PYTHON PROGRAMMING
DEPARTMENT OF COMPUTER SCIENCE

AND

APPLICATIONS

Islamiah College Vaniyambadi (Autonomous)

By

P. Magizhan

UNIT – I

INTRODUCTION DATA, EXPRESSIONS, STATEMENTS

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

Introduction to Python and installation:

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

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

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

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

Beginning with Python programming:
1) Finding an Interpreter:
Before we start Python programming, we need to have an interpreter to interpret and run our programs. There are certain online interpreters like **https://ide.geeksforgeeks.org/**, http://ideone.com/ or http://codepad.org/ that can be used to start Python without installing an interpreter.
*Windows***:** There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from **http://python.org/downloads/**

2) Writing first program:

```
# Script Begins

    Statement1
```

```
    Statement2

    Statement3

# Script Ends
```

Differences between scripting language and programming language:

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

Why to use Python:

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

1. **Python is object-oriented**
   Structure supports such concepts as polymorphism, operation overloading and multiple inheritance.
2. **Indentation**
   Indentation is one of the greatest feature in python

3. **It's free (open source)**
Downloading python and installing python is free and easy
4. **It's Powerful**
    * Dynamic typing
    * Built-in types and tools
    * Library utilities
    * Third party utilities (e.g. Numeric, NumPy, sciPy)
    * Automatic memory management
5. **It's Portable**
    * Python runs virtually every major platform used today
    * As long as you have a compaitable python interpreter installed, python programs will run in exactly the same manner, irrespective of platform.
6. **It's easy to use and learn**
    * No intermediate compile
    * Python Programs are compiled automatically to an intermediate form called byte code, which the interpreter then reads.
    * This gives python the development speed of an interpreter without the performance loss inherent in purely interpreted languages.
    * Structure and syntax are pretty intuitive and easy to grasp.
7. **Interpreted Language**
Python is processed at runtime by python Interpreter
8. **Interactive Programming Language**
Users can interact with the python interpreter directly for writing the programs
9. **Straight forward syntax**
The formation of python syntax is simple and straight forward which also makes it popular.

Installation:

There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from http://python.org/downloads/
Steps to be followed and remembered:
Step 1: Select Version of Python to Install.
Step 2: Download Python Executable Installer.
Step 3: Run Executable Installer.
Step 4: Verify Python Was Installed On Windows.
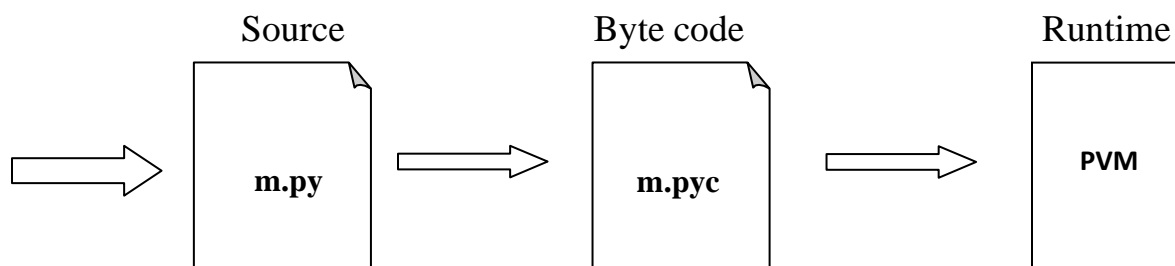
Step 5: Verify Pip Was Installed.

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

Working with Python

Python Code Execution:

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

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

There are two modes for using the Python interpreter:

- Interactive Mode
- Script Mode

Running Python in interactive mode:

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

>>> print("hello world")

