रा.इ.सू.प्रौ.सं NIELIT

Contents to be covered

- Introduction
- Features of Python
- Applications
- Difference between C and Python
- Difference between Java and Python
- Byte Code
- Memory Management in Python
- Garbage Collection in Python

Introduction

- Python is a popular programming language.
- It combines the features of C and Java.
- It was created by Guido van Rossum, and released in 1991.
- The name "Python" was adopted from the Rossum's favourite comedy series "Monty Python's Flying Circus".
- Python is mainly interpreted language.
- It is an open source software.
- It is used for web development (server-side), software development, mathematics, system scripting etc.

Introduction Contd..

- 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.

Features of Python

The following are some of the important features of

Simple: It is a simple programming language. When we read a Python program, we feel like reading English sentences.

Easy to learn: It uses very few keywords. Its programs use very simple structure. So, developing programs in Python become easy.

Open Source: There is no need to pay for Python software. It can be freely downloaded from www.python.org. website.

Dynamically Typed: In python, we need not declare anything. An assignment statement binds a name to an object, and the object can be of any type.

Features of Python Contd..

Platform independent: Python program are not dependent on any specific operating system, we can use Python almost all operating system like Unix, Linux, Windows, Macintosh etc.

Portable: When a program yields the same result on any computer in the world, then it is called a portable program. Python programs will give the same result since they are platform independent.

Procedure and object oriented: Python is a procedure oriented as well as an object oriented programming language.

Features of Python Contd..

Huge Library: It has a big library which can be used on any Operating System. Programmers can develop programs easily using the modules available in the Python Library.

Database Connectivity: Python provides interfaces to connect its programs to all major databases like Oracle, Sybase or MySQL.

Application of Python

- General purpose language: Used to create Machine learning, Web applications/development, GUI, Software development.
- Used alongside software to create workflows.
- Connect to database systems. It can also read and modify files.
- Can be used to handle big data and perform complex mathematics.
- Can be used for rapid prototyping, or for production-ready software development.
- Top companies using Python: Google, Dropbox, Youtube, Quora, Yahoo, NASA, Reddit

Flavors of Python

Flavors Name	Descriptions
CPython	This is the standard Python Compiler implemented in C language.
Jython	This is earlier known as **JPython**. This is the implementation of Python programming language which is designed to run on Java.
IronPython	This is the implementation of Python language for .NET framework.
PyPy	This is the Python implementation using Python language. PyPy is written in a language called **RPython** which was created in Python language.
AnacondaPython	When Python is redeveloped for handling large scale data processing, predictive analytics and scientific computing, it is called Anaconda Python. This implementation mainly focus on large scale of data.

Difference Between C and Python

C	Python
Procedure-oriented language	Object oriented language
Compiled language	Interpreted language
Saved with .c extension	Saved with .py extension
Variables are declared in C	No need of declaration
Pointers are available	No pointers functionality
Limited number of built-in functions	large library of built-in functions
Does not have complex data structures	Have some complex data structures
Statically typed	Dynamically typed
Syntax of C is complex	Simple, easy to learn, write and read
Faster	Slower

Difference Between Java and Python

Java	Python
Compiled+ interpreted Language.	It is an Interpreted Language
Statically typed	Dynamically typed
Complex learning curve	Python is easy to learn and use
It is verbose. It means they contain more number of lines	Programs are concise and compact
It is multi-platform, object-oriented Language.	High-level object-oriented programming language.
It uses curly braces to define the beginning and end of each function and class definition.	It uses indentation to separate code into separate blocks.
Multiple inheritances is partially done through interfaces.	Supports both single and multiple inheritances.
Limited string related functions.	Lots of string related functions.
Java is faster.	It is slower because python is an interpreter
Desktop GUI apps, Embed Systems, Web application.	Excellent for scientific and numeric computing, ML.

राइसूप्रौसं NIELIT

Byte Code

- Python is usually called an interpreted language, however, it combines compiling and interpreting. When we execute a source code (a file with a .py extension),
- Python first compiles it into a byte code.
- The byte code is a low-level platform-independent representation of your source code, however, it is not the binary machine code and cannot be run by the target machine directly.
- In fact, it is a set of instructions for a virtual machine which is called the Python Virtual Machine (PVM).

रा.इ.सू.प्रौ.सं
NIELIT

Byte Code (Contd...)

- After compilation, the byte code is sent for execution to the PVM.
- The PVM is an interpreter that runs the byte code and is part of the Python system.
- The byte code is platform-independent, but PVM is specific to the target machine.
- The default implementation of the Python programming language is CPython which is written in the C programming language.
- CPython compiles the python source code into the byte code, and this bytecode is then executed by the CPython virtual machine.

Memory Management in Python

- Memory allocation and de-allocation are done during run time automatically.
- The programmer need not allocate memory while creating objects or de-allocate memory when deleting the objects.
- Python PVM will take care of such issues.
- Everything is considered as an object in Python. For every object memory should be allocated.
- Memory manager inside the PVM allocates memory required for objects created in a python program.
- All these objects are stored on a separate memory called ***heap***.
- ***Heap*** is the memory which allocated during run time.
- The size of the ***heap*** memory depends on the RAM of our computer and it can increase or decrease.

रा.इ.सू.प्रौ.सं NIELIT

Garbage Collection in Python

- A module represents Python code that performs a specific task. Garbage collector is a module in Python that is useful to delete objects from memory which are not used in the program.
- The module that represents the garbage collector is named as ***gc***.
- Garbage collector in the simplest way to maintain a count for each object regarding how many times that object is referenced (or used).
- When an object is referenced twice, its reference count will be 2. when an object has some count, it is being used in the program and hence garbage collector will not remove it from memory.

Garbage Collection in Python Contd..

- When an object is found with reference count 0, garbage collector will understand that the object is not used by the program and hence it can be deleted from memory.
- Garbage collector can detect reference cycles. A reference cycle is a cycle of references pointing to the first object from last object.

Contents to be covered

- Python Virtual Machine (PVM)
- Installation
- Execution of program

Internal Working of Python

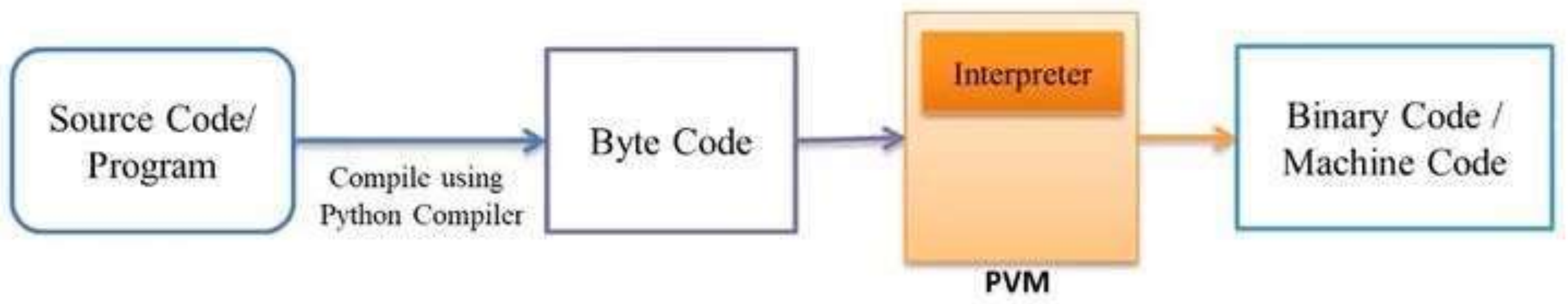
- Python uses code modules that are interchangeable instead of a single long list of instructions that was standard for functional programming languages.
- The standard implementation of python is called “cpython”. It is the default and widely used implementation of the Python.
- Python doesn’t convert its code into machine code, something that hardware can understand.
- It is into byte code and this byte code can’t be understood by CPU. So we need actually an interpreter called the python virtual machine

Internal Working of Python

- The python virtual machine executes the byte codes.
- **PVM** is nothing but a software/interpreter that converts the byte code to machine code for given operating system.
- **PVM** is also called **Python** Interpreter and this is the reason **Python** is called an Interpreted language.

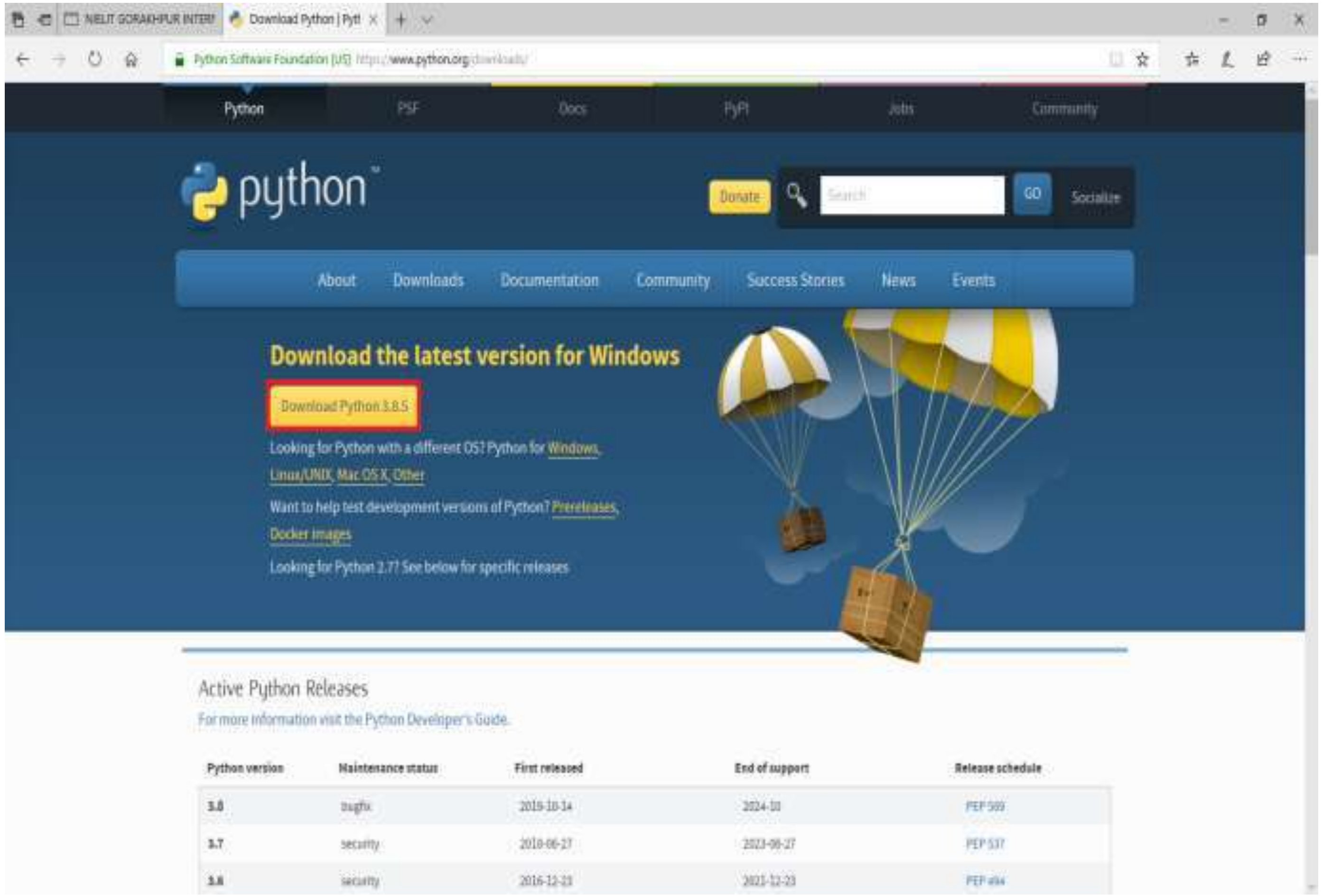
Python Virtual Machine

- Python file first get compiled to give us byte code and that byte code is interpreted into machine language.
- This is performed by PVM.
- Execution:



Installation of Python

- Go to: www.python.org/downloads
- Click the “download python” button, save it and run.
- Follow the step by step process of installation wizard.



Installation of Python (Contd..)

- An **Open File - Security Warning** pop-up window will appear.
- Click **Run**. A **Python 3.8.5 Setup** pop-up window will appear.

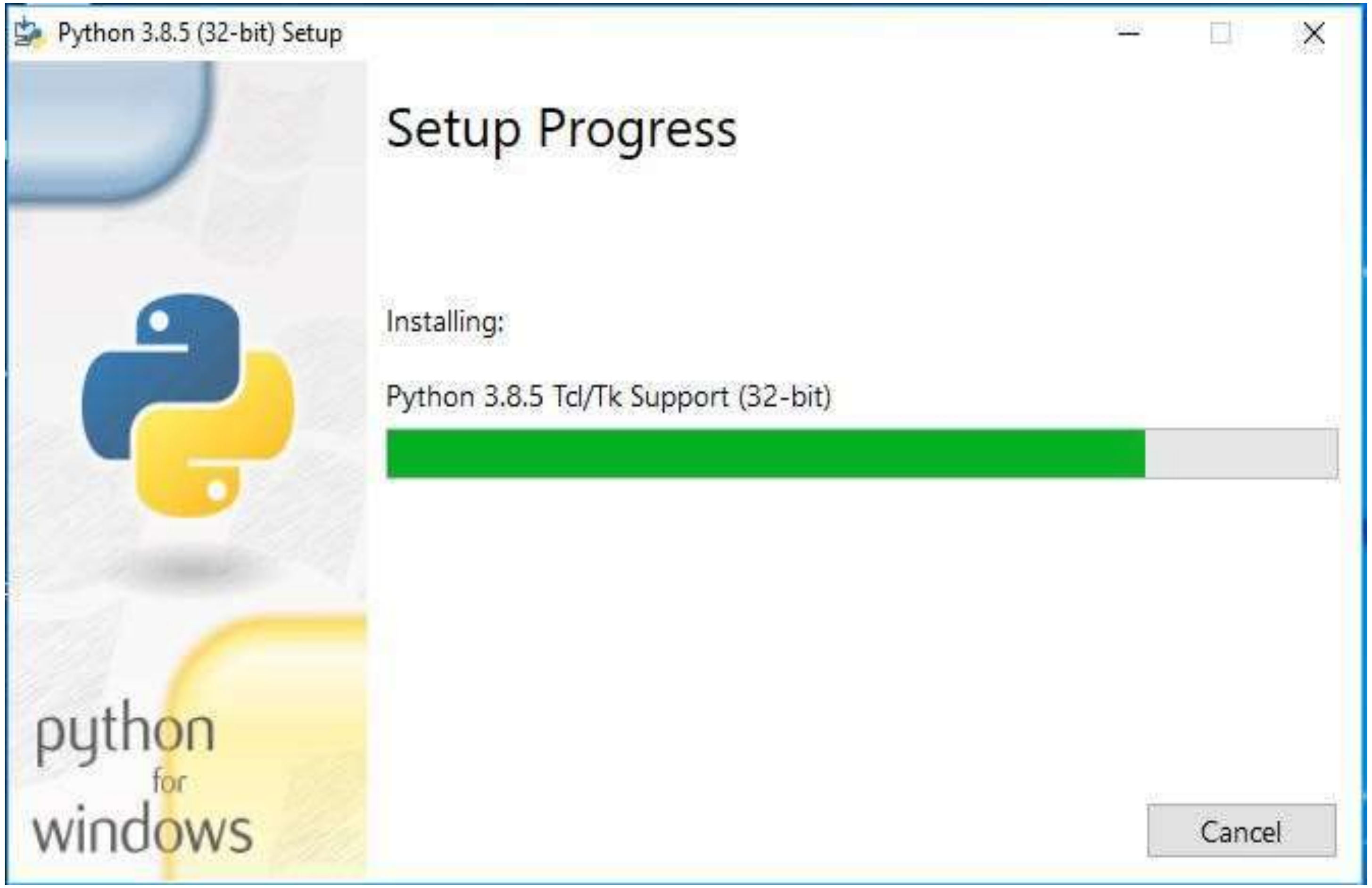
रा.इ.सू.प्रौ.सं
NIELIT

Installation of Python (Contd..)

- Ensure that the **Install launcher for all users (recommended)** and the **Add Python 3.8 to PATH** checkboxes at the bottom are checked.
- Highlight the **Install Now** (or **Upgrade Now**) message, and then click it.

Installation of Python (Contd..)

- A new **Python 3.8.5 Setup** pop-up window will appear with a **Setup Progress** message and a progress bar.



Installation of Python (Contd..)

- Pop-up window will appear with a **Setup was successfully** message.
- Click the **Close** button.
- Python should now be installed.

Executing a Python Program

There are three ways of executing a python program.

- ❑ Using Python's Command line window
- ❑ Using Python IDLE graphics window
- ❑ Directly from System prompt

The first two are called interactive modes where we can type the program one line at a time and the PVM executes it immediately.

If you execute complete program at a time this one is called non-interactive mode where the PVM executes program after typing the entire program.

Using Python’s Command line window

- ❑ Open the Python command line window
- ❑ >>> (symbol) which is called Python prompt
- ❑ Type program at the >>> prompt

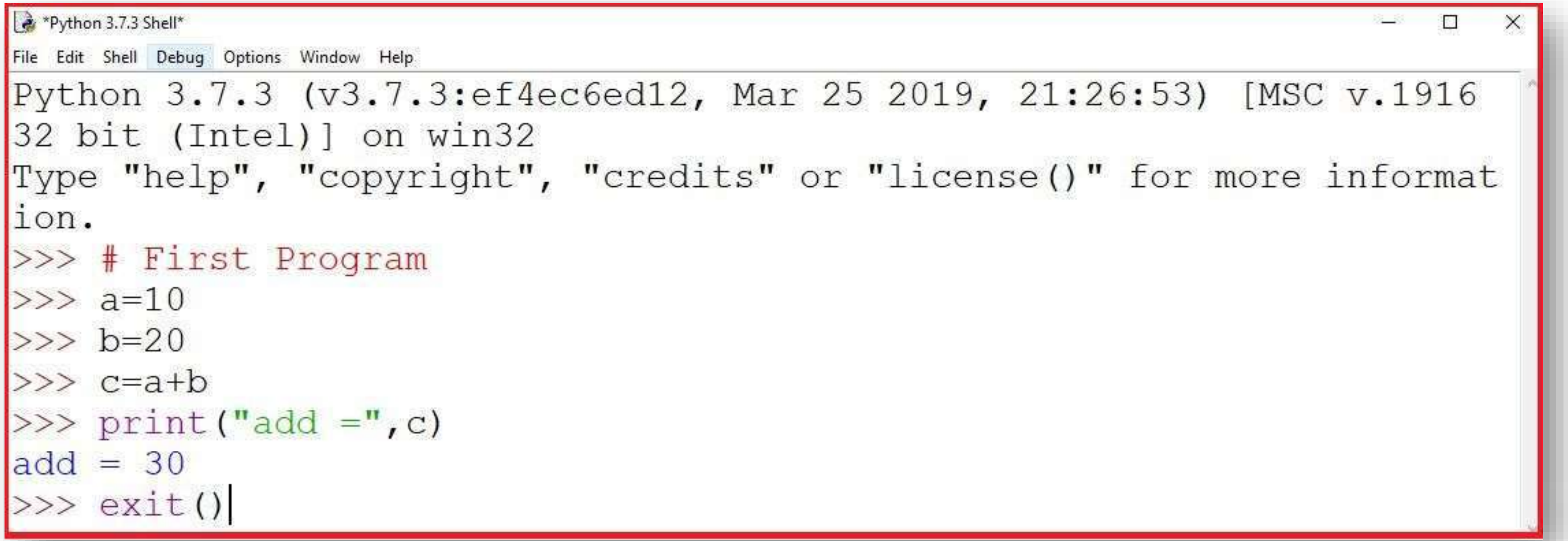
Python 3.7 (32-bit)

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> # First Program
... a=10
>>> b=20
>>> c=a+b
>>> print("add =",c)
add = 30
>>> exit()
```

- ❑ After typing the last line and pressing enter button, it display the result. After that, type exit() or quit() to close Python command prompt.

Using Python IDLE graphics window

- ❑ Click the Python’s IDLE (Integrated Development Environment) window
- ❑ Type program :



```
*Python 3.7.3 Shell*
File  Edit  Shell  Debug  Options  Window  Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916
32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more informat
ion.
>>> # First Program
>>> a=10
>>> b=20
>>> c=a+b
>>> print("add =",c)
add = 30
>>> exit()
```

- ❑ To terminate the IDLE window, type exit() or quit(). It will display a message as to kill the program or not.

Directly from System prompt

- ❑ Open a text editor and type the program
- ❑ Save the program by clicking File → Save As, type the program name with extension .py
- ❑ Open the command prompt
- ❑ Go to that directory where the program is saved
- ❑ Execute the program by calling the python command

  c:\xyz>python pro_name.py

रा.इ.सू.प्रौ.सं NIELIT

Contents to be covered

- Variable
- Literals, Constants, Identifiers
- Keywords
- Operators
- Data types
- Naming Conventions with examples
- Basic programming Examples

Variables

- A variable is a name which is used to refer memory location and used to hold value.
- In Python variable is also called as an **identifier**.

Rules of Python variables:

- A variable name must start with a letter or the underscore character.
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z,0-9, and _)
- Variable names are case sensitive.
- Example:

```
x=10
y=20
```

Literal

- Literals can be defined as a data that is given in a variable or constant.
- Python Support the following literals:

String Literals

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes for a string. A string literal can also span multiple lines.

first.py - C:\Users\Nielit-042\AppData\Local\Programs\Python\Python38-32\first.py (3.8.5)

File Edit Format Run Options Window Help

```
x='Hello'
y="Welcome to NIELIT"
z="""Web Development
  using Python & Django"""

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

Literal

Numeric Literals

Numbers stores numeric values. Python create number objects when a number is assigned to a variable.

Numeric literals can belong to following four different numerical types.

- **int** (Singed integer)
- **long** (Long integers used for higher range)
- **float** (Float is used to store floating point numbers)
- **complex** (Complex number)

Boolean Literals

A Boolean literal can have any of the two value: **True** or **False**

Special Literals

Python contain one special literal **None.** It is used to specify to that field that is not created. It is also used for end of lists in Python

रा.इ.सू.प्रौ.सं NIELIT

Keywords

- Python keywords are special reserved words which convey a special meaning to the compiler/interpreter.
- Each keyword has a special meaning and a specific operation.
- Keywords can not be used as a variable.

List of Python Keywords

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Operators & Operand

- Operators are special symbols which represents computation.
- They are applied on operand(s), which can be values or variables. Operators when applied on operands form an expression.
- Operators are categorized as Arithmetic, Relational, Logical and Assignment.
- Value and variables when used with operator are known as operands.

Mathematical/Arithmetic Operators

- Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.
- **Assume a=5 and b=3**

Operator	Meaning	Example	Result
+	Addition Operator	a+b	8
-	Subtraction Operator	a-b	2
*	Multiplication Operator	a*b	15
/	Division Operator	a/b	1.666
%	Modulus Operator	a%b	2
**	Exponent Operator	a**b	125
//	Floor Division	a//b	1

Assignment Operators

- These operator are useful to store the right side value into a left side variable
- Assume a=20, y=10 and z=5

Operator	Meaning	Example	Result
=	Assignment Operator		
+=	Addition Assignment Operator	z+=x	z=25
-=	Subtraction Assignment Operator	z-=x	z=-15
*=	Multiplication Assignment Operator	z*=x	z=100
/=	Division Assignment Operator	z/=x	z=0.25
%=	Modulus Assignment Operator	z%=x	z=5
**=	Exponentiation Assignment Operator	Z**=y	z=9765625

Relational Operators

- Relational operators compares the values.
- It either returns **True** or **False** according to the condition.

Symbol	Description	Example 1	Example 2

<	Less than	>>>7<10
		True
		>>> 7<5
		False
		>>> 7<10<15
		True
		>>>7<10 and 10<15
		True
			>>>‘Hello’< ‘Goodbye’
			False
			>>>'Goodbye'< 'Hello'
			True

>	Greater than	>>>7>5 True >>>10<10 False	>>>‘Hello’> ‘Goodbye’ True >>>'Goodbye'> 'Hello' False
<=	less than equal to	>>> 2<=5	>>>‘Hello’<= ‘Goodbye’

Relational Operators

		True >>> 7<=4 False	False >>>'Goodbye' <= 'Hello' True
>=	greater than equal to	>>>10>=10 True >>>10>=12 False	>>>'Hello'>= 'Goodbye' True >>>'Goodbye' >= 'Hello' False
! =, <>	not equal to	>>>10!=11 True >>>10!=10 False	>>>'Hello'!= 'HELLO' True >>> 'Hello' != 'Hello' False
==	equal to	>>>10==10 True >>>10==11 False	>>>'Hello' == 'Hello' True >>>'Hello' == 'Good Bye' False

Logical Operators

- In the case of logical operators, False indicates 0 and True indicates any other number.

Operator	Example	Meaning
And	x and y	If x is true then return y, otherwise x
Or	x or y	If x is true then return x, otherwise y
Not	not x	If x is true then return false

```
x=1
y=2
z=x and y
print(z)
z=x or y
print(z)
z=not x
print(z)
```

Output

2

1

False

Logical Operators Contd..

```
x=1
y=2
z=x and y and x
print(z)
z=x and y or x
print(z)
z=not not x
print(z)
```

Output

```
1
2
True
```

Boolean Type

- There are two bool type literals. They are True & False

```
x=False
y=True
z=x and y
print(z)
z=x or y
print(z)
z=not x
print(z)
```

Operator	Example	Meaning
And	x and y	If both x and y are true then it return True otherwise False
Or	x or y	If either x or y is true then return True otherwise False
Not	not x	If x is true then return false

Output

False
True
True

Precedence of operator:

- Listed from high precedence to low precedence

Operator	Description
**	Exponentiation (raise to the power)
+ , -	unary plus and minus
* , /, %, //	Multiply, divide, modulo and floor division
+ , -	Addition and subtraction
<, <=, >, >=	Comparison operators
==, !=	Equality operators
% =, / =, // = , -=, + =, * =	Assignment operators
not and or	Logical operators

Special Operator:

- **Identity operators:**
  - is and is not are the identity operators in Python.
  - They are used to check if two values (or variables) are located on the same part of the memory.
  - Two variables that are equal does not imply that they are identical.

