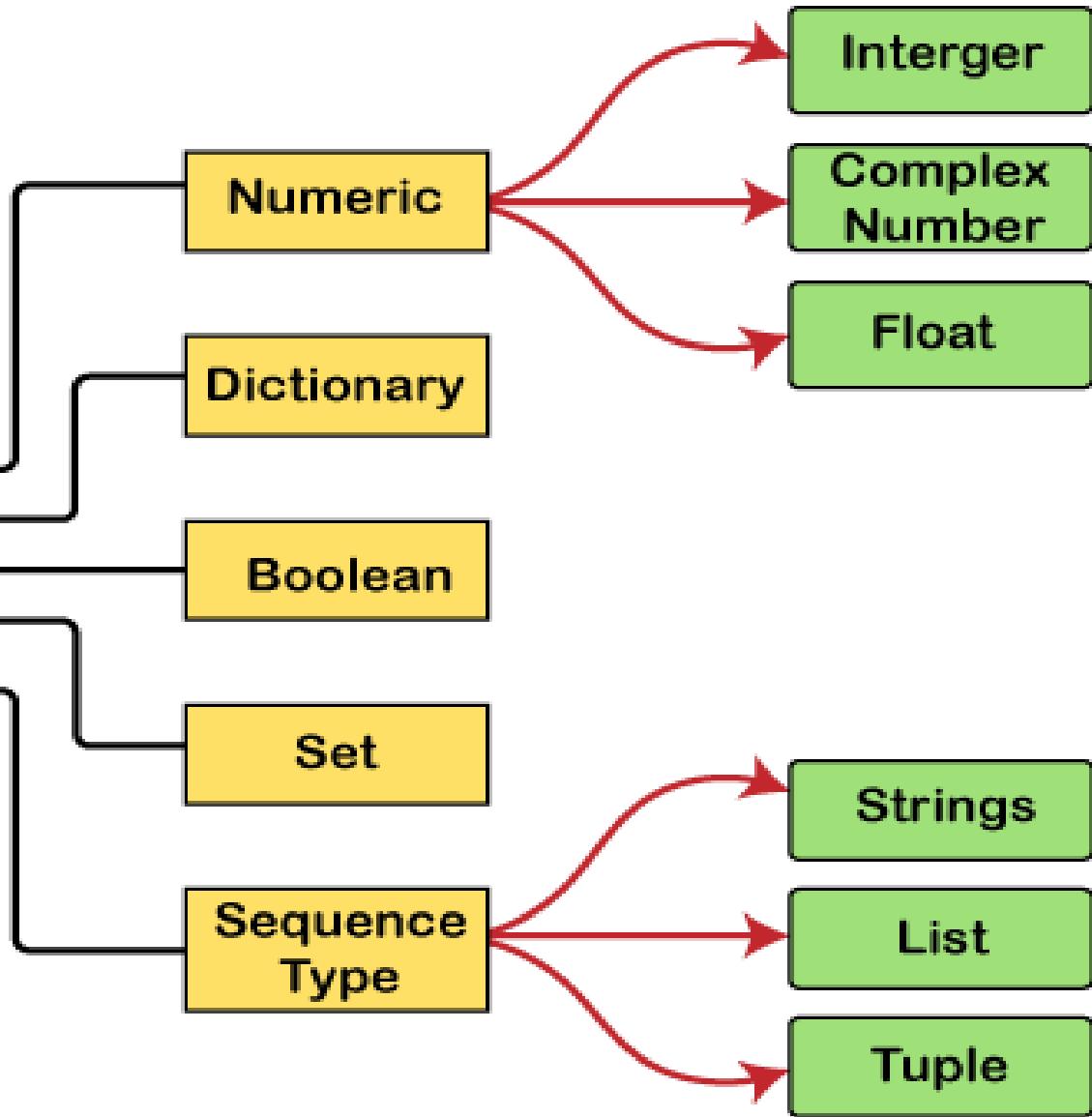
- Data Types:
 1. Single-Value data types - int, float, Complex and Boolean
 2. Multi-Valued Data - Lists, Tuples, Sets, Dictionaries

Standard data types

- A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

[Numbers](#) , [Sequence Type](#), [Boolean](#), [Set](#),[Dictionary](#)

Python - Data Types



Python has the following data types built-in by default, in these categories:

Text Type : str

Numeric Types : int, float, complex

Sequence Types : list, tuple, range

Mapping Type : dict

Set Types : set, frozenset

Boolean Type : bool

Binary Types : bytes, bytearray,
memoryview

None Type : NoneType

- **Integer data type:**
- a = 5
- The variable a holds integer value five. Python interpreter will automatically interpret variables a as an integer type.

a=10

b="Hi Python"

c = 10.5

print(type(a))

print(type(b))

print(type(c))

Output:

<type 'int'> <type 'str'> <type 'float'>

- **Numbers**
- Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data-type. Python provides the **type()** function to know the data-type of the variable.
- Similarly, the **isinstance()** function is used to check an object belongs to a particular class.
- Python creates Number objects when a number is assigned to a variable.

- For example;
- `a = 5`
- `print("The type of a", type(a))`
-
- `b = 40.5`
- `print("The type of b", type(b))`
-
- `c = 1+3j`
- `print("The type of c", type(c))`
- `print(" c is a complex number", isinstance(1+3j,complex))`
- **Output:**
- The type of a <class 'int'> The type of b <class 'float'> The type of c <class 'complex'> c is complex number: True

- **Int** - Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to **int**
- **Float** - Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.
- **complex** - A complex number contains an ordered pair, i.e., $x + iy$ where x and y denote the real and imaginary parts, respectively. The complex numbers like $2.14j$, $2.0 + 2.3j$, etc.

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple

x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview
x = None	NoneType

- **Python Lists**
- `mylist = ["apple", "banana", "cherry"]`
- **List**
- Lists are used to store multiple items in a single variable.
- Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are [Tuple](#), [Set](#), and [Dictionary](#), all with different qualities and usage.
- Lists are created using square brackets:
- **Example**
- Create a List:
- `thislist = ["apple", "banana", "cherry"]`
`print(thislist)`

- **List Items**
- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0], the second item has index [1] etc.
- **Ordered**
- When we say that lists are ordered, it means that the items have a defined order, and that order will not change.
- If you add new items to a list, the new items will be placed at the end of the list.
- **Note:** There are some [list methods](#) that will change the order, but in general: the order of the items will not change.

- **Changeable**
- The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.
- **Allow Duplicates**
- Since lists are indexed, lists can have items with the same value:
- **Example**
- Lists allow duplicate values:
- ```
thislist = ["apple", "banana", "cherry", "apple",
"cherry"]
print(thislist)
```

# List Length

To determine how many items a list has, use the `len()` function:

## Example

Print the number of items in the list:

```
thislist = ["apple", "banana", "cherry"]
```

```
print(len(thislist))
```

O/P 3

# Python Tuples

```
mytuple = ("apple", "banana", "cherry")
```

## Tuple

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are [List](#), [Set](#), and [Dictionary](#), all with different qualities and usage.

A tuple is a collection which is ordered and unchangeable.

Tuples are written with round brackets.

## Example

```
thistuple = ("apple", "banana", "cherry")
```

```
print(thistuple)
```

```
o/p ('apple', 'banana', 'cherry')
```

- **Tuple Items**
- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

- **Ordered**
- When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
- **Unchangeable**
- Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- **Allow Duplicates**
- Since tuples are indexed, they can have items with the same value:

## Example

Tuples allow duplicate values:

```
thistuple = ("apple", "banana", "cherry", "apple",
 "cherry")
```

```
print(thistuple)
```

O/P

```
('apple', 'banana', 'cherry', 'apple', 'cherry')
```

# **Tuple Items - Data Types**

Tuple items can be of any data type:

## **Example**

String, int and boolean data types:

```
tuple1 = ("apple", "banana", "cherry")
```

```
tuple2 = (1, 5, 7, 9, 3)
```

```
tuple3 = (True, False, False)
```

```
print(tuple1)
```

```
print(tuple2)
```

```
print(tuple3)
```

o/p ('apple', 'banana', 'cherry')

(1, 5, 7, 9, 3)

(True, False, False)

# Python Sets

```
myset = {"apple", "banana", "cherry"}
```

## Set

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are [List](#), [Tuple](#), and [Dictionary](#), all with different qualities and usage.

A set is a collection which is *unordered*, *unchangeable*\*, and *unindexed*.

\* **Note:** Set *items* are unchangeable, but you can remove items and add new items.

- **Set Items**
- Set items are unordered, unchangeable, and do not allow duplicate values.
- **Unordered**
- Unordered means that the items in a set do not have a defined order.
- Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

## **Unchangeable**

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

Once a set is created, you cannot change its items, but you can remove items and add new items.

## **Duplicates Not Allowed**

Sets cannot have two items with the same value.

## **Example**

Duplicate values will be ignored:

```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)
```

A set can contain different data types:

## Example

A set with strings, integers and boolean values:

```
set1 = {"abc", 34, True, 40, "male"}
```

```
print(set1)
```

o/p

```
{"abc", 34, True, 40, "male"}
```

# Python Dictionaries

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

- **Dictionary**
- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered\*, changeable and do not allow duplicates.

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

Dictionaries are written with curly brackets, and have keys and values:

## Example

Create and print a dictionary:

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

```
print(thisdict)
```

o/p {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

- **Ordered or Unordered?**
- As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.
- When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.
- Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.
- **Changeable**
- Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

# Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

## Example

Duplicate values will overwrite existing values:

```
thisdict = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964,
 "year": 2020
}

print(thisdict)
o/p {'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

## Strings

**Strings** in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:

### Example

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

### Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

### Example

```
a = "Hello"
print(a)
```

# Multiline Strings

You can assign a multiline string to a variable by using three quotes:

## Example

You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

# Slicing

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

## Example

Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"
print(b[2:5])
```

**Note:** The first character has index 0.

- **Slice From the Start**
- By leaving out the start index, the range will start at the first character:
- **Example**
- Get the characters from the start to position 5 (not included):
  - b = "Hello, World!"  
`print(b[:5])`
  - o/p Hello

## Slice To the End

By leaving out the *end* index, the range will go to the end:

### Example

Get the characters from position 2, and all the way to the end:

```
b = "Hello, World!"
```

```
 print(b[2:])
```

```
o/p Ilo, World!
```

# Negative Indexing

Use negative indexes to start the slice from the end of the string: **Example**

Get the characters:

From: "o" in "World!" (position -5)

To, but not included: "d" in "World!" (position -2):

```
b = "Hello, World!"
```

```
print(b[-5:-2])
```

o/p orl

**SIDDARTHA INSTITUTE OF SCIENCE AND  
TECHNOLOGY: PUTTUR**  
**(AUTONOMOUS)**  
**II B.Tech. – I Sem.**

**(20CS0511) PYTHON PROGRAMMING**

# COURSE OBJECTIVES

**The objectives of this course:**

- *Introduce Scripting Language*
- *Exposure to various problem solving approaches of computer science*
- *Introduce function-oriented programming paradigm*
- *Exposure to solve the problems using object oriented concepts, exceptional handling*
- *Exposure to solve the problems using Files, Regular Expressions and, Standard Libraries*

## COURSE OUTCOMES

- On successful completion of this course, the student will be able to
- *Solve the problems using control structures, input and output statements.*
- *Summarize the features of lists, tuples, dictionaries, strings and files*
- *Experience the usage of standard libraries, objects, and modules*
- *Solve the problems using Object Oriented Programming Concepts*
- *Build the software for real time applications using python*
- *Install various Python packages*

## **UNIT – II**

- Operators and Expressions: Operators-Arithmetic Operators, Comparison Operators, Assignment Operators, Logical Operators, Bitwise Operators, Membership Operators, Identity Operators- Expressions and order of evaluations
- Control Flow: Branching- simple if, if-else, if-elif-else, nested if, looping-while and for-jumping – break- continue and pass

# **Python Operators**

Operators are used to perform operations on variables and values.

## **Example**

```
print(10 + 5)
```

Python divides the operators in the following groups:

Arithmetic operators

Assignment operators

Comparison operators

Logical operators

Identity operators

Membership operators

Bitwise operators

# Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

| Operator | Name           | Example  |
|----------|----------------|----------|
| +        | Addition       | $x + y$  |
| -        | Subtraction    | $x - y$  |
| *        | Multiplication | $x * y$  |
| /        | Division       | $x / y$  |
| %        | Modulus        | $x \% y$ |
| **       | Exponentiation | $x ** y$ |
| //       | Floor division | $x // y$ |

x = 5

y = 3

print(x + y)

print(x - y)

print(x \* y)

print(x / y)

print(x % y)

print(x \*\* y)

print(x // y)

# Python Assignment Operators

Assignment operators are used to assign values to variables:

| Operator | Example | Same As   |
|----------|---------|-----------|
| =        | x = 5   | x = 5     |
| +=       | x += 3  | x = x + 3 |
| -=       | x -= 3  | x = x - 3 |
| *=       | x *= 3  | x = x * 3 |
| /=       | x /= 3  | x = x / 3 |

| Operator               | Example                    | Same As                       |
|------------------------|----------------------------|-------------------------------|
| <code>%=</code>        | <code>x %= 3</code>        | <code>x = x % 3</code>        |
| <code>//=</code>       | <code>x //= 3</code>       | <code>x = x // 3</code>       |
| <code>**= x</code>     | <code>**= 3</code>         | <code>x = x ** 3</code>       |
| <code>&amp;=</code>    | <code>x &amp;= 3</code>    | <code>x = x &amp; 3</code>    |
| <code> =</code>        | <code>x  = 3</code>        | <code>x = x   3</code>        |
| <code>^=</code>        | <code>x ^= 3</code>        | <code>x = x ^ 3</code>        |
| <code>&gt;&gt;=</code> | <code>x &gt;&gt;= 3</code> | <code>x = x &gt;&gt; 3</code> |
| <code>&lt;&lt;=</code> | <code>x &lt;&lt;= 3</code> | <code>x = x &lt;&lt; 3</code> |

Ex:1    x = 5                                 output

            print(x)                                 5

Ex:2    x = 5

            x += 3

            print(x)                                 8

Ex:3    x = 5

            x -= 3

            print(x)                                 2

Ex:4 x = 5 output

$$x^* = 3$$

`print(x)`

15

**Ex:5**    x = 5

$$x \neq 3$$

`print(x)`

1.6666666666666667

Ex:6 x = 5

$$x \% = 3$$

`print(x)`

2

Ex:7    x = 5                                                  output

    x \*\*= 3

    print(x)