hello world

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

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

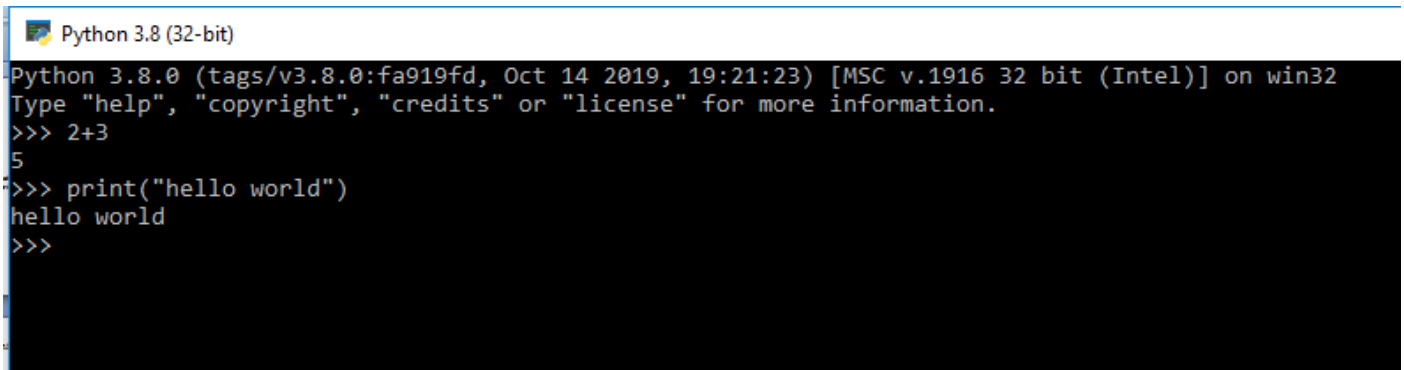# Quantities stored in memory are not displayed by default.

>>> x

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

[0, 1, 2]

>>> 2+3

5

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

Running Python in script mode:

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

python MyFile.py

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

Example:

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

Data types:

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

Int:

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

>>> print(24656354687654+2)

24656354687656

>>> print(20)

20

>>> print(0b10)

2

```
>>> print(0B10)
2
>>> print(0X20)
32
>>> 20
20
>>> 0b10
2
>>> a=10
>>> print(a)
10
```

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

```
>>> type(10)
<class 'int'>
>>> a=11
>>> print(type(a))
<class 'int'>
```

<u>Float:</u>

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

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

```
>>> y=2.8
>>> y
2.8
>>> y=2.8
>>> print(type(y))
<class 'float'>
>>> type(.4)
<class 'float'>
>>> 2.
```

2.0

Example:

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

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

Output:

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

Boolean:

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

>>> type(True)

<class 'bool'>

>>> type(False)

<class 'bool'>

String:

1. Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.

- 'hello' is the same as "hello".

- Strings can be output to screen using the print function. **For example: print("hello").**

>>> print(" college")

college

>>> type("college")

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

college

```
>>> " "
```

' '

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

```
>>> print("Islamiah is an autonomous (')
```

college") Islamiah is an autonomous (') college

```
>>> print('Islamiah is an autonomous (")
```

college')Islamiah is an autonomous (") college

Suppressing Special Character:

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

```
>>> print("Islamiah is an autonomous (\')
```

college")Islamiah is an autonomous (') college

```
>>> print('Islamiah is an autonomous (\")
```

college')Islamiah is an autonomous (") college

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

```
>>> print('a\
....b')
a. ..b
>>> print('a\
b\
c')
```

abc

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

a

 b

>>> print("Islamiah \n

college")Islamiah

 college

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

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

>>> print("a\tb")

a   b

List:

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

Ex:

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

>>> print(list1)

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

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

>>> x

[]

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

>>> x=list(tuple1)

>>> x

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

Variables:

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

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

Rules for Python variables:

- A variable name must start with a letter or the underscore character

- A variable name cannot start with a number

- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)

- Variable names are case-sensitive (age, Age and AGE are three different variables)

Assigning Values to Variables:

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

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

For example −

a= 100   # An integer assignment

b = 1000.0   # A floating point

c = "John"   # A string

print (a)

print (b)

print (c)

This produces the following result −

100

1000.0

John

Multiple Assignment:

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

For example :

a = b = c = 1

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

For example −

a,b,c = 1,2,"Islamiah"

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

Output Variables:

The Python print statement is often used to output variables.

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

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

Output: Islamiah

To combine both text and a variable, Python uses the **"+"** character:

Example

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

Output

Python is awesome

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

Example

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

Output:

Python is awesome

Expressions:

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

Examples: Y=x + 17

>>> x=10

>>> z=x+20

>>> z

30

>>> x=10

>>> y=20

>>> c=x+y

>>> c

30

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

>>> y=20

>>> y

20

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

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

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

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

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

Some of the python expressions are:

Generator expression:

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

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

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

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

Conditional expression:

Syntax: true_value if Condition else false_value

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

>>> x

'1'

Statements:

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

Examples:

An assignment statement creates new variables and gives them values:

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

>>> college="Islamiah"

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

>>> print("Islamiah

colege")Islamiah college

Precedence of Operators:

Operator precedence affects how an expression is evaluated.