- **Membership operators:**
  - in and not in are the membership operators in Python.
  - They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

रा.इ.सू.प्रौ.सं NIELIT

Identity Operator:

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

```
a=10
b=10
c=a is b
print(c)
print(id(a))
print(id(b))
```

Output

True
140735281390512
140735281390512

Identity Operator:

```
#for Array
from array import*
a=array('i',[1,2,3])
b=array('i',[1,2,3])
c= a is b
print(c)
print(id(a))
print(id(b))

#for list
x=[1,2]
y=[1,2]
z= x is y
print(z)
print(id(x))
print(id(y))
```

Output

```
False
1692318410608
1692318410672
False
1692313475784
1692313474888
```

Membership Operator:

- x='Hello Python'
- y=2
- z=[1,2,3,4]
- print('H' in x)
- print('y' in x)
- print('p' not in x)
- print('h' not in x)
- print(y in z)
- print(5 in z)
- print(2 not in z)

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Output:
True
True
True
False
True
False
False

Bitwise Operator:

- Bitwise operators acts on bits and performs bit by bit operation.
- For example, 2 is 0000 0010 in binary and 7 is 0000 0111

Operator	Meaning	Example
&	Bitwise AND	z=x &y **Output:** z=2 binary equivalent is (0000 0010)
|	Bitwise OR	z=x | y **Output:** z=7 binary equivalent is (0000 0111)
^	Bitwise XOR	z=x ^y **Output:** z=5 binary equivalent is (0000 0101)
~	Bitwise NOT	z=~y **Output:** z=8 binary equivalent is (1111 1000)
>>	Bitwise Right Shift	z=y>>1 **Output:** z=3 binary equivalent is (0000 0011)
<<	Bitwise Right Shift	z=y<<1 **Output:** z=14 binary equivalent is (0000 1110)

Example of Bitwise Operator:

```
x=2        #Binary Equivalent of 2 is 0010
y=7        #Binary Equivalent of 7 is 0111
z=x&y
print(z)
z=x|y
print(z)
z=x^y
print(z)
z=~y
print(z)
```

Output

2
7
5
-8

Example of Bitwise Operator:

```
x=2            #Binary Equivalent of 2 is 0010
y=7            #Binary Equivalent of 7 is 0111
```

Right Shift

```
z=y>>1
print(z)
z=y>>2
print(z)
z=y>>3
print(z)
```

Output

```
3
1
0
```

Left Shift

```
z=y<<1
print(z)
z=y<<2
print(z)
z=y<<3
print(z)
```

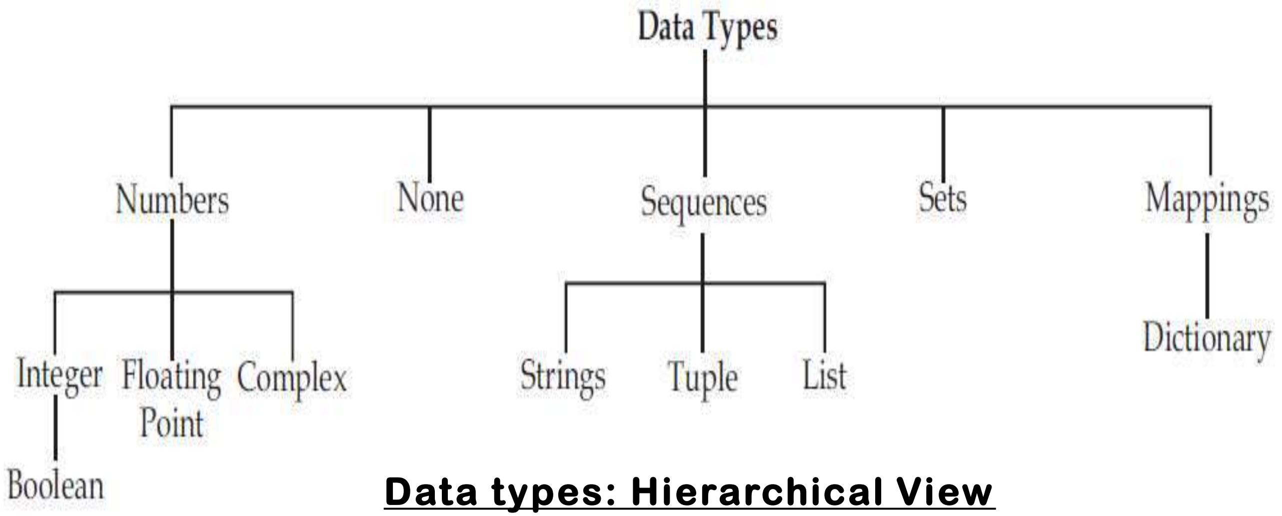
Output

```
14
28
56
```

Data Types in Python

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

- None type
- Numeric type
- Sequences
- Sets
- Mapping



Data types: Hierarchical View

Numeric Types

Integers, floating point numbers and complex numbers falls under Python numbers category. They are defined as int, float and complex class in Python.

- int
- float
- complex

We can use the **type()** function to know which class a variable or a value belongs to and the **isinstance()** function to check if an object belongs to a particular class.

Examples Numeric Types

```
a = 2
print(a, "is of type", type(a))
a = 2.0
print(a, "is of type", type(a))
a = 1+2j
print(a, "is complex number?", isinstance(1+2j,int))
```

- Integers can be of any length, it is only limited by the memory available.
- A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is integer, 1.0 is floating point number.
- Complex numbers are written in the form, x + yj, where *x* is the real part and *y* is the imaginary part.

रा.इ.सू.प्रौ.सं NIELIT

Converting the Datatypes

Sometimes, we want to convert one datatype into another. This is called type conversion or coercion. For this purpose mention the datatype with parenthesis

Ex: **int(x) is used to convert it is in integer type**

```
x=15.56
int(x)                # will display 15
```

float(num) is used to convert it is in float type

```
num=15
float(num)            #will display 15.0
```

complex(n) is used to convert it is in complex type

```
n=10
complex(n)            #will display (10+0j)
```

bool Datatypes

The bool datatype in python represents boolean values. There are only two boolean value True or False that can be represented by this datatype. A blank string like “” is also represented as False.

Ex:

```
a=10>5
print(a)        # display True
a=6>10
print(a)        # display False
```

Contents to be covered

- Input Function
- Output Function

Input Function

- To accept input from keyboard, Python provides the **input()** function. This function takes a value from the keyboard and returns it as a string.

Example:

```
str=input('enter you city: ')
print(str)

str=input('enter a number: ')
x=int(str)
print(x)

x=int(input('enter any no.: '))
print(x)

x=float(input('enter any no.: '))
print(x)
```

Output:

```
enter you city: gorakhpur
gorakhpur
enter a number: 4
4
enter any no.: 54
54
enter any no.: 37.5
37.5
```

Output Function

print() function is used to output data to the standard output device (screen).

```
print('This sentence is output to the screen')
```

 # Output: This sentence is output to the screen

```
a = 5
print('The value of a is', a)
```

 # Output: The value of a is 5

Examples of Output Function

```
print('hello')
print('hello \tPython')
print('hello '*3)
print('hello'+'Python')

a,b=2,4
print(a)
print(a,b)

print(a,b,sep=",")
print(a,b,sep=":")
print(a,b,sep="____")

print("hello")
print("welcome",end="")
print("python")
print(1,2,3,sep=",",end="&")
```

```
hello
hello    Python
hello hello hello
helloPython
2
2 4
2,4
2:4
2____4
hello
welcomepython
1,2,3&
```

Program to sum of two numbers

```
a = int(input("enter first number: "))
b = int(input("enter second number: "))
sum= a + b
print(“Sum of two numbers is:", sum)
```

Problems

- The length & breadth of a rectangle and radius of a circle are input through the keyboard.
- Write a program to calculate the area & perimeter of the rectangle, and the area & circumference of the circle.
- Employee's basic salary is input through the keyboard. His dearness allowance is 40% of basic salary, and house rent allowance is 20% of basic salary. Write a program to calculate his gross salary.
- If the marks obtained by a student in five different subjects are input through the keyboard, find out the aggregate marks and percentage marks obtained by the student. Assume that the maximum marks that can be obtained by a student in each subject is 100.
- The distance between two cities (in km.) is input through the keyboard. Write a program to convert and print this distance in meters, feet, inches and centimeters.
- Two numbers are input through the keyboard into two locations a and b. Write a program to interchange the contents of a and b.

Contents to be covered

- Control Statement
- Concept of Indentation
- if, if...else, if…elif…else statement with examples

Control Statement

- **Control statements** decides the direction of flow of program execution.

- **Decision making:**
  - if statement
  - if...else statement
  - if…elif…else statement

The if statement

- It is used to execute one or more statement depending upon whether condition is true or not.
- **Syntax**:-

```
if condition:                    num=1
    statements                   if num==1:
                                     print('one')
```

Identation

- In Python, the body of the if statement is indicated by the indentation. Body starts with an indentation and the first un-indented line marks the end.
- It refers to spaces that are used in the beginning of a statement. The statements with same indentation belong to same group called a suite.
- By default, Python uses 4 spaces but it can be increased or decreased by the programmers.

```
if x==1:
        print('a')
        print('b')
        If y==2:
        print('c')
        print('d')
        print('end')
print (''end)
```

The if..else statement

- The if..else statement evaluates test expression and will execute body of if only when test condition is True. If the condition is False, body of else is executed. Indentation is used to separate the blocks.

- **Syntax**:-

```
If condition:
        Satement1
else:
        Statement2
```

Example:

```
num = 3
if num >= 0:
        print("Positive or Zero")
else:
         print("Negative number")
```

राइसूप्रौसं NIELIT

if...elif...else Statement

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

Syntax

```
if condition1:
        Statement1
elif condition2:
        Statement2
elif condition3:
        Statement3
else:
        Body of else
```

Example:

```
num = 3.4
 if num > 0:
        print("Positive number")
elif num == 0:
        print("Zero")
else:
        print("Negative number")
```

Problem

- WAP to input the number and check it is even or odd.
- WAP to check a number is positive or negative.
- WAP to check greatest among two numbers.
- WAP to check a year leap year or not.
- WAP to check greatest among three numbers.
- WAP to check a three digit number is palindrome or not.
- WAP to input the cost price and selling price of an item and check for profit or loss. Also calculate it.

Contents to be covered

- While loop
- For loop
- range() function
- break and continue

Loop

- 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.
- For this purpose, The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times.
- The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times.

Advantages Loops

There are the following advantages of loops in Python.

- It provides code re-usability.
- Using loops, we do not need to write the same code again and again.
- Using loops, we can traverse over the elements of sequence data types. (array, list, tuple, set and dictionary).

While Loop

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

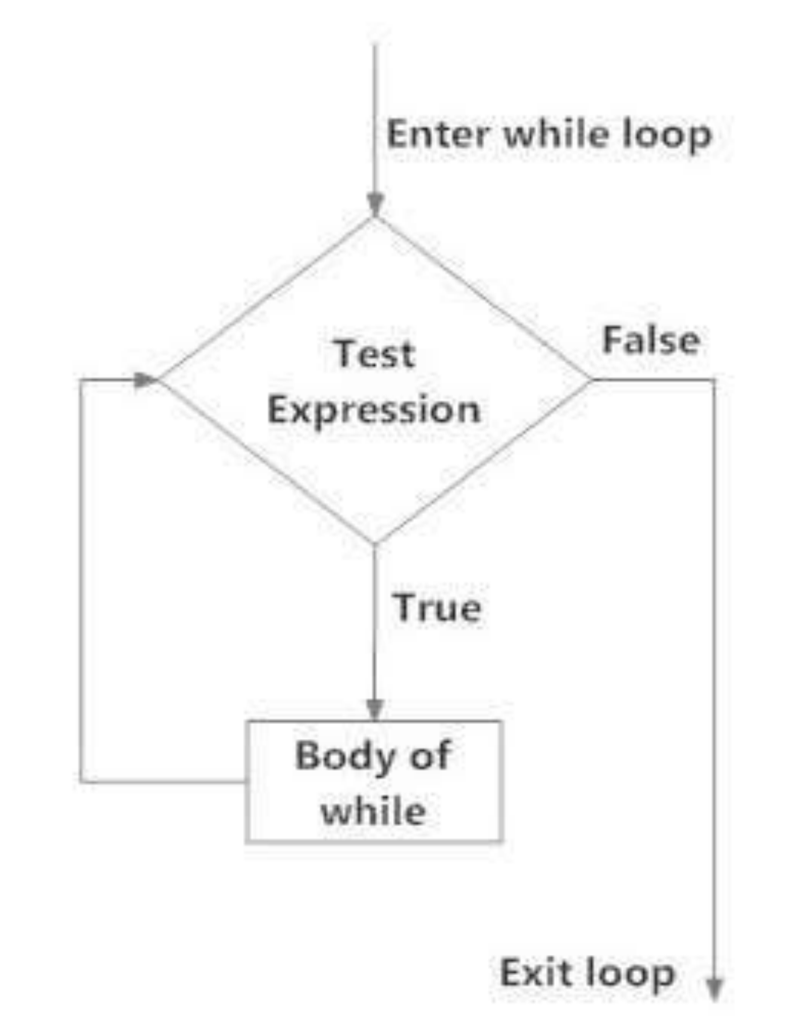
```
while condition:
        Body of while
```

- **Working:**

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

राइसूप्रौसं NIELIT

Flow Chart:



Example:

```
n = 10
sum = 0
i = 1
while i <= n:
        print(i)
        sum = sum + i
        i = i+1
print("The sum is", sum)
```

Output:

```
1
2
3
4
5
6
7
8
9
10
The sum is 55
```

राइसूप्रौसं NIELIT

While loop with else

- We can also use an optional else block with while loop.
- The else part is executed if the condition in the while loop evaluates to False.

Example:

```
n = 10
sum = 0
i = 1
while i <= n:
        print(i)
        sum = sum + i
        i = i+1
else:
    print('End of loop')
print("The sum is", sum)
```

Output:

```
1
2
3
4
5
6
7
8
9
10
End of loop
The sum is 55
```

The for loop

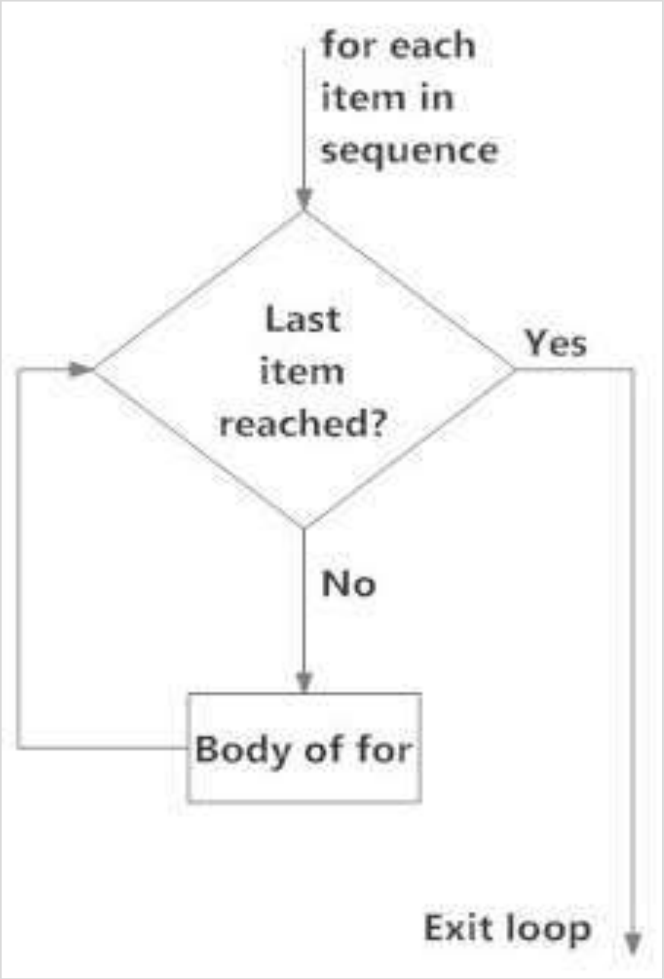
- The for loop is used to iterate (repeat) over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.
- Syntax:

```
for val in sequence:
        Body of for
```

(val is the variable that takes the value of the item inside the sequence on each iteration)

- Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Flow Chart:



Example:

```
str='Hello'
for ch in str:
    print(ch)
```

Output:

```
H
e
l
l
o
>>>
```

रा.इ.सू.प्रौ.सं NIELIT

for loop with else

- A for loop can have an optional else block. The else part is executed if the items in the sequence used in for loop exhausts.
- break statement can be used to stop the loop. In such case, the else part is ignored.
- Hence, a for loop's else part runs if no break occurs.

Example:

```
for x in range(10,0,-1):
    print(x)
else:
    print('out of for loop')
```

Output:

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

रा.इ.सू.प्रौ.सं NIELIT

Range () Function

- It is used to provide a sequence of numbers.
- range(n) gives numbers from 0 to n-1.
- For example, range(10) returns numbers from 0 to 9.
- For example, range(5, 10) returns numbers from 5 to 9.
- We can also define the start, stop and step size as **range(start, stop, step size)**, step size defaults to 1 if not provided.
- Step size represents the increment in the value of the variable at each step.

Example:

```
for i in range(1,10,2):
    print(i)
```

Output:

```
1
3
5
7
9
```

Example:

```
for x in range(10,0,-1):
    print(x)
```

Output:

```
10
9
8
7
6
5
4
3
2
1
```

Break statement

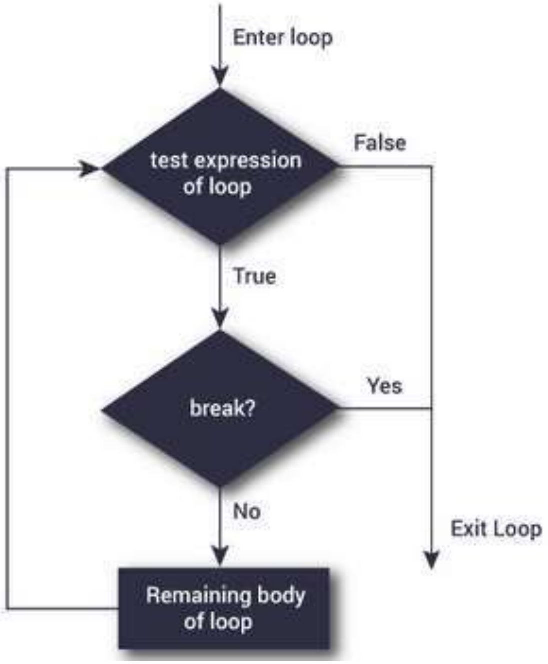
- break and continue statements can alter the flow of a normal loop.

  break statement:-

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

```
break
```

Flow Chart:



Working:

```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

Continue statement

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

Problem

- Program to print numbers between 1 to 10 by using while and for loop.
- Program to print even and odds numbers between two values input by user.
- Program to check whether a number is prime or not.
- Program to print all prime numbers between two values input by user.
- Program to print Fibonacci Series upto 10 terms.
- Program to check whether a number is palindrome or not.
- Program to check whether a number is Armstrong or not.

राइसूप्रौसं NIELIT

Contents to be covered

- Nesting of while loop
- Nesting of for loop

Nesting of Loop

- Python programming language allows the usage of one loop inside another loop.
- A nested loop is a loop inside a loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop”.

Syntax for Nested While loop

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

Example of Nested While loop

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

Syntax for Nested for loop

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

राइसूप्रौसं NIELIT

Example of Nested for loop

```
for i in range(1,11):
    for j in range(1,11):
        k = i*j
        print (k, end=' ')
    print()
```

- The print() function inner loop has **end=' '** which appends a space instead of default newline. Hence, the numbers will appear in one row.
- Last print() will be executed at the end of inner for loop.

रा.इ.सू.प्रौ.सं
NIELIT

Problem

- Program to print Factorial value b/w 1 to 100.
- Program to print all the prime number b/w 1 to 300.
- Program to print all the palindrome number b/w 1 to 500.

Contents to be covered

- Concept of Array and its operations
- Handling Strings and Characters

Array

- An array is an object that stores a group of elements (or values) of same datatype.
- The size of the array is not fixed in python. Hence, we need not specify how many elements we are going to store into an array in the beginning.
- Arrays can grow or shrink in memory dynamically (during runtime).
- **Creating an Array:**

arrayname=array(type code, [elements])

a=array('i', [4,6,2]) -*integer type array*

arr= array('d',[1.5,-2.2,3,5.75]) -*double type array*

Importing the Array Module

- import array

  a= array.array('i',[4,6,2,9])

- Import array as ar

  a=ar.array('i',[4,6,2,9])

- from array import*

  a=array('i',[4,6,2,9])

  (* symbol represents all)

Example:

Output:

```
#Program to create an integer type array

from array import *
a=array('i',[5,6,-7,8])
print('the array elements are:')
for element in a:
    print(element)
```

```
the array elements are:
5
6
-7
8
```

```
#Array of charcters

from array import *
arr=array('u',['a','b','c','d','e'])
print('the array elements are:')
for ch in arr:
    print (ch)
```

```
the array elements are:
a
b
c
d
e
```

Indexing & Slicing on Array

- An index represents the position number of an element in an array. For example, the following integer type array:

X=array('i', [10, 20, 30, 40, 50])

10	20	30	40	50
X[0]	X[1]	X[2]	X[3]	X[4]

Allocates 5 blocks of memory each of 2 bytes of size and stores the elements 10, 20, 30, 40, 50.

Example:

```
#Create one array from another array

from array import *
arr1=array('d',[1.5, 2.5, -3.5, 4])
arr2=array(arr1.typecode,(a*3 for a in arr1))
print('The arr2 elements are:')
for i in arr2:
    print(i)
```

Output:

```
The arr2 elements are:
4.5
7.5
-10.5
12.0
```

The len(a) function returns the number of elements in the array 'a' into n.

```
from array import *
a = array('i', [10, 20, 30, 40, 50])
n=len(a)
for i in range (n):
    print (a[i], end='\t')
```

```
10	20	30	40	50
```

Type Codes to Create Array

Type code	C Type	Python Type	Minimum size in bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'f'	float	float	4
'd'	double	float	8

Example:

#Accessing array elements

```
from array import *
a = array('i', [2, 4, 6, 8])
print("First element:", a[0])
print("Second element:", a[1])
print("Second last element:", a[-1])
```

Output:

```
First element: 2
Second element: 4
Second last element: 8
```

#Slicing Arrays

```
from array import *
a = array('i', [10, 20, 30, 40, 50, 60, 70, 80])
print("Element:", a[1:4])
print("Element:", a[0:])
print("Element:", a[:4])
print("Element:", a[-4:])
print("Element:", a[-4:-1])
print("Element:", a[0:7:2])
```

```
Element: array('i', [20, 30, 40])
Element: array('i', [10, 20, 30, 40, 50, 60, 70, 80])
Element: array('i', [10, 20, 30, 40])
Element: array('i', [50, 60, 70, 80])
Element: array('i', [50, 60, 70])
Element: array('i', [10, 30, 50, 70])
```

Change or add elements in the array

```
from array import *

numbers = array('i', [1, 2, 3, 5, 7, 10])

# changing first element
numbers[0] = 0
print(numbers)      # Output: array('i', [0, 2, 3, 5, 7, 10])

# changing 3rd to 5th element
numbers[2:5] = array('i', [4, 6, 8])
print(numbers)      # Output: array('i', [0, 2, 4, 6, 8, 10])
```

```
array('i', [0, 2, 3, 5, 7, 10])
array('i', [0, 2, 4, 6, 8, 10])
```

We can add one item to a list using append() method or add several items using extend() method.

```
from array import *

numbers = array('i', [1, 2, 3])

numbers.append(4)
print(numbers)      # Output: array('i', [1, 2, 3, 4])

# extend() appends iterable to the end of the array
numbers.extend([5, 6, 7])
print(numbers)      # Output: array('i', [1, 2, 3, 4, 5, 6, 7])
```

```
array('i', [1, 2, 3, 4])
array('i', [1, 2, 3, 4, 5, 6, 7])
```

Insert at particular location

Example:

```python
# vowel list
vowel = ['a', 'e', 'i', 'u']

# inserting element to list at 4th position
vowel.insert(3, 'o')

print('Updated List: ', vowel)
```

Output:

```
Updated List:  ['a', 'e', 'i', 'o', 'u']
```

राइसूप्रौसं
NIELIT

Remove/delete element

We can delete one or more items from an array using Python's del statement.

Example:

```
from array import *

number = array('i', [1, 2, 3, 3, 4])

del number[2] # removing third element
print(number) # Output: array('i', [1, 2, 3, 4])

del number # deleting entire array
print(number) # Error: array is not defined
```

Output:

```
array('i', [1, 2, 3, 4])
Traceback (most recent call last):
  File "B:/python/programs/array9.py", line 9, in <module>
    print(number) # Error: array is not defined
NameError: name 'number' is not defined
```

We can use the remove() method to remove the given item, and pop() method to remove an item at the given index.