125

Ex:8    x = 5

    x //= 3

    print(x)

1

Ex:9    x = 5

    x &= 3

    print(x)

1

Ex:10 x = 5

    x |= 3

    print(x)

7

Ex:11 x = 5

output

x ^= 3

print(x)

6

Ex:12 x = 5

x >>= 3

print(x)

0

Ex:13 x = 5

x <<= 3

print(x)

40

# Python Comparison Operators

- Comparison operators are used to compare two values

| Operator           | Name                     | Example                |
|--------------------|--------------------------|------------------------|
| <code>==</code>    | Equal                    | <code>x == y</code>    |
| <code>!=</code>    | Not equal                | <code>x != y</code>    |
| <code>&gt;</code>  | Greater than             | <code>x &gt; y</code>  |
| <code>&lt;</code>  | Less than                | <code>x &lt; y</code>  |
| <code>&gt;=</code> | Greater than or equal to | <code>x &gt;= y</code> |
| <code>&lt;=</code> | Less than or equal to    | <code>x &lt;= y</code> |

|               |        |
|---------------|--------|
| x = 5         | output |
| y = 3         |        |
| print(x == y) | false  |
| print(x != y) | true   |
| print(x > y)  | true   |
| print(x < y)  | false  |
| print(x >= y) | true   |
| print(x <= y) | false  |

- **Python Logical Operators**
- Logical operators are used to combine conditional statements:

| Operator | Description                                             | Example                                  |
|----------|---------------------------------------------------------|------------------------------------------|
| and      | Returns True if both statements are true                | $x < 5$ and $x < 10$                     |
| or       | Returns True if one of the statements is true           | $x < 5$ or $x < 4$                       |
| not      | Reverse the result, returns False if the result is true | <code>not(x &lt; 5 and x &lt; 10)</code> |

- **Python Identity Operators**
- Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

| Operator | Description                                            | Example    |
|----------|--------------------------------------------------------|------------|
| is       | Returns True if both variables are the same object     | x is y     |
| is not   | Returns True if both variables are not the same object | x is not y |

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x
print(x is z)
returns True because z is the same object as x
print(x is y)
returns False because x is not the same object
as y, even if they have the same content
print(x == y)
to demonstrate the difference betweeen "is"
and "==": this comparison returns True because
x is equal to y
```

# Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

| Operator | Description                                                                      | Example    |
|----------|----------------------------------------------------------------------------------|------------|
| in       | Returns True if a sequence with the specified value is present in the object     | x in y     |
| not in   | Returns True if a sequence with the specified value is not present in the object | x not in y |

```
Ex: x = ["apple", "banana"]
 print("banana" in x)
```

```
returns True because a sequence with the value
"banana" is in the list
```

```
Ex: x = ["apple", "banana"]
 print("pineapple" not in x)
```

```
returns True because a sequence with the value
"pineapple" is not in the list
```

# Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

| Operator | Name | Description                                     |
|----------|------|-------------------------------------------------|
| &        | AND  | Sets each bit to 1 if both bits are 1           |
|          | OR   | Sets each bit to 1 if one of two bits is 1      |
| ^        | XOR  | Sets each bit to 1 if only one of two bits is 1 |
| ~        | NOT  | Inverts all the bits                            |

| Operator | Name                 | Description                                                                                             |
|----------|----------------------|---------------------------------------------------------------------------------------------------------|
| <<       | Zero fill left shift | Shift left by pushing zeros in from the right and let the leftmost bits fall off                        |
| >>       | Signed right shift   | Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off |

# **Operator Precedence**

The precedence of the operators is essential to find out since it enables us to know which operator should be evaluated first.

## Operator

`**`

## Description

The exponent operator is given priority over all the others used in the expression.

`~ + -`

The negation, unary plus, and minus.

`* / % //`

The multiplication, divide, modules, reminder, and floor division.

| Operator      | Description                                                                                |
|---------------|--------------------------------------------------------------------------------------------|
| + -           | Binary plus, and minus                                                                     |
| >> <<         | Left shift. and right shift                                                                |
| &             | Binary and.                                                                                |
| ^             | Binary xor, and or                                                                         |
| <, <= , >, >= | Comparison operators (less than, less than equal to, greater than, greater then equal to). |

## Operator

## Description

`== !=`

Equality operators.

`= %= /= //=- +=`

Assignment operators

`is is not`

Identity operators

`in not in`

Membership operators

`not or and`

Logical operators

# Control Flow

Branching: Simple if

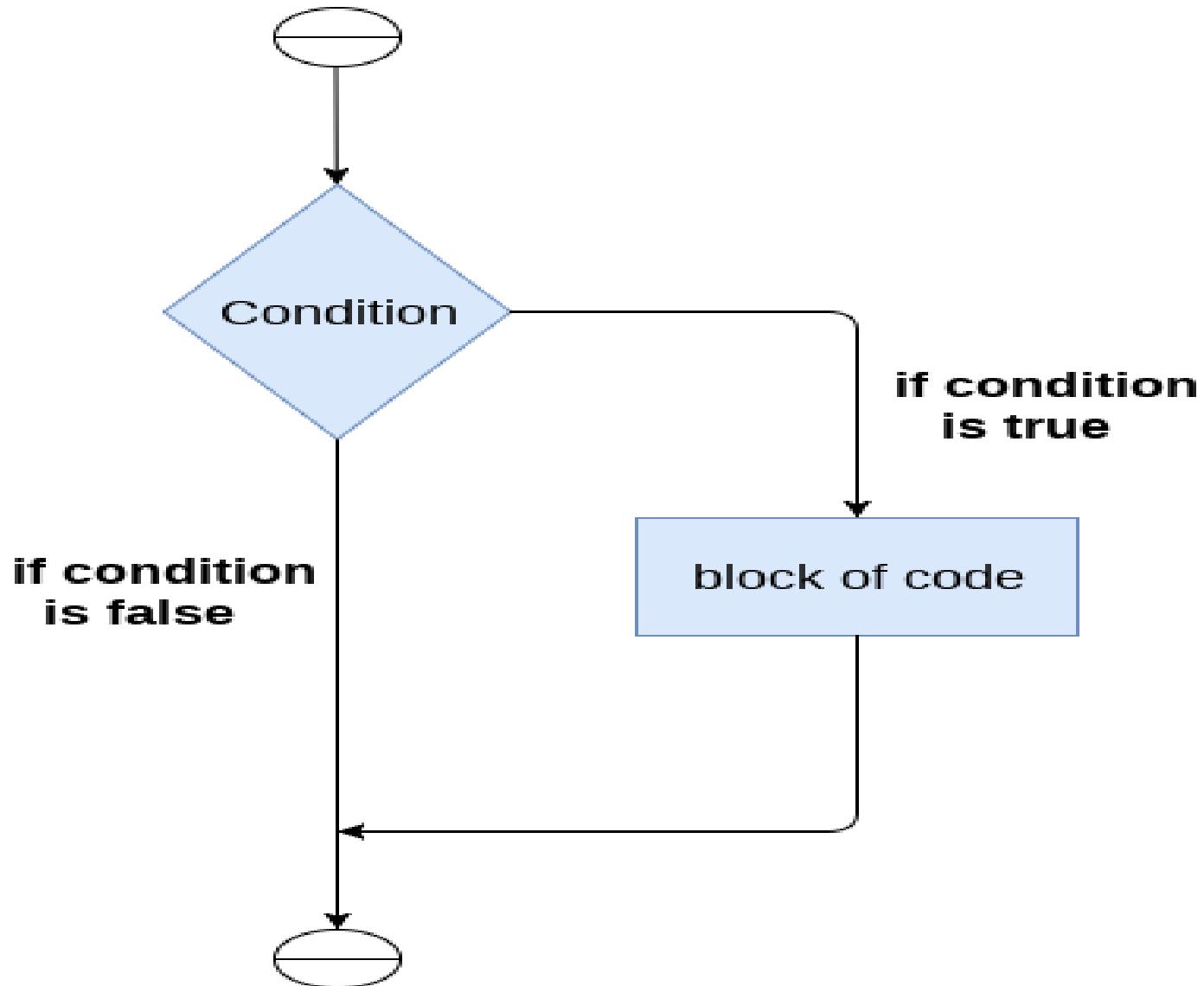
## **The if statement**

The **if** statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block.

The condition of if statement can be any valid logical expression which can be either evaluated to true or false.

The syntax of the if-statement is given below.

```
if expression:
 statement
```



## **Example 1**

```
num = int(input("enter the number?"))

if num%2 == 0:
 print("Number is even")
```

## **Output:**

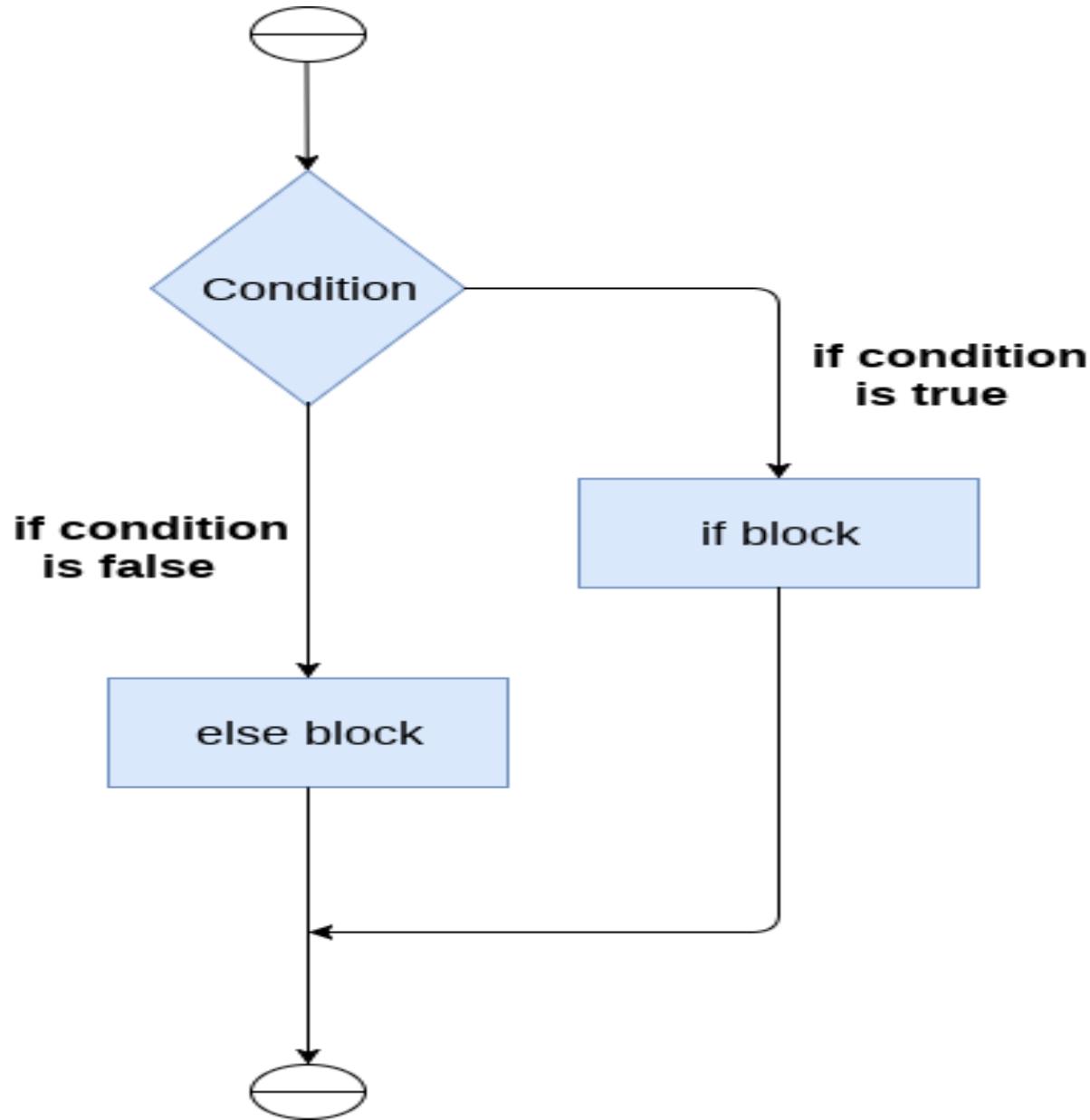
enter the number?10

Number is even

## **if-else statement**

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.