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

Example 1:

>>> 3+4*2

11

Multiplication gets evaluated before the addition operation

>>> (10+10)*2

40

Parentheses () overriding the precedence of the arithmetic operators

Example 2:

```
a = 20
b = 10
c = 15
d = 5
e = 0

e = (a + b) * c / d       #( 30 * 15 ) / 5
print("Value of (a + b) * c / d is ",  e)

e = ((a + b) * c) / d     # (30 * 15 ) / 5
print("Value of ((a + b) * c) / d is ",  e)

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

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

e = a + (b * c) / d;      # 20 + (150/5)
print("Value of a + (b * c) / d is ",  e)
```
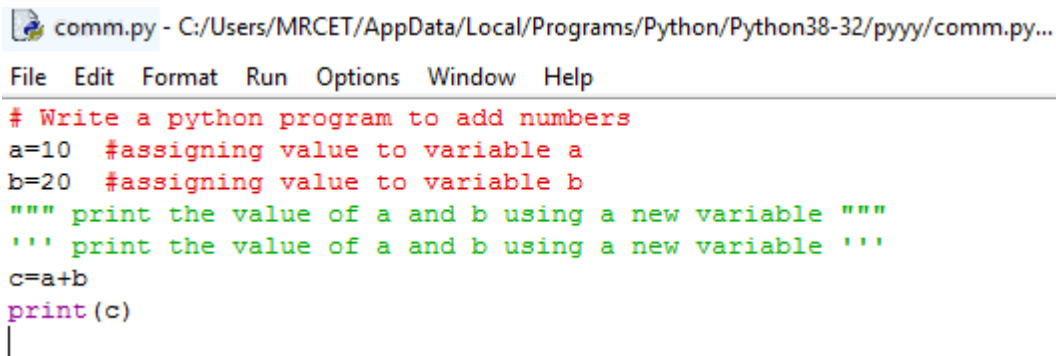
Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/opprec.py
Value of (a + b) * c / d is 90.0
Value of ((a + b) * c) / d is 90.0
Value of (a + b) * (c / d) is 90.0
Value of a + (b * c) / d is 50.0

Comments:

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

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

Example:

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

File  Edit  Format  Run  Options  Window  Help
# Write a python program to add numbers
a=10  #assigning value to variable a
b=20  #assigning value to variable b
""" print the value of a and b using a new variable """
''' print the value of a and b using a new variable '''
c=a+b
print(c)
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/comm.py 30

Modules:

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

Syntax:

import <module-name>

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

Note: In python we have help ()

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

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

```
>>> import sys
>>> print(sys.version)
3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)]
>>> print(sys.version_info)
sys.version_info(major=3, minor=8, micro=0, releaselevel='final', serial=0)
>>> print(calendar.month(2021,5))
      May 2021
Mo Tu We Th Fr Sa Su
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
>>> print(calendar.isleap(2020))
True
>>> print(calendar.isleap(2017))
False
```

Functions:

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

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

1. **<u>Built-in functions</u> -** Functions that are built into Python.

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

   integer = -20

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

   Output:

   Absolute value of -20 is: 20

2. **<u>User-defined functions</u> -** Functions defined by the users themselves.

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

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

   Output:

   The sum is 25

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

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

Example:

#example for flow of execution

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

Output:

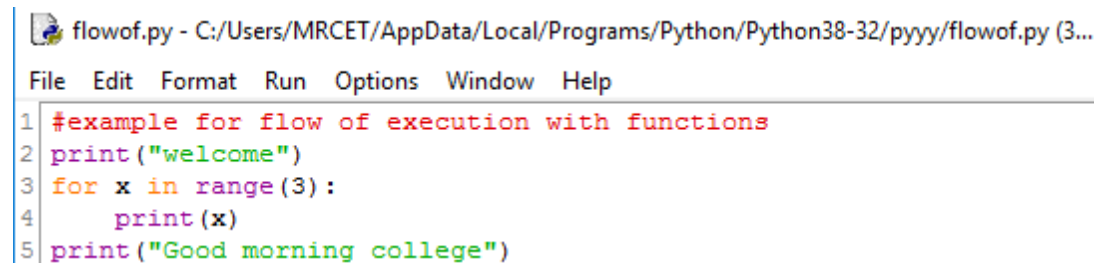C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py