```
from array import *
numbers = arr.array('i', [10, 11, 12, 12, 13])
numbers.remove(12)
print(numbers)   # Output: array('i', [10, 11, 12, 13])

print(numbers.pop(2))   # Output: 12
print(numbers)   # Output: array('i', [10, 11, 13])
```

```
array('i', [10, 11, 12, 13])
12
array('i', [10, 11, 13])
```

Example:

```
#Program to store store student's marks into an array
#and find total marks and percentage of marks.

from array import *
str=input('Enter marks:').split(' ')
marks=[int(num) for num in str]
sum=0
for x in marks:
    print(x)
    sum+=x
print('Total marks: ', sum)
n=len(marks)
percent=sum/n
print('Percentage: ',percent)
```

```
Enter marks:60 50 70 80 45 60 90
60
50
70
80
45
60
90
Total marks:  455
Percentage:  65.0
```

NIELIT

```python
# To search for the element in the array

from array import *
x=array('i',[])
print('How many elements?', end=' ')
n=int(input())
for i in range (n):
    print('enter element :', end=' ')
    x.append(int(input()))

print('original array: ',x)
print('enter element to search: ',end=' ')
s=int(input())
flag=False
for i in range(len(x)):
    if s==x[i]:
        print('found at position=', i+1)
        flag=True
if flag==False:
    print('Not found in the array')
```

```
How many elements? 5
enter element : 5
enter element : 6
enter element : 7
enter element : 8
enter element : 4
original array:  array('i', [5, 6, 7, 8, 4])
enter element to search:  7
found at position= 3
```

रा.इ.सू.प्रौ.सं NIELIT

Problem

1. WAP to store student's marks (60,70,75,45 and 50) into an array and find total marks and percentage of marks.
2. Take an array of elements 5,10,15,20,25,30,35,40,45,50 and perform the following operations:
   a) Print the 3rd and 5th element.
   b) Slice the array into two arrays from 0 to 3 and 5 onwards.
   c) Change the element of 4th position by element 32
   d) Delete the 6th element
   e) Insert a new element 55 at last position of the array.
   f) Delete the element 40 from the array.
   g) Extend the array by elements 60,65,70 and 75.
   h) Insert a new element 18 at 4th position of the array.
3. Program to search for the position of an element in an array.

Contents to be covered

- List
- Tuple

List

- A sequence is a data types that represents a group of elements.
- The purpose of any sequence is to store and process a group of elements.
- In python, strings, lists, tuples and dictionaries are very important sequence data types.
- All sequence allow some common operations.

List Contd..

- A list is similar to an array that consists of a group of elements or items.
- Just like array, a list can store elements.
- But, there is one major difference between an array and a list.
- An array can store only one type of elements whereas a list can store different types of elements.
- Hence lists are more versatile and useful than an array.
- Perhaps lists are the most used data types in python programming.

Student = [10, ‘venu gopal’, ‘M’, 50, 55, 62, 74, 66]

List Contd..

- Please observe that the elements of the student list are stored in square bracket [ ].
- We can create an empty list without any elements by simply writing empty square braces as:

```
e_lst= [        ]                    #this an empty list

print(student[1])                    # Output : venu gopal
```

List Contd..

One of the most useful data structures that you will find are lists.

List can contains both number and string and can hold many of it.

#List with numbers

lst1 = [1,2,3,4]

#List with strings

lst2 = [‘string’ , ’can’ , ’go’ , ’another string’]

#List with strings and numbers

lst3 = [1,2, ‘string’ , ‘another string’]

List Contd..

#Print each out

```
print(lst1)
print(lst2)
print(lst3)
```

Outputs:

[1,2,3,4]

[‘string’ , ’can’ , ’go’ , ’another string’]

[1,2, ‘string’ , ‘another string’]

राइसूप्रौसं NIELIT

List Contd..

Since, we just introduced a list, One of the most useful part of list is that they are iterable. An iterable is a collection of data that you can move through using a for loop.

```
#for loop through a list of numbers
    lst = [1,2,3,4]
#iterate through each number in the list
  for number in lst:
#put the number along iteration
  print(number)
```

Output:

```
1
2
3
4
```

Creating for loops and putting it into function

Printing Word in the List

```
#Declare the list of words
      lst_string=['Hello','World')
#Iterate through each word in the list
  for word in lst_string:
#Print the word during each iteration
  print (word)
```

Output :

hello
world

Output:
1
2
3
4

The range() function

We can use range() function to generate a sequence of integers which can be stored in a list. The format of the range() function is:

range(start, stop, stepsize)

If we do not mention the ‘start’, it is assumed to be 0 and the ‘stepsize’ is taken as 1. The range of numbers stops one element prior to ‘stop’.

range(0, 10, 1)

The range() function Contd..

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

The range() function return a sequence of numbers, starting from 0 by default and increment by one is default.

```
# create an iterable by using range
    for x in range(6):

# print the value of the iterable during each loop
    print(x):
```

Output:

0
1
2
3
4
5

The range() function Contd..

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

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

Output:

```
2
3
4
5
```

The range() function Contd..

The range() function default to increment by 1 in the sequence, however it is possible to specify the increment value by adding a third parameter

range(2,30,3):

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

Output:

5
8
11
14
17
20
23
26
29

Tuples

A tuple is a collection where is ordered and unchangeable in python tuples are written in brackets

```
tuples=("apple", "banana", "cherry")
print(tuples)

                  #It will be return (apple, banana, cherry)
```

Tuples Contd..

Tuples are sequence, just like list the difference between tuples and list are the tuple can not be change unlike lists and tuples use parenthesis, where as list are use sequence brackets

```
#declare tuples
        tuple1 = (1,2,3,4)
        tuple2 =  tuple ([1,2,3,4])

#Print the tuples
        print(tuple1)
        print(tuple2)
```

Output:

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

Tuples Contd..

Just like with list, you access the data using indexes and you can iterate through the data. The biggest difference between a tuples and list, the list are mutable and tuple are immutable.

This means that in python given a list and tuple

```
# declare list and tuple
        list1=[1,2,3,4]
        tuple1=(1,2,3,4)

        print(list1)
        print(tuple1)
```

Output:

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

Using tuple in your code

1. Length of tuples

```
tup=(1,2,3)
print(len(tup))                    #It will be return 3
```

2. Concatenate of tuples

```
tup1=(1,2,3)
tup2=(3,4,5)
print(tup1+tup2)                   #It will be return (1,2,3,3,4,5)
```

Using tuple in your code

3. Repetion of tuples

    tup=('hello') * 4

    print(tup) #It will be return (hello, hello, hello, hello)

4. Membership

    print(3 in (1,2,3)) #It will be return true

5. Iteration

    for x is (1,2,3):

        print(x) #It will be return 1,2,3

Built in function

➢ len (tuple)	give the length value
➢ max (tuple)	give the max value
➢ min (tuple)	give the min value
➢ tuple (seq)	turn sequence in tuple

Contents to be covered

- Sets
- Dictionary

Set

A set is an unordered collection of items.

Every element is unique (no duplicates).

The set itself is mutable. We can add or remove items from it. Does not support indexing. Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

- Any immutable data type can be an element of a set: a number, a string, a tuple. Mutable (changeable) data types cannot be elements of the set. The elements in the set cannot be duplicates.
- The elements in the set are immutable(cannot be modified) but the set as a whole is mutable.
- There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.
- A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

Set Example

- thisset = {"apple", "banana", "cherry"}
  print(thisset)
- **Access Items**
- You cannot access items in a set by referring to an index, since sets are unordered the items has no index.
- But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

Set Contd..

Example: thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)

- Once a set is created, you cannot change its items, but you can add new items.
- Add Items

1. To add one item to a set use the add() method.
2. To add more than one item to a set use the update() method.

Set Contd..

Programs

1. ```
   thisset = {"apple", "banana", "cherry"}
   thisset.add("orange")
   print(thisset)
   ```
2. ```
   thisset = {"apple", "banana", "cherry"}
   thisset.update(["orange", "mango", "grapes"])
   print(thisset)
   ```
3. ```
   print(sorted(thisset))
   ```
   ```

Set Operation

```
a={2,3,4,5,12}
b={3,4,7,8,2,5}
c=a|b                    #union
print(c)
c=a&b                    #intersection
print(c)
c=b-a                    #diffence
print(c)
c=a-b
print(c)
c=a<=b                   # b is supperset
print(c)
```

Set Operation

The set() Constructor

- It is also possible to use the set() constructor to make a set.
- Using the set() constructor to make a set:"""

- #Same as {"a", "b","c"}
- normal_set = set(["a", "b","c"])

- # Adding an element to normal set is fine
- normal_set.add("d")
- print("Normal Set")
- print(normal_set)

Set Operation Contd..

#3 using constructor

- thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
- print(thisset)
- #why set in python

Set Operation Contd..

Advantages of Python Sets

1. Sets cannot have multiple occurrences of the same element, it makes sets highly useful to efficiently
3. Remove duplicate values from a list or tuple and to perform common
4. To perform math operations like unions and intersections. ..

• int	Immutable	float	Immutable
• Bool	Immutable	tuple	Immutable
• Str	Immutable	frozeset	Immutable
• set	Mutable(changeable)	list	Mutable
• dictionary	Mutable		

रा.इ.सू.प्रौ.सं NIELIT

Set Operation Contd..

- thisset = {"apple", "banana", "cherry“}
- Remove "banana" by using remove() method:
- ```
  thisset = {"apple", "banana", "cherry"}
  thisset.remove("banana")
  print(thisset)

  print(len(thisset))
  ```
- POP-You can also use the pop(), method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.
- The return value of the pop() method is the removed item.
  ```

Example of Pop

- thisset = {"apple", "banana", "cherry"}
  x = thisset.pop()
  print(x)
  print(thisset)
- **Note:** Sets are *unordered*, so when using the pop() method, you will not know which item that gets removed.
- The clear() method empties the set:
- thisset = {"apple", "banana", "cherry"}
  thisset.clear()
  print(thisset)
- The del keyword will delete the set completely:
  thisset = {"apple", "banana", "cherry"}
  del thisset
  print(thisset)

Set Operation Contd..

This is how you perform the well-known operations on sets in Python:

A | B **A.union(B)**	Returns a set which is the union of sets A and B.
A |= B **A.update(B)**	Adds all elements of array B to the set A.
A & B **A.intersection(B)**	Returns a set which is the intersection of sets A and B.
A - B **A.difference(B)**	Returns the set difference of A and B (the elements included in A, but not included in B).
A ^= B **A.symmetric_difference_update(B)**	Writes in A the symmetric difference of sets A and B.
A <= B **A.issubset(B)**	Returns true if A is a subset of B.
A >= B **A.issuperset(B)**	Returns true if B is a subset of A.
A < B	Equivalent to A <= B and A != B
A > B	Equivalent to A >= B and A != B

Set operation(union , intersection etc)

- a={2,3,4,5}
- b={3,4,7,8,2,5}
- c=a|b #union
- print(c)
- c=a&b #intersection
- print(c)
- c=a-b #diffence
- print(c)
- c=a<=b# b is supperset
- print(c)

Dictionary

- **Dictionary** is an unordered collection of data values.
- A Dictionary can be created by placing sequence of elements within curly **{}** braces, separated by ‘comma’.
- A dictionary has a **key: value** pair. Each key-value pair is separated by a colon **:**, whereas each key is separated by a ‘comma’.
- Values in a dictionary can be of any datatype and can be duplicated, whereas keys can’t be repeated and must be immutable.
- Dictionary can also be created by the built-in function dict().

Dictionary Example

```
dy= {}                                # empty dictionary

dy= {1: 'apple', 2: 'ball'}           # dictionary with integer keys

dy= {'name': 'John', 1: [2, 4, 3]}    #dictionary with mixed keys

dy= dict({1:'apple', 2:'ball'})       # using dict()

dy= dict([(1,'apple'), (2,'ball')])   # from sequence having each item as a pair
```

Accessing elements from a dictionary

- Key can be used either inside square brackets or with the get() method. (The difference while using get() is that it returns None instead of KeyError, if the key is not found.)

```
my_dict = {'name':'Ram', 'age': 26}
print(my_dict['name'])
print(my_dict.get('age'))

my_dict.get('address')  # Trying to access keys which doesn't exist throws error
my_dict['address']
```

```
Ram
26
Traceback (most recent call last):
  File "B:/python/programs/Dictionary/Dict1.py", line 6, in <module>
    my_dict['address']
KeyError: 'address'
```

Change or add elements in a dictionary

```
dt = {'name':'Ram', 'age': 26}
dt['age'] = 27
print(dt)
dt['address'] = 'Gorakhpur'
print(dt)
```

```
{'name': 'Ram', 'age': 27}
{'name': 'Ram', 'age': 27, 'address': 'Gorakhpur'}
```

Delete or Remove Elements

```
squares = {1:1, 2:4, 3:9, 4:16, 5:25}
print(squares.pop(4))
print(squares)
print(squares.popitem())
print(squares)
del squares[5]
print(squares)
squares.clear()
print(squares)
del squares
print(squares)        # Throws Error
```

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

राइसूप्रौसं NIELIT

Example

```
original = {1:'one', 2:'two'}
new = original.copy()

print('Orignal: ', original)
print('New: ', new)
```

```
Orignal:  {1: 'one', 2: 'two'}
New:  {1: 'one', 2: 'two'}
```

```
dt = {'name':'Ram', 'age': 26}
print('the existing dictionary is:',dt)
dt['age'] = 27
print(dt)
dt['address'] = 'Gorakhpur'
dt['mob no.']=1234567890
print('the final updated dictionary is ',dt)
```

```
person =  {'name': 'Ram', 'salary': None}
salary =  None
person =  {'name': 'Ram', 'salary': None, 'age': 22}
age =  22
```

Program

```
person = {'name': 'Ram‘}                    # key is not in the dictionary
salary = person.setdefault('salary')
print('person = ',person)
print('salary = ',salary)                   # key is not in the dictionary
                                            # default_value is provided
age = person.setdefault('age', 22)
print('person = ',person)
print('age = ',age)
```

रा.इ.सू.प्रौ.सं
NIELIT

Example

```
squares = {1:1, 2:4, 3:9, 4:16, 5:25}
print(squares.pop(4))
print(squares)
print(squares.popitem())
print(squares)
del squares[3]
print(squares)
squares.clear()
print(squares)
del squares
print(squares)      # Throws Error
```

Example

```
dt = {'name':'Ram', 'age': 26}
print('the existing dictionary is:',dt)
dt['age'] = 27
print(dt)
dt['address'] = 'Gorakhpur'
dt['mob no.']=1234567890
print('the final updated dictionary is ',dt)
my_dict = {'name':'Ram', 'age': 26}
print(my_dict['name'])
print(my_dict.get('age'))
my_dict.get('address')  # Trying to access keys which doesn't exist throws
  error
my_dict['address']
```

Example

```
squares = {1:1, 2:4, 3:9, 4:16, 5:25}
print(squares.pop(4))
print(squares)
print(squares.popitem())
print(squares)
del squares[3]
print(squares)
squares.clear()
print(squares)
del squares
print(squares)      # Throws Error
```

Example

```
person = {'name': 'Ram‘}                    # key is not in the dictionary
salary = person.setdefault('salary')
print('person = ',person)
print('salary = ',salary)                   # key is not in the dictionary
                                            # default_value is provided
age = person.setdefault('age', 22)
print('person = ',person)
print('age = ',age)
```

Python String

Objectives-Following objective of the python string

- ❖ Concept of String.
- ❖ String manipulating & Indexing
- ❖ Creating String & Deleting String
- ❖ Various String Functions

Python String

Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes. The computer does not understand the characters; internally, it stores manipulated character as the combination of the 0's and 1's.

Each character is encoded in the ASCII or Unicode character. So we can say that Python strings are also called the collection of Unicode characters.

In Python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

Example:

```
str="Nielit"

print(type(str))
```

Output:

```
<class 'str'>
```

In Python, strings are treated as the sequence of characters, which means that Python doesn't support the character data-type; instead, a single character written as 'N' is treated as the string of length 1.

Creating String in Python

We can create a string by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string or docstrings.

Example:

```
#Use single quotes

s1 = 'Python Programming'

print(s1)

print(type(s1))

print("***********************")

#Use double quotes
```

```
s2 = "Python Programming"
print(s2)
print(type(s2))
print("***********************")

#Use triple quotes
s3 = """Triple quotes are generally used for
    represent the multiline or
    docstring'''
print(s3)
print(type(s3))
```

Output:

Python Programming

<class 'str'>

****

Python Programming

<class 'str'>

****

"Triple quotes are generally used for

represent the multiline or

docstring

<class 'str'>

Strings indexing and splitting

Like other languages, the indexing of the Python strings starts from 0. For example, the string "HELLO" is indexed as given in the below figure.

str[0] = 'H'

str[1] = 'E'

str[2] = 'L'

str[3] = 'L'

str[4] = 'O'

Example:

```
str = "PYTHON"
print(str[0])
print(str[1])
print(str[2])
print(str[3])
print(str[4])
print(str[5])
# It returns the IndexError because 6th index doesn't exist
print(str[6])
```

Output:

P

Y

T

H

O

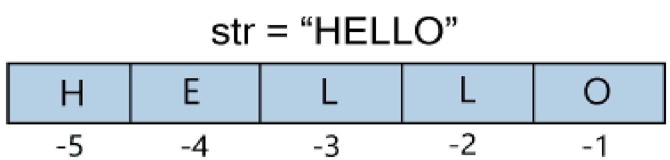
N

IndexError: string index out of range

Slice operator [] in String

As shown in Python, the slice operator [] is used to access the individual characters of the string. However, we can use the: (colon) operator in Python to access the substring from the given string.

Example.



str[-1] = 'O' str[-3:-1] = 'LL'

str[-2] = 'L' str[-4:-1] = 'ELL'

str[-3] = 'L' str[-5:-3] = 'HE'

str[-4] = 'E' str[-4:] = 'ELLO'

str[-5] = 'H' str[::-1] = 'OLLEH'

Example:

```
str ='HELLOWORLD'
print(str[-1])
print(str[-3])
print(str[-2:])
print(str[-4:-1])
print(str[-7:-2])
# Reversing the given string
print(str[::-1])
# Search Character out of index
print(str[-12])
```

Output:

D

R

LD

ORL

LOWOR

DLROWOLLEH

print(str[-12])

IndexError: string index out of range

Reassigning Strings

Updating the content of the strings is as easy as assigning it to a new string. The string object doesn't support item assignment i.e., A string can only be replaced with new string since its content cannot be partially replaced. Strings are immutable in Python.

Example

```
str = "PYTHON"
str[0] = "p"
print(str)
```

Output

str[0] = "p“

TypeError: 'str' object does not support item assignment

Example:

```
str = "PYTHON"

print(str)

str= "python"

print(str)
```

output:

PYTHON

python

Deleting the String

As we know that strings are immutable. We cannot delete or remove the characters from the string. But we can delete the entire string using the del keyword.

```
str="PYTHON"
print(str)
del str[0]
#print String after delete
print("*******")
print(str)
```

Output:

```
del str[0]
TypeError: 'str' object doesn't support item deletion
```

Example:

```
str="PYTHON"

print(str)

del str

#print String after delete

print("*******")

print(str)
```

Output:

```
PYTHON

*******

<class 'str'>
```

String Operators

Operator	Description
+	It is known as concatenation operator used to join the strings given either side of the operator.
*	It is known as repetition operator. It concatenates the multiple copies of the same string.
[]	It is known as slice operator. It is used to access the sub-strings of a particular string.
[:]	It is known as range slice operator. It is used to access the characters from the specified range.
in	It is known as membership operator. It returns if a particular sub-string is present in the specified string.

String Operators

Operator	Description
not in	It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string.
%	It is used to perform string formatting. It makes use of the format specifies used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python.

Example on String Operator:

```
str1 = "Python"
str2 = "Program"
print(str1*3) # prints PythonPythonPython
print(str1+str2)# prints PythonProgram
print(str1[4]) # prints o
print(str2[2:4]); # prints og
print('h' in str1) # prints True as "h" is  present in str1
print('m' in str1) # prints False as "m" is not present in str1
print('am' not in str2) # prints False as "am" is present in str2.
print('n' not in str2) # prints True as "n" is not present in str2.
print("The string str : %s"%(str1)) # prints The string str : Python
```

Output:

PythonPythonPython

PythonProgram

o

og

True

False

False

True

The string str : Python

String Functions

1.count()

The count() method returns the number of times a specified value appears in the string.

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

Parameter	**Description**
value	Required. A String. The string to value to search for
start	Optional. An Integer. The position to start the search. Default is 0
end	Optional. An Integer. The position to end the search. Default is the end of the string

String Functions

Example:

Return the number of times the value "apple" appears in the string:

```
txt = "I love apples, apple are my favorite fruit"
x = txt.count("apple")
print(x)
```

Output: 2

Example: Search from index 10 to 24:

```
txt = "I love apples, apple are my favorite fruit"
x = txt.count("apple", 10, 24)
print(x)
```

Output: 1

String Functions

2.find()

The find() method finds the first occurrence of the specified value.

The find() method returns -1 if the value is not found.

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

Parameter	**Description**
Value	Required. The value to search for
Start	Optional. Where to start the search. Default is 0
End	Optional. Where to end the search. Default is to the end of the string

String Functions

Example: To find the first occurrence of the letter "e" in txt:

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

Output: 1

Example 2:

Where in the text is the first occurrence of the letter "e" when you only search between position 5 and 10?:

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

Output: 8

String Functions

3-rfind(): The rfind() searches the string for a specified value and returns the last position of where it was found.

Example:

- The rfind() method finds the last occurrence of the specified value.
- The rfind() method returns -1 if the value is not found.

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

Parameter	Description
value	Required. The value to search for
start	Optional. Where to start the search. Default is 0
end	Optional. Where to end the search. Default is to the end of the string

String Functions

Example: Where in the text is the last occurrence of the string "nielit"?:

```
txt = "nielit gorakhpur has started o level course. nielit gorakhpur"
x = txt.rfind("nielit")
print(x)
```

Output:

43

String Functions

Example: Where in the text is the last occurrence of the letter "e" when you only search between position 5 and 10?:

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

Output: 8

Example: If the value is not found, the rfind() method returns -1

```
txt = "Hello, welcome to NIELIT gorakhpur."
x = txt.rfind('nielit')
print(x)
```

Output: -1

String Functions

4-capitalize():

This method converts the first character to upper case. The capitalize() method returns a string where the first character is upper case.

Example: Upper case the first letter in this sentence:

```
txt = "hello, welcome to NIELIT gorakhpur."

x = txt.capitalize()

print (x)
```

Output: Hello, welcome to nielit gorakhpur.

String Functions

5-title()

The title() method returns a string where the first character in every word is upper case. Like a header, or a title.

Example:

```
txt = "python programming using string"
x = txt.title()
print(x)
```

Output: Python Programming Using String

If the word contains a number or a symbol, the first letter after that will be converted to upper case.

String Functions

Example:

```
txt = " 3rd generation python"
x = txt.title()
print(x)
```

Output:3Rd Generation Python

Example: Note that the first letter after a non-alphabet letter is converted into a upper case letter:

```
txt = "hello b2b2b2 and 3g3g3g"
x = txt.title()
print(x)
```

Output: Hello B2B2B2 And 3G3G3G

String Functions

6-lower()

The lower() method returns a string where all characters are lower case. Symbols and Numbers are ignored.

Example:

```
txt = "Welcome To NIELIT gorakhpur"
x = txt.lower()
print(x)
```

Output: welcome to nielit gorakhpur

String Functions

7-upper()

The upper() method returns a string where all characters are in upper case. Symbols and Numbers are ignored.

Example:

```
txt = "Welcome To NIELIT gorakhpur"
x = txt.upper()
print(x)
```

Output: WELCOME TO NIELIT gorakhpur

String Functions

8-islower()

The islower() method returns True if all the characters are in lower case, otherwise False. Numbers, symbols and spaces are not checked, only alphabet characters.

Example:

```
txt = "hello world!"
x = txt.islower()
print(x)
```

Output:True

String Functions

9-isupper()

The isupper() method returns True if all the characters are in upper case, otherwise False. Numbers, symbols and spaces are not checked, only alphabet characters.

Example:

```
txt = "PYTHON PROGRAM"
x = txt.isupper()
print(x)
```

Output: True

String Functions

10-istitle()

The istitle() method returns True if all words in a text start with a upper case letter, AND the rest of the word are lower case letters, otherwise False. Symbols and numbers are ignored.

Example:

a = "HELLO, AND WELCOME TO MY WORLD"

b = "Hello"

c = "22 Names"

d = "This Is %'!?"

String Functions

```
print(a.istitle())
print(b.istitle())
print(c.istitle())
print(d.istitle())
```

Output:

```
False
True
True
True
```

String Functions

11-replace()

The replace() method replaces a specified phrase with another specified phrase.

Syntax: string.replace(oldvalue, newvalue, count)

Parameter Values

Parameter	Description
oldvalue	Required. The string to search for
newvalue	Required. The string to replace the old value with
count	Optional. A number specifying how many occurrences of the old value you want to replace. Default is all occurrences

String Functions

Example: Replace all occurrence of the word "one":

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

Output: three three was a race horse, two two was three too.

Example: Replace the two first occurrence of the word "one":

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

Output: three three was a race horse, two two was one too.

String Functions

12-strip()

The strip() method removes any leading (spaces at the beginning) and trailing (spaces at the end) characters (space is the default leading character to remove)

Syntax **string.strip(characters)**

Parameter Values

Parameter	Description
characters	Optional. A set of characters to remove as leading/trailing characters

Example:

Remove spaces at the beginning and at the end of the string:

```
txt = "     banana     "
x = txt.strip()
print(x)
```

Output:banana

Example:

Remove the leading and trailing characters other than space

```
txt = ",,,,,rrttgg.....apple....rrr"
x = txt.strip(",.grt")
print(x)
```

Output: apple

String Functions

lstrip()

The lstrip() method removes any leading characters (space is the default leading character to remove)

Syntax: string.lstrip(characters)

Where, character is Optional. A set of characters to remove as leading characters

Example:

```
txt = ",,,,,ssaaww.....banana..    "
x = txt.lstrip(",.asw")
print(x)
```

Output: **banana..**

Note: Only leading character on left side will be removed.

String Functions

rstrip()

The rstrip() method removes any trailing characters (characters at the end a string), space is the default trailing character to remove.

Syntax: **string.rstrip(characters)**

Where, characters is optional. A set of characters to remove as trailing characters

Example:

```
txt = "banana,,,,,ssaaww....."
x = txt.rstrip(",.asw")
print(x)
```

Output: **banana..**

Note: Only leading character on right side will be removed.

String Functions

split():

The split() method splits a string into a list. You can specify the separator, default separator is any whitespace.

Syntax string.split(separator, maxsplit)

Parameter	Description
separator	Optional. Specifies the separator to use when splitting the string. By default any whitespace is a separator
maxsplit	Optional. Specifies how many splits to do. Default value is -1, which is "all occurrences"

String Functions

Example: txt = "hello, my name is Peter, I am 26 years old"

```
x = txt.split(", ")
print(x)
```

Output: ['hello', 'my name is Peter', 'I am 26 years old']

String Functions

partition()

The partition() method searches for a specified string, and splits the string into a tuple containing three elements.

- The first element contains the part before the specified string.
- The second element contains the specified string.
- The third element contains the part after the string.

Syntax **string.partition(value)**

Where, value is required. The value is the string to search for

String Functions

Example

```
txt = "I could eat bananas all day"
x = txt.partition("bananas")
print(x)
```

Output: ('I could eat ', 'bananas', ' all day')

Search for the word "bananas", and return a tuple with three elements:

1 - everything before the "banana"

2 - the "banana"

3 - everything after the "banana"

String Functions

join()

The join() method takes all items in an iterable and joins them into one string. A string must be specified as the separator.