```
if condition:
 #block of statements
else:
 #another block of statements (else-block)
```

**Example 1 : Program to check whether a person is eligible to vote or not.**

```
age = int (input("Enter your age? "))
if age>=18:
 print("You are eligible to vote !!");
else:
 print("Sorry! you have to wait !!");
```

**Output:**

```
Enter your age? 19
You are eligible to vote !!
```

## The elif statement

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them.

We can have any number of elif statements in our program depending upon our need.  
However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C.

The syntax of the elif statement is given below.

if expression 1:

    # block of statements

elif expression 2:

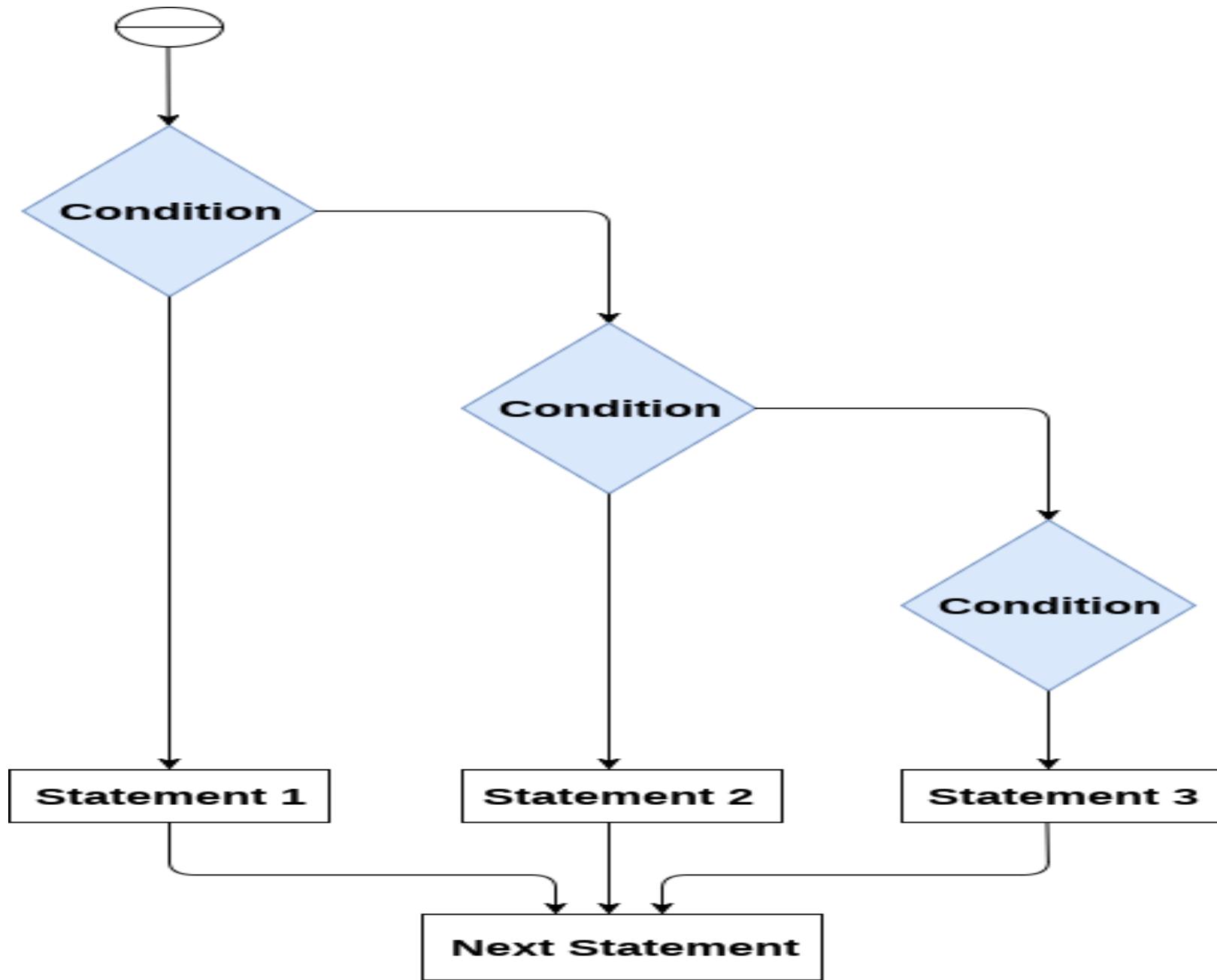
    # block of statements

elif expression 3:

    # block of statements

else:

    # block of statements



## **Example 1**

```
number = int(input("Enter the number?"))
if number==10:
 print("number is equals to 10")
elif number==50:
 print("number is equal to 50");
elif number==100:
 print("number is equal to 100");
else:
 print("number is not equal to 10, 50 or 100");
```

## **Output:**

Enter the number?15

number is not equal to 10, 50 or 100

## Example 2

```
marks = int(input("Enter the marks? "))

if marks > 85 and marks <= 100:
 print("Congrats ! you scored grade A ...")
elif marks > 60 and marks <= 85:
 print("You scored grade B + ...")
elif marks > 40 and marks <= 60:
 print("You scored grade B ...")
elif (marks > 30 and marks <= 40):
 print("You scored grade C ...")
else:
 print("Sorry you are fail ?")
```

# Python Nested If

## If Inside If

You can have if statements inside if statements, this is called *nested* if statements.

### Example

```
x = 41
```

```
if x > 10:
```

```
 print("Above ten,")
```

```
 if x > 20:
```

```
 print("and also above 20")
```

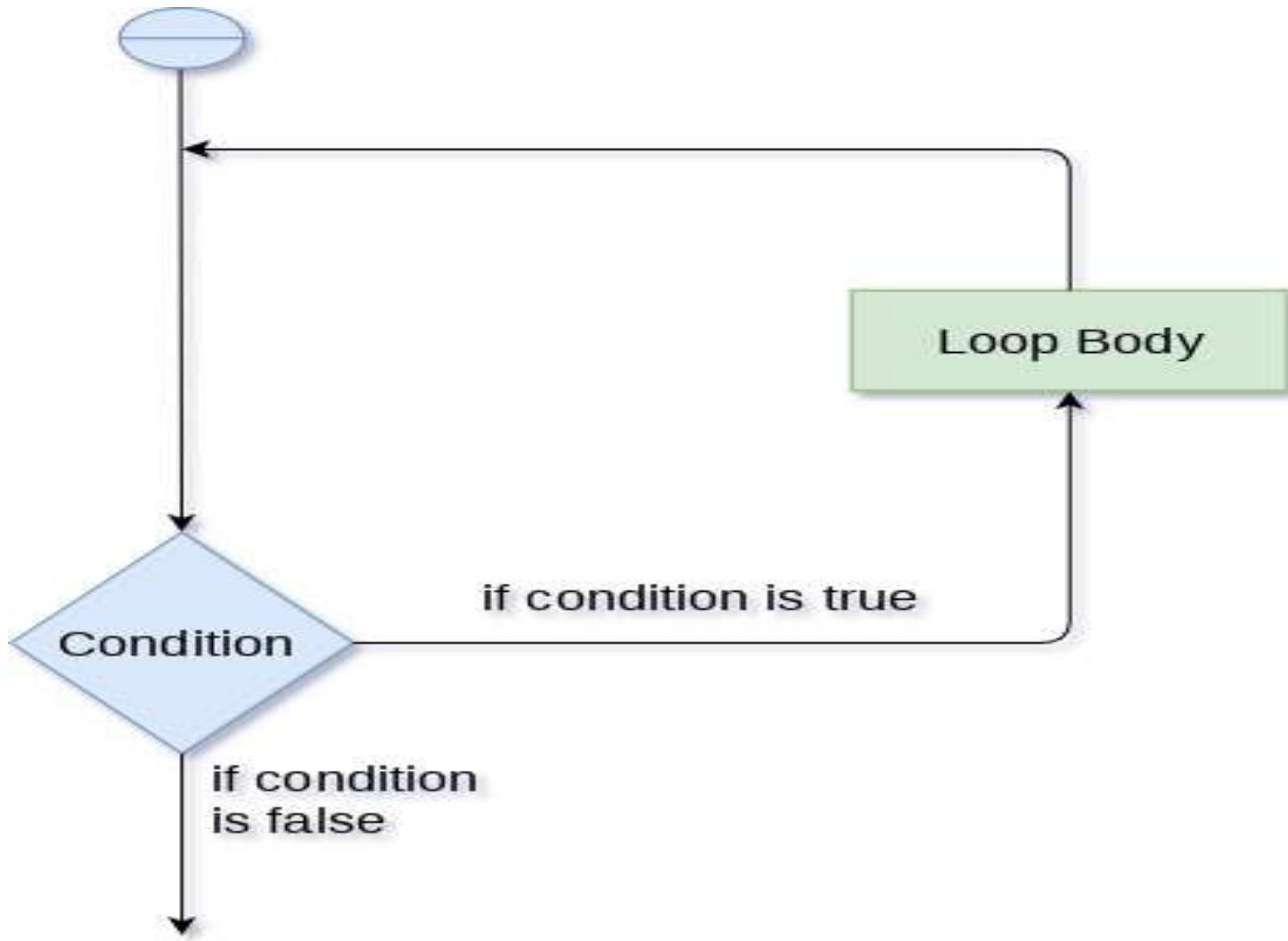
```
 else:
```

```
 print("but not above 20.")
```

# Looping

The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.

The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times.



- **for loop** The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied.
- The for loop is also called as a pre-tested loop. It is better to use for loop if the number of iteration is known in advance.

**do-while loop** The do-while loop continues until a given condition satisfies. It is also called post tested loop.

It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

**while loop** The while loop is to be used in the scenario where we don't know the number of iterations in advance.

The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

# Python While loop

The Python while loop allows a part of the code to be executed until the given condition returns false. It is also known as a **pre-tested** loop.

When we don't know the number of iterations then the while loop is most effective to use.

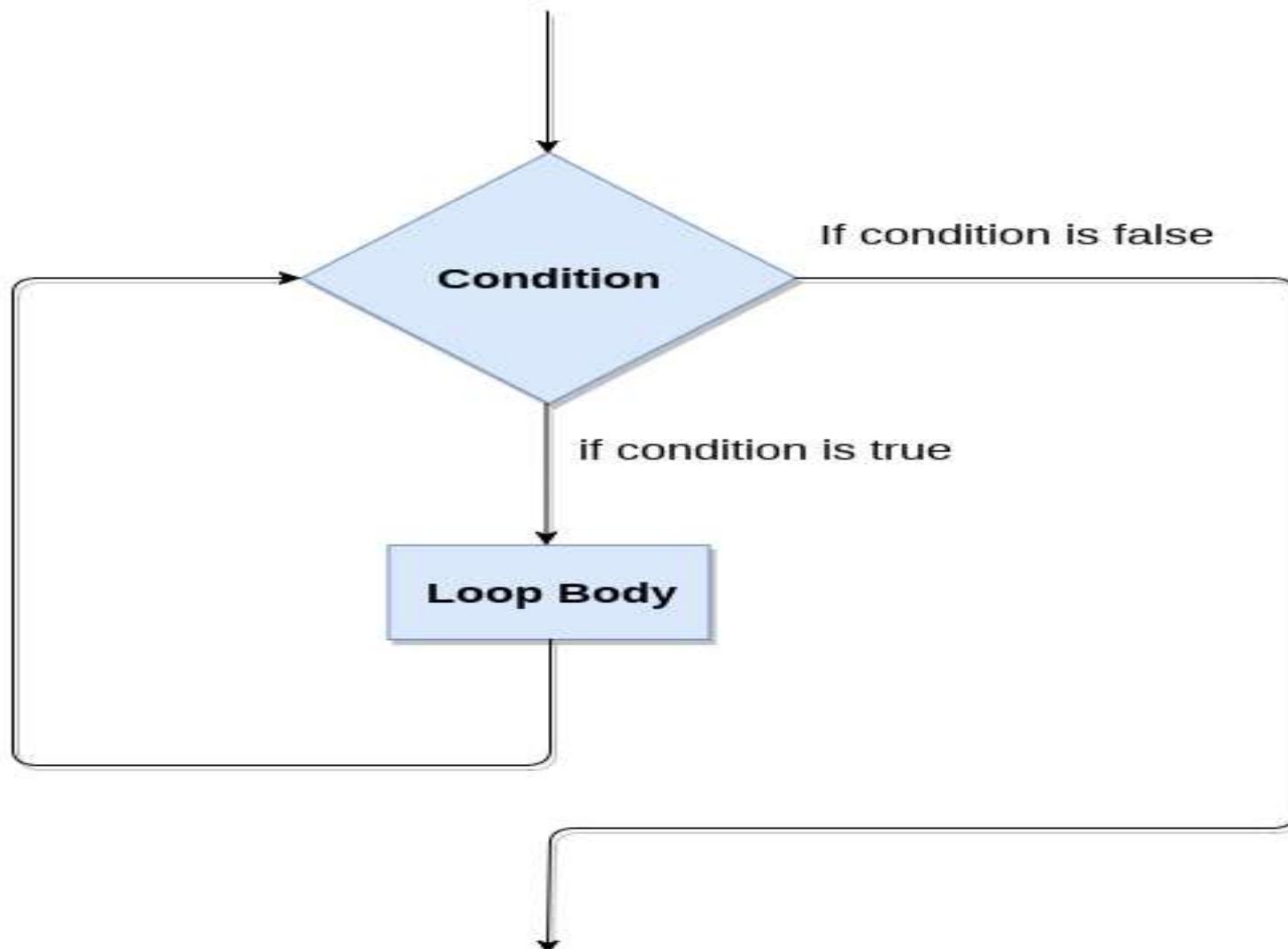
The syntax is given below:

**while expression:**

**statements**

Here, the statements can be a single statement or a group of statements.

# While loop Flowchart



# Loop Control Statements

We can change the normal sequence of **while** loop's execution using the loop control statement. When the while loop's execution is completed, all automatic objects defined in that scope are demolished.

1. **Continue Statement** - When the continue statement is encountered, the control transfer to the beginning of the loop.

## Example:

```
prints all letters except 'a' and 't'
```

```
i = 0
```

```
str1 = 'javatpoint'
```

```
while i < len(str1):
```

```
 if str1[i] == 'a' or str1[i] == 't':
```

```
 i += 1
```

```
 continue
```

```
 print('Current Letter :', a[i])
```

```
 i += 1
```

## **Output:**

Current Letter : j

Current Letter : v

Current Letter : p

Current Letter : o

Current Letter : i

Current Letter : n

**Break Statement** - When the break statement is encountered, it brings control out of the loop.

**Example:**

```
i = 0
```

```
str1 = 'javatpoint'
```

```
while i < len(str1):
```

```
 if str1[i] == 't':
```

```
 i += 1
```

```
 break
```

```
 print('Current Letter :', str1[i])
```

```
 i += 1
```

## **Output:**

Current Letter : j

Current Letter : a

Current Letter : v

Current Letter : a

**Pass Statement** - The pass statement is used to declare the empty loop. It is also used to define empty class, function, and control statement.

**Example** -

```
An empty loop
str1 = 'javatpoint'
i = 0
while i < len(str1):
 i += 1
 pass
print('Value of i :', i)
```

**Output:**

Value of i : 10

# **Program to print 1 to 10 using while loop**

i=1

#The while loop will iterate until condition becomes false.

While(i<=10):

    print(i)

    i=i+1

## **Output:**

1 2 3 4 5 6 7 8 9 10

# Program to print table of given numbers.

i=1

number=0

b=9

```
number = int(input("Enter the number:"))
```

```
while i<=10:
```

```
 print("%d X %d = %d \n"%(number,i,number*i))
```

```
i = i+1
```

## **Output:**

Enter the number:10

10 X 1 = 10

10 X 2 = 20

10 X 3 = 30

10 X 4 = 40

10 X 5 = 50

10 X 6 = 60

10 X 7 = 70

10 X 8 = 80

10 X 9 = 90

10 X 10 = 100

# Python For Loops

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

## Example

```
Print each fruit in a fruit list:
```

```
fruits = ["apple", "banana", "cherry"]
```

```
for x in fruits:
```

```
 print(x)
```

o/p

apple

banana

cherry

The for loop does not require an indexing variable to set beforehand.

# Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

## Example

#Loop through the letters in the word "banana":

```
for x in "banana":
```

```
 print(x)
```

b

a

n

a

n

a

# The range() Function

To loop through a set of code a specified number of times, we can use the range() function,

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

## Example

#Using the range() function:

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

```
 print(x)
```

**o/p**

0

1

2

3

4

5

- Note that `range(6)` is not the values of 0 to 6, but the values 0 to 5.

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

## **Example**

Using the start parameter:

```
for x in range(2, 6):
 print(x)
```

```
2
3
4
5
```

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, 3):

## Example

```
Increment the sequence with 3 (default is 1):
```

```
for x in range(2, 30, 3):
 print(x)
```

```
2
5
8
11
14
17
20
23
26
29
```

# Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

## Example

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):
 print(x)
else:
 print("Finally finished!")
```

o/p

0

1

2

3

4

5

Finally finished!

# Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

## Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
```

```
for x in adj:
 for y in fruits:
 print(x, y)
```

o/p

red apple

red banana

red cherry

big apple

big banana

big cherry

tasty apple

tasty banana

tasty cherry

# **The pass Statement**

for loops cannot be empty, but if we use for some reason, have a for loop with no content, put in the pass statement to avoid getting an error.

## **Example**

```
for x in [0, 1, 2]:
 pass
```

**SIDDARTHA INSTITUTE OF SCIENCE AND  
TECHNOLOGY: PUTTUR**  
**(AUTONOMOUS)**  
**II B.Tech. – I Sem.**

**(20CS0511) PYTHON PROGRAMMING**

# COURSE OBJECTIVES

**The objectives of this course:**

- *Introduce Scripting Language*
- *Exposure to various problem solving approaches of computer science*
- *Introduce function-oriented programming paradigm*
- *Exposure to solve the problems using object oriented concepts, exceptional handling*
- *Exposure to solve the problems using Files, Regular Expressions and, Standard Libraries*

## COURSE OUTCOMES

- On successful completion of this course, the student will be able to
- *Solve the problems using control structures, input and output statements.*
- *Summarize the features of lists, tuples, dictionaries, strings and files*
- *Experience the usage of standard libraries, objects, and modules*
- *Solve the problems using Object Oriented Programming Concepts*
- *Build the software for real time applications using python*
- *Install various Python packages*

## **UNIT – III**

Functions: Defining Functions, Calling Functions, Passing Arguments, Keyword Arguments, Default Arguments, Variable-length arguments, Anonymous Functions, Fruitful Functions- Nested functions, Recursive functions- Scope of the Variables in a Function.

Object Oriented Programming in Python: Classes and Objects- self-variable- Methods –Constructor-Inheritance-polymorphism- Method Overloading- Method Overriding.

# Functions

- Functions are the most important aspect of an application. A **function can be defined as the organized block of reusable code**, which can be called whenever required.
- Python allows us to divide a large program into the basic building blocks known as a function. A function can be called multiple times to provide reusability and modularity to the Python program.

# Defining Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

# Creating a Function

In Python a function is defined using the **def** keyword:

Python provides the **def** keyword to define the function.

The syntax of the define function is given below.

## Syntax:

```
def my_function(parameters):
 function_block
```

```
return expression
```

## Example

```
def my_function():
 print("Hello from a function")
```

## Example

```
def my_function():
 print("Hello from a function")
```

```
my_function()
```

## Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

There are mainly two types of functions.

**User-defined functions** - The user-defined functions are those defined by the **user** to perform the specific task.

**Built-in functions** - The built-in functions are those functions that are **pre-defined** in Python.

# Example

```
def my_function(fname):
 print(fname + " Reference")
```

```
my_function("Emil")
```

```
my_function("Tobias")
```

```
my_function("Linus")
```

o/P

Emil Reference

Tobias Reference

Linus Reference

# Calling Functions

- In Python, after the function is created, we can call it from another function. A function must be defined before the function call; otherwise, the Python interpreter gives an error.
- To call the function, use the function name followed by the parentheses.

```
#function definition
def hello_world():
 print("hello world")
```

```
function calling
```

```
hello_world()
```

**Output:**

```
hello world
```

# **The return statement**

The return statement is used at the end of the function and returns the result of the function. It terminates the function execution and transfers the result where the function is called. The return statement cannot be used outside of the function.

## **Syntax**

`return [expression_list]`

## **Example 1**

```
Defining function
```

```
def sum():
```

```
 a = 10
```

```
 b = 20
```

```
 c = a+b
```

```
 return c
```

```
calling sum() function in print statement
```

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

**Output:**

The sum is: 30

# **Creating function without return statement**

```
Defining function
```

```
def sum():
```

```
 a = 10
```

```
 b = 20
```

```
 c = a+b
```

```
calling sum() function in print statement
```

```
print(sum())
```

**Output:**

None

# **Arguments in function**

The arguments are types of information which can be passed into the function. The arguments are specified in the parentheses. We can pass any number of arguments, but they must be separate them with a comma.

## **Example 1**

```
#defining the function
def func (name):
 print("Hi ",name)
#calling the function
func("Devansh")
```

## **Output:**

Hi Devansh

## **Example 2**

```
#Python function to calculate the sum of two variables
#defining the function
def sum (a,b):
 return a+b;
#taking values from the user
a = int(input("Enter a: "))
b = int(input("Enter b: "))
#printing the sum of a and b
print("Sum = ",sum(a,b))
```

### **Output:**

Enter a: 10

Enter b: 20

Sum = 30

# Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

## Example

```
def my_function(food):
```

```
 for x in food:
```

```
 print(x)
```

```
fruits = ["apple", "banana", "cherry"]
```

```
my_function(fruits)
```

o/p

apple

banana

cherry

# **Pass by Reference vs. Value**

In the Python programming language, all arguments are supplied by reference. It implies that if we modify the value of an argument within a function, the change is also reflected in the calling function. For instance,

## **Code**

```
defining the function
def square(my_list):
 """This function will find the square of items in list"""
 squares = []
 for l in my_list:
 squares.append(l**2)
 return squares
```

```
calling the defined function
list_ = [45, 52, 13];
result = square(list_)
print("Squares of the list is: ", result)
```

## **Output:**

Squares of the list is: [2025, 2704, 169]

# Keyword Arguments

You can also send arguments with the *key = value* syntax.

This way the order of the arguments does not matter.

## Example

```
def my_function(child3, child2, child1):
 print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias",
 child3 = "Linus")
```

o/p The youngest child is Linus

# Default Arguments

A default argument is a kind of parameter that takes as input a default value if no value is supplied for the argument when the function is called. Default arguments are demonstrated in the following instance.

## Code

```
Python code to demonstrate the use of default argument
s
defining a function
def function(num1, num2 = 40):
 print("num1 is: ", num1)
 print("num2 is: ", num2)
```

```
Calling the function and passing only one argument
```

```
print("Passing one argument")
```

```
function(10)
```

```
Now giving two arguments to the function
```

```
print("Passing two arguments")
```

```
function(10,30)
```

## **Output:**

Passing one argument

num1 is: 10

num2 is: 40

Passing two arguments

num1 is: 10

num2 is: 30

# Required Arguments

The arguments given to a function while calling in a pre-defined positional sequence are required arguments.

The count of required arguments in the method call must be equal to the count of arguments provided while defining the function.

We must send two arguments to the function `function()` in the correct order, or it will return a syntax error.

## Code

```
Python code to demonstrate the use of default arguments
```

```
Defining a function
```

```
def function(num1, num2):
```

```
 print("num1 is: ", num1)
```

```
 print("num2 is: ", num2)
```

```
Calling function and passing two arguments out of order, we need num1 to be 20 and num2 to be 30
```

```
print("Passing out of order arguments")
```

```
function(30, 20)
```

```
Calling function and passing only one argument
print("Passing only one argument")
try:
 function(30)
except:
 print("Function needs two positional arguments
")
```

## **Output:**

Passing out of order arguments

num1 is: 30

num2 is: 20

Passing only one argument

Function needs two positional arguments

# Variable-Length Arguments

We can use special characters in Python functions to pass as many arguments as we want in a function. There are two types of characters that we can use for this purpose:

**\*args** -These are Non-Keyword Arguments

**\*\*kwargs** - These are Keyword Arguments.

## Code

```
Python code to demonstrate the use of variable-length arguments

Defining a function

def function(*args_list):

 ans = []
 for l in args_list:
 ans.append(l.upper())
 return ans

Passing args arguments

object = function('Python', 'Functions', 'tutorial')

print(object)
```

```
defining a function
def function(**kargs_list):
 ans = []
 for key, value in kargs_list.items():
 ans.append([key, value])
 return ans

Paasing kwargs arguments
object = function(First = "Python", Second = "Func
tions", Third = "Tutorial")
print(object)
```

- **Output:**
- `['PYTHON', 'FUNCTIONS', 'TUTORIAL']`
- `[['First', 'Python'], ['Second', 'Functions'], ['Third', 'Tutorial']]`

# The Anonymous Functions

These types of Python functions are anonymous since we do not declare them, as we declare usual functions, using the def keyword.

We can use the lambda keyword to define the short, single output, anonymous functions.

Lambda expressions can accept an unlimited number of arguments; however, they only return one value as the result of the function.

# Syntax

Lambda functions have exactly one line in their syntax:

```
lambda [argument1 [,argument2... .argumentn]] :
expression
```

Ex:

```
Defining a function
```

```
lambda_ = lambda argument1, argument2: argument1 + argument2
```

```
Calling the function and passing values
print("Value of the function is : ", lambda_(20,30))
print("Value of the function is : ", lambda_(40,50))
```

## **Output:**

Value of the function is : 50

Value of the function is : 90

Fruitful functions:

## **Python Function within Another Function**

Inner or nested function refers to a function defined within another defined function. Inner functions can access the parameters of the outer scope.

Inner functions are constructed to cover them from the changes that happen outside the function. Many developers regard this process as encapsulation.

Ex:

```
defining a nested function
def function1():
 string = 'Python functions tutorial'
 def function2():
 print(string)
 function2()
function1()
```

**Output:**

Python functions tutorial

# Scope and Lifetime of Variables

The scope of a variable refers to the domain of a program wherever it is declared. A function's arguments and variables are not accessible outside the defined function. As a result, they only have a local domain.

The period of a variable's existence in RAM is referred to as its lifetime. Variables within a function have the same lifespan as the function itself.

When we get out of the function, they are removed. As a result, a function does not retain a variable's value from earlier executions.

#defining a function to print a number.

```
def number():
```

```
 num = 30
```

```
 print("Value of num inside the function: ", num)
```

```
num = 20
```

```
number()
```

```
print("Value of num outside the function:", num)
```

**Output:**

Value of num inside the function: 30

Value of num outside the function: 20

## **Recursive functions:**

Recursion is one of an important concept of programming to solve problems.

It is a function that is called itself.

**Example -**

```
def factorial(n):
```

```
 if n==0:
```

```
 return 1
```

```
 else:
```

```
 return n*factorial(n-1)
```

```
num = int(input("Enter a number: "))
```

```
print("The factorial of a {0} is: ".format(num), facto
rial(num))
```

## **Output:**

Enter a number: 6 The factorial of a 6 is: 720

## Example -

```
def fib(n) :
 if n==0:
 return 0
 elif n ==1 :
 return 1
 else :
 return fib(n-1) +fib(n-2)
num = int(input("Enter a number: "))
print("The {num} element of fibonacci series is: ".fo
rmat(num), fib(7))
```

## **Output:**

Enter a number: 7

The 7 element of fibonacci series is: 13

# Object Oriented Programming in Python

## Classes and Objects:

### **Python Classes/Objects**

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

To create a class, use the keyword `class`:

Example

Create a class named `MyClass`, with a property named `x`:

```
class MyClass:
```

```
 x = 5
```

```
print(MyClass)
```

o/P

```
<class '__main__.MyClass'>
```

## Create Object

we can use the class named MyClass to create objects:

### Example

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

```
class MyClass:
```

```
 x = 5
```

```
p1 = MyClass()
```

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

```
O/P : 5
```

## The `__init__()` Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the **built-in `__init__()`** function.

All classes have a function called `__init__()`, which is always executed when the class is being initiated.

Use the **`__init__()` function to assign values to object properties**, or other operations that are necessary to do when the object is being created

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

```
class Person:
```

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

```
 self.name = name
```

```
 self.age = age
```

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

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

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

**o/p**

John

36

**Note:** The `__init__()` function is called automatically every time the class is being used to create a new object.

## Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

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

```
class Person:
 def __init__(self, name, age):
 self.name = name
 self.age = age
 def myfunc(self):
 print("Hello my name is " + self.name)
p1 = Person("John", 36)
p1.myfunc()
```

O/P

Hello my name is John

## The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

```
class Person:
 def __init__(mysillyobject, name, age):
 mysillyobject.name = name
 mysillyobject.age = age
 def myfunc(abc):
 print("Hello my name is " + abc.name)
p1 = Person("John", 36)
p1.myfunc()
```

O/P

Hello my name is John

## Modify Object Properties

You can modify properties on objects like this:

### Example

Set the age of p1 to 40:

```
p1.age = 40
```

## Delete Object Properties

You can delete properties on objects by using the del keyword:

### Example

Delete the age property from the p1 object:

```
del p1.age
```

```
class Person:
 def __init__(self, name, age):
 self.name = name
 self.age = age
 def myfunc(self):
 print("Hello my name is " + self.name)
p1 = Person("John", 36)
p1.age = 40
print(p1.age)
```

O/P

40

# Constructors

Python facilitates a special type of method, also called as Python Constructors, to initialize the instance members of the class and to verify enough object resources for executing any startup task.

## **Types of Constructors:**

Parameterized Constructor

Non- Parameterized Constructor

## **Features of Python Constructors:**

In Python, a Constructor begins with double underscore (\_) and is always named as `__init__()`.

In python Constructors, arguments can also be passed.

In Python, every class must necessarily have a Constructor.

If there is a Python class without a Constructor, a default Constructor is automatically created without any arguments and parameters.