welcome

0

1

2
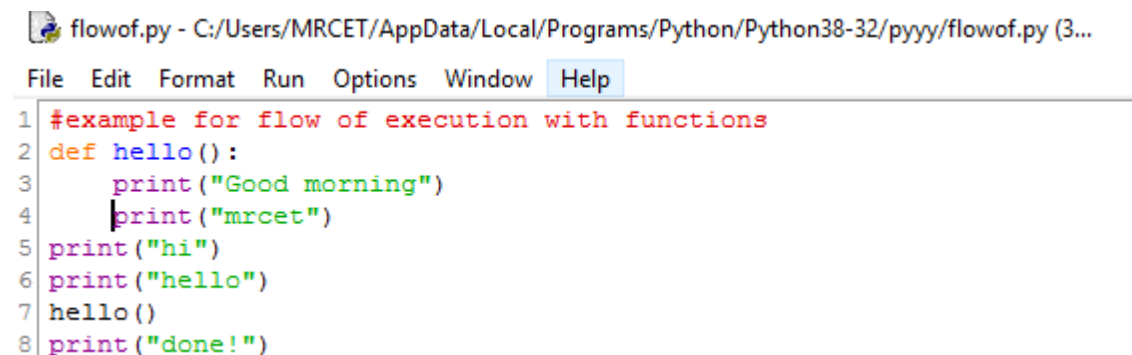
Good morning college

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

```
flowof.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py (3...
File  Edit  Format  Run  Options  Window  Help
1 #example for flow of execution with functions
2 print("welcome")
3 for x in range(3):
4     print(x)
5 print("Good morning college")
```

- -

```
flowof.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py (3...
File  Edit  Format  Run  Options  Window  Help
1 #example for flow of execution with functions
2 def hello():
3     print("Good morning")
4     print("mrcet")
5 print("hi")
6 print("hello")
7 hello()
8 print("done!")
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/flowof.py hi

hello

Good morning

Islamiah

done!

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

Parameters and arguments:

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

Example:
#here a and b are parameters

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

#12 and 13 are arguments
#function call
result=add(12,13)
print(result)

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/paraarg.py 25

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

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

Syntax:

def functionname():

statements

.

.

.

functionname()

Function definition consists of following components:

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

Example:

def hf():

   hello world

hf()

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

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

#calling function in python:

def hf():

   print("hello world")

hf()

Output:

hello world

- - - - - - - - - - - - - - - - - -

```python
def hf():

    print("hw")

    print("gh kfjg 66666")

hf()

hf()

hf()
```

Output:

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

- -

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

    c=x+y

    print(c)

add(5,4)
```

Output:

9

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

    c=x+y

    return c

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

Output:

9

- -

```
def add_sub(x,y):

    c=x+y

    d=x-y

    return c,d

print(add_sub(10,5))
```

Output:

 (15, 5)

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

```
def hf():

    return "hw"

print(hf())
```

Output:

hw

```
def hf():

    return "hw"

hf()
```

Output:

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

>>>

```
- - - - - - - - - - - - - - - - - - - - - - -
```

def hello_f():

   return "hellocollege"

print(hello_f().upper())

Output:

HELLOCOLLEGE

 # Passing Arguments

def hello(wish):

   return '{}'.format(wish)

print(hello("Islamiah"))

Output:

Islamiah

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

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

def   wish(name,msg):

  """This function greets to

  the person with the provided message"""

    print("Hello",name + ' ' + msg)

wish("ISLAMIAH","Good morning!")

Output:

Hello ISLAMIAH Good morning!

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

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

```
def hello(wish,hello):

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

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

Output:

hiIslamiah,college

#Keyword Arguments

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

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

(Or)

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

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

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

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

Output:

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

Note:

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

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

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

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

For example: if you define the function like below

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

- -

```
def print_name(name1, name2):

  """ This function prints the name """

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

#calling the function

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

Output:

B and A are friends

#Default Arguments

Function arguments can have default values in Python.