Example: join all items in a dictionary into a string, using a the word “and" as separator:

```
List1 =("apple","Bannana")
mySeparator = " and "
x = mySeparator.join(List1)
print(x)
```

Output: apple and Bannana

String Functions

isspace()

The isspace() method returns True if all the characters in a string are whitespaces, otherwise False.

Example: txt = "   s   "

x = txt.isspace()

print(x)

Output: False

Advantages of Using Function

1.Ease of Use: This allows ease in debugging the code and prone to less error.

2.Reusability: It allows the user to reuse the functionality with a different interface without typing the whole program again.

3.Ease of Maintenance: It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

Functions

A function can be defined as the organized block of reusable code which can be called whenever required. A function is a block of code which only runs when it is called. Basically two types of function.

1-SDF-System Defined Function

2-UDF-User Defined Function.

- Python allows us to divide a large program into the basic building blocks known as function.
- A function can be called multiple times to provide reusability and modularity to the python program.
- The idea is to put some commonly or repeatedly done task together and make a function.

Function Types(UDF):

Function can be categorized in to:

- ❑ Non-Parameterized Function
- ❑ Parameterized Function

Function definition

In python, we can use def keyword to define the function.

Syntax:

```
def function_name(parameter_list):
        function-definition
return <expression>
```

- The function block is started with the colon (:)
- All the same level block statements remain at the same indentation.
- A function can accept any number of parameters that must be the same in the definition and function calling.

Function calling

In python, a function must be defined before the function calling otherwise the python interpreter gives an error.

Once the function is defined, we can call it from another function or the python prompt.

To call the function, use the function name followed by the parentheses.

```
def hello_world():
    #function declaration
    print("This is first statement")
   #function definition
    print("This is second statement")
hello_world()
```

Non-Parameterized Function

The non-parameterized function does not require any variable name in their declaration and calling.

Example:

```
def area_circle():
    r=float(input("Enter Radius of Circle:"))
    a=3.14*r*r
    print("Area of circle:",a)
area_circle()     #function calling
```

Parameterized Function

The parameterized function require variable name in their declaration and calling.

Function parameters

The information into the functions can be passed as the parameters. The parameters are specified in the parentheses.

- A function may have any number of parameters.
- Multiple parameters are separated with a comma

Example:

```
def area_circle(r):
    a=3.14*r*r
    print("Area of circle:",a)
radius=float(input("Enter Radius:"))
area_circle(radius)
```

- Here, the function named area_circle() is declared with empty().
- This means, the function called with empty() i.e. calling does not requires any parameter.

Example 2:

Python function to calculate area of rectangle using parameterized function

```
def area_rect(l,b):                    #parameterized function
        area_lb=l*b
        print("Area of Rectangle:",area_lb)
len=float(input("Enter Length of rectangle:"))
brth=float(input("Enter Breadth of rectangle:"))
area_rect(len, brth)
```

Output:

```
Enter Length of rectangle:30
Enter Breadth of rectangle:20
Area of Rectangle: 600.0
```

Returning a value

A function may return a value using return keyword. When a function produces a result, there are two possibilities:

a) The value of output is preserved within the function. As in above example.

b) Transfer the value of output to calling function.

i) Return statement is used in this scenario.

ii) A return statement is used to end the execution of the function call and “returns” the result (value of the expression following the return keyword) to the caller.

c. The statements after the return statements are not executed.

d. If the return statement is without any expression, then the special value None is returned.

Example1

```
def si_int(p,r,t):
        si=(p*r*t)/100
        return si  #returning the value of si to the calling function
s=si_int(20000,8.5,3)
print("Simple Interest=",s)
```

- In above example, the function si_int() will calculate the simple interest in variable named si and returns the value of si to the function calling.
- At function calling, the returned value is stored in variable named s.
- Now, we can use the value of s as per requirement. In above example, the value of s gets printed using print statement.

Example2

```
def area_rect(l,b): #parameterized function
        area_lb=l*b
        return area_lb  #returning the value of area_lb to calling function
len=float(input("Enter Length of rectangle:"))
brth=float(input("Enter Breadth of rectangle:"))
# Ar will store the value of area_lb, which is returned by the function
Ar=area_rect(len, brth)
```

Output:

Enter Length of rectangle:20

Enter Breadth of rectangle:10

Area of Rectangle: 200.0

Types of parameters:

There may be several types of arguments which can be passed at the time of function calling.

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

1.Required arguments

The required arguments are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, then the python interpreter will show the error.

Example1

```
def calculate(a,b):
    return a+b
sum1=calculate(10) # this causes an error as we are missing a required arguments b.
print("Sum=",sum)
```

Output:

calculate() missing 1 required positional argument: 'b'

Example2

```
def calculate(a,b):
        return a+b
sum1=calculate(10,10)
print(“Sum=“,sum)
```

Output:

Sum=20

2. Keyword arguments

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

Example1

#The function simple_interest(p, t, r) is called with the keyword arguments the order of arguments doesn't matter in this case

```
def simple_interest(p,t,r):
    return (p*t*r)/100
print("Simple Interest: ",simple_interest(t=10,r=10,p=1900))
```

Output:

Simple Interest: 1900.0

Note: In this case the name of argument is same in calling and definition.

2. Keyword arguments

Example2

#The function simple_interest(p, t, r) is called with the keyword arguments the order of arguments doesn't matter in this case

```
def simple_interest(p,t,r):
    return (p*t*r)/100
x=float(input("Enter Amount"))
y=float(input("Enter Time"))
z=float(input("Enter Rate"))
print("Simple Interest: ",simple_interest(t=y,r=z,p=x))
```

Output:

```
Enter Amount12000
Enter Time5
Enter Rate5
Simple Interest:  3000.0
```

2. Keyword arguments

- **If we provide the different name of arguments at the time of function call, an error will be thrown.**

  simple_interest(20000,rate=7.5, time=6)   #error

- **The python allows us to provide the mix of the required arguments and keyword arguments at the time of function call.**

  simple_interest(20000,t=5,r=6.5)

- **The required argument must not be given after the keyword argument.**

  simple_interest(20000,r=7.5,6)   #error

3. Default Arguments

Python allows us to initialize the arguments at the function definition. If the value of any of the argument is not provided at the time of function call, then the default value for the argument will be used.

Example1

```
def printme(name,age=22):
        print("Name:",name,"\nAge:",age)
printme("Ravi")                  # name=Ravi age=22 (default value)
printme("Sachin", 33)            #name =Sachin age=33
```

Output:

Name: Ravi

Age: 22

Name: Sachin

Age: 33

4. Variable length Arguments

Variable length argument is a feature that allows a function to receive any number of arguments. However, at the function definition, we have to define the variable with * (star) as *<variable - name >.

Example1

```
def printme(*names):
    print("type of passed argument is ",type(names))
    print("printing the passed arguments...")
    for name in names:
        print(name)
#calling printme function
printme("Rahul","Prashant","sunita","Sandeep")
```

Output:

type of passed argument is <class 'tuple'>

printing the passed arguments...

Rahul

Prashant

sunita

Sandeep

4. Variable length Arguments

Example2

```
def adder(*num):
    sum = 0
    for n in num:
        sum = sum + n
    print("Sum:",sum)
adder(3,5)              #calling adder with 2 argument
adder(4,5,6,7)          #calling adder with 4 argument
adder(1,2,3,5,6)        #calling adder with 5 argument
```

Output:

Sum: 8

Sum: 22

Sum: 17

Local Variable

- A local variable is a type of variable declared within programming block or function.
- It can only be used only inside that function or code block in which they were declared.
- The local variable exists until the block of the function is in under execution. After that, it will be destroyed automatically.

Example:1

```
def func():
    a=10
    print("Value of a in function:",a)
func()
print("Value of ‘a’ outside function:",a)
```

Output:

name 'a' is not defined

In example1, it is clearly shown that the value of variable ‘a’ is only accessible inside the function func(). The value variable ‘a’ cannot be accessible outside the function because variable ‘a’ is a local variable of the function func(). It can only be accessible inside the function func().

Global variable

- Global variables are defined outside of a subroutine or function.
- The global variable will hold its value throughout the lifetime of a program.
- They can be accessed within any function defined for the program.

Example:2

```
a=10
def func():
    print("Value of a in function:",a)
func()
print("Value of ‘a’ outside function:",a)
```

Output:

Value of a in function: 10

Value of ‘a’ outside function: 10

Here, variable ‘a’ is defined outside the function func(). The variable ‘a’ will now become a global variable. It can be accessed inside any function as well as outside the function.

Global keyword

So far, we haven’t had any kind of a problem with global scope. So let’s take an .

Example:3

```
i=10      #global variable
def counter():
    i=20 #local variable of function counter
    print("Value of i in function:",i)
counter()
print("Value of i Outside Function:",i)
```

Output:

Value of i in function: 20

Value of i Outside Function: 10

Now, when we make a reference to ‘i’ outside this function, we get 10 instead of 20.

Global keyword is a keyword that allows a user to modify a variable outside of the current scope.

- It is used to create global variables from a non-global scope i.e. inside a function.
- Global keyword is used inside a function only when we want to do assignments or when we want to change a variable.
- Global is not needed for printing and accessing.

Rules of global keyword:

If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.

- Variables that are only referenced inside a function are implicitly global.
- We Use global keyword to use a global variable inside a function.
- There is no need to use global keyword outside a function.

Use of global keyword:

To access a global variable inside a function there is no need to use global keyword.

Example:4

```
# global variable
a = 15
b = 10
 # function to perform addition
def add():
    c = a + b
    print(c)
# calling a function
add()
```

Output: 25

•If we need to assign a new value to a global variable then we can do that by declaring the variable as global.

Code 2: Without global keyword

```
a = 15
  # function to change a global value
def change():
      # increment value of a by 5
    a = a + 5
    print(a)
 change()
```

•**If we need to assign a new value to a global variable then we can do that by declaring the variable as global.**

Code 2: Without global keyword

```
a = 15
  # function to change a global value
def change():
        a = a + 5  # increment value of a by 5
        print(a)
change()
```

Output: UnboundLocalError: local variable 'a' referenced before assignment

This output is an error because we are trying to assign a value to a variable in an outer scope. This can be done with the use of global variable.

Code 2: With global keyword

```
# Python program to modify a global
# value inside a function
x = 15
def change():
    global x  # using a global keyword
    x = x + 5 # increment value of a by 5
    print("Value of x inside a function :", x)
change()
print("Value of x outside a function :", x)
```

Output:

Value of x inside a function : 20

Value of x outside a function : 20

Python Docstrings

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods.

General Rules:

- ❑ The doc string line should begin with a capital letter and end with a period.
- ❑ The first line should be a short description.
- ❑ If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description.
- ❑ The following lines should be one or more paragraphs describing the object’s calling conventions, its side effects, etc.

Python Docstrings

Declaring Docstrings: The docstrings are declared using “””triple double quotes””” just below the class, method or function declaration. All functions should have a docstring.

Accessing Docstrings: The docstrings can be accessed

- using the __doc__ method of the object or
- using the help function.

Python Docstrings

Example:

```
def my_function():
    """Demonstrate docstrings and does nothing really."""
    return None
print("Printing DocString Using __doc__:")
print(my_function.__doc__ )
print("Printing DocString Using help function:")
help(my_function)
```

Output:

```
Printing DocString Using __doc__:
Demonstrate docstrings and does nothing really.
Printing DocString Using help function:
Help on function my_function in module __main__:
my_function()
        Demonstrate docstrings and does nothing really.
```

Python Anonymous/Lambda Function

In this article, you'll learn about the anonymous function, also known as **lambda functions.**

- What are lambda functions in Python?
- In Python, an anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword in Python, anonymous functions are defined using the **lambda** keyword.
- Hence, anonymous functions are also called lambda functions.
- How to use lambda Functions in Python?
- A lambda function in python has the following syntax.

Syntax of Lambda Function in python

```
lambda arguments: expression
```

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Example of Lambda Function in python

Here is an example of lambda function that doubles the input value.

```
# Program to show the use of lambda functions


double = lambda x: x * 2

print(double(5))
```

Run Code

Output

10

In the above program, lambda x: x * 2 is the lambda function. Here x is the argument and x * 2 is the expression that gets evaluated and returned.

This function has no name. It returns a function object which is assigned to the identifier double. We can now call it as a normal function. The statement

```
double = lambda x: x * 2
```

is nearly the same as:

```
def double(x):
   return x * 2
```

Use of Lambda Function in python

We use lambda functions when we require a nameless function for a short period of time.

In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like **filter(), map()** etc.

Example use with filter()

The **filter()** function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

Here is an example use of **filter()** function to filter out only even numbers from a list.

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)
```

Run Code

Output

[4, 6, 8, 12]

Example use with map()

The **map()** function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of map() function to double all the items in a list.

 # Program to double each item in a list using map()

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)
```

Run Code

Output

[2, 10, 8, 12, 16, 22, 6, 24]

Recursion Function:

What is recursion?

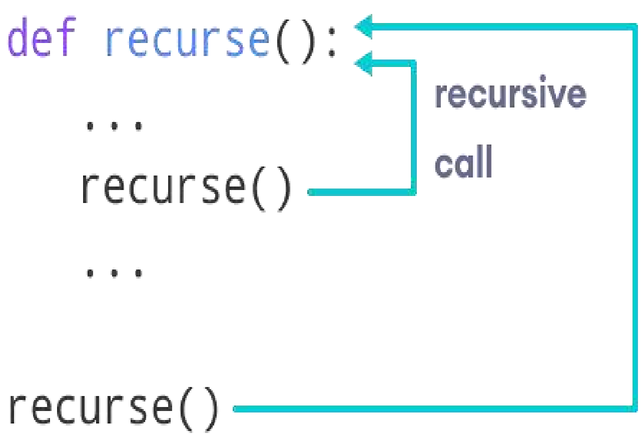
Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Python Recursive Function

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

The following image shows the working of a recursive function called recurse.



Recursive Function in Python

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

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

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

Output

The factorial of 3 is 6

In the above example, **factorial()** is a recursive function as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

```
factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call
```

Let's look at an image that shows a step-by-step process of what is going on:

Factorial by a recursive method

Working of a recursive factorial function

Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

Introduction to File Handling

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).

- Since Random Access Memory (RAM) is volatile, we use files for future use of the data by permanently storing them.

- File handling provides a mechanism to store the output of a program in a file and to perform various operations on it.

File Types

- **Text files**

  In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character (‘\n’) in python by default. Text file has extension .txt.

- **Binary files**

In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language. Binary files have an extension .bin.

Files Operations

In Python, a file operation takes place in the following order:

1. Open a file
2. Read or write (perform operation)
3. Close the file

File Handling Method

Python provides the **open()** function which accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

The syntax to use the open() function----

file object = open(<file-name>, <accessmode>, <buffering>)

The files can be accessed using various modes like read, write, or append.

The following are the details about the access mode to open a file.

SN	Access mode	Description
1	r	It opens the file to read-only. 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.
3	r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
4	rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
5	w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.

6	**wb**	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
7	**w+**	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
8	**wb+**	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	**a**	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.
10	**w**	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.

10	**ab**	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
11	**a+**	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
12	**ab+**	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

Example:

To open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

```
#opens the file file.txt in read mode
fileptr = open("file.txt","r")
if fileptr:
    print("file is opened successfully")
```

Output:

```
<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.

We can perform any operation on the file externally in the file system is the file is opened in python, hence it is good practice to close the file once all the operations are done.

The syntax to use the close() method is ---

fileobject.close()

Example:

```
# opens the file file.txt in read mode
fileptr = open("file.txt","r")

if fileptr:
    print("file is opened successfully")

#closes the opened file
fileptr.close()
```

Reading the file

To read a file using the python script, the python provides us the read() method. The read() method reads a string from the file. It can read the data in the text as well as binary format.

The syntax of the read() method is ----

fileobj.read(<count>)

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Reading from a file

There are three ways to read data from a text file.

1. Using read()
2. Using readline()
3. Using readlines()

Files

read() :

The read() method returns the specified number of bytes from the file. Default is -1 which means the whole file.

```
File.read([size])
```

Where, size is optional. The number of bytes to return. Default -1, which means the whole file.

Example: #opening file in reading mode

```
file1 = open(“mydata.txt","r")

print("Output of read()")

print("------------------------------")

#To read entire file using read()

print(file1.read())

file1.close() # closing the file
```

To run this program,

- Create new file using notepad.
- Type some content in the file.
- Save the file with name – mydata.
- Location of mydata file and program must be the same.
- If you save the mydata file at desktop, then save the python file also on desktop.d

Example: Reading 7 character using read(7)

```
#opening file in reading mode
file1 = open("mydata.txt","r")
print("Output of read()")
print("-----------------------------")
#print("Output of read(7)")
print(file1.read(7))
file1.close()
```

readline() :

The readline() method returns one line from the file. You can also specified how many bytes from the line to return, by using the size parameter

Syntax: File.readline([size])

Where, size is optional. The number of bytes from the line to return. Default -1, which means the whole line.

Example 1:

```
file1 = open("mydata.txt","r")

print("Calling readline() to return  the first line only:")

print(file1.readline())

file1.close()         # closing the file
```

Example:

```
print("Return only the six first bytes from the first line:")

print(file1.readline(6))
```

readlines():

Reads all the lines and return them as each line a string element in a list.

Syntax: file.readlines(hint)

Where, hint is optional. If the number of bytes returned exceed the hint number, no more lines will be returned. Default value is -1, which means all lines will be returned.

Example 1:

```
file1 = open("mydata.txt","r")

print("Return all lines in the file, where each line is an item in the list:")

print(file1.readlines())

file1.close() # closing the file
```

Example 2: Do not return the next line if the total number of returned bytes are more than 10:

```
file1 = open("mydata.txt","r")

print("Do not return the next line if the total number of returned bytes are more than 10:")

print(file1.readlines(10))

file1.close() # closing the file
```

Redirecting the output:

We can redirect the output from output screen to a file by specifying the file object in print().

- **Redirecting text to a file**

```
print(“text”,file=file_object)
```

- **Redirecting variable value to a file**

```
print(var_name, file=file_object)
```

#Writing output to a file

```
f = open(“output.txt", "w")
r=5
a=3.14*r*r
print("Area=", a, file=f)
print(“New line.", file=f)
f.close()
```

#Writing output to a file with more result

```
ch='y'
f = open("output.txt", "w")
while ch=='Y' or ch =='y':
    print("Enter radius")
    r=float(input())
    a=3.14*r*r
    print("Area=", a, file=f)
    print("enter your choice")
    ch=input()
```

Writing to a file

There are two methods to write in a file.

1. Using write()
2. Using writeline()

Write():

The write() method writes a specified text to the file. Where the specified text will be inserted depends on the file mode and stream position. A file can be written in two mode:

- **"a":** The text will be inserted at the current file stream position, default at the end of the file.
- **"w":** The file will be emptied before the text will be inserted at the current file stream position, default 0. The contents of existing file will be overwritten.

Note:

- If file does not exist, the new file will be created using “a” or “w” mode.
- If you want to create a new file, then use “a” or “w” mode.
- If you want to append the contents at the end of the existing file, use “a” mode.
- If you want to overwrite the content of an existing file, use “w” mode.

Note: "x" mode is similar to "w " mode.

- For "x " mode, if the file exists, raise FileExistsError.
- For "w" mode, it will simply create a new file or overwrite the existed file.

Syntax : file.write(byte)

Where, byte is the text or byte object that will be inserted.

Example 1: Open the file with "a" mode for appending, then add some text to the file:

#opening file in read mode to see contents of original file

```
f = open("mydata.txt","r")
print("Original File:")
print("---------------------------------------------")
print(f.read())
f.close()
```

#opening file in append mode to write into file

```
f1 = open(“mydata.txt","a")
```

#writing into file

```
f1.write("This line inserted today")

f1.close()
```

#open and read the file after the appending:

```
print("File After appending the content to the file-")

print("-------------------------------------------")

f = open(“mydata.txt", "r")

print(f.read())

f.close()
```

creating a new file named new_file.txt and typing the content using keyboard

f1 = open(r"C:\Users\nielit\Documents\new_file.txt","w")

#writing content from keyboard in file

s=input("Enter text to be inserted in file:")

f1.write(s)

f1.close()

#open and read the file after the appending:

print("File After writing the content to the file-")

print("---")

f = open(r"C:\Users\nielit\Documents\new_file.txt", "r")

print(f.read())

Writelines()

The writelines() method writes the items of a list to the file. Where the texts will be inserted depends on the file mode and stream position. It is used to write more than one line to the file. The file can be opened in two modes:

- **"a": append mode**
- **"w": write mode**

Example 2: creating a new file named writelines_ex.txt and typing the content using keyboard

f1 = open(“writelines_ex.txt","w")

#writing content from keyboard in file

s=input("Enter text to be inserted and press enter key to insert the content in the file:")

f1.writelines(s)

f1.close()

#open and read the file after the writing:

print("File After writing the content to the file-")

print("--")

f = open(“writelines_ex.txt", "r")

print(f.read())

Example 1: Open the file with "a" for appending, then add a list of texts to append to the file:

#Open the file with "a" for appending, then add a list of texts to append to the file:

f = open("demofile3.txt", "a")

#demofile3 file will now be created where this program is stored

f.writelines(["See you soon!", "Over and out."])

f.close()

#open and read the file after the appending:

f = open("demofile3.txt", "r")

print(f.read())

The difference between Write() and WriteLine() method is based on new line character.

- write(arg) expects a string as argument and writes it to the file. If you provide a list of strings, it will raise an exception.

- writelines(arg) expects an iterable as argument (an iterable object can be a tuple, a list, a string). Each item contained in the iterator is expected to be a string.

Seek():

The seek() method sets the current file position in a file stream. The seek() method also returns the new position.

Syntax: file.seek(offset)

Where, offset is required. Offset is the number representing the position to set the current file stream position.

#Using seek method to place cursor at 4th position and then reading:

```
f = open("demofile3.txt", "r")

print("Original Content:")

print(f.read())

print("----------------------------------------------")

print("Reading the content from 4th position:")

f.seek(4)

print(f.readline())

f.close()
```

Tell()

The tell() method returns the current file position in a file stream.

Syntax: file.tell()

It has no parameter.

Example: Open the file and check the cursor position:

```
f = open("demofile3.txt", "r")

print("Current Location of cursor",f.tell())

print(f.readline())

print("Updated Location of cursor after reading",f.tell())
```

Python os module

The os module provides us the functions that are involved in file processing operations like renaming, deleting, etc.

Let's look at some of the os module functions.

Renaming the file

The os module provides us the rename() method which is used to rename the specified file to a new name.

The syntax to use the rename() method is…

rename(?current-name?, ?new-name?)

Example:

```
import os;
#rename file2.txt to file3.txt
os.rename("file2.txt","file3.txt")
```

Removing the file

The os module provides us the remove() method which is used to remove the specified file.

The syntax to use the remove() method is--

remove(?file-name?)

Example:

```
import os;
#deleting the file named file3.txt
os.remove("file3.txt")
```

Creating the new directory

The mkdir() method is used to create the directories in the current working directory.

The syntax to create the new directory is

mkdir(?directory name?)

Example:

```
import os;
#creating a new directory with the name new
os.mkdir("new")
```

Changing the current working directory

The chdir() method is used to change the current working directory to a specified directory.

The syntax to use the chdir() method is

```
chdir("new-directory")
```

Example:

```python
import os;
#changing the current working directory to new
os.chdir("new")
```

Changing the current working directory

The chdir() method is used to change the current working directory to a specified directory.

The syntax to use the chdir() method is

```
chdir("new-directory")
```

Example:

```
import os;
#changing the current working directory to new
os.chdir("new")
```

The getcwd() method

This method returns the current working directory.

The syntax to use the getcwd() method is

os.getcwd()

Example:

```
import os;
#printing the current working directory
print(os.getcwd())
```

Python Modules

A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.

Modules in Python provides us the flexibility to organize the code in a logical way.

To use the functionality of one module into another, we must have to import the specific module.

Example

In this example, we will create a module named as file.py which contains a function func that contains a code to print some message on the console.

Let's create the module named as file.py.