```
class Employees():

 def __init__(self, Name, Salary):
 self.Name = Name
 self.Salary = Salary

 def details(self):
 print ("Employee Name : ", self.Name)
 print("Employee Salary: ", self.Salary)
 print ("\n")
```

```
first = Employees("Khush", 10000)
```

```
second = Employees("Raam", 20000)
```

```
third = Employees("Lav", 10000)
```

```
fourth = Employees("Sita", 30000)
```

```
fifth = Employees("Lucky", 50000)
```

```
first.details()
```

```
second.details()
```

```
third.details()
```

```
fourth.details()
```

```
fifth.details()
```

```
vibhuti@kmradi-Inspiron-3541:~/Desktop$ python pythonexample.py
```

```
Employee Name : Khush
Employee Salary: 10000
```

```
Employee Name : Raam
Employee Salary: 20000
```

```
Employee Name : Lav
Employee Salary: 10000
```

```
Employee Name : Sita
Employee Salary: 30000
```

```
Employee Name : Lucky
Employee Salary: 50000
```

```
vibhuti@kmradi-Inspiron-3541:~/Desktop$ █
```

## Inheritance

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

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

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

## Example

```
class Person:
```

```
 def __init__(self, fname, lname):
```

```
 self.firstname = fname
```

```
 self.lastname = lname
```

```
 def printname(self):
```

```
 print(self.firstname, self.lastname)
```

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

```
x.printname()
```

O/P

John Doe

## Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

### Example

Create a class named Student, which will inherit the properties and methods from the Person class:

```
class Student(Person):
```

```
 pass
```

**Note:** Use the pass keyword when you do not want to add any other properties or methods to the class.

```
class Person:
 def __init__(self, fname, lname):
 self.firstname = fname
 self.lastname = lname
 def printname(self):
 print(self.firstname, self.lastname)
class Student(Person):
 pass
x = Student("Mike", "Olsen")
x.printname()
O/P Mike Olsen
```

- **Add the `__init__()` Function**
- So far we have created a child class that inherits the properties and methods from its parent.
- We want to add the `__init__()` function to the child class (instead of the `pass` keyword).
- **Note:** The `__init__()` function is called automatically every time the class is being used to create a new object.
-

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

class Student(Person):
 def __init__(self, fname, lname):
 Person.__init__(self, fname, lname)

x = Student("Mike", "Olsen")
x.printname()
```

## Use the super() Function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Ex:

```
class Person:
```

```
 def __init__(self, fname, lname):
```

```
 self.firstname = fname
```

```
 self.lastname = lname
```

```
 def printname(self):
```

```
 print(self.firstname, self.lastname)
```

```
class Student(Person):
```

```
 def __init__(self, fname, lname):
```

```
 super().__init__(fname, lname)
```

```
x = Student("Mike", "Olsen")
```

```
x.printname()
```

Ex:

```
class Person:
```

```
 def __init__(self, fname, lname):
```

```
 self.firstname = fname
```

```
 self.lastname = lname
```

```
 def printname(self):
```

```
 print(self.firstname, self.lastname)
```

```
class Student(Person):
```

```
 def __init__(self, fname, lname, year):
```

```
 super().__init__(fname, lname)
```

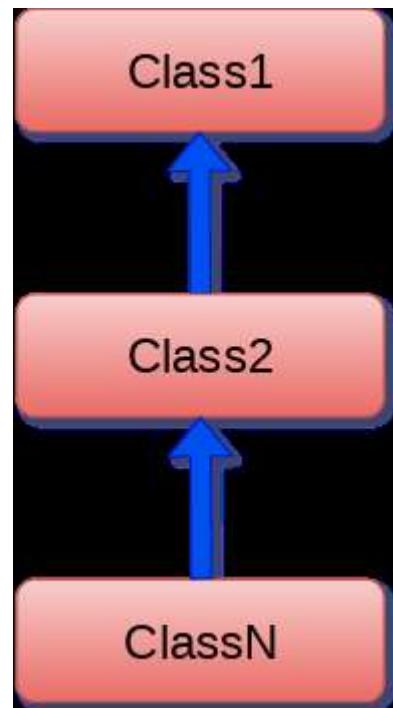
```
 self.graduationyear = year
```

```
def welcome(self):
 print("Welcome", self.firstname, self.lastname,
"to the class of", self.graduationyear)
x = Student("Mike", "Olsen", 2019)
x.welcome()
```

```
class Animal:
 def speak(self):
 print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
 def bark(self):
 print("dog barking")
d = Dog()
d.bark()
d.speak()
O/P
dog barking
Animal Speaking
```

# Python Multi-Level inheritance

Multi-level inheritance is achieved when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is achieved in python.



```
class Animal:
 def speak(self):
 print("Animal Speaking")

class Dog(Animal):
 def bark(self):
 print("dog barking")

class DogChild(Dog):
 def eat(self):
 print("Eating bread...")
```

```
d = DogChild()
```

```
d.bark()
```

```
d.speak()
```

```
d.eat()
```

## **Output:**

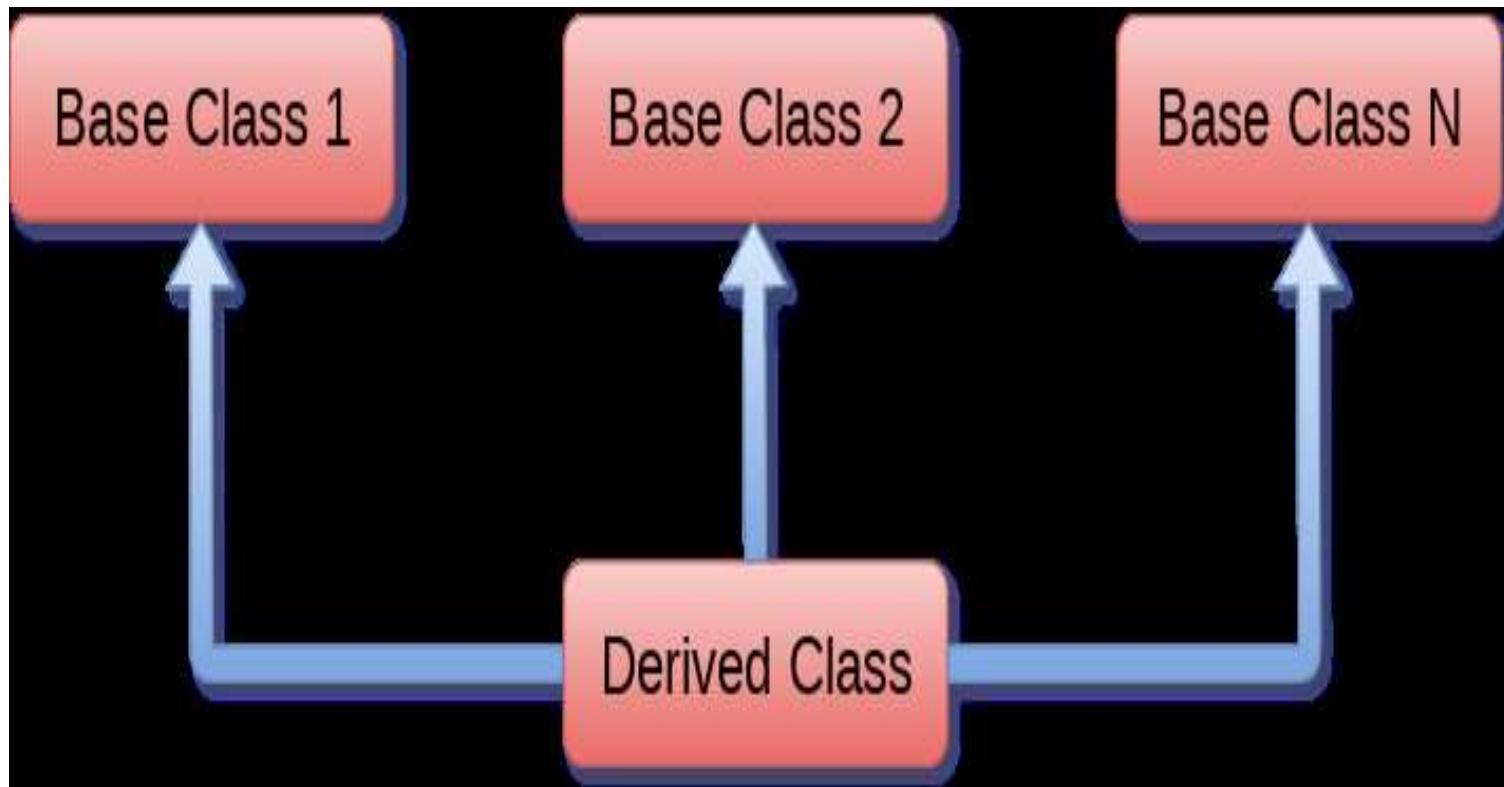
dog barking

Animal Speaking

Eating bread...

# Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.



class Base1:

<class-suite>

class Base2:

<class-suite>

. class BaseN:

<class-suite>

class Derived(Base1, Base2, ..... BaseN):

<class-suite>

```
class Calculation1:
 def Summation(self,a,b):
 return a+b;
class Calculation2:
 def Multiplication(self,a,b):
 return a*b;
class Derived(Calculation1,Calculation2):
 def Divide(self,a,b):
 return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

### **Output:**

30  
200  
0.5

- **Method Overriding**
- We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.
- We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

## **Example**

```
class Animal:
```

```
 def speak(self):
```

```
 print("speaking")
```

```
class Dog(Animal):
```

```
 def speak(self):
```

```
 print("Barking")
```

```
d = Dog()
```

```
d.speak()
```

## **Output:**

Barking

**Polymorphism** defines the ability to take different forms. Polymorphism in [Python](#) allows us to define methods in the child class with the same name as defined in their parent class

- There are different methods to use polymorphism in Python. You can use different function, class methods or objects to define polymorphism.

# Polymorphism with Class Methods

```
class India():
```

```
 def capital(self):
```

```
 print("New Delhi")
```

```
 def language(self):
```

```
 print("Hindi and English")
```

```
class USA():
```

```
 def capital(self):
```

```
 print("Washington, "Newyork")
```

```
 def language(self):
```

```
 print("English")
```

```
obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
 country.capital()
 country.language()
```

## **Output:**

New Delhi

Hindi and English

Washington, Newyork

English

**SIDDARTHA INSTITUTE OF SCIENCE AND  
TECHNOLOGY: PUTTUR**  
**(AUTONOMOUS)**  
**II B.Tech. – I Sem.**

**(20CS0511) PYTHON PROGRAMMING**

# COURSE OBJECTIVES

**The objectives of this course:**

- *Introduce Scripting Language*
- *Exposure to various problem solving approaches of computer science*
- *Introduce function-oriented programming paradigm*
- *Exposure to solve the problems using object oriented concepts, exceptional handling*
- *Exposure to solve the problems using Files, Regular Expressions and, Standard Libraries*

## COURSE OUTCOMES

- On successful completion of this course, the student will be able to
- *Solve the problems using control structures, input and output statements.*
- *Summarize the features of lists, tuples, dictionaries, strings and files*
- *Experience the usage of standard libraries, objects, and modules*
- *Solve the problems using Object Oriented Programming Concepts*
- *Build the software for real time applications using python*
- *Install various Python packages*

## **UNIT – IV**

- **Modules:** Creating modules, import statement, from...import statement and name spacing.
- **Python packages:** Introduction to PIP, Installing Packages via PIP (Numpy, Pandas, Matplotlib etc.,) Using Python Packages.
- **Exception Handling:** Introduction to Errors and Exceptions, Handing Exceptions, RaisingExceptions, User Defined Exceptions, Regular Expressions-Searching and Matching.

## **Modules:** Creating modules

- Consider a module to be the same as a code library.
- A file containing a set of functions you want to include in your application.
- Create a Module
- To create a module just save the code you want in a file with the file extension .py:

## example

Save this code in a file named **mymodule.py**

```
def greeting(name):
 print("Hello, " + name)
```

## Use a Module

Now we can use the module we just created, by using the import statement:

Example

Import the module named **mymodule**, and call the greeting function:

```
import mymodule
mymodule.greeting("Jack")
```

O/P

Hello, Jack

# Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

## Example

Save this code in the file **mymodule.py**

```
person1 = {
 "name": "John",
 "age": 36,
 "country": "Norway"
}
```

## Example

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule
a = mymodule.person1["age"]
print(a)
```

O/P

36

## Naming a Module

You can name the module file whatever you like, but it must have the **file extension .py**

## Re-naming a Module

You can create an alias when you import a module, by using the **as** keyword:

Create an alias for **mymodule** called **mx**:

```
import mymodule as mx
a = mx.person1["age"]
print(a)
```

O/P

# Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

## Example

Import and use the **platform module**:

```
import platform
x = platform.system()
print(x)
```

o/p

Windows

## Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module. The dir() function:

### Example

List all the defined names belonging to the platform module:

```
import platform
x = dir(platform)
print(x)
```

O/P

```
DEV_NULL', '_UNIXCONFDIR', 'WIN32_CLIENT_RELEASES',
'WIN32_SERVER_RELEASES', '__builtins__', '__cached__
'__spec__', '__version__', '_default_architecture',
'_dist_try_harder', '_follow_symlinks',
'_ironpython26_sys_version_parser',
'_ironpython_sys_version_parser', '_java_getprop', '_libc
```

## Import From Module

You can choose to import only parts from a module, by using the `from` keyword.

### Example

The module named `mymodule` has one function and one dictionary:

```
def greeting(name):
 print("Hello, " + name)
person1 = {
 "name": "John",
 "age": 36,
 "country": "Norway"
}
```

## Example

Import only the person1 dictionary from the module:

```
from mymodule import person1
print (person1["age"])
```

O/P

36

**Note:** When importing using the from keyword, do not use the module name when referring to elements in the module.

Example: **person1["age"]**, **not** mymodule.person1["age"]

## Name space:

A namespace is a way of providing the unique name for each object in Python. Everything in Python is an object, i.e., a variable or a method. In other words, **it is a collection of the defined symbolic names along with the information about the object that each name references.**

A namespace can be understood as a dictionary where a name represents a key and objects are values. Let's understand it with a real-life example –

A namespace is like a surname. A "Peter" name might be difficult to find in the class if there are multiple "Peter," but when we particularly ask for "Peter Warner" or "Peter Cummins.". It might be rare to find the same name and surname in a class for multiple students.