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

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

   return '{},{}'.format(wish,name)

print(hello("good morning"))

Output:

good morning,you

- -

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

   return '{},{}'.format(wish,name)   //print(wish + ' ' + name)

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

Output:

good morning,nirosha   //   good morning nirosha

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

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

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

Syntax Error: non-default argument follows default argument

- -

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

""" This function will print sum of two numbers

    if the arguments are not supplied

    it will add the default value """

  print (a+b)

sum(1,2) #calling with arguments

sum( )   #calling without arguments

Output:

3

6

Variable-length arguments

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

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

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

```
def wish(*names):
  """This function greets all
  the person in the names tuple."""

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

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

Output:

Hello

ISLAMIAH

Hello CSE

Hello SIR

Hello MADAM

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

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

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

print(areaOfCircle(r))
```

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/fu1.py Enter radius of circle 3
28.259999999999998

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

```
def calculete(a,b):

    total=a+b

    diff=a-b

    prod=a*b

    div=a/b

    mod=a%b
```

```
   return total,diff,prod,div,mod

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

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

#function call

s,d,p,q,m = calculete(a,b)

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

#print("diff= ",d)

#print("mul= ",p)

#print("div= ",q)

#print("mod= ",m)
```

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

#program to find biggest of two numbers using functions.

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


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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/fu1.py Enter a value 5
Enter b value-2
big number= 5

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

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


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

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/fu1.py Enter a value 5
Enter b value -6
Enter c value 7
big number= 7

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

```
def result(a):
    if a>40:
```

```
return "pass"
```

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

print(result(a))
```

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

CONTROL FLOW, LOOPS

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

Control Flow, Loops:

Boolean Values and Operators:

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

>>> 5 == 5

True

>>> 5 == 6

False

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

>>> type(True)

<class 'bool'>

>>> type(False)

<class 'bool'>

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

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

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

Note:

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

Conditional (if):

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

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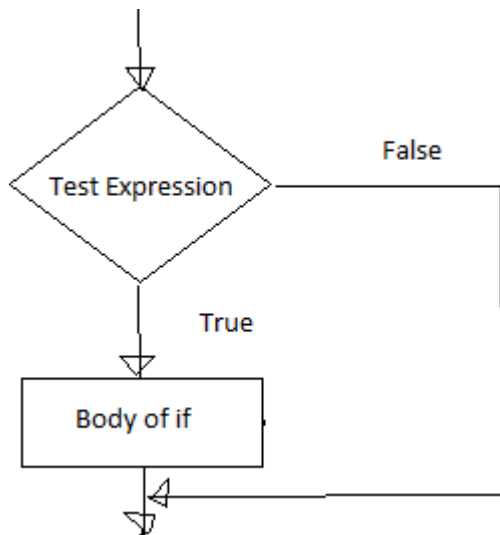
if Statement Flowchart:

Fig: Operation of if statement

Example: Python if Statement

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

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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/if1.py 3 is greater
done
-1 a is smaller
Finish
- - - - - - - - - - - - - - - - - - - -

a=10

if a>9:

   print("A is Greater than 9")

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/if2.py A is Greater than 9

Alternative if (If-Else):

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

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

Syntax of if - else :

     if test expression:

         Body of if stmts

     else:

         Body of else stmts

If - else Flowchart :
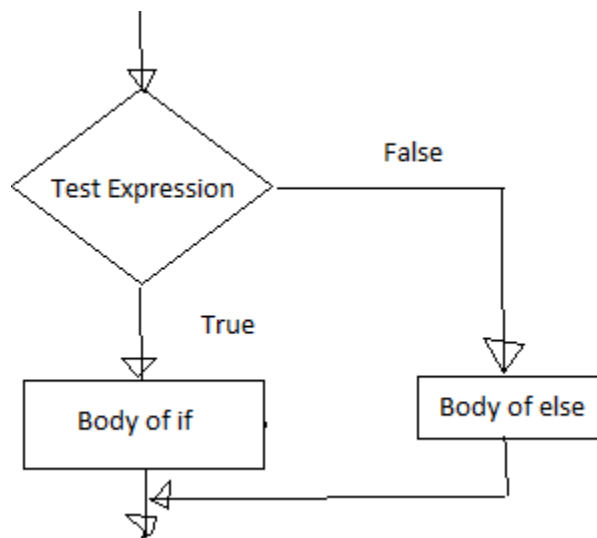
Fig: Operation of if – else statement

Example of if - else:

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

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py
enter the number 2
a is smaller than the input given