```
1.  #displayMsg prints a message to the name being passed.
2.  def displayMsg(name)
3.      print("Hi "+name);
```

Here, we need to include this module into our main module to call the method **displayMsg()** defined in the module named file.

Loading the module in our python code

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

1. The import statement
2. The from-import statement

The import statement

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.

The syntax to use the import statement is given below.

1. import module1,module2,........ module n

Hence, if we need to call the function **displayMsg()** defined in the file file.py, we have to import that file as a module into our module as shown in the example below.

Example:

1. import file;
2. name = input("Enter the name?")
3. file.displayMsg(name)

Output:

Enter the name?John

Hi John

The from-import statement

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement. The syntax to use the from-import statement is given below.

1. from < module-name> import <name 1>, <name 2>..,<name n>

Consider the following module named as calculation which contains three functions as summation, multiplication, and divide.

calculation.py:

```
#place the code in the calculation.py
def summation(a,b):
    return a+b
def multiplication(a,b):
    return a*b;
def divide(a,b):
    return a/b;
```

Main.py:

1. **from** calculation **import** summation
2. #it will import only the summation() from calculation.py
3. a = int(input("Enter the first number"))
4. b = int(input("Enter the second number"))
5. print("Sum = ",summation(a,b)) #we do not need to specify the module name while accessing summation()

Output:

Enter the first number10

Enter the second number20

Sum = 30

The from...import statement is always better to use if we know the attributes to be imported from the module in advance. It doesn't let our code to be heavier. We can also import all the attributes from a module by using *.

Consider the following syntax.

1. **from <module> import ***

Renaming a module

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

The syntax to rename a module is given below.

1. import <module-name> as <specific-name>

Example

1. #the module calculation of previous example is imported in this example as cal.
2. import calculation as cal;
3. a = int(input("Enter a?"));
4. b = int(input("Enter b?"));
5. print("Sum = ",cal.summation(a,b))

Output:

Enter a?10

Enter b?20

Sum = 30

Using dir() function

The dir() function returns a sorted list of names defined in the passed module. This list contains all the sub-modules, variables and functions defined in this module.

Consider the following example.

Example

1. import json

3. List = dir(json)

5. print(List)

Output:

['JSONDecoder', 'JSONEncoder', '__all__', '__author__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__path__', '__spec__', '__version__', '_default_decoder', '_default_encoder', 'decoder', 'dump', 'dumps', 'encoder', 'load', 'loads', 'scanner']

The reload() function

As we have already stated that, a module is loaded once regardless of the number of times it is imported into the python source file. However, if you want to reload the already imported module to re-execute the top-level code, python provides us the reload() function. The syntax to use the reload() function is given below.

1. **reload(<module-name>)**

for example, to reload the module calculation defined in the previous example, we must use the following line of code.

1. **reload(calculation)**

Python packages

The packages in python facilitate the developer with the application development environment by providing a hierarchical directory structure where a package contains sub-packages, modules, and sub-modules. The packages are used to categorize the application level code efficiently.

Let's create a package named Employees in your home directory. Consider the following steps.

1. Create a directory with name Employees on path /**home.**
2. Create a python source file with name ITEmployees.py on the path /**home/Employees.**

ITEmployees.py

```
1.     def getITNames():
2.         List = ["John", "David", "Nick",    "Martin"]
3.         return List;
```

3. Similarly, create one more python file with name BPOEmployees.py and create a function getBPONames().

4. Now, the directory Employees which we have created in the first step contains two python modules. To make this directory a package, we need to include one more file here, that is __init__.py which contains the import statements of the modules defined in this directory.

__init__.py

1. from ITEmployees import getITNames
2. from BPOEmployees import getBPONames

5. Now, the directory Employees has become the package containing two python modules. Here we must notice that we must have to create __init__.py inside a directory to convert this directory to a package.

6. To use the modules defined inside the package Employees, we must have to import this in our python source file. Let's create a simple python source file at our home directory (/home) which uses the modules defined in this package.

Test.py

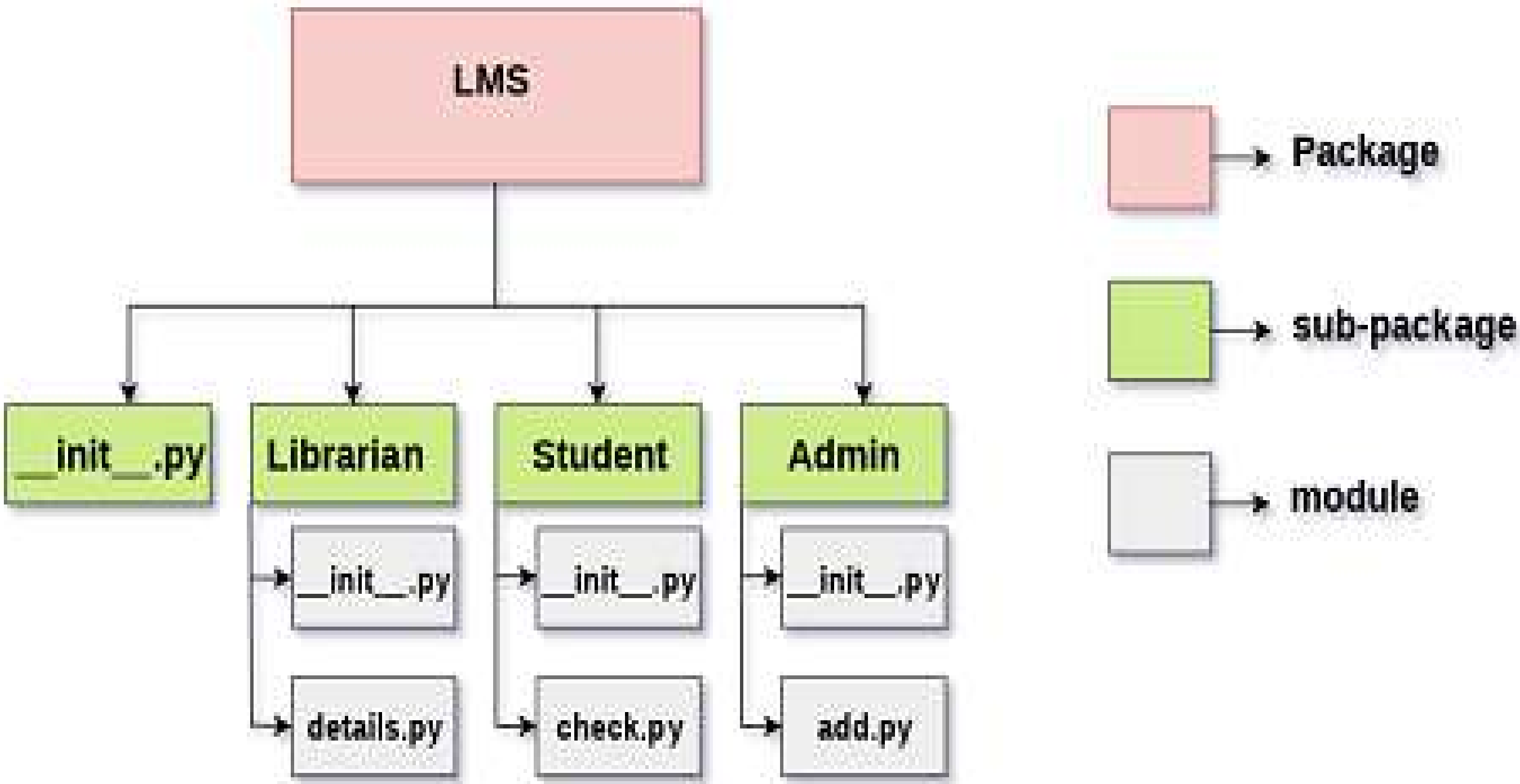
```
1.	import Employees
2.	print(Employees.getNames())
```

Output:

['John', 'David', 'Nick', 'Martin’]

We can have sub-packages inside the packages. We can nest the packages up to any level depending upon the application requirements.

The following image shows the directory structure of an application Library management system which contains three sub-packages as Admin, Librarian, and Student. The sub-packages contain the python modules.


LMS
__init__.py
Librarian
Student
Admin
__init__.py
details.py
__init__.py
check.py
__init__.py
add.py
Package
sub-package
module

Python datetime

Python provides the datetime module work with real dates and times. In real-world applications, we need to work with the date and time. Python enables us to schedule our Python script to run at a particular timing.

In Python, the date is not a data type, but we can work with the date objects by importing the module named with datetime, time, and calendar.

Example1-Get Current date & Time

```
import datetime

cdtime = datetime.datetime.now()

print(cdtime)
```

Output:

2020-09-01 10:18:36.461122

we have imported **datetime** module using **import datetime** statement.

One of the classes defined in the datetime module is datetime class. We then used now() method to create a datetime object containing the current local date and time.

Example2-Get Current date

```
import datetime
date1 = datetime.date.today()
print(date1)
```

Output:

2020-09-01

In this program, we have used **today()** method defined in the date class to get a date object containing the current local **date**.

Commonly used classes in the datetime module are:

- date Class
- time Class
- datetime Class
- timedelta Class

datetime.date Class

You can instantiate date objects from the date class. A date object represents a date (year, month and day).

Example 3: Date object to represent a date

```
import datetime

d = datetime.date(2020, 1, 7)

print(d)
```

Output:

2020-01-07

We can only import **date** class from the **datetime** module.

Here's how:

```
from datetime import date

a = date(2020, 4, 13)

print(a)
```

Output:

2020-04-13

Example 6: Print today's year, month and day

We can get year, month, day, day of the week etc. from the date object easily.

```
from datetime import date

# date object of today's date

today = date.today()

print("Current year:", today.year)

print("Current month:", today.month)

print("Current day:", today.day)
```

Output:

```
Current year: 2020

Current month: 9

Current day: 1
```

Example 7: Print hour, minute, second and microsecond

time object, you can easily print its attributes such as **hour**, **minute** etc.

```
from datetime import time

a = time(11, 34, 56)

print("hour =", a.hour)

print("minute =", a.minute)

print("second =", a.second)

print("microsecond =", a.microsecond)
```

Output:

```
hour = 11

minute = 34

second = 56

microsecond = 0
```

The calendar module

Python provides a calendar object that contains various methods to work with the calendars.

Example: Print the calendar for the month August-2020.

```
import calendar
cal = calendar.month(2020,8)
#printing the calendar of August 2020
print(cal)
```

Output:

Example : Print the calendar of year2020

```
import calendar

#printing the calendar of the year 2020

s = calendar.prcal(2020)
```

-:OOPs:-

Object Oriented programming is a programming style that is associated with the concept of Class, Objects and various other concepts revolving around these two, like Inheritance, Polymorphism, Abstraction, Encapsulation etc

.

Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

-:OOPs:-

OOPs Concept Definitions

The main features of Object Oriented Programming which you will be using in Python.

- ❖ Objects
- ❖ Classes
- ❖ Abstraction
- ❖ Encapsulation
- ❖ Inheritance
- ❖ Polymorphism

-:OOPs:-

Objects

It is a basic unit of Object Oriented Programming and represents the real life entities
The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.
Everything in Python is an object, and almost everything has attributes and methods.
all Objects shares two characteristics, they all have attributes and they all have behaviors. for example dogs have attributes like (name, color, type) and behaviors like (Running, barking, fetching etc).
Another Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

-:OOPs:-

Object is *a real world entity*.
Object is *a run time entity*.
Object is *an entity which has state and behavior*.
Object is *an instance of a class*.
Object is group of data and functions.
Object is minimal identifiable component in OOPS program.

Object

An object consists of:
State : It is represented by attributes of an object. It also reflects the properties of an object.

Behavior : It is represented by methods of an object. It also reflects the response of an object with other objects.

Identity : It gives a unique name to an object and enables one object to interact with other objects.

-:OOPs:-

Example of an object: dog

Identity

Name of dog

State/Attributes

Breed

Age

Color

Behaviors

Bark

Sleep

Eat

-:OOPs:-

Class-

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

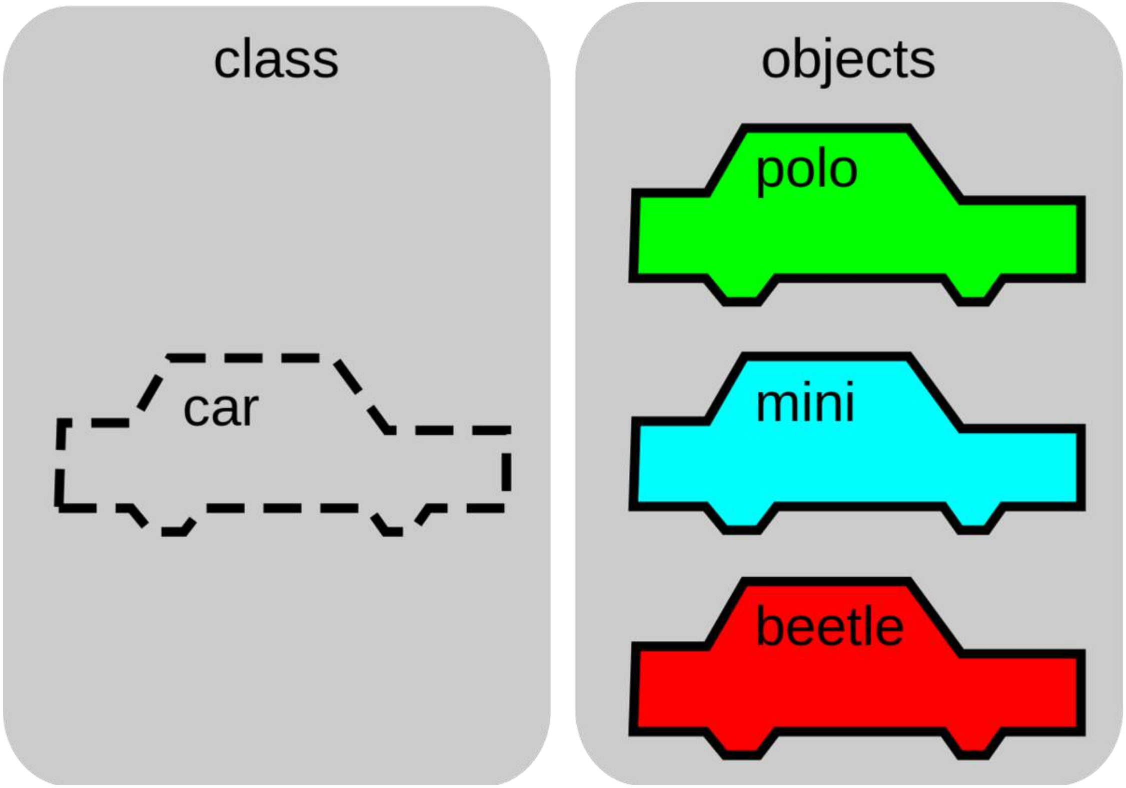
-:OOPs:-

Class-

Modifiers: A class can be public or has default access (Refer this for details).
Class name: The name should begin with a initial letter (capitalized by convention).
Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
Body: The class body in indent.

-:OOPs:-

Class



-:OOPs:-

Method

- The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

- The methods are used to implement the functionalities of an object.

- For example if we created ”start()” and ”stop()” methods for the ”Car” class

-:OOPs:-

Method

```
class Car:
 # create class attributes
     name = "c200"
     make = "mercedez"
     model = 2008

 # create class methods
     def start(self):
         print ("Engine started")
     def stop(self):
         print ("Engine switched off")
```

-:OOPs:-

Method

- In Above Program there is two method are created named "Start " and "Stop" under "Car" Class.We can call them with the help of object of the "Car" class .

- You can create more method or function in any class for reducing complexity and simplifying the code.

- Actually method is also called building block of the program.

-:OOPs:-

Abstraction

- Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonym because data abstraction is achieved through encapsulation.

- Abstraction is used to hide internal details and show only functionalities.

- Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does

-:OOPs:-

Abstraction

- ❖ Data abstraction allows us to transform a complex data structure into one that's simple and easy to use.
- ❖ Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation

-:OOPs:-

Abstraction

Functions as Abstraction Mechanisms

- An **abstraction** hides detail
  - Allows a person to view many things as just one thing
- We use abstractions to refer to the most common tasks in everyday life
  - For example, the expression “doing my laundry”
- Effective designers must invent useful abstractions to control complexity

-:OOPs:-

Encapsulation

- Data encapsulation is one of the fundamentals of OOP (object-oriented programming).
- It refers to the bundling of data with the methods that operate on that data.
- Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them

-:OOPs:-

Encapsulation

- Abstraction and Encapsulation are two important Object Oriented Programming (OOPS) concepts. Encapsulation and Abstraction both are interrelated terms.

- Real Life Difference Between Encapsulation and Abstraction is that Encapsulate means to hide. Encapsulation is also called data hiding.

- You can think Encapsulation like a capsule (medicine tablet) which hides medicine inside it

-:OOPs:-

Basic Different Between Abstraction and Encapsulation

- Encapsulation is used for hide the code and data in a single unit to protect the data from the outside the world. Class is the best example of encapsulation.
- Abstraction refers to showing only the necessary details to the intended user.
- . Abstraction is implemented using interface and abstract class
- Encapsulation is implemented using private and protected access modifier

-:OOPs:-

Inheritance

- Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in Python by which one class is allow to inherit the features(fields and methods) of another class.
- Inheritance is an OOPS concept in which one object acquires the properties and behaviors of the parent object. It's creating a parent-child relationship between two classes. It offers robust and natural mechanism for organizing and structure of any software.

-:OOPs:-

Inheritance

Important terminology:

Super Class: The class whose features are inherited is known as superclass(or a base class or a parent class).

Sub Class: The class that inherits the other class is known as subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

-:OOPs:-

Inheritance

- Inheritance allows us to inherit attributes and methods from the base/parent class.
- This is useful as we can create sub-classes and get all of the functionality from our parent class.
- Then we can overwrite and add new functionalities without affecting the parent class.

-:OOPs:-

Inheritance

- A class which inherits the properties is known as **Child** Class.
- A class whose properties are inherited is known as **Parent** class.
- Inheritance refers to the ability to create **Sub-classes** that contain specializations of their parents.

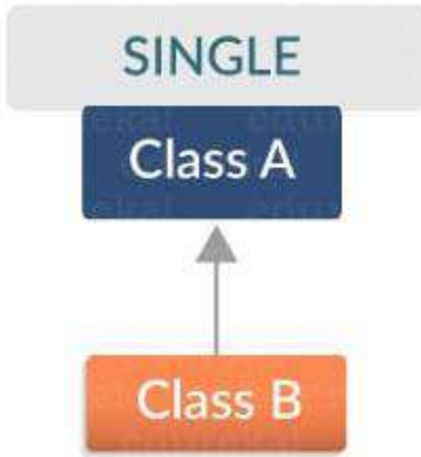
-:OOPs:-

Inheritance

-:OOPs:-

Inheritance

Single inheritance: When a child class inherits from only one parent class, it is called as single inheritance.



-:OOPs:-

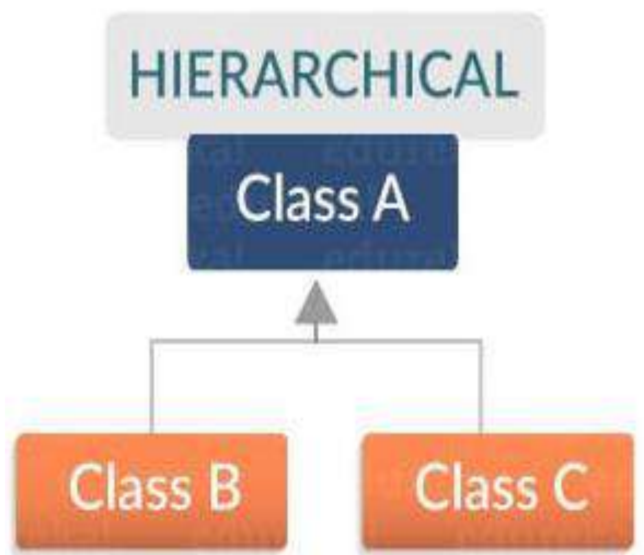
Inheritance

Multi-level inheritance :is archived 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 archived in python.

-:OOPs:-

Inheritance

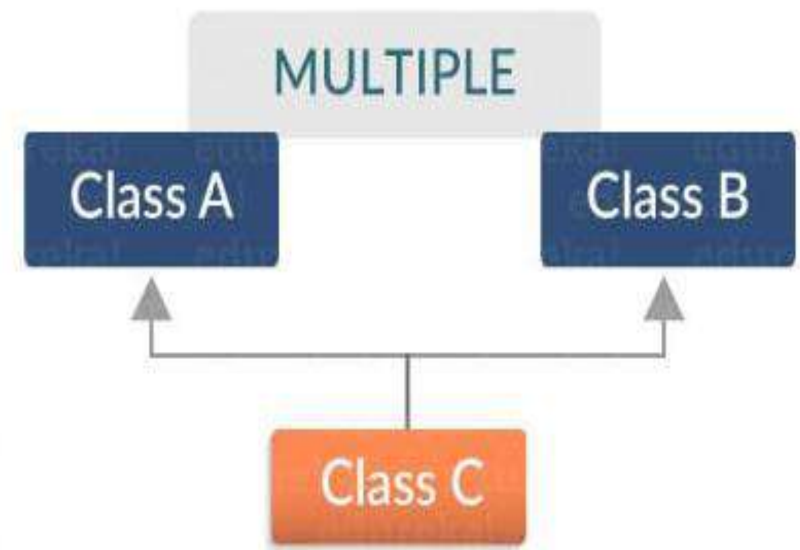
hierarchical inheritance: When more than one derived classes are created from a single base – it is called hierarchical inheritance.



-:OOPs:-

Inheritance

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



-:OOPs:-

Polymorphism

- Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

- Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

-:OOPs:-

Polymorphism

- Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).
- Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

-:OOPs:-

Advantage of OOPs

- OOP offers easy to understand and a clear modular structure for programs.
- Objects created for Object-Oriented Programs can be reused in other programs. Thus it saves significant development cost.
- Large programs are difficult to write, but if the development and designing team follow OOPS concept then they can better design with minimum flaws.
- It also enhances program modularity because every object exists independently

Sample program

```
class Add:


  def input(self, c, d):
    self.a = c
    self.b = d
  def process(self):
    self.sum = self.a + self.b
  def  show(self) :
    print(self.sum)

p1 = Add()
p1.input(10,20)
p1.process()
p1.show()
```

Object-oriented vs. Procedure-oriented Programming languages

The difference between object-oriented and procedure-oriented programming is given below:

Index	Object-oriented Programming	Procedural Programming
1.	Object-oriented programming is the problem-solving approach and used where computation is done by using objects.	Procedural programming uses a list of instructions to do computation step by step.
2.	It makes the development and maintenance easier.	In procedural programming, It is not easy to maintain the codes when the project becomes lengthy.

3.	It simulates the real world entity. So real-world problems can be easily solved through oops.	It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions.
4.	It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere.	Procedural language doesn't provide any proper way for data binding, so it is less secure.
5.	Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc.	Example of procedural languages are: C, Fortran, Pascal, VB etc.

Python Class and Objects

- We have already discussed in previous tutorial, a class is a virtual entity and can be seen as a blueprint of an object.
- The class came into existence when it instantiated. Let's understand it by an example.
- Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc.
- we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.
- On the other hand, the object is the instance of a class. The process of creating an object can be called instantiation.

Creating Classes

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows −
To create a class, use the keyword class:

```
class MyClass:
  x = 5
print(MyClass)
```

Create Object

You can access the object's attributes using the dot operator with object. Class variable would be accessed using class name. Now we can use the class named myClass to create objects:

```
class MyClass:
  x = 5

p1 = MyClass()
print(p1.x)
```

Object Functons

```
class MyClass:
  x = 5
  age=20

p1 = MyClass()

print(hasattr(p1, 'age'))                    # Returns true if 'age' attribute exists

print(getattr(p1 , 'age'))                   # Returns value of 'age' attribute

print(setattr(p1, 'age', 8))                 # Set attribute 'age' at 8

print("Set New value of Age Attributes")

print(getattr(p1 , 'age'))          #check the value is set or not

print(delattr(p1, 'age'))                    # Delete attribute 'age'

print("Value after deletion Display the old value")

print(getattr(p1 , 'age'))
```

The __init__() Function

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:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

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

Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

__dict__ − Dictionary containing the class's namespace.
__doc__ − Class documentation string or none, if undefined.
__name__ − Class name.
__module__ − Module name in which the class is defined. This attribute is "__main__" in interactive mode.
__bases__ − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Built-In Class Attributes

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print("Name : ", self.name,  ", Salary: ", self.salary)

print("Employee.__doc__:", Employee.__doc__)
print("Employee.__name__:", Employee.__name__)
print("Employee.__module__:", Employee.__module__)
print("Employee.__bases__:", Employee.__bases__)
print("Employee.__dict__:", Employee.__dict__)
```

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:

```
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()
```

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

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()
```

Delete Object Properties

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

Delete the age property from 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)

del (p1.age)

print(p1.age)
```

Delete Object

You can delete objects by using the del keyword:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)
    print("Age is ",self.age)

p1 = Person("John", 36)
p1.myfunc()
del p1
p1.myfunc()
```

The self-parameter

The self-parameter refers to the current instance of the class and accesses the class variables. We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.

Python – Access Specifier

Various object-oriented languages like C++, Java, Python control access modifications which are used to restrict access to the variables and methods of the class. Most programming languages has three forms of access modifiers, which are Public, Protected and Private in a class.

Python uses '_' symbol to determine the access control for a specific data member or a member function of a class. Access specifiers in Python have an important role to play in securing data from unauthorized access and in preventing it from being exploited.

A Class in Python has three types of access modifiers –

- ❖ Public Access Modifier
- ❖ Private Access Modifier
- ❖ Protected Access Modifier

Public Access Modifier

The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

Example1:

```
class employee:
   def __init__(self, name, sal):
      self.name=name
      self.salary=sal
e1=employee("Shiv",12000)
print(e1.name)
print(e1.salary)
```

Python – Access Specifier

Example2:

```
class employee:
   def __init__(self, name, sal):
      self.name=name
      self.salary=sal
   def display(self):
      print(self.name)
      print(self.salary)
e1=employee("Shiv",12000)
print(e1.name)
print(e1.salary)
print("**********")
print(e1.display())
```

Python – Access Specifier

Example3: Public data members can also be access on python shell.

You can access employee class's attributes and also modify their values,

```
>>> e1=employee ("Kiran", 10000)
>>> e1.salary
10000
>>> e1.salary=20000
>>> e1.salary
20000
```

Private Access Modifier:

The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore ‘__’ symbol before the data member of that class.

Example1:

```
class employee:
   def __init__(self,name,sal):
      self.__name=name
      self.__salary=sal
   def display(self):
      print(self.__name)
      print(self.__salary)
e1=employee("Shiv",12000)
print(e1.display())
```

Private Access Modifier:

Note in the example1 we see that the value of variable is access through member function. If you access the variable direct through object

Example:

```
class employee:
    def __init__(self,name,sal):
        self.__name=name
        self.__salary=sal
    def display(self):
        print(self.__name)
        print(self.__salary)
e1=employee("Shiv",12000)
print(e1.name)
print(e1.salary)
```

Output:

'employee' object has no attribute 'name'

Private Access Modifier:

If you try to access variable on Python shell

Example3:

```
>>> e1=employee("Bill",10000)
>>> e1.__salary
```

AttributeError: 'employee' object has no attribute '__salary'

Protected Access Modifier

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore '_' symbol before the data member of that class.

Example3:

```
class employee:
   def __init__(self,name,sal,age):
      self._name=name
      self._salary=sal
      self._age=age
class person(employee):
   def display(self):
      print(self._name)
      print(self._salary)
      print(self._age)
p=person("John",20000,29)
print(p.display())
```

Python Constructor

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.

Constructors can be of two types.

1. **Parameterized Constructor**
2. **Non-parameterized Constructor**

Constructor definition is executed when we create the object of this class.

Constructors also verify that there are enough resources for the object to perform any start-up task.

Creating the constructor in python

In Python, the method the **__init__()** simulates the constructor of the class. This method is called when the class is instantiated. It accepts the **self**-keyword as a first argument which allows accessing the attributes or method of the class.

We can pass any number of arguments at the time of creating the class object, depending upon the **__init__()** definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the **Employee** class attributes.

Example

```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))


emp1 = Employee("John", 101)
emp2 = Employee("David", 102)
```

accessing display() method to print employee 1 information

emp1.display()

accessing display() method to print employee 2 information

emp2.display()

```
Output:
ID: 101
Name: John
ID: 102
Name: David
```

Counting the number of objects of a class

The constructor is called automatically when we create the object of the class. Consider the following example.

Example

```
class Student:
    count = 0
    def __init__(self):
        Student.count = Student.count + 1
s1=Student()
s2=Student()
s3=Student()
print("The number of students:",Student.count)
```

Output:
The number of students: 3

Python Non-Parameterized Constructor

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument. Consider the following example.

Example