The namespace helps the Python interpreter to understand what exact method or variable is trying to point out in the code. So its name gives more information - **Name** (which means name, a unique identifier) + **Space** (related to scope).

In Python, there are four types of namespaces which are given below.

Built-in

Global

Enclosing

Local

# The Built-in Namespace

As its name suggests, it contains pre-defined names of all of Python's built-in objects already available in Python. Let's list these names with the following command.

Open the Python terminal and type the following command.

**Command -**

```
dir(__builtins__)
```

O/P

- ['ArithmetricError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundException', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSErr', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '\_\_build\_class\_\_', '\_\_debug\_\_', '\_\_doc\_\_', '\_\_import\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
- T

# The Global Namespace

The global namespace consists of any names in Python at any level of the main program. It is created when the main body executes and remains in existence until the interpreter terminates.

The Python interpreter creates a global namespace for any module that our Python loads with the import statement.

## The Local and Enclosing Namespaces

The function uses the local namespaces; the Python interpreter creates a new namespace when the function is executed.

The local namespaces remain in existence until the function terminates. The function can also consist of another function. We can define one function inside another as below.

## **Example -**

```
def f():
 print('Initiate f()')
 def g():
 print('Initiate g()')
 print('End g()')
 return
 g()
 print('Initiate f()')
return
f()
```

- In the above example, the function **g()** is defined within the body of **f()**. Inside the **f()** we called the **g()** and called the main **f()** function. Let's understand the working of the above function -
- When we calls **f()**, Python creates a new namespace for **f()**.
- Similarly, the **f()** calls **g()**, **g()** gets its own separate namespace.
- Here the **g() is a local namespace created for f() is the enclosing namespace.**
-

- **Exception Handling:**
- Introduction to Errors and Exceptions
- Exceptions are events that are used to modify the flow of control through a program when the error occurs. Exceptions get triggered automatically on finding errors in Python.

Errors mostly occur at runtime that's they belong to an unchecked type.

Exceptions are the problems which can occur at runtime and compile time. It mainly occurs in the code written by the developers.

| Sr. No. | Key                        | Error                             | Exception                                  |
|---------|----------------------------|-----------------------------------|--------------------------------------------|
| 1       | Type                       | Classified as an unchecked type   | Classified as checked and unchecked        |
| 3       | Recoverable/ Irrecoverable | It is irrecoverable               | It is recoverable                          |
| 4       |                            | It can't be occur at compile time | It can occur at run time compile time both |
| 5       | Example                    | OutOfMemoryError , IOException    | NullPointerException , SQLException        |

- **These exceptions are processed using five statements. These are:**
- **try/except**: catch the error and recover from exceptions hoist by programmers or Python itself.
- **try/finally**: Whether exception occurs or not, it automatically performs the clean-up action.
- **assert**: triggers an exception conditionally in the code.
- **raise**: manually triggers an exception in the code.

- **Roles of exception Handler**
- **Error handling:** The exceptions get raised whenever Python detects an error in a program at runtime. As a programmer, if you don't want the default behavior, then code a 'try' statement to catch and recover the program from an exception. Python will jump to the 'try' handler when the program detects an error; the execution will be resumed.
- **Event Notification:** Exceptions are also used to signal suitable conditions & then passing result flags around a program and text them explicitly.

- **Terminate Execution:** There may arise some problems or errors in programs that it needs a termination. So try/finally is used that guarantees that closing-time operation will be performed.
- **Exotic flow of Control:** Programmers can also use exceptions as a basis for implementing unusual control flow. Since there is no 'go to' statement in Python so that exceptions can help in this respect.

(a,b) = (6,0)

try:# simple use of try-except block for handling  
errors

g = a/b

except ZeroDivisionError:

print ("This is a DIVIDED BY ZERO error")

Output:

This is a DIVIDED BY ZERO error

## Ex: Syntax error

```
string = "Python Exceptions"
```

```
for s in string:
```

```
 if (s != o:
```

```
 print(s)
```

## Output:

```
if (s != o:
```

^

SyntaxError: invalid syntax

Ex:

```
string = "Python Exceptions"
```

```
for s in string:
```

```
 if (s != o):
```

```
 print(s)
```

**Output:**

```
2 string = "Python Exceptions"
```

```
4 for s in string:
```

```
----> 5 if (s != o):
```

```
 6 print(s)
```

NameError: name 'o' is not defined

- **Try and Except Statement - Catching Exceptions**
- In Python, we catch exceptions and handle them using try and except code blocks.
- The **try** clause contains the code that can **raise an exception**, while the **except** clause contains the code lines that **handle the exception**.
-

Ex: try except block

```
a = ["Python", "Exceptions", "try and except"]
```

```
try:
```

```
 #looping through the elements of the array a,
```

```
 #choosing a range that goes beyond the length of the array
```

```
 for i in range(4):
```

```
 print("The index and element from the array is", i, a[i])
```

```
#if an error occurs in the try block, then except block will be
```

```
executed by the Python interpreter
```

```
except:
```

```
 print ("Index out of range")
```

## **Output:**

The index and element from the array is 0 Python

The index and element from the array is 1

## Exceptions

The index and element from the array is 2

try and except

Index out of range

## Raise an Exception

we can intentionally raise an exception in python using the raise keyword. We can use a customized exception in conjunction with the statement.

```
#Python code to show how to raise an exception in Python
```

```
num = [3, 4, 5, 7]
```

```
if len(num) > 3:
```

```
 raise Exception(f"Length of the given list must be less than or equal to 3 but is {len(num)}")
```

## Output:

1 num = [3, 4, 5, 7]

2 if len(num) > 3:

3 raise Exception( f"Length of the given list must  
be less than or equal to 3 but is {len(num)}" )

Exception: Length of the given list must be less  
than or equal to 3 but is 4

## Assertions in Python

When we're finished verifying the program, an assertion is a consistency test that we can switch on or off.

The simplest way to understand an assertion is to compare it with an if-then condition. An exception is thrown if the outcome is false when an expression is evaluated.

Assertions are made via the assert statement, which was added in Python 1.5 as the latest keyword.

Assertions are commonly used at the beginning of a function to inspect for valid input and at the end of calling the function to inspect for valid output.

**The syntax for the assert clause is –**

**assert Expressions[, Argument]**

Python uses ArgumentException, if the assertion fails, as the argument for the Assertion Error.

## Code

```
#Python program to show how to use assert
#keyword
defining a function
def square_root(Number):
 assert (Number < 0), "Give a positive integer"
 return Number**(1/2)
#Calling function and passing the values
print(square_root(36))
print(square_root(-36))
```

## Output:

```
7 #Calling function and passing the values
8 print(square_root(36))
9 print(square_root(-36))
3 def square_root(Number):
4 assert (Number < 0), "Give a positive integer"
5 return Number**(1/2)
6
AssertionError: Give a positive integer
```

## Try with Else Clause

Python also supports the else clause, which should come after every except clause, in the try, and except blocks.

Only when the try clause fails to throw an exception the Python interpreter goes on to the else block.

## **Ex Code:**

```
def reciprocal(num1):
try:
 reci = 1 / num1
except ZeroDivisionError:
 print("We cannot divide by zero")
else:
 print (reci)
reciprocal(4)
reciprocal(0)
```

## **Output:**

0.25

We cannot divide by zero

## **Finally Keyword in Python**

The finally keyword is available in Python, and it is always used after the try-except block.

The finally code block is always executed after the try block has terminated normally or after the try block has terminated for some other reason.

## **Ex Code:**

**try:**

```
 div = 4 // 0
```

```
 print(div)
```

**except ZeroDivisionError:**

```
 print("Attempting to divide by zero")
```

**finally:**

```
 print('This is code of finally clause')
```

## **Output:**

Attempting to divide by zero

This is code of finally clause

## **User-Defined Exceptions**

By inheriting classes from the typical built-in exceptions, Python also lets us design our customized exceptions.

We raise a user-defined exception in the try block and then handle the exception in the except block.

## Ex Code

```
class EmptyError(RuntimeError):
 def __init__(self, argument):
 self.arguments = argument
var = " "

try:
 raise EmptyError("The variable is empty")
except (EmptyError, var):
 print(var.arguments)
```

## **Output:**

2 try:

3 raise EmptyError( "The variable is empty" )

4 except (EmptyError, var):

EmptyError: The variable is empty

## Example

Search the string to see if it starts with "The" and ends with "Spain":

```
txt = "The rain in Spain"
```

```
x = re.search("^The.*Spain$", txt)
```

```
if x:
```

```
 print("YES! We have a match!")
```

```
else:
```

```
 print("No match")
```

```
o/P
```

YES! We have a match!

# RegEx Functions

The `re` module offers a set of functions that allows us to search a string for a match:

| Function                       | Description                                                                              |
|--------------------------------|------------------------------------------------------------------------------------------|
| <a href="#"><u>findall</u></a> | Returns a list containing all matches                                                    |
| <a href="#"><u>search</u></a>  | Returns a <a href="#"><u>Match object</u></a> if there is a match anywhere in the string |
| <a href="#"><u>split</u></a>   | Returns a list where the string has been split at each match                             |
| <a href="#"><u>sub</u></a>     | Replaces one or many matches with a string                                               |

# Metacharacters

Metacharacters are characters with a special meaning:

| Character | Description                                                                | Example       | Try it                   |
|-----------|----------------------------------------------------------------------------|---------------|--------------------------|
| []        | A set of characters                                                        | "[a-m]"       | <a href="#">Try it »</a> |
| \         | Signals a special sequence (can also be used to escape special characters) | "\d"          | <a href="#">Try it »</a> |
| .         | Any character (except newline character)                                   | "he..o"       | <a href="#">Try it »</a> |
| ^         | Starts with                                                                | "^hello"      | <a href="#">Try it »</a> |
| \$        | Ends with                                                                  | "planet\$"    | <a href="#">Try it »</a> |
| *         | Zero or more occurrences                                                   | "he.*o"       | <a href="#">Try it »</a> |
| +         | One or more occurrences                                                    | "he.+o"       | <a href="#">Try it »</a> |
| ?         | Zero or one occurrences                                                    | "he.?o"       | <a href="#">Try it »</a> |
| {}        | Exactly the specified number of occurrences                                | "he.{2}o"     | <a href="#">Try it »</a> |
|           | Either or                                                                  | "falls stays" | <a href="#">Try it »</a> |
| ( )       | Capture and groups                                                         |               |                          |

```
import re
txt = "The rain in Spain"
#Find all lower case characters alphabetically
between "a" and "m":
x = re.findall("[a-m]", txt)
print(x)
O/P
['h', 'e', 'a', 'i', 'i', 'a', 'i']
```

```
import re
txt = "That will be 59 dollars"
#Find all digit characters:
x = re.findall("\d", txt)
print(x)

O/P
['5', '9']
```

```
import re
txt = "hello planet"
#Check if the string starts with 'hello':
x = re.findall("^hello", txt)
if x:
 print("Yes, the string starts with 'hello'")
else:
 print("No match")
```

O/P

Yes, the string starts with 'hello'

# Python packages: Introduction to PIP

## PIP

PIP is a package manager for Python packages, or modules if you like.

**Note:** If you have Python version 3.4 or later, PIP is included by default.

## Package

A package contains all the files you need for a module.

Modules are Python code libraries you can include in your project.

## Check if PIP is Installed

Navigate your command line to the location of Python's script directory, and type the following:

### Example

Check PIP version:

C:\Users\Your  
Name\AppData\Local\Programs\Python\Python  
\Scripts>pip --version

## Install PIP

If you do not have PIP installed, you can download and install it from this page: <https://pypi.org/project/pip/>

### Download a Package

Downloading a package is very easy.

Open the command line interface and tell PIP to download the package you want.

Navigate your command line to the location of Python's script directory, and type the following:

## Example

Download a package named "camelcase":

C:\Users\Your  
*Name*\AppData\Local\Programs\Python\Python\  
Scripts>pip install camelcase

- Now you have downloaded and installed your first package!

# Numpy

- NumPy is a Python library used for working with arrays.
- It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.
- NumPy stands for Numerical Python.

# Why Use NumPy?

- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- Arrays are very frequently used in data science, where speed and resources are very important

# Why is NumPy Faster Than Lists?

- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- This behavior is called locality of reference in computer science.

# Which Language is NumPy written in?

- NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

# Installation of NumPy

- pip install numpy
- It will take 2 minutes time to download the numpy library in google colab

# Example:1

- import numpy

```
arr = numpy.array([1, 2, 3, 4, 5])
```

```
print(arr)
```

- o/p: 1, 2, 3, 4, 5

# Checking NumPy Version

- import numpy as np

```
print(np.__version__)
```

- Output:
- 1.16.3

# NumPy Array Indexing

- Access Array Elements
- Array indexing is the same as accessing an array element.
- You can access an array element by referring to its index number.
- The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

- import numpy as np

```
arr = np.array([1, 2, 3, 4])
```

```
print(arr[0])
```

- Output:1

# Access 2-D Arrays

- import numpy as np

```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
```

```
print('2nd element on 1st row: ', arr[0, 1])
```

- import numpy as np
  - arr = np.array([1, 2, 3, 4, 5, 6])
  - newarr = np.array\_split(arr, 3)
  - print(newarr)

• o/p:[array([1, 2]), array([3, 4]), array([5, 6])]

- import numpy as np
  - arr = np.array([1, 2, 3, 4, 5, 4, 4])
  - x = np.where(arr == 4)
  - print(x)

• o/p:(array([3, 5, 6]),)

# NumPy Sorting Arrays

- Sorting Arrays
- Sorting means putting elements in an *ordered sequence*.
- *Ordered sequence* is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.
- The NumPy ndarray object has a function called `sort()`, that will sort a specified array.

# Sorting Arrays

- Sorting means putting elements in an *ordered sequence*
- import numpy as np

```
arr = np.array([3, 2, 0, 1])
```

```
print(np.sort(arr))
```

- Output:0,1,2,3

**Pandas** is a Python library.

Pandas is used to analyze data.

## **Pandas Series**

A Pandas Series is like a column in a table.

It is a one-dimensional array holding data of any type.

```
import pandas as pd
```

```
a = [1, 7, 2]
```

```
myvar = pd.Series(a)
```

```
print(myvar)
```

```
o/p: 0 1
```

```
1 7
```

```
2 2 dtype:
```

```
int64
```

# Pandas DataFrames

- A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

- import pandas as pd

```
data = {
 "calories": [420, 380, 390],
 "duration": [50, 40, 45]
}
```

```
#load data into a DataFrame object:
df = pd.DataFrame(data)
```

```
print(df)
```

# Output

|   | calories | duration |
|---|----------|----------|
| 0 | 420      | 50       |
| 1 | 380      | 40       |
| 2 | 390      | 45       |

# Pandas - Analyzing DataFrames

- Viewing the Data
- One of the most used method for getting a quick overview of the DataFrame, is the head() method.
- The head() method returns the headers and a specified number of rows, starting from the top.

# Head()

- import pandas as pd

```
df = pd.read_csv('data.csv')
```

```
print(df.head(10))
```

- Note: Head means first ten rows will collect from the dataset

# Output

| Duration | Pulse | Maxpulse | Calories |
|----------|-------|----------|----------|
| 0        | 60    | 110      | 409.1    |
| 1        | 60    | 117      | 479.0    |
| 2        | 60    | 103      | 340.0    |
| 3        | 45    | 109      | 282.4    |
| 4        | 45    | 117      | 406.0    |
| 5        | 60    | 102      | 300.5    |
| 6        | 60    | 110      | 374.0    |
| 7        | 45    | 104      | 253.3    |
| 8        | 30    | 109      | 195.1    |
| 9        | 60    | 98       | 269.0    |

# Tail()

- import pandas as pd

```
df = pd.read_csv('data.csv')
```

```
print(df.Tail(10))
```

Note: Tail means last ten rows will collect from the dataset

# Output

| Duration | Pulse | Maxpulse | Calories |
|----------|-------|----------|----------|
| 159      | 30    | 80       | 240.9    |
| 160      | 30    | 85       | 250.4    |
| 161      | 45    | 90       | 260.4    |
| 162      | 45    | 95       | 270.0    |
| 163      | 45    | 100      | 280.9    |
| 164      | 60    | 105      | 290.8    |
| 165      | 60    | 110      | 300.4    |
| 166      | 60    | 115      | 310.2    |
| 167      | 75    | 120      | 320.4    |
| 168      | 75    | 125      | 330.4    |

# Matplotlib Pyplot

- Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias:
- import matplotlib.pyplot as plt

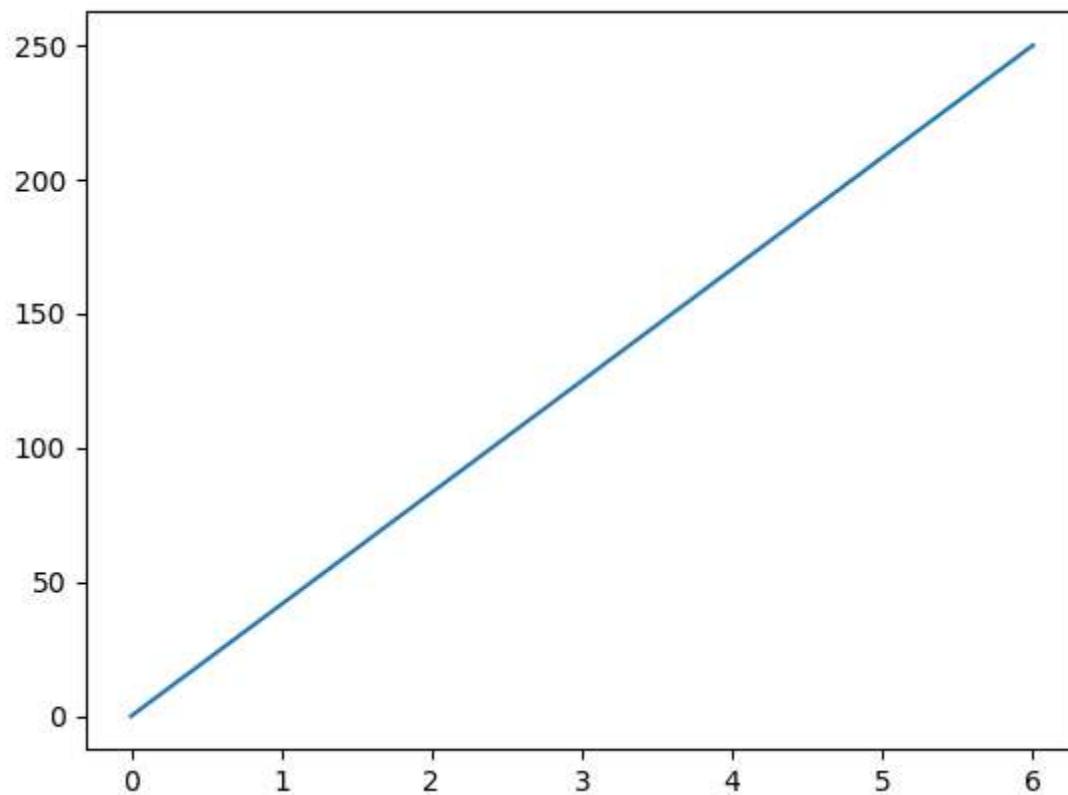
# Example :1

- import matplotlib.pyplot as plt  
import numpy as np

```
xpoints = np.array([0, 6])
ypoints = np.array([0, 250])
```

```
plt.plot(xpoints, ypoints)
plt.show()
```

# Output



# Matplotlib Plotting

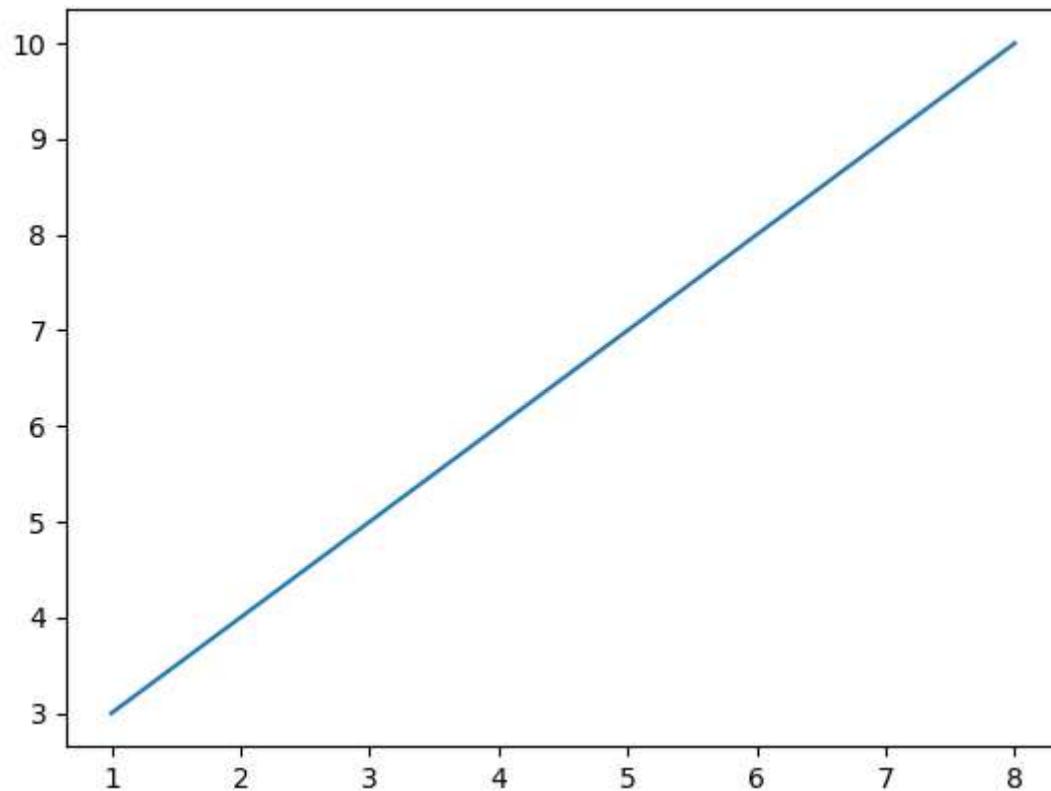
- Plotting x and y points
- The plot() function is used to draw points (markers) in a diagram.
- By default, the plot() function draws a line from point to point.
- The function takes parameters for specifying points in the diagram.
- Parameter 1 is an array containing the points on the **x-axis**.
- Parameter 2 is an array containing the points on the **y-axis**.

- import matplotlib.pyplot as plt  
import numpy as np

```
xpoints = np.array([1, 8])
ypoints = np.array([3, 10])
```

```
plt.plot(xpoints, ypoints)
plt.show()
```

# Output



**SIDDARTHA INSTITUTE OF SCIENCE AND  
TECHNOLOGY: PUTTUR**  
**(AUTONOMOUS)**  
**II B.Tech. – I Sem.**

**(20CS0511) PYTHON PROGRAMMING**

# COURSE OBJECTIVES

**The objectives of this course:**

- *Introduce Scripting Language*
- *Exposure to various problem solving approaches of computer science*
- *Introduce function-oriented programming paradigm*
- *Exposure to solve the problems using object oriented concepts, exceptional handling*
- *Exposure to solve the problems using Files, Regular Expressions and, Standard Libraries*

## COURSE OUTCOMES

- On successful completion of this course, the student will be able to
- *Solve the problems using control structures, input and output statements.*
- *Summarize the features of lists, tuples, dictionaries, strings and files*
- *Experience the usage of standard libraries, objects, and modules*
- *Solve the problems using Object Oriented Programming Concepts*
- *Build the software for real time applications using python*
- *Install various Python packages*

## **UNIT – V**

Functional Programming: Iterators and Generators -  
Maps and Filters.

Files: Text files- Reading and Writing files- Command  
line arguments.

Brief Tour of the Standard Library: Dates and Times-  
Data Compression- Python Runtime Services-  
Data Management and Object Persistence.

GUI Programming - Turtle Graphics

Functional programming is designed to handle the symbolic computation and application processing and it is based on mathematical work.

The most popular functional programming languages are Python, Lisp, Haskell, Clojure, Erlang etc.

# Python Iterators

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

# Example 1

- mytuple = ("apple", "banana", "cherry")
- myit = iter(mytuple)

```
print(next(myit))
```

```
print(next(myit))
```

```
print(next(myit))
```

Output:Apple,banana,cherry

## Example 2

- mytuple = ("apple", "banana", "cherry")

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

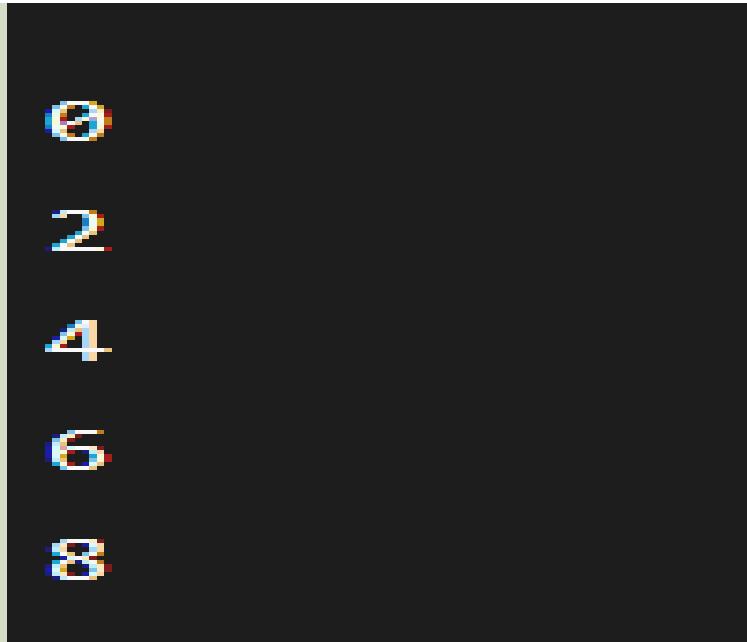
- Output:Apple,banana,cherry

# Generators

- Python Generators are the functions that return the traversal object and used to create iterators. It traverses the entire items at once. The generator can also be an expression.
- we need to implement `__iter__()` and `__next__()` method to keep track of internal states.

```
def simple():
 for i in range(10):
 if(i%2==0):
 yield i
#Successive Function call using for loop
for i in simple():
 print(i)
Note: yield is similar to return stmt in function
```

# Output



## MAP's

The python **map()** function is used to return a list of results after applying a given function to each item of an iterable(list, tuple etc.)

Syntax:

`map(function, iterables)`

### Parameters

**function**- It is a function in which a map passes each item of the iterable.

**iterables**- It is a sequence, collection or an iterator object which is to be mapped.

### Return

It returns a list of results after applying a given function to each item of an iterable(list, tuple etc.)

```
def calculateAddition(n):
 return n+n

numbers = (1, 2, 3, 4)

result = map(calculateAddition, numbers)
print(result)

converting map object to set
numbersAddition = set(result)
print(numbersAddition)
```

```
<map object at 0x7fb04a6bec18>
```

```
{8, 2, 4, 6}
```

```
numbers = (1, 2, 3, 4)
result = map(lambda x: x-x, numbers)
print(result)

converting map object to set
numbersSubtract = set(result)
print(numbersSubtract)
```

O/P

```
<map 0x7f53009e7cf8>
{0}
```

# Python filter() Function

Python filter() function is used to get filtered elements. This function takes two arguments, first is a function and the second is iterable. The filter function returns a sequence from iterable for which function returns True.

The first argument can be None if the function is not available and returns only elements that are True.

## Syntax

`filter (function, iterable)`

# Example 1

```
Python filter() function example
```

```
def filterdata(x):
```

```
 if x>5:
```

```
 return x
```

```
Calling function
```

```
result = filter(filterdata,(1,2,6))
```

```
Displaying result
```

```
print(list(result))
```

Output:6

## Example 2

```
result1 = filter(None,(1,0,6))
returns all non-zero values

result2 = filter(None,(1,0,False,True))
returns all non-zero and True values

Displaying result

result1 = list(result1)
result2 = list(result2)
print(result1)
print(result2)
```

O/P

[1, 6]

[1, True]

**Files:**Sometimes, it is not enough to only display the data on the console.

The data to be displayed may be very large, and only a limited amount of data can be displayed on the console.

A file is a named location on disk to store related information.

We can access the stored information (non-volatile) after the program termination.