```
class Student:
    # Constructor - non parameterized
    def __init__(self):
        print("This is non parametrized constructor")
    def show(self,name):
        print("Hello",name)
student = Student()
student.show("John")
```

Python Parameterized Constructor

The parameterized constructor has multiple parameters along with the **self**. Consider the following example.

Example

```
class Student:
    # Constructor - parameterized
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name
    def show(self):
        print("Hello",self.name)
student = Student("John")
student.show()
```

Output:
This is parametrized constructor
Hello John

Python Default Constructor

When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor. It does not perform any task but initializes the objects. Consider the following example.

Example

```
class Student:
    roll_num = 101
    name = "Joseph"

    def display(self):
        print(self.roll_num,self.name)

st = Student()
st.display()
```

Output:
101 Joseph

More than One Constructor in Single class

Let's have a look at another scenario, what happen if we declare the two same constructors in the class.

Example

```
class Student:
    def __init__(self):
        print("The First Constructor")
    def __init__(self):
        print("The second contructor")

st = Student()
```

Output:

The Second Constructor

In the above code, the object **st** called the second constructor whereas both have the same configuration. The first method is not accessible by the **st** object. Internally, the object of the class will always call the last constructor if the class has multiple constructors.

Note: The constructor overloading is not allowed in Python.

Destructors in Python

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically.

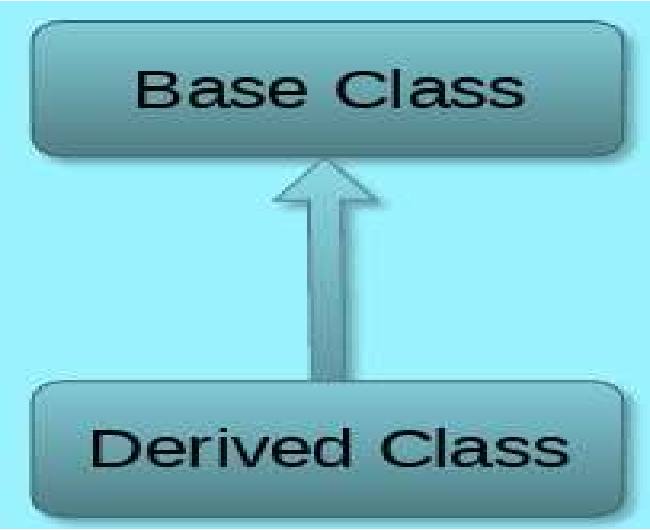
The __del__() method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Example

```
class student:
   # Initializing
  def __init__(self):
     print('Student Data created created.')
   # Deleting (Calling destructor)
  def __del__(self):
     print('Destructor called, Student Data deleted.')
 obj = student()
del obj
```

Python Inheritance

- Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class.
- A child class can also provide its specific implementation to the functions of the parent class.
- In this section of the tutorial, we will discuss inheritance in detail.
- In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name.
- Consider the following syntax to inherit a base class into the derived class.



Syntax

class derived-class(base class):

<class-suite>

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

Syntax

```
class derive-class(<base class 1>, <base class 2>, ..... <base class n>):
    <class - suite>
```

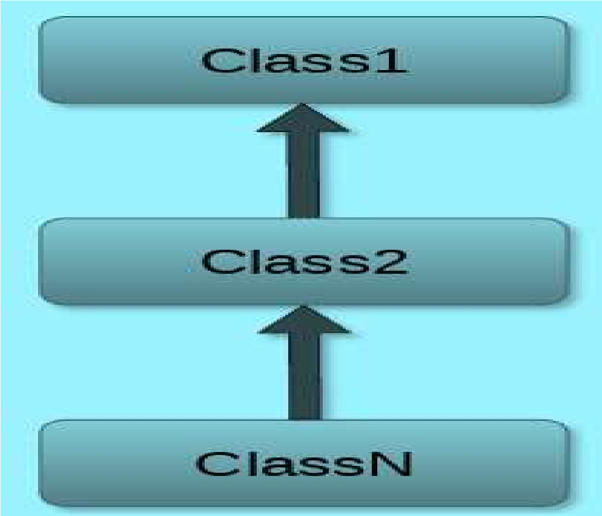
Example 1

```
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()
```

Output:
dog barking
Animal Speaking

Python Multi-Level inheritance

- Multi-Level inheritance is possible in python like other object-oriented languages.
- Multi-level inheritance is archived 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 archived in python.



The syntax of multi-level inheritance is given below.

Syntax

```
1.      class class1:
2.          <class-suite>
3.      class class2(class1):
4.          <class suite>
5.      class class3(class2):
6.          <class suite>
```

Example

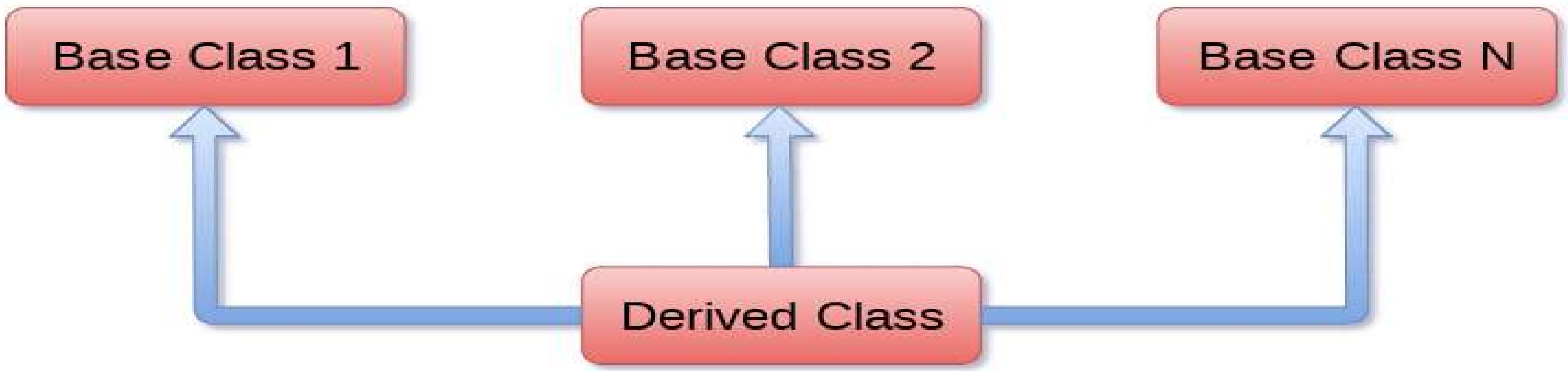
```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating 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.



The syntax to perform multiple inheritance is given below.

Syntax

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ...... BaseN):
    <class-suite>
```

Example

```
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

The issubclass(sub,sup) method

The issubclass(sub, sup) method is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise.

Consider the following example.

Example

```
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(issubclass(Derived,Calculation2))
print(issubclass(Calculation1,Calculation2))
```

Output:

True

False

The isinstance (obj, class) method

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

Consider the following example.

Example

```
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(isinstance(d,Derived))
```

Output:

```
True
```

Method Resolution Order (MRO)

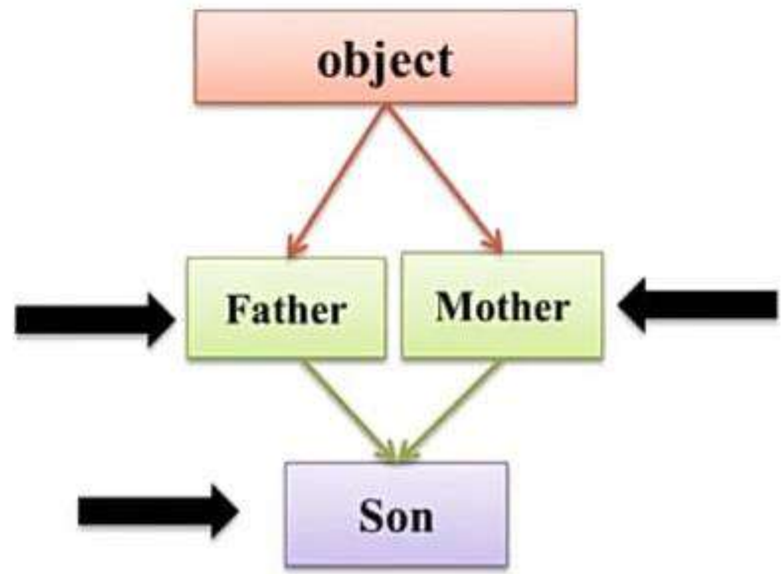
In the multiple inheritance scenario members of class are searched first in the current class. If not found, the search continues into parent classes in depth-first, left to right manner without searching the same class twice.

- Search for the child class before going to its parent class.
- When a class is inherited from several classes, it searches in the order from left to right in the parent classes.
- It will not visit any class more than once which means a class in the inheritance hierarchy is traversed only once exactly.

Method Resolution Order (MRO)

s = Son()

- The search will start from Son. As the object of Son is created, the constructor of Son is called.
- Son has super().__init__() inside his constructor so its parent class, the one in the left side 'Father' class's constructor is called.
- Father class also has super().__init__() inside his constructor so its parent 'object' class's constructor is called.
- Object does not have any constructor so the search will continue down to right hand side class (Mother) of object class so Mother class's constructor is called.
- As Mother class also has super().__inti__() so its parent class 'object' constructor is called but as object class already visited, the search will stop here.



Python Exceptions

We can make certain mistakes while writing a program that lead to errors when we try to run it. A python program terminates as soon as it encounters an unhandled error. These errors can be broadly classified into two classes:

- Syntax errors
- Logical errors (Exceptions)

Python Syntax Errors

Error caused by not following the proper structure (syntax) of the language is called **syntax error** or **parsing error**.

Python Exceptions

Python Logical Errors (Exceptions)

Errors that occur at runtime (after passing the syntax test) are called **exceptions** or **logical errors**.

For instance, they occur when we try to open a file(for reading) that does not exist (FileNotFoundError), try to divide a number by zero (ZeroDivisionError), or try to import a module that does not exist (ImportError).
Whenever these types of runtime errors occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Python Exceptions

- Example

#Demo without exception handling	#Demo with exception handling

<pre>a =12
b = 0
c = a/b
print(c)
print("Hello")
d =a+b
print(d)</pre>
	<pre>try :
	    a =12
	    b = 0
	    c = a/b
	    print(c)
	except:
	    print("Exception found ")
	print("Hello")
	d =a+b
	print(d)</pre>

Python Exceptions

An exception can be defined as an abnormal condition in a program resulting in the disruption in the flow of the program.

Whenever an exception occurs, the program halts the execution, and thus the further code is not executed. Therefore, an exception is the error which python script is unable to tackle with.

Python provides us with the way to handle the Exception so that the other part of the code can be executed without any disruption. However, if we do not handle the exception, the interpreter doesn't execute all the code that exists after the that.

Common Exceptions

A list of common exceptions that can be thrown from a normal python program is given below.

- **ZeroDivisionError:** Occurs when a number is divided by zero.
- **NameError:** It occurs when a name is not found. It may be local or global.
- **IndentationError:** If incorrect indentation is given.
- **IOError:** It occurs when Input Output operation fails.
- **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

List of Exceptions

Exception Base class for all exceptions	**StopIteration** Raised when the next() method of an iterator does not point to any object.
SystemExit Raised by the sys.exit() function.	**StandardError** Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError Base class for all errors that occur for numeric calculation.	**OverflowError** Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError Raised when a floating point calculation fails.	**ZeroDivisionError** Raised when division or modulo by zero takes place for all numeric types.
AssertionError Raised in case of failure of the Assert statement.	**AttributeError** Raised in case of failure of attribute reference or assignment.
EOFError Raised when there is no input from either the raw_input() or input() function and the end of file is reached.	**ImportError** Raised when an import statement fails.

List of Exceptions

IndexError Raised when an index is not found in a sequence.	**LookupError** Base class for all lookup errors.
KeyboardInterrupt Raised when the user interrupts program execution, usually by pressing Ctrl+c.	**UnboundLocalError** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
KeyError Raised when the specified key is not found in the dictionary.	**NameError** Raised when an identifier is not found in the local or global namespace.
EnvironmentError Base class for all exceptions that occur outside the Python environment.	**IOError** Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
IOError Raised for operating system-related errors.	**SyntaxError** Raised when there is an error in Python syntax.
IndentationError Raised when indentation is not specified properly.	**SystemError** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.

List of Exceptions

SystemExit Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit**.**	**TypeError** Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.	**RuntimeError** Raised when a generated error does not fall into any category.
NotImplementedError Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.	

Problem without handling exceptions

As we have already discussed, the exception is an abnormal condition that halts the execution of the program. Consider the following example.

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = a/b;
print("a/b = %d"%c)

#other code:
print("Hi I am other part of the program")
```

Output

```
Enter a: 10
Enter b: 0
Traceback (most recent call last):
        File "exception-test.py", line 3, in <module>
                c = a/b;
ZeroDivisionError: division by zero
```

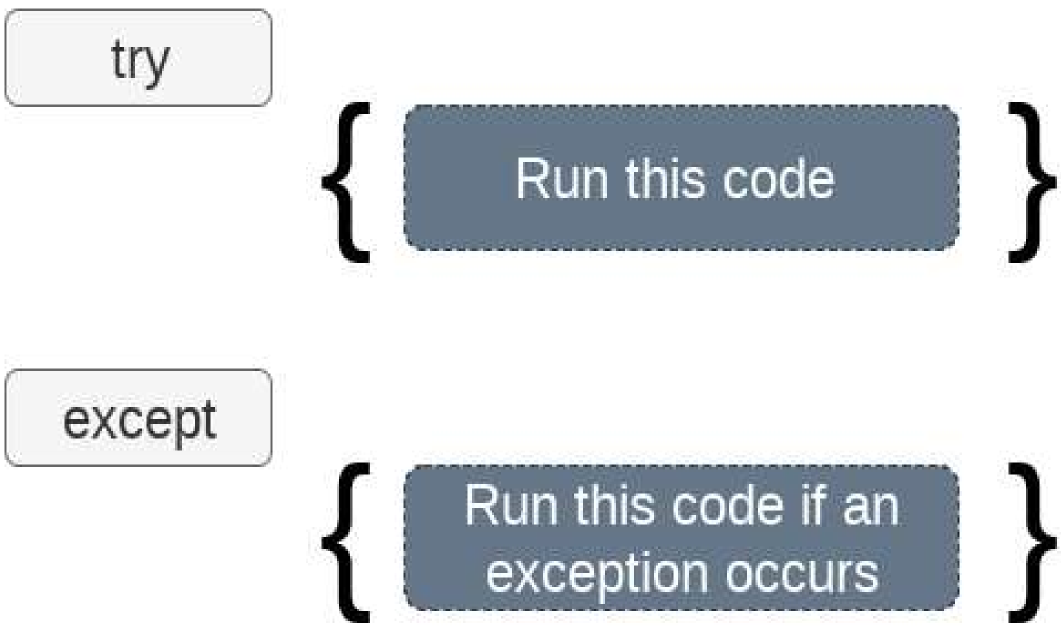
Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

```
try:
        You do your operations here;
        ......................

except ExceptionI:
        If there is ExceptionI, then execute this block.

except ExceptionII:
        If there is ExceptionII, then execute this block.
        ......................

else:
        If there is no exception then execute this block.
```

Exception Handling

If the python program contains suspicious code that may throw the exception, we must place that code in the try block. The try block must be followed with the except statement which contains a block of code that will be executed if there is some exception in the try block.



Try…. except

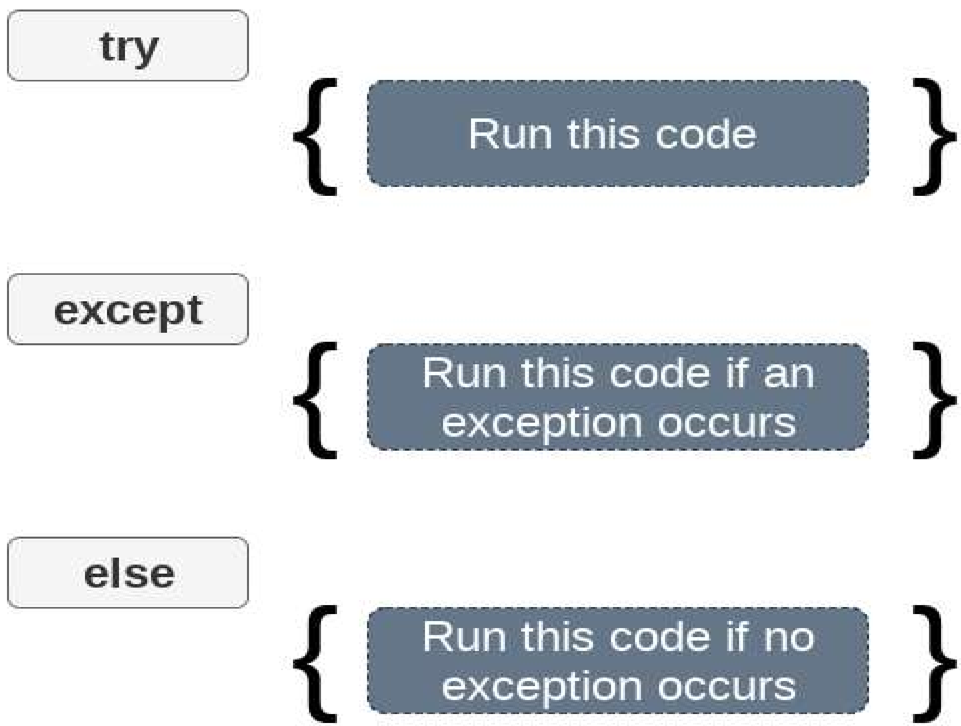
```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b;
    print("a/b = %d"%c)
except Exception:
    print("can't divide by zero")
else:
    print("Hi I am else block")
```

Output

Enter a: 10
Enter b: 0
can’t divide by zero

Try except Statement

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.



The except statement with no exception

Python provides the flexibility not to specify the name of exception with the except statement.
Consider the following example.

```
try:
   a = int(input("Enter a:"))
   b = int(input("Enter b:"))
   c = a/b;
   print("a/b = %d"%c)
except:
   print("can't divide by zero")
else:
   print("Hi I am else block")
```

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

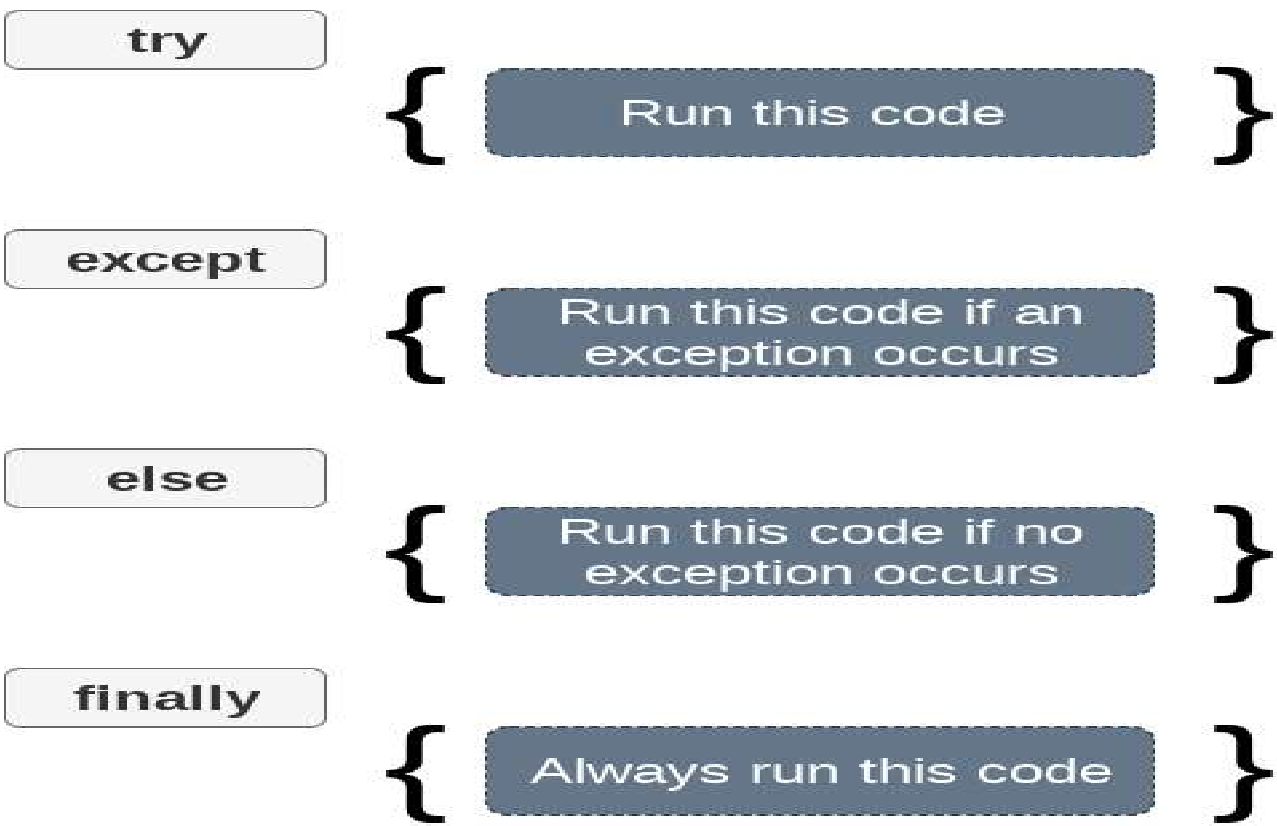
Declaring multiple exceptions

The python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions.

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

The finally block

The finally block will always be executed, no matter if the try block raises an error or not:



The finally block

We can use the finally block with the try block in which, we can pace the important code which must be executed before the try statement throws an exception.

The syntax to use the finally block is given below.

```
try:
    fileptr = open("file.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
```

Exception IndexError

An IndexError is raised when a sequence is referenced which is out of range.

Example :

```
import array as arr
x=arr.array('i',[10,20,30,40,50])
print(x[10])
```

Output :

```
print(x[10])
IndexError: array index out of range
```

Exception KeyError

A KeyError is raised when a mapping key is not found in the set of existing keys.

Example :

```
array = { 'a':1, 'b':2 }
print array(['c'] )
```

Output :

Traceback (most recent call last): File "exceptions_KeyError.py", line 13, in print array['c'] KeyError: 'c'

Exception NameError

This error is raised when a local or global name is not found. For example, an unqualified variable name.

Example :

```
def func():
    print ans

func()
```

Output :

Traceback (most recent call last): File NameError: global name 'ans' is not defined

Exception ValueError

A ValueError is raised when a built-in operation or function receives an argument that has the right type but an invalid value.

Example :

print int('a')

Output :

Traceback (most recent call last): File ValueError: invalid literal for int() with base 10: 'a'

Exception ZeroDivisionError

A ZeroDivisionError is raised when the second argument of a division or modulo operation is zero. This exception returns a string indicating the type of the operands and the operation.

Example :

```
print 1/0
```

Output :

Traceback (most recent call last): File "line 1, in print 1/0
ZeroDivisionError: integer division or modulo by zero

Exception ImportError

An ImportError is raised when the import statement is unable to load a module or when the “from list” in from … import has a name that cannot be found.

Example :

import module_does_not_exist

Output :

Traceback (most recent call last): File "exceptions_ImportError_nomodule.py", line 12, in import module_does_not_exist ImportError: No module named module_does_not_exist

Exception TypeError

TypeErrors are caused by combining the wrong type of objects, or calling a function with the wrong type of object.

Example :

```
ny = 'Statue of Liberty'
my_list = [3, 4, 5, 8, 9]
print  my_list + ny
```

Output:

```
Traceback (most recent call last):
  File "C:/Users/NIELIT/Desktop/test1.py", line 3, in <module>
    print (my_list + ny)
TypeError: can only concatenate list (not "str") to list
```

Raising exceptions

An exception can be raised by using the raise clause in python. The syntax to use the raise statement is given below.

syntax

raise Exception_class,<value>

Points to remember

- To raise an exception, raise statement is used. The exception class name follows it.
- An exception can be provided with a value that can be given in the parenthesis.
- To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.

Raising exceptions

Example 1

```
try:
    age = int(input("Enter the age?"))
    if age<18:
        raise ValueError;
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

Example 2

```
try:
    a = int(input("Enter a?"))
    b = int(input("Enter b?"))
    if b is 0:
        raise ArithmeticError;
    else:
        print("a/b = ",a/b)
except ArithmeticError:
    print("The value of b can't be 0")
```

Custom Exception

The python allows us to create our exceptions that can be raised from the program and caught using the except clause. However, we suggest you read this section after visiting the Python object and classes.

```
class UnderAge(Exception):
  pass
def verify_age(age):
   try :
     if int(age) < 18:
        raise UnderAge
     else:
        print('Age: '+str(age))
   except :
      print("Exception Under age found ")
# main program
verify_age(23)  # won't raise exception
verify_age(17)  # will raise exception
print("Hello")
```

Example of Control structure(if)

EX-

```
a = 10
b = 20
if a == b:
   print('yes')
else:
   print('no')
```

Ex- 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 !!");
```

Ex- non zero represent true

```
n=1
if n :
   print("True")
else :
   print("False")
```

Ex- Zero represent false

```
n=0
if n :
   print("True")
else :
   print("False")
```

Ex If with true condition

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

Ex If with indentation

```
a = 33
b = 200
if b > a:
     print("b is greater than a")
     print("This is if part ")
print("This is outside if ")
```

Ex Simple if else demo

```
a = 33
b = 200
if b > a:
     print("b is greater than a")
else :
    print("This is outside if part")
```

Ex if else if demo

```
a = 33
b = 200
if b > a:
     print("b is greater than a")
elif a==b :
    print("both are same")
elif b > a :
    print("a is greater than b")
```

Ex- Nested If

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

Ex one line if else demo

```
a = 2
```

Ex if with two condition using “and”

```
a = 200
b = 33
c = 500
```

```
b = 330
print("A") if a > b else print("B")
```

```
if a > b and c > a:
  print("Both conditions are True")
```

Ex if with two condition using “or”

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the
conditions is True")
```

Ex nested if demo

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

Ex-Check Greater Number between Two Number:

```
a=int(input("Enter First Number"))
b=int(input("Enter Second Number"))
if a>=b :
   print(" A is Greater= ",a)
else:
   print(" B is greater= ",b)
```