In Python, files are treated in two modes as **text or binary**. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

Open a file

Read or write - Performing operation

Close the file

- **Syntax:**
- file object = open(<file-name>, <access-mode>, <buffering>)

| SN | Access | Description                                                                                                                                          |
|----|--------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1  | r      | It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed. |
| 2  | rb     | It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.                                               |

```
fileptr = open("D:\\file.txt","r")
```

```
if fileptr:
```

```
 print("file is opened successfully")
```

O/P

```
<class '_io.TextIOWrapper'> file is opened
successfully
```

## The **close()** method

Once all the operations are done on the file, we must close it through our Python script using the **close()** method.

Any unwritten information gets destroyed once the **close()** method is called on a file object.

```
fileptr = open("file.txt","r")
if fileptr:
 print("file is opened successfully")
#closes the opened file
fileptr.close()
```

# Writing the file

- To write some text to a file, we need to open the file using the `open` method with one of the following access modes.
- **w**: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.
- **a**: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

# Example 1

```
fileptr = open("D:\file2.txt", "w")
appending the content to the file
fileptr.write("""Python is the modern day language. It makes things so simple.
It is the fastest-growing programing language""")
closing the opened the file
fileptr.close()
```

O/P

File2.txt

Python is the modern-day language. It makes things so simple. It is the fastest growing programming language.

## Reading Lines Using `readlines()` function

```
fileptr = open("file2.txt","r");
#stores all the data of the file into the
#variable content
content = fileptr.readlines()
#prints the content of the file
print(content)
#closes the opened file
fileptr.close()
```

## **Command Line Arguments**

**Python command line arguments** provides a user-friendly interface to your text-based command line program.

Python exposes a mechanism to capture and extract your Python command line arguments.

These values can be used to modify the behaviour of a program.

For example, if your program processes [data read from a file](#), then you can pass the name of the file to your program

Python allows for command line input.

That means we are able to ask the user for input.

The method is a bit different in Python 3.6 than Python 2.7.

Python 3.6 uses the `input()` method.

Python 2.7 uses the `raw_input()` method.

- **Python 3.6**
- `print("Enter your name:")`  
`x = input()`  
`print("Hello ", x)`

## Python 2.7

```
print("Enter your name:")
x = raw_input()
print("Hello ", x)
```

Save this file as **demo\_string\_input.py**, and load it through the command line:

**C:\Users\Your Name>python demo\_string\_input.py**

Our program will prompt the user for a string:

Enter your name:

- The user now enters a name:
- Linus

Then, the program prints it to screen with a little message:

Hello Linus

Python stops executing when it comes to the `input()` function, and continues when the user has given some input.

- The Python **sys** module provides access to any command-line arguments via the **sys.argv**. This serves two purposes –
- **sys.argv** is the list of command-line arguments.
- **len(sys.argv)** is the number of command-line arguments.
- Here **sys.argv[0]** is the program ie. script name.

## Example

Consider the following script test.py –

```
#!/usr/bin/python

import sys

print 'Number of arguments:', len(sys.argv),
'arguments.'

print 'Argument List:', str(sys.argv)
```

Now run above script as follows –

```
$ python test.py arg1 arg2 arg3
```

- This produce following result –
- Number of arguments: 4 arguments.
- Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
- **NOTE** – As mentioned above, first argument is always script name and it is also being counted in number of arguments.

# Date and Time

## Python Dates

A date in Python is not a data type of its own, but we can import a module named `datetime` to work with dates as date objects.

# Example 1

```
import datetime
x = datetime.datetime.now()
print(x)
```

Output:

2022-09-21 11:54:45.686028

## Example 2

- Date Output
- When we execute the code from the example above the result will be:
- 2022-09-21 11:54:06.435142
- The date contains year, month, day, hour, minute, second, and microsecond.

- **Creating Date Objects**
- To create a date, we can use the `datetime()` class (constructor) of the `datetime` module.
- The `datetime()` class requires three parameters to create a date: year, month, day.

## Example

Create a date object:

```
import datetime
x = datetime.datetime(2020, 5, 17)
print(x)
```

O/P

2020-05-17 00:00:00

# The strftime() Method

The datetime object has a method for formatting date objects into readable strings.

## Example

Display the name of the month:

```
import datetime
x = datetime.datetime(2018, 6, 1)
print(x.strftime("%B"))
```

O/P

June

## Example

Return the year and name of weekday:

```
import datetime
x = datetime.datetime.now()
print(x.year)
print(x.strftime("%A"))
```

O/P

2022

Wednesday

## Python – Data Compression:

- In this tutorial, we will learn about the data compression in Python programming language. In python, the data can be archived, compressed using the modules like zlib, gzip, bz2, lzma, zipfile and tarfile.
- To use the respective module, you need to import the module first. Let us look at below example.

# Example 1

- import zlib >>> s = b'you learn learnt learning  
the data daily '
- Output:len(s) 41

## Example 2

- `t = zlib.compress(s)`
- `len(t)`
- Output : 39

# Python Run Time Service

- The modules described in this chapter provide a wide range of services related to the Python interpreter and its interaction with its environment. Here's an overview:
- [sys — System-specific parameters and functions](#)
- [sysconfig — Provide access to Python's configuration information](#)
  - [Configuration variables](#)
  - [Installation paths](#)
  - [Other functions](#)
  - [Using sysconfig as a script](#)

# Cont,

- `dataclasses` — Data Classes
  - Module contents
  - Post-init processing
  - Class variables
  - Init-only variables
  - Frozen instances
  - Inheritance
  - Re-ordering of keyword-only parameters in `__init__()`
  - Default factory functions
  - Mutable default values
  - Descriptor-typed fields

|                                   |                                                                                   |
|-----------------------------------|-----------------------------------------------------------------------------------|
| <a href="#"><u>gc</u></a>         | Interface to the cycle-detecting garbage collector.                               |
| <a href="#"><u>weakref</u></a>    | Support for weak references and weak dictionaries.                                |
| <a href="#"><u>fpectl</u></a>     | Provide control for floating point exception handling.                            |
| <a href="#"><u>atexit</u></a>     | Register and execute cleanup functions.                                           |
| <a href="#"><u>types</u></a>      | Names for built-in types.                                                         |
| <a href="#"><u>UserDict</u></a>   | Class wrapper for dictionary objects.                                             |
| <a href="#"><u>UserList</u></a>   | Class wrapper for list objects.                                                   |
| <a href="#"><u>UserString</u></a> | Class wrapper for string objects.                                                 |
| <a href="#"><u>operator</u></a>   | All Python's standard operators as built-in functions.                            |
| <a href="#"><u>inspect</u></a>    | Extract information and source code from live objects.                            |
| <a href="#"><u>traceback</u></a>  | Print or retrieve a stack traceback.                                              |
| <a href="#"><u>linecache</u></a>  | This module provides random access to individual lines from text files.           |
| <a href="#"><u>pickle</u></a>     | Convert Python objects to streams of bytes and back.                              |
| <a href="#"><u>cPickle</u></a>    | Faster version of <a href="#"><u>pickle</u></a> , but not subclassable.           |
| <a href="#"><u>copy_reg</u></a>   | Register pickle support functions.                                                |
| <a href="#"><u>shelve</u></a>     | Python object persistence.                                                        |
| <a href="#"><u>copy</u></a>       | Shallow and deep copy operations.                                                 |
| <a href="#"><u>marshal</u></a>    | Convert Python objects to streams of bytes and back (with different constraints). |
| <a href="#"><u>warnings</u></a>   | Issue warning messages and control their disposition.                             |

# Data Management and Object Persistence

- During the course of using any software application, user provides some data to be processed. The data may be input, using a standard input device (keyboard) or other devices such as disk file, scanner, camera, network cable, WiFi connection, etc.
- Data so received, is stored in computer's main memory (RAM) in the form of various data structures such as, variables and objects until the application is running. Thereafter, memory contents from RAM are erased.

# Data Persistence

The word ‘persistence’ means "the continuance of an effect after its cause is removed".

The term data persistence means it continues to exist even after the application has ended.

Thus, data stored in a non-volatile storage medium such as, a disk file is a persistent data storage.

Using Python’s built-in File object, it is possible to write string data to a disk file and read from it. Python’s standard library, provides modules to store and retrieve serialized data in various data structures such as JSON(JSON stands for JavaScript Object Notation ) and XML.

- **Python Data Persistence - File API**
- Python uses built-in **input()** and **print()** functions to perform standard input/output operations. The **input()** function reads bytes from a standard input stream device, i.e. keyboard.
- The **print()** function on the other hand, sends the data towards standard output stream device i.e. the display monitor. Python program interacts with these IO devices through standard stream objects **stdin** and **stdout** defined in **sys** module.
- The **input()** function is actually a wrapper around **readline()** method of **sys.stdin** object. All keystrokes from the input stream are received till 'Enter' key is pressed.
- `>>> import sys`
- `>>> x=sys.stdin.readline()`
- `Welcome to cse dept>>>`
- `X`
- Output :'Welcome to cse dept'

- **Python Data Persistence - dbm Package**
- The dbm package presents a dictionary like interface DBM style databases. **DBM stands for DataBase Manager.** This is used by UNIX (and UNIX like) operating system. The dbbm library is a simple database engine written by Ken Thompson. These databases use binary encoded string objects as key, as well as value.
- The database stores data by use of a single key (a primary key) in fixed-size buckets and uses hashing techniques to enable fast retrieval of the data by key.
- The dbm package contains following modules –

- **dbm.gnu(generally used database manager)** module is an interface to the DBM library version as implemented by the GNU project.
- **dbm.ndbm(new database manageer)** module provides an interface to UNIX ndbm implementation.
- **dbm.dumb(data base manager for unix)** is used as a fallback option in the event,when other dbm implementations are not found. This requires no external dependencies but is slower than others.

# GUI PROGRAMMING

- Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below.
- **Tkinter** – Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look this option in this chapter.
- **wxPython** – This is an open-source Python interface for wxWindows <http://wxpython.org>.
- **JPython** – JPython is a Python port for Java which gives Python scripts access to Java class libraries on the local machine <http://www.jython.org>

**Turtle** is a special feathers of Python. Using Turtle, we can easily draw in a drawing board.

First we import the turtle module.

Then create a window,

next we create turtle object

and using turtle method we can draw in the drawing board.

| METHOD       | PARAMETER  | DESCRIPTION                                                |
|--------------|------------|------------------------------------------------------------|
| Turtle()     | None       | It creates and returns a new turtle object                 |
| forward()    | amount     | It moves the turtle forward by the specified amount        |
| backward()   | amount     | It moves the turtle backward by the specified amount       |
| right()      | angle      | It turns the turtle clockwise                              |
| left()       | angle      | It turns the turtle counter clockwise                      |
| penup()      | None       | It picks up the turtle's Pen                               |
| pendown()    | None       | Puts down the turtle's Pen                                 |
| up()         | None       | Picks up the turtle's Pen                                  |
| down()       | None       | Puts down the turtle's Pen                                 |
| color()      | Color name | Changes the color of the turtle's pen                      |
| fillcolor()  | Color name | Changes the color of the turtle will use to fill a polygon |
| heading()    | None       | It returns the current heading                             |
| position()   | None       | It returns the current position                            |
| goto()       | x, y       | It moves the turtle to position x,y                        |
| begin_fill() | None       | Remember the starting point for a filled polygon           |
| end_fill()   |            |                                                            |

- Turtle is a Python library which used to create graphics, pictures, and games. It was developed by **Wally Feurzeig, Seymour Parpet** and **Cynthia Solomon** in 1967. It was a part of the original Logo programming language.
- Turtle is a pre-installed library in Python that is similar to the virtual canvas that we can draw pictures and attractive shapes. It provides the onscreen pen that we can use for drawing.
-

- Before working with the turtle library, we must ensure the two most essential things to do programming.
- **Python Environment** - We must be familiar with the working Python environment. We can use applications such as **IDLE** or **Jupiter Notebook**. We can also use the Python interactive shell.
- **Python Version** - We must have Python 3 in our system; if not, then download it from Python's official website.
-

- **import turtle**
- Now, we can access all methods and functions. First, we need to create a dedicated window where we carry out each drawing command. We can do it by initializing a variable for it.
- `s = turtle.getscreen()`
-

 Python Turtle Graphics

— □ ×



- **import** turtle
- **# Creating turtle screen**
- **s = turtle.getscreen()**
- **# To stop the screen to display**
- **turtle.mainloop()**

- **Programming with turtle**
- First, we need to learn to move the turtle all direction as we want. We can customize the pen like turtle and its environment. Let's learn the couple of commands to perform a few specific tasks.
- Turtle can be moved in four directions.
- Forward
- Backward
- Left
- Right

- **Turtle motion**
- The turtle can move forward and backward in direction that it's facing. Let's see the following functions.
- **forward(*distance*) or turtle.fd(*distance*)** - It moves the turtle in the forward direction by a certain distance. It takes one parameter **distance**, which can be an integer or float.
-

- **import** turtle
- # Creating turtle screen
- t = turtle.Turtle()
- # To stop the screen to display
- t.forward(100)
- turtle.mainloop()
-

Python Turtle Graphics

— □ ×



## **Example - 2:**

```
import turtle

Creating turtle screen
t = turtle.Turtle()

Move turtle in opposite direction
t.backward(100)

To stop the screen to display
turtle.mainloop()
```



Python Turtle Graphics



```
import turtle

Creating turtle screen
t = turtle.Turtle()

t.heading()

Move turtle in opposite direction
t.right(25)

t.heading()

To stop the screen to display
turtle.mainloop()
```

```
import turtle
Creating turtle screen
t = turtle.Turtle()
t.heading()
Move turtle in left
t.left(100)
t.heading()
To stop the screen to display
turtle.mainloop()
```



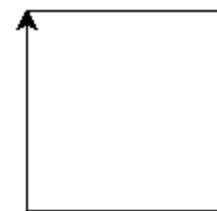
Python Turtle Graphics



- Drawing a Shape
- We discussed the movement of the turtle. Now, we learn to move on to making actual shape. First, we draw the **polygon** since they all consist of straight lines connected at the certain angles. Let's understand the following example.
- **Example -**
- t.fd(100)
- t.rt(90)
- t.fd(100)
- t.rt(90)
- t.fd(100)
- t.rt(90)
- t.fd(100)



Python Turtle Graphics



- **import** turtle
- # Creating turtle screen
- t = turtle.Turtle()
- t.circle(50)
- turtle.mainloop()



Python Turtle Graphics

- □ ×