Ex- Enter a number & Check the number is even or odd.

```
num=int(input("Enter  Number"))
if num%2==0 :
    print(" Number is Even= ",num)
else:
    print(" Number is Odd= ",num)
```

Ex Largest among three number

```
a=int(input("Enter First Number"))
b=int(input("Enter Second Number"))
c=int(input("Enter Third Number"))
if a>=b and a>=c:
   print(" A is Greater= ",a)
elif b>=c and b>=a:

   print(" B is greater= ",b)
else:
   print(" C is greater= ",c)
```

Ex

```
a = 10
b = 11
c = 10
if a == b:
    print('first condition is true')
elif a == c:
    print('second condition is true')
else:
    print('nothing is true. existence is
pain.')
```

Ex Simple calculator

```
A=int(input("Enter First Number"))
B=int(input("Enter Second Number"))
C=input("Enter Operation Like (+,-,*,/,//,%)
As per Your Choice")
if C=='+' :
   sum=A+B
   print(" Sum of these Two Number= ",sum)
elif C=='-':
   sub=A-B
   print(" Subtraction of these Two Number=
",sub)
elif C=='*':
   multi=A*B
```

Ex-

```
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");
```

print(" Multiplication of these Two Number= ",multi)
elif C=='/':
   div=A/B
   print(" Division of these Two Number= ",div)
elif C=='//':
   div1=A/B
   print(" Integer Division of these Two Number= ",div1)
elif C=='%':
   Mod1=A%B
   print(" Modules of these Two Number= ",Mod1)
else:
   print("Your choice is wrong Try Again")	

1-Program for Calculating Factorial.

```
n=int(input("Enter number for
factorial"))
f=1
for i in range(1,n+1,1):
   f=f*i
print("Factorial= ",f)
```

2-Using Else Statement in Loop

```
x=['Delhi','Mumbai','Chennai','Kanpur','Noida','Lucknow']
city_name=input("Enter City Name ")

for i in x:
   if i==city_name:
      print("City Found")
      break
else:
   print("City Not Found")
```

While Loop

1-Print a message 5 times using while loop.

```
i=1
while i<=5:
   print("Hello")
   i=i+1
```

2-Method2-

```
i=1
while i in range(11):
   print("Hello")
   i=i+1
```

3- Factorial By Using While Loop

```
num=int(input("Enter Number"))
i=1
f=1
while i<=num:
   f=f*i
   i=i+1
print("Factorial Number=",f)
```

4-Program to Reverse of a number

```
num=int(input("Enter Number"))
rev=0
while num>0:
   a=num%10
   rev=rev*10+a
   num=num//10
print("Reverse of Number ",rev)
```

5-Program to Find Sum of Digit:

```
num=int(input("Enter Number"))
sum=0
while num>0:
   a=num%10
   sum=sum+a
   num=num//10
print("Reverse of Number ",sum)
```

6-Program for Palindrome Number

```
num=int(input("Enter Number"))
num1=num
rev=0
while num>0:
   a=num%10
   rev=rev*10+a
   num=num//10
print("Reverse of Number ",rev)

if(rev==num1):
   print("Number is Palindrome Number")
else:
   print("Number is not Palindrome Number")
```

7-Fibnoacci Series:

```
N=5

f1 = 0
f2 = 1

print( f1 )
print( f2 )

for k in range(N):
```

8-Concept Odd Loop

```
ch='Y'
while ch=='Y' or ch=='y':
   p=int(input("Enter Amount"))
   t=int(input("Enter Time"))
   r=int(input("Enter Rate"))
   s=p*r*t/100
   print("Simple Intrest=",s)
   print("Do you want to another Calculation Y/N? ")
   ch=input()
```

f_next = f2 +f1 f1 =f2 f2 =f_next print(f_next)	

8-Prime_Number
 	
num = int(input("Enter a number (greater than 1)"))
f = 0
i = 2
while i <= num / 2:
  if num % i == 0:
    f=1
    break
  i=i+1
 	
if f==0:
  print("The entered number is a PRIME number")
else:
  print("The entered number is not a PRIME number")	

Ex1 – Print First 10 natural numbers using for loop

```
for x in range(10):
    print("The Value of X", x)
```

The Output of this program is 0 to 9 .Because loop start intial value 0 show loop execute till 9. If we want to start loop from 1 to 10 then the code written as follows.

```
for x in range(1,11):
        print("The Value of X", x)
```

Because its execute less than 1 so the value written 11 replace of 10

Ex7- python program to print odd numbers within a given range.

```
lower=int(input("Enter the lower limit for the range:"))
upper=int(input("Enter the upper limit for the range:"))
for i in range(lower,upper+1):
    if(i%2!=0):
        print("Number is Even",i)
    else:
        print("Number is Odd= ",i)
```

EX2-Program to write a table using for loop.

```
#table of two
for x in range(2,22,2):
    print(x,end=' ')
```

Loop Start with initial value 2 & end with 20 & skip value by 2

Output:

2 4 6 8 10 12 14 16 18 20

Ex8-Program to Find Sum of even & Odd number between 1 to 20.

```
even=0
odd=0
for i in range(1,21,1):
    if(i%2==0):
        even=even+i
    else:
        odd=odd+i
print("Sum of Even Number= ",even)
print("Sum of Odd Number= ",odd))
```

Output:

Sum of Even Number= 110

Sum of Odd Number= 100

EX3-Program to write a table using for loop.(Any Number)

```
#Write Table of Any Number
num=int(input("Enter Number to Calculate table of Any Number Input by User"))
for x in range(1,11,1):
    t=x*num
    print(t,end=" ")
```

Ex9-Find the Fibonacci Sequence

```
nterms = int(input("How many terms? "))
# first two terms
n1=0
n2 = 1
count = 0
```

Output:
Enter Number to Calculate table of Any Number Input by User5
5 10 15 20 25 30 35 40 45 50

```
# check if the number of terms is valid
if nterms <= 0:
  print("Please enter a positive integer")
elif nterms == 1:
  print("Fibonacci sequence upto",nterms,":")
  print(n1)
else:
  print("Fibonacci sequence:")
  while count <= nterms:
     print(n1)
     nth = n1 + n2
     # update values
     n1 = n2
     n2 = nth
     count += 1
```

Ex4-Program to find sum of First 10 natural Number Sum.

```
#Program to Find First 10 natural number sum using for loop.
sum=0
for x in range(11):
   sum=sum+x
   print("Sum=",sum)
```

Output:
0 1 3 6 10 15 21 28 36 45 55

Sum= 55

Ex10-Program to check a input number is prime number or not.

```
# Program to check if a number is prime or not

#num = 407

# To take input from the user
num = int(input("Enter a number: "))

# prime numbers are greater than 1
if num > 1:
  # check for factors
  for i in range(2,num):
     if (num % i) == 0:
        print(num,"is not a prime number")

#print(i,"times",num//i,"is",num)
        break
```

```
  else:
     print(num,"is a prime number")

# if input number is less than
# or equal to 1, it is not prime
else:
  print(num,"is not a prime number")
```

Ex5-Program to Find natural number sum using for loop Where number is input through keyboard.

```
num=int(input("How many number to be want to sum"))
sum=0
for x in range(num+1):
   sum=sum+x
   print(sum,end=" ")

print("\n","Sum= ",sum)
```

Output:
How many number to be want to sum -5
0 1 3 6 10 15
Sum= 15

Ex11- Using Odd Loop.

```
ch='y'
while(ch=='y' or ch=='Y'):
   num = int(input("Enter a number: "))

   if num > 1:
      for i in range(2,num):
         if (num % i) == 0:
            print(num,"is not a prime
number")

print(i,"times",num//i,"is",num)
            break
      else:
         print(num,"is a prime number")

   else:
      print(num,"is not a prime
number")
   print("Do You Want to Check
Another Number (Y/N)? ")
   ch=input(" ")
```

Ex6-Program to Find Odd & Even Number
Between 1 to 20.

#Program to Find even & odd Number between 1 to 20.
for x in range(1,21,1):

<table>
<tr><td><pre>
    if(x%2==0):
        print("Even Number= ",x)
    else:
        print("Odd Number= ",x)
</pre>
Output: Display Number Between From 1 to 20</td><td></td></tr>
<tr><td></td><td></td></tr>
</table>

1-Create Array INTEGER VALUE (I)

```
import array as arr
x=arr.array('i',[10,20,30,40,50])
print(type(x))
print(x)
```

(II) **Create Array Character value**

```
mport array as ar
newarr = ar.array('u', ['N','I','E','L','I','T'])

print(newarr)

for i in range(len(newarr)):
   print(newarr[i],end="")
```

2-Print array elements using index

```
import array as arr
x=arr.array('i',[10,20,30,40,50])
print(type(x))
print(x)
print(x[0])
print(x[1])
print(x[2])
print(x[3])
print(x[4])
```

3-Display Array Elements using for Loop

```
import array as arr
x=arr.array('i',[10,20,30,40,50])
for i in range(len(x)):
   print(x[i],end=' ')
```

4-Enter Array element by using loop.

```
import array as arr
m=int(input("Enter the size of the array"))
x=arr.array('i',[])
for i in range(m):
   x.append(int(input("Enter Array Elements")))
```

5-Enter Array element by using loop & Display

```
import array as arr
m=int(input("Enter the size of the array"))
x=arr.array('i',[])
for i in range(m):
```

10-Program to add new elements in existing array at the end

```
import array as arr
x=arr.array('i',[10,20,30])
print("Array Before inserting Array Elements")
print(x)
print("Array after inserting Array Elements")
print(x)
x.append(60) x.append(50)
print(x)
```

11-Searching Elements in an Array

```
from array import*
x=array('i',[20,10,25,60,70,85,45])
print("Enter Element to be Searched")
num=int(input())
for i in range(len(x)):
   if(x[i]==num):
      print("Element Found")
      break
else:
   print("Element not  Found")
```

12-Insert element using insert method.

```
import array as ar
num = ar.array('i', [1, 2, 3, 5, 7, 10])
num.insert(1,9)
for x in num:
 print(x)
```

13-Delete Array element By Value

```
import array as ar
num = ar.array('i', [1, 2, 3, 5, 7, 10])
print("Array before delete Element")
for x in num:
 print(x)
num.remove(7)
for x in num:
 print(x)
```

14-Delete Array element By Index

```
import array as ar
num = ar.array('i', [1, 2, 3, 5, 7, 10])
print("Array before delete Element")
for x in num:
 print(x)
print("Array After delete Element")
```

```
   x.append(int(input("Enter Array
Elements")))
for i in range(len(x)):
   print(x[i],end=" ")
```

6-Slicing Python arrays

```
import array as ar
newarr = ar.array('d', [2.1, 4.5,3.5,4.2,3.3, 5.5])
print("First element:", newarr[0])
print("Second element:", newarr[1])
print("Last element:", newarr[-1])
print(newarr[2:5]) # 3rd to 5th
print(newarr[:])   # beginning to end
```

7-Changing Array Elements

```
import array as ar
num = ar.array('i', [1, 2, 3, 5, 7, 10])
num[0] = 0
print(num)
```

8-Adding Array Elements

We can add one item to the array using the append () method, or add several items using the extend () method.

Example1-

```
import array as ar
num = ar.array('i', [1, 2, 3])
num.append(4)
print("Display After Adding 4 in Array")
print(num)
print("Display After Extending Array")
num.extend([5, 6, 7])
print(num)
```

Example2-

```
import array as ar
num = ar.array('i', [1, 2, 3])
ln=len(num)
print(ln) #length
num[0] = 0  #changing first element
print("After Changing the First Element")
print(num)
print("After Appending the Element at last Position
By Default")
num.append(4) #appending 4 to array
print(num)
print("After Extending The Array Elements")
num.extend([5, 6, 7]) #extending numbers with
5,6,7
print(num)
print("Changing the Array Elements in a range")
```

```
num.pop(0)
for x in num:
 print(x)
```

15-Python Array Program to Find out a Student Marks & grade.

```
import array as arr
m=int(input("Enter the Number of Subject"))
x=arr.array('i',[])
sum=0
for i in range(m):
   x.append(int(input("Enter Each Subject
marks")))
   sum=sum+x[i]
avg=sum/m
print("Total Marks=",sum)
print("Average Marks=",avg)
if avg>=90:
   print("Grade= S")
   print("Remarks=Excellent")
elif avg>=80 and avg<90:
   print("Grade= A")
   print("Remarks=Very Good")
elif avg>=70 and avg<80:
   print("Grade= B")
   print("Grade= Good")
elif avg>=60 and avg<70:
   print("Grade= C")
   print("Grade= Satisfaction")
elif avg>=50 and avg<60:
   print("Grade= Pass")
   print("Grade= D")
else:
   print("Grade= F")
   print("Grade= Fail")
```

```
num[2:5]=ar.array('i',[25,35,40]) #Changing
Element 3,4,5
print(num)
```

9- Storing text Element in an array

```
import array as ar
num = ar.array('u', ['N','I','E','L','I','T'])
ln=len(num)
for i in range(ln):
   print(num[i])
```

16-Sorting Array Elements in Ascending Order

```
from array import *
#Initialize array
arr =array("i",[5, 2, 8, 7, 1]);
temp = 0;
#Displaying elements of original array
print("Elements of original array: ");
for i in range(0, len(arr)):
   print(arr[i], end=" ");
#Sort the array in ascending order
for i in range(0, len(arr)):
   for j in range(i+1, len(arr)):
      if(arr[i] > arr[j]):
         temp = arr[i];
         arr[i] = arr[j];
         arr[j] = temp;
print();
#Displaying elements of the array after sorting
print("Elements of array After in sorted in
ascending order: ");
for i in range(0, len(arr)):
   print(arr[i], end=" ");
```

17-Sorting Array Elements Descending Order

```
from array import *
#Initialize array
arr =array("i",[5, 2, 8, 7, 1]);
temp = 0;
#Displaying elements of original array
print("Elements of original array: ");
for i in range(0, len(arr)):
   print(arr[i], end=" ");
#Sort the array in ascending order
for i in range(0, len(arr)):
   for j in range(i+1, len(arr)):
      if(arr[i] < arr[j]):
         temp = arr[i];
         arr[i] = arr[j];
         arr[j] = temp;
print();
#Displaying elements of the array after sorting
print("Elements of array After in sorted in
ascending order: ");
for i in range(0, len(arr)):
   print(arr[i], end=" ");
```

1-#Program to Concept of single,Double & Triple Quotes

```
x='Python'
y="Python"
z="""Python"""
print("Data Type of X= ",type(x))
print("Data Type of Y= ",type(y))
print("Data Type of Z= ",type(z))
```

Output:

Data Type of X= <class 'str'>

Data Type of Y= <class 'str'>

Data Type of Z= <class 'str'>

8-#Python Program to Print String Using Loop.

```
#First Method
x="Python"
print("First Method>>>>>")
print("\n")
for c in x:
   print(c,end="")
#IIND METHOD
print("\n")
print("IInd Method>>>>>")
for i in range(len(x)):
   print(x[i],end="")
#print Character By using while loop
print("\n")
print("By Using While Loop>>>>>")
i=0
while i<len(x):
   print(x[i],end="")
   i=i+1
```

2-#Program to Find Out Length of String

```
x="Python Programming"
print("Length of X = ",len(x))
count=0
for i in x:
   count=count+1
print("Length Using For Loop",count)
```

Output:

Length of X = 18

Length Using For Loop 18

9-#Python program to Check Palindrome of string

```
s=input("Enter String")
str = ""
for i in s:
    str = i + str
print("The original string  is : ",end="")
print(s)
print("The reversed string(using loops) is : ",end="")
print(str)
if s==str:
    print("String is Plaindrome")
else:
    print("String is not plaindrome")
```

3-#Program to Concept of String Indexing
str1 ="PYTHON"
print("String Length",len(str1))
print(str1[0])
print(str1[1])
print(str1[2])
print(str1[3])
print(str1[4])
print(str1[5])
# It returns the IndexError because 6th index doesn't exist 	
print(str1[6])
Output:
String Length 6
P
Y
T
H
O
N
IndexError: string index out of range
	10-#program to count No. of vowels of string
	string=input("Enter string:")
	vowels=0
	for i in string:
	    if(i=='a' or i=='e' or i=='i' or i=='o' or i=='u'or i=='A' or
	    i=='E' or i=='I' or i=='O' or i=='U'):
	        vowels=vowels+1
	print("Number of vowels are:")
	print(vowels)
	Output:
	Enter string:Hello
	Number of vowels are:
	2
	
	String Function:

4-#Program to Concept of String slicing
str ='HELLOWORLD'
print(str[-1])
print(str[-3])
print(str[-2:])
print(str[-4:-1])
print(str[-7:-2])
# Reversing the given string 	
print(str[::-1])
# Search Character out of index 	
print(str[-12])
	1-Example: count()
	Return the number of times the value "apple" appears in the string:
	txt = "I love apples, apple are my favorite fruit"
	x = txt.count("apple")
	print(x)
	Output: 2
	
	2-Example: Search from index 10 to 24:

Output:
D
R
LD
ORL
LOWOR
DLROWOLLEH
print(str[-12])
IndexError: string index out of range
 	
Reverse of String:
s=input("Enter String")
str = ""
for i in s:
    str = i + str
    print(str)
print("The original string is : ",end="")
print(s)
	txt = "I love apples, apple are my favorite fruit"
	x = txt.count("apple", 10, 24)
	print(x)
	Output: 1
	find()
	3-Example: To find the first occurrence of the letter "e" in txt:
	txt = "Hello, welcome to my world."
	x = txt.find("e")
	print(x)
	Output: 1
	4-Example-
	Where in the text is the first occurrence of the letter "e" when you only search between position 5 and 10?
	txt = "Hello,welcome to my world."
	x = txt.find("e", 5, 10)
	print(x)
	Output: 7

5-#Update String Value
str1="python"
print("String=",str1)
str1="PYTHON"
print("String= ",str1)
#Update a Particular charcater give error
str1[0]='P'
print(str1)
Output
String= python
	rfind()
	Example: Where in the text is the last occurrence of the string "nielit"?:
	txt ="nielit Centre has started Python Programming course. nielit Centre"
	x = txt.rfind("nielit")
	print(x)
	Output:
	53

String= PYTHON **str1[0]='P'** **TypeError: 'str' object does not support item assignment**	
6a-#Delete String str1="python" print("String=",str1) del str1 print("String= ",str1) **Output:** **name 'str1' is not defined** **6b-#Delete a Particular charcater give error** str1="python" del str1[0] print(str1) **Output** **del str1[0]** **TypeError: 'str' object doesn't support item deletion**	**Example: Where in the text is the last occurrence of the letter "e" when you only search between position 5 and 10?** txt = "Hello, welcome to NIELIT" x = txt.rfind("e", 5, 10) print(x) **Output: 8** **Example: If the value is not found, the rfind() method returns -1** txt = "Hello, welcome to NIELIT." x = txt.rfind('nielit') print(x) **Output:** -1

7-#Example on String Operation

```
str1 = "Python"
str2 = "Program"
print(str1*3)                #prints PythonPythonPython
print(str1+str2)             #prints PythonProgram
print(str1[4])               #prints o
print(str2[2:4]);            #prints og
print('h' in str1)           #prints True as "h" is  present in str1
print('m' in str1)           #prints False as "m" is not present in str1
print('am' not in str2)      #prints False as "am" is present in str2.
print('n' not in str2)       #prints True as "n" is not present in str2.
print("The string str : %s"%(str1)) #prints The string str : Python
```

<table>
<tr><td colspan="2">

<u>Output:</u>
PythonPythonPython
PythonProgram
o
og
True
False
False
True
The string str : Python
</td></tr>
<tr><td>
capitalize():
Example: Upper case the first letter in this sentence:
txt = "hello, welcome to NIELIT Lucknow."
x = txt.capitalize()
print (x)
Output:
Hello, welcome to nielit lucknow.
lower()
txt = "Welcome To NIELIT Lucknow"
x = txt.lower()
print(x)
Output:
welcome to nielit lucknow
upper()
txt = "Welcome To NIELIT Lucknow"
x = txt.upper()
print(x)
Output:
WELCOME TO NIELIT LUCKNOW
</td><td>
title()

Example:
txt = "python programming using string"
x = txt.title()
print(x)
Output: Python Programming Using String
Example:
txt = " 3rd generation python"
x = txt.title()
print(x)
Output:
3Rd Generation Python

Example:
txt ="hello b2b2b2 and 3g3g3g"
x = txt.title()
print(x)
Output:
Hello B2B2B2 And 3G3G3G
</td></tr>
<tr><td></td><td></td></tr>
</table>

islower() Example: txt = "hello world!" x = txt.islower() print(x) **Output:** **True** **Example:** txt = "hello World!" x = txt.islower() print(x) **Output:** **False** **istitle()** **Example:** a ="HELLO, AND WELCOME TO MY WORLD" b ="Hello" c ="22 Names" d ="This Is %'!?" print(a.istitle()) print(b.istitle()) print(c.istitle()) print(d.istitle()) **Output:** **False** **True** **True** **True**	**isupper()** **Example:** txt ="PYTHON PROGRAM" x = txt.isupper() print(x) **Output:** **True** **Example:** txt ="PYTHON pROGRAM" x = txt.isupper() print(x) **Output:** **False** **replace()** **Example:** txt = "one one was a race horse, two two was one too." x = txt.replace("one", "three") print(x) **Output:** **three three was a race horse, two two was three too.** **Example:** txt = "one one was a race horse, two two was one too." x = txt.replace("one", "three",2) print(x) **Output:** **three three was a race horse, two two was one too.**
strip() **Example** txt = " banana " print(txt) x = txt.strip() print(x)	**lstrip()** **Example** txt = ",,,,,ssaaww.....banana.. " x = txt.lstrip(",.asw") print(x) **Output:** **banana..**

Output: **banana** **banana** **Example** txt = ",,,,,rrttgg.....apple....rrr" print(txt) x = txt.strip(",.grt") print(x) **Output:** **,,,,,rrttgg.....apple....rrr** **apple**	**rstrip()** **Example** txt = "banana,,,,,ssaaww....." x = txt.rstrip(",.asw") print(x)**Output:** **banana**
split(): **Example:** txt = "hello, my name is Peter, I am 26 years old" x = txt.split(", ") print(x) **Output:** **['hello', 'my name is Peter', 'I am 26 years old']**	**partition()** txt = "I could eat bananas all day" x = txt.partition("bananas") print(x) **Output:** **('I could eat ', 'bananas', ' all day')**
join() **Example:** List1 =("apple","Bannana") mySeparator = " and " x = mySeparator.join(List1) print(x) **Output:** **apple and Bannana** **Example:** List1 =["apple","bannana"] mySeparator = " and " x = mySeparator.join(List1) print(x) **Output:** **apple and banana**	**isspace()** **Example** txt = " s " x = txt.isspace() print(x) **Output:** **False** **Example** txt = "\n" x = txt.isspace() print(x) **Output:** **True**

Example: List1 ="apple" List2="bannana" mySeparator = " and " x = mySeparator.join(List1,List2) print(x) **Output:** **join() takes exactly one argument (2 given)**	

1-Program for print character based on ASCII Value

```
print (chr(65))
```

Output

A

2-Program to print ascii value to character & Corresponding character of ASCII Value using ordinal(ord)

```
ch="A"

print (chr(65))

print(ord(ch))
```

3-Display ASCII Value from 0 to 255

```
for x in range(0,255,1):
   print(chr(x), end=" ")
```

ord()-Used to Display a character ASCII Value
chr()-Used to Display Corresponding character of given ASCII Value

1-#Function to create Display message

```
def display():
   print("Hello, How are You")
display()
```

11 A-# Example of required parameter

```
def add(a,b):
   c=a+b;
   print("Sum=",c)
add(10,20)
```

2-#Function to Call more times to Display message

```
def display():
   print("Hello, How are You")
display()
display()
display()
```

11 B-# Example of required parameter

```
def add(a,b):
   c=a+b;
   print("Sum=",c)
add(10)
```

Output:
add() missing 1 required positional argument: 'b'

3-#Function to Call more times Using Loop

```
def display():
   print("Hello, How are You")
for i in range(1,10):
   display()
```

12A) # Example of Keyword Arguments

```
def si(p,r,t):
   s=p*r*t/100
   print("Simple Intrest=",s)
si(p=20000,r=10,t=5)
```

4-#Function to Add two number without using argument & No Return Type

```
def add(x,y):
   x=10
   y=20
   c=x+y
   print(c)
add(10,20)
```

12B-You Can also change the position of Keyword Arguments

```
def si(p,r,t):
   s=p*r*t/100
   print("Simple Intrest=",s)
si(r=10,t=5,p=20000)
```

5- A) #Function to Add two number without using argument & No Return Type

```
def add():
   x=int(input("Enter Firs Number= "))
   y=int(input("Enter Firs Number= "))
   c=x+y
   print(c)
add()
```

12C) You Can also Used Keyword; Argument Like this.

```
def si(p,r,t):
   s=p*r*t/100
   print("Simple Intrest=",s)
si(20000,r=10,t=5)
```

5-B) -#Function to Add two number with using argument & No Return Type

```
def add(x,y):
   c=x+y
   print(c)
x=int(input("Enter First Number"))
y=int(input("Enter Second Number"))
add(x,y)
```

12D) You Cannot used default argument after Used Keyword; Argument Like this.

```
def si(p,r,t):
   s=p*r*t/100
   print("Simple Intrest=",s)
si(,r=10,t=5, 20000)
```

Output:
positional arguments Follows Keyword arguments

6-#Function to Add two number with using argument & Return Type

```
def add(x,y):
   c=x+y
   return c
x=int(input("Enter First Number"))
y=int(input("Enter Second Number"))
```

12E) You Cannot used default argument after Used Keyword; Argument Like this.

```
def si(p,r,t):
   s=p*r*t/100
   print("Simple Intrest=",s)
si(r=10,20000,t=5)
```

z=add(x,y) print("Sum of Two Number=",z)	**Output:** **positional arguments Follows Keyword arguments**

7-#Function to Calculate Area of Circle
def area(r):
   area1=3.14*r*r
   return area1
 	
x=float(input("Enter Radius of Circle"))
y=area(x)
print("Area of Circle=",y)
	13- Default Argument Used For if no argument passed
	def si(p,r,t=5):
	   s=p*r*t/100
	   print("Simple Intrest=",s)
	si(2000,10)

8-#Function to Calculate Simple Intrest
def Simple_Intrest(p,r,t):
   s=p*r*t/100
   return s
x=int(input("Enter Principal Amount"))
y=float(input("Enter Rate of Intrest"))
z=float(input("Enter Time"))
S=Simple_Intrest(x,y,z)
print("Simple Intrest=",S)
	14-A-Variable Length of Argument
	def printname(name):
	   print(name)
	printname("Shiv")

9-#Return more then one value
def add(x,y):
   return(x+y,x-y,x*y,x/y)
a,b,c,d=add(10,5)
print("A=",a)
print("B=",b)
print("C=",c)
print("D=",d)
print("The Airthmetic Value are:",add(10,5))
# This return a Tuple value 	
	14-B) You can Pass any number of Arguments
	def printname(*name):
	   print(name)
	printname("Shiv","Mani","Ravi","Nitin","Mukesh")

10-# Example2-Return more then one value
def add(x,y):
   add=x+y
   multi=x*y
   return add,multi
a=int(input("Enter First Number"))
b=int(input("Enter Second Number"))
m,n=add(a,b)
print("Addition=",m)
print("Multiply=",n)
 	
print("The Airthmetic Value are:",add(a,b))
# This return a Tuple value 	
	9-#Function to calculate Factorial of a number
	def fact(num):
	   if num<=0:
	      print("Not Calculate Factorial")
	   else:
	      i=1
	      f=1
	      while(i<=num):
	         f=f*i
	         i=i+1
	      print("factorial of Given Number=",f)
	x=int(input("Enter Number For Factorial"))
	fact(x)

Global Variable & Local Variable

1-Program for Global & Local Variable Concept.

```
i=10     #global variable
def counter():
   i=20 #local variable of function counter
   print("Value of i in function:",i)
counter()
print("Value of i Outside Function:",i)
```

2-# global variable

```
a = 15
b = 10
 # function to perform addition
def add():
   c = a + b
   print(c)
# calling a function
```

3- You can update the value of global variable inside function

```
a = 15
 # function to change a global value
def change():
     # increment value of a by 5
   a = a+10
   print(a)
change()
```

Output:

```
 a = a+10
UnboundLocalError: local variable 'a' referenced before assignment
```

4-Using Global Keyword Inside a Function

```
#Python program to modify a global
#value inside a function
x=15
def change():
   global x  # using a global keyword
   x = x + 5 # increment value of a by 5
   print("Value of x inside a function :", x)
change()
print("Value of x outside a function :", x)
```

5-Concept of Doc String -Example1

```
def my_function():
   """Demonstrate docstrings and does nothing really."""
   return None
print("Printing DocString Using __doc__:")
print(my_function.__doc__ )
print("Printing DocString Using help function:")
help(my_function)
```

Example2:

```
def msg1():
     '''This Function to design for Display message'''

#print(msg1.__doc__)

help(msg1)
```

6- Concept of Doc String-Example3

```
def add():
   '''the add function is used to add two number'''
   return None
   x=10
   y=20
   z=x+y
   print("Sum=",z)
print(add())
print(add.__doc__)
print("Doc String Value by Using help Function")
help(add)
```

Program to Perform Arithmetic Operation by Using Function

```
ch='y'
def add():
   x=int(input("Enter First Number"))
   y=int(input("Enter Second Number"))
   c=x+y
   print("Sum= ",c)
def Sub():
   x=int(input("Enter First Number"))
   y=int(input("Enter Second Number"))
   c=x-y
   print("Subtraction= ",c)
def Multi():
   x=int(input("Enter First Number"))
   y=int(input("Enter Second Number"))
   c=x*y
   print("Multiply= ",c)
def div():
   x=int(input("Enter First Number"))
   y=int(input("Enter Second Number"))
   c=x//y
   print("Division = ",c)

while (ch=='Y' or ch=='y'):
   print("1-ADD")
   print("2-ADD")
   print("3-ADD")
   print("4-ADD")
   print("5-Exit")
   print("Enter Your Choice")
   x=int(input())
   if(x==1):
      add()
   elif (x==2):
      Sub()
   elif (x==3):
      Multi()
   elif (x==4):
      Div()
   elif(x==5):
      break
```

```
else:
   print("Your Choice is Wrong try Again")
print("Do You Want to another Calculation(Y/N):  ")
ch=input()
```

1. Calculate area of circle using return statement.

2. calculate area of circle using lambda function.

3. calculate perimeter of a circle using lambda function.

4. Create a lambda Function to calculate remainder.

5. Find the Square of a number by using lambda function.

1-Program to accessing class variable.

```
class num:
    a=0
    b=0
    def display():
        print("A= ",a)
        print("B= ",b)
n=num()
n.display()
```

Output:

n.display()
TypeError: display() takes 0 positional arguments but 1 was given

2- Program to accessing class variable. With self-Keyword.

```
class num:
    a=0
    b=0
    def display(self):
        print("A= ",self.a)
        print("B= ",self.b)
n=num()
n.display()
```

Output:

A=0
B=0

3- Access Class variable without object

```
class num:
    a=10
    b=20
print("A=",a)
print("B=",b)
```

Output:

print("A=",a)
NameError: name 'a' is not defined

4- **Access Class variable with object**

```
class num:
    a=10
    b=20

a1=num() #creating Object
print("A=",a1.a)
print("B=",a1.b)
```

Output:

A= 10
B= 20

5-Program to add two numbers by using class & object concept.

```
class num:
    a=10
    b=20
    def sum1(self):
        self.c=self.a+self.b
    def display(self):
        print("Sum=",self.c)

a1=num() #creating Object
a1.sum1() # Calling Sum1 function
a1.display()# Calling display Function
```

Output:

Sum=30

6-Mehod2- Program to add two numbers by using class & object concept.

```
class num:
    a=int(input("Enter First Number"))
    b=int(input("Enter Second Number"))
    def sum1(self):
        self.c=self.a+self.b
    def display(self):
        print("Sum=",self.c)

a1=num() #creating Object
a1.sum1() # Calling Sum1 function
a1.display()# Calling display Function
```

7- **Mehod3- Program to add two numbers by using class & object concept.**

```
class num:
    def input(self):
```

8- **Mehod3- Program to add two numbers by using class & object concept.**

```
        self.a=int(input("Enter First Number"))
        self.b=int(input("Enter Second Number"))
    def sum1(self):
        self.c=self.a+self.b
    def display(self):
        print("Sum=",self.c)

a1=num() #creating Object
a1.input()
a1.sum1() # Calling Sum1 function
a1.display()# Calling display Function
```

Output:
Enter First Number12
Enter Second Number12
Sum= 24

```
class num:
    def input(self,x,y):
        self.a=x
        self.b=y

    def sum1(self):
        self.c=self.a+self.b
    def display(self):
        print("Sum=",self.c)


m=int(input("Enter First Number"))
n=int(input("Enter Second Number"))
a1=num() #creating Object
a1.input(m,n)
a1.sum1() # Calling Sum1 function
a1.display()# Calling display Function
```

output:
Enter First Number12
Enter Second Number12
Sum= 24

9-Program to calculate simple Interest using oops Concept.

```
class num:
    def input(self,x,y,z):
        self.a=x
        self.b=y
        self.c=z

    def sum1(self):
        self.s=(self.a*self.b*self.c)/100
    def display(self):
        print("Simple Interest=",self.s)
p=int(input("Enter Amount"))
r=int(input("Enter Rate"))
t=int(input("Enter Time"))

a1=num() #creating Object
a1.input(p,r,t)
a1.sum1() # Calling Sum1 function
a1.display()# Calling display Function
```

10-Program to calculate Area of Circle using oops Concept.

```
class num:
    def input(self,radius):
        self.r=radius
    def sum1(self):
        self.area=(3.14*self.r*self.r)
    def display(self):
        print("Area of Circle=",self.area)

t=eval(input("Enter Radius of Circle= "))

a1=num() #creating Object
a1.input(t)
a1.sum1() # Calling Sum1 function
a1.display()# Calling display Function
```

Output:
Enter Radius of Circle= 2
Area of Circle= 12.56

Note: We can also use our self-referential structure with any meaning full name at the replacement of self.

Example:

```
class name1:
    def input1(s,x,y):
        s.x=x
        s.y=y
    def display(s):
        print("The Value of X=",s.x)
        print("The Value of X=",s.y)
n=name1()
n.input1(10,20)
n.display()
```

5-

```
class MyClass:
  x = 5
  age=20

p1 = MyClass()

print(hasattr(p1, 'salary'))          # Returns true if 'age' attribute exists

print(getattr(p1 , 'age'))            # Returns value of 'age' attribute

print(setattr(p1, 'age', 8))          # Set attribute 'age' at 8
```

```python
print("Set New value of Age Attributes")

print(getattr(p1 , 'age'))            #check the value is set or not

print(delattr(p1, 'age'))              # Delete attribute 'age'

print("Value after deletion Display the old value")

print(getattr(p1 , 'age'))
```

Constructor/Destructor

<table>
<tr>
<td>
1-Constructor-Example1

class abc:
   def __init__(self):
      a=0
      b=0
      print("A=",a)
      print("B=",b)

a=abc()

output:
A=0
B=0
</td>
<td>
2-Parametrized Constructor
class abc:
   def __init__(self,a,b):
      self.a=a
      self.b=b
      print("A=",a)
      print("B=",b)
   def process1(self):
      c=self.a+self.b
      print(c)
a=abc(0,0)
a.process1()
a1=abc(10,20)
a1.process1()
Output:
A= 0
B= 0
0
A= 10
B= 20
30
</td>
</tr>
<tr>
<td>
3-Calculate are of circle using constructor method.
class abc:
   def __init__(self,radius):
      self.radius=radius
      print("Radius=",self.radius)
   def process1(self):
      self.area=self.radius*self.radius*3.14
   def display(self):
      print("Area of Circle=",self.area)

a=abc(5)
a.process1()
a.display()
a=abc(10)
a.process1()
a.display()

Output:
Radius= 5
Area of Circle= 78.5
Radius= 10
Area of Circle= 314.0
</td>
<td>
4-Area of Rectangle by using Parametrized Constructor.
class abc:
   def __init__(self,a,b):
      self.x=a
      self.y=b
      print("Length=",self.x)
      print("Width=",self.y)
   def process1(self):
      self.area=self.x*self.y
   def permi(self):
      self.p=2*(self.x+self.y)
   def display(self):
      print("Area of Circle=",self.area)
      print("Permiter of Circle=",self.p)

a=abc(5,5)
a.process1()
a.permi()
a.display()
a=abc(10,10)
a.process1()
a.permi()
a.display()
</td>
</tr>
<tr>
<td>
5-Parameterized constructor (input example)
class abc:
   def __init__(self,a,b):
      self.x=a
      self.y=b
      print("Length=",self.x)
      print("Width=",self.y)
   def process1(self):
      self.area=self.x*self.y
   def permi(self):
      self.p=2*(self.x+self.y)
</td>
<td>
Output:
Length= 5
Width= 5
Area of Circle= 25
Permiter of Circle= 20
Length= 10
Width= 10
Area of Circle= 100
Permiter of Circle= 40
</td>
</tr>
</table>

```
    def display(self):
        print("Area of Rectangle=",self.area)
        print("Permiter of Rectangle=",self.p)

m=int(input("Enter The Length of Rectangle"))
n=int(input("Enter The Width of Rectangle"))
a=abc(m,n)
a.process1()
a.permi()
a.display()
m=int(input("Enter The Length of Rectangle"))
n=int(input("Enter The Width of Rectangle"))
a=abc(m,n)
a.process1()
a.permi()
a.display()
```

Output:

```
Enter The Length of Rectangle12
Enter The Width of Rectangle12
Length= 12
Width= 12
Area of Rectangle= 144
Permiter of Rectangle= 48
Enter The Length of Rectangle25
Enter The Width of Rectangle25
Length= 25
Width= 25
Area of Rectangle= 625
Permiter of Rectangle= 100
```

6-Destructor is Program

```
class student:

    # Initializing
    def __init__(self):
        print('Student Data created created.')

    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Student Data
deleted.')

obj = student()
del obj
```

Output:

Student Data created created.
Destructor called, Student Data deleted.

Example7-Destroy object

```
class abc:
    def __init__(self,a,b):
        self.x=a
        self.y=b
        print("Length=",self.x)
        print("Width=",self.y)
    def process1(self):
        self.area=self.x*self.y
    def permi(self):
        self.p=2*(self.x+self.y)
    def display(self):
        print("Area of Rectangle=",self.area)
        print("Permiter of Rectangle=",self.p)
    def __del__(self):
        print("Destructor is Destroyed")

m=int(input("Enter The Length of Rectangle"))
n=int(input("Enter The Width of Rectangle"))
a=abc(m,n)
a.process1()
a.permi()
a.display()
del a
print(type(a))
```

Example8-Destroy object

```
class abc:
    def __init__(self,a,b):
        self.x=a
        self.y=b
        print("Length=",self.x)
        print("Width=",self.y)
    def process1(self):
        self.area=self.x*self.y
    def permi(self):
        self.p=2*(self.x+self.y)
    def display(self):
        print("Area of Rectangle=",self.area)
        print("Permiter of Rectangle=",self.p)

m=int(input("Enter The Length of Rectangle"))
n=int(input("Enter The Width of Rectangle"))
a=abc(m,n)
a.process1()
a.permi()
a.display()
del a
print(type(a))
```

Details About class:

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print("Name : ", self.name,  ", Salary: ", self.salary)

print("Employee.__doc__:", Employee.__doc__)
print("Employee.__name__:", Employee.__name__)
print("Employee.__module__:", Employee.__module__)
print("Employee.__bases__:", Employee.__bases__)
print("Employee.__dict__:", Employee.__dict__)
```

Inheritance Example:

1-# Single inheritance

```
class Add:
   def input(self, c, d):
      self.a = c
      self.b = d
   def process1(self):
      self.sum = self.a + self.b
      print(self.sum)
class Sub(Add):
   def process2(self):
      self.sub = self.a - self.b
      print(self.sub)

p1 = Sub()
p1.input(10,20)
p1.process1()
p1.process2()
```

2-Example:

```
class A:
   def input(self,a,b):
      self.a=a
      self.b=b
class C (A):
   def process(self):
      c=self.a+self.b
      print(c)
a1=C()
x=int(input("Enter First Number"))
y=int(input("Enter Second Number"))
a1.input(x,y)
a1.process()
```

3-Example

 # Another Example of Inheritance

```
class circle :

   def input(self,a) :
      self.r = a
   def process_area(self) :
      self.area =  3.14*self.r *self.r

   def show(self) :
      print("Ärea is",self.area)

class peri(circle):
   def process_peri(self):
      self.p = 2*3.14 * self.r
   def showp(self) :
      print("Perimeter is ",self.p)

aa = circle()
aa.input(5)
aa.process_area()
aa.show()
p = peri()
p.input(10)
p.process_peri()
p.showp()
```

4-Example

 # Another Example of Inheritance

```
class circle :

   def input(self,a) :
      self.r = a
   def process_area(self) :
      self.area =  3.14*self.r *self.r

   def show(self) :
      print("Ärea is",self.area)

class peri(circle):
   def process_peri(self):
      self.p = 2*3.14 * self.r
   def showp(self) :
```

5-Accessing Function of base class with object of derived class

class circle :

```
   def input(self,a) :
      self.r = a
   def process_area(self) :
      self.area =  3.14*self.r *self.r

   def show(self) :
      print("Ärea is",self.area)

class peri(circle):
   def process_peri(self):
      self.p = 2*3.14 * self.r
   def showp(self) :
      print("Perimeter is ",self.p)
```

```
        print("Perimeter is ",self.p)

aa = circle()
r1=float(input("Enter Radius"))
aa.input(5)
aa.process_area()
aa.show()
p = peri()
r2=float(input("Enter Radius"))
p.input(10)
p.process_peri()
p.showp()
```

```
p = peri()
r2=float(input("Enter Radius"))
p.input(r2)
p.process_peri()
p.showp()
p.process_area()
p.show()
```

6-EXAMPLE

```
class Rect:
    def input(self,x,y):
        self.a=x
        self.b=y
    def ReactArea(self):
        self.z=self.a*self.b
        print("Area of Reactangle=",self.z)
class Permi(Rect):
    def per(self):
        self.p=2*(self.a+self.b)
        print("Permeter of  Reactangle=",self.p)
p1=Permi()
x1=int(input("Enter Length of Reactangle"))
x2=int(input("Enter Width of Reactangle"))
p1.input(x1,x2)
p1.ReactArea()
p1.per()
```

7-Multilevel Inheritance

```
class Add:
    def input(self,a,b):
        self.x=a
        self.y=b
    def add(self):
        self.z=self.x+self.y
        print("Addition=",self.z)
class Sub(Add):
    def sub(self):
        self.z=self.x-self.y
        print("Subtraction=",self.z)
class Multi(Sub):
    def multi(self):
        self.z=self.x*self.y
        print("Multiply=",self.z)
m=Multi()
m.input(10,20)
m.add()
m.sub()
m.multi()
```

8-Multilevel Inheritance

```
class student:
    def shows(self):
        print("This is student class")
class persion(student):
    def showp(self):
        print("This is persion class")
class group1(persion):
    def showg(self):
        print("This is group class")
g=group1()
g.shows()
g.showp()
g.showg()
```

Multiple Inheritances:

<table>
<tr>
<td>

1-Example1-

```
class class1:
    def showmain(self):
        print("This is main class")
class class2:
    def showchild(self):
        print("This is Child Class")
class c(class1,class2):
    def showmultiple(self):
        print("This is Inherited Class")
c1=c()
c1.showmain()
c1.showchild()
c1.showmultiple()
```

</td>
<td>

Example2-Constructor of inherited Class

```
class mainclass:
    def __init__(self):
        print("This is main class Constructor")
    def showmain(self):
        print("This is main class")
class subclass:
    def __init__(self):
        print("This is sub class Constructor")
    def showsub(self):
        print("This is Sub Class")
class inherit1(mainclass,subclass):
    def __init__(self):
        print("Constructor of Inherited class")
    def showinheri(self):
        print("this is inherited class")
inh1=inherit1()
inh1.showmain()
inh1.showsub()
inh1.showinheri()
```

Output:

Constructor of Inherited class
This is main class
This is Sub Class
this is inherited class

</td>
</tr>
<tr>
<td>

Resolving Constructor Problem with multiple inherite-Example1

```
class mainclass:
    def __init__(self):
        super().__init__()
        print("This is main class Constructor")
    def showmain(self):
        print("This is main class")
class subclass:
    def __init__(self):
        super().__init__()
        print("This is sub class Constructor")
    def showsub(self):
        print("This is Sub Class")
class inherit1(mainclass,subclass):
    def __init__(self):
        super().__init__()
        print("Constructor of Inherited class")
    def showinheri(self):
        print("this is inherited class")
inh1=inherit1()
inh1.showmain()
```

</td>
<td>

Resolving Constructor Problem with multiple inherite-Example2

```
class mainclass:
    def __init__(self):
        super().__init__()
        print("This is main class Constructor")
    def showmain(self):
        print("This is main class")
class subclass:
    def __init__(self):
        super().__init__()
        print("This is sub class Constructor")
    def showsub(self):
        print("This is Sub Class")
class inherit1(subclass,mainclass):
    def __init__(self):
        super().__init__()
        print("Constructor of Inherited class")
    def showinheri(self):
        print("this is inherited class")
```

</td>
</tr>
</table>

inh1.showsub() inh1.showinheri() **Output:** **This is sub class Constructor** **This is main class Constructor** **Constructor of Inherited class** **This is main class** **This is Sub Class** **this is inherited class**	inh1=inherit1() inh1.showmain() inh1.showsub() inh1.showinheri() **Output:** **This is main class Constructor** **This is sub class Constructor** **Constructor of Inherited class** **This is main class** **This is Sub Class** **this is inherited class**

class A:
  def __init__(self):
    print('I am in A Class')
 	
# B class inheriting A 	
class B(A):
  def __init__(self):
    print('I am in B class')
    super().__init__()
b1=B()	

Overriding

1-Example1

```
class Add:
    def result(self,a,b):
        print("Sum=",a+b)
class multiply(Add):
    def result(self,a,b):
        print("Multiply Result= ",a*b)
m=multiply()
m.result(10,10)
```

Output:

Multiply Result= 100

Example2:

```
class Add:
    def result(self,a,b):
        print("Sum=",a+b)
class multiply(Add):
    pass
m=multiply()
m.result(10,10)
```

Output:

Sum= 20

Example3: Another Way to Accessing parent class method.

```
class Add:
    def result(self,a,b):
        print("Sum=",a+b)
class multiply(Add):
    def result(self,a,b):
        print("Multiply=",a*b)

m=multiply()
m.result(10,10)
a=Add()
a.result(10,10)
```

Output:

Multiply= 100

Sum= 20

Note: But This is not a good method Because here we create both class object

Example3: Another Way to Accessing parent class method. By using super()

```
class Add:
    def result(self,a,b):
        print("Sum=",a+b)
class multiply(Add):
    def result(self,a,b):
        super().result(a,b)
        print("Multiply=",a*b)

m=multiply()
m.result(10,10)
```

Output:

Sum= 20

Multiply= 100

Overloading:

1- Example2: Not Access Generate Error

```
class hello:
    def show(self,a):
        print("The value of A=",a)
    def show(self):
        print("Method Overloading")
h=hello()
h.show()
h.show(10)
```

Output:

Method Overloading

h.show(10)

TypeError: show() takes 1 positional argument but 2 were given

Example2:

```
class Human:
    def sayHello(self, name=None):
      if name is not None:
        print('Hello',name)
      else:
        print('Hello ')

obj = Human()
print(obj.sayHello())
print(obj.sayHello('Rambo'))
```

Output

Hello

None

Hello Rambo

None

Example3-
class numeric:
    def sum(self,a=None,b=None,c=None):
        if a!=None and b!=None and c!=None:
            result=a+b+c
            return result
        elif a!=None and b!=None:
            result=a*b
            return result
        else:
            print("Enter At Least Two Arguments")
 	
n=numeric()
print(n.sum(10,20,20))
print(n.sum(10,20))
n.sum(10)
 	
Output:
50
200
Enter At Least Two Arguments	

```
class Sub:

    def result(self,a,b):

        print("Sub=",a-b)



class Add(Sub):

    def result(self,a,b):

        super().result(a,b)

        print("Sum=",a+b)

class multiply(Add):

    def result(self,a,b):

        super().result(a,b)

        print("Multiply=",a*b)



m=multiply()

m.result(10,10)
```

#Demo without exception handling
 	
a =12
b = 0
c = a/b
print(c)
print("Hello")
d =a+b
print(d)
	#Demo with exception handling
	
	try :
	   a =12
	   b = 0
	   c = a/b
	   print(c)
	except:
	   print("Exception found ")
	print("Hello")
	d =a+b
	print(d)

#Multiple Except
 	
try:
   a=5
   b=0   # if b ="das" than raise Type Error
   print (a/b)
except TypeError:
   print('Unsupported operation')
except ZeroDivisionError:
   print ('Division by zero not allowed')
 	
print ('Out of try except blocks')
	#Demo finally block
	
	try :
	   a =12
	   b = 0
	   c = a/b
	   print(c)
	except:
	   print("Exception found ")
	
	finally :
	   print("This is finally block")
	
	print("Hello")
	d =a+b
	print(d)

#Uses of Else
try:
   a=5
   b=0   # if b ="das" than raise Type Error
   print (a/b)
except TypeError:
   print('Unsupported operation')
except ZeroDivisionError:
   print ('Division by zero not allowed')
else:
   print("No exceptio found ")
print ('Out of try except blocks')
	#Use of no exception
	
	try :
	   a =12
	   b = 0
	   c = a/b
	   print(c)
	except:
	   print("Exception found ")
	else :
	   print("No exception found")

#Demo try, Except , Else finally
 	
try :
   a =12
   b = 0
   c = a/b
   print(c)
except:
   print("Exception found ")
	#Demo try, Except , Else finally
	
	try :
	   a =12
	   b = 0
	   c = a/b
	   print(c)

```
else :
   print("No exception found")
finally :
   print("This is finally block")

print("Hello")
d =a+b
print(d)
```

```
except TypeError:
   print('Unsupported operation')
except ZeroDivisionError:
   print ('Division by zero not allowed')

else :
   print("No exception found")

finally :
   print("This is finally block")

print("Hello")
d =a+b
print(d)
```

#Raise an exception

```
try:
    age = int(input("Enter the age?"))
    if age<18:
        raise ValueError;
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid",ValueError)
```

#Raise an exception

```
try:
    x=int(input('Enter age (0-50): '))
    if x > 50:
        raise ValueError(x)
except ValueError:
    print(x, "is out of allowed range")
else:
    print(x, "is within the allowed
range",ValueError(x))
```

#Raise an exception

```
try:
    a = int(input("Enter a?"))
    b = int(input("Enter b?"))
    if b is 0:
        raise ArithmeticError;
    else:
        print("a/b = ",a/b)
except ArithmeticError:
    print("The value of b can't be 0")
```

#Demo-Custom Exception

```
class UnderAge(Exception):
  pass

def verify_age(age):

  if int(age) < 18:
     raise UnderAge
  else:
     print('Age: '+str(age))

# main program

verify_age(23)  # won't raise exception
verify_age(17)  # will raise exception
```

#Demo-Custom Exception Handling

```
class UnderAge(Exception):
  pass

def verify_age(age):
   try :
     if int(age) < 18:
        raise UnderAge
     else:
        print('Age: '+str(age))
   except :
      print("Exception Under age found ")

# main program
verify_age(23)  # won't raise exception
verify_age(17)  # will raise exception

print("Hello")
```

Example2-

```
class arith(Exception):
    pass
def verify_number():
    a = int(input("Enter a?"))
    b = int(input("Enter b?"))
    if b is 0:
        raise arith;
    else:
        print("a/b = ",a/b)

verify_number()
verify_number()
```

Example3:

```
class arith(Exception):
    pass
def verify_number():
    a = int(input("Enter a?"))
    b = int(input("Enter b?"))
    try:
        if b is 0:
            raise arith;
        else:
            print("a/b = ",a/b)
    except:
        print("Exception Found Value of B is 0")

verify_number()
verify_number()
```

Check all built in exceptions

```
"""locals()['__builtins__'] will
return a module of built-in
exceptions, functions, and
attributes.
dir allows us to list these
attributes as strings."""

print(dir(locals()['__builtins__']))
```