```
a=10
b=20
if a>b:
    print("A is Greater than B")
else:
    print("B is Greater than A")
```

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/if2.py B is Greater than A

Chained Conditional: (If-elif-else):

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

Syntax of if – elif - else :

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

Flowchart of if – elif - else:
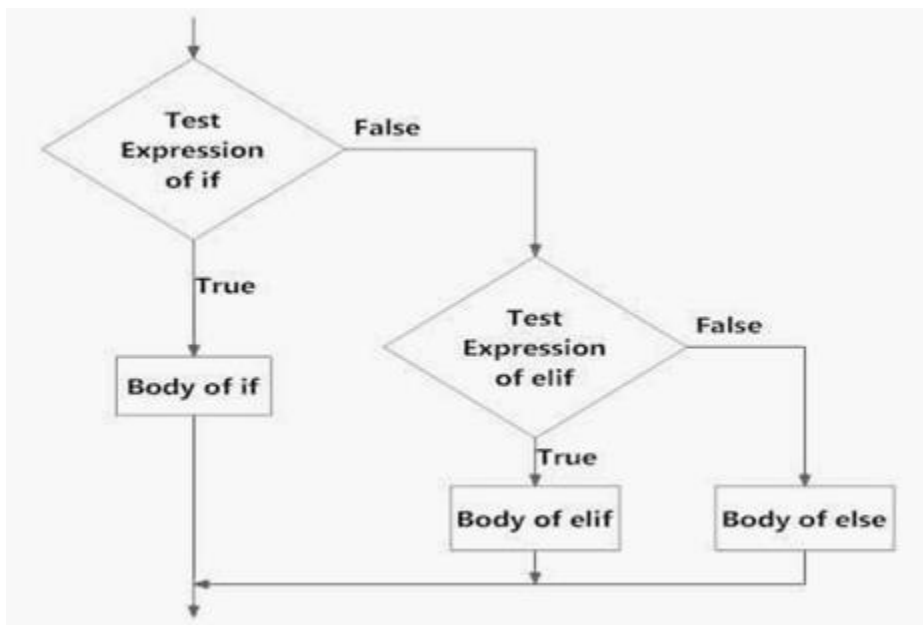

Fig: Operation of if – elif - else statement

Example of if - elif – else:

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

```
if a>b:
```

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

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py
enter the number5
enter the number2
enter the number9
a is greater
>>>
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py
enter the number2
enter the number5
enter the number9
c is greater

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

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/ifelif.py 3 - Got a true expression value

100

Good bye!

Iteration:

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

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

Statements:

In Python Iteration (Loops) statements are of three types:
1. While Loop
2. For Loop
3. Nested For Loops

While loop:

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

   Syntax:

   while(expression):
         Statement(s)

   Flowchart:
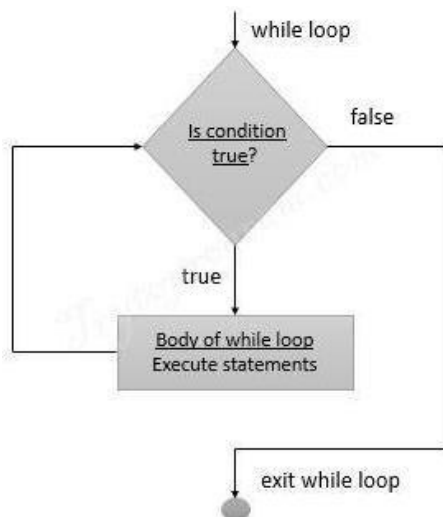
Example Programs:

1._____

```
i=1
while i<=6:
        print("Islamiah
        college")i=i+1
```

output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/wh1.py Islamiah college

Islamiah

college

Islamiah

college

Islamiah

college

Islamiah

college

Islamiah

college

2._____

```
i=1

while i<=3:
   print("ISLAMIAH",end="
   ")j=1
   while j<=1:
     print("CSE DEPT",end="")
     j=j+1
   i=i+1
   print()
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/wh2.py ISLAMIAH CSE DEPT

ISLAMIAH CSE

DEPTISLAMIAH

CSE DEPT

3.

```
i=1
```

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

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

Output:

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

ISLAMIAH CSE
DEPTISLAMIAH
ISLAMIAH

4. _____

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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/wh4.py 1
2
3
4
5
6
7
8
9

2. _____

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

b = b + 1

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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/wh5.py
Iteration 1
Iteration 2
Iteration 3
While loop terminated
- - - - - - - - - - - - - - - - -
```
count = 0
while (count < 9):
    print("The count is:", count)
    count = count + 1
print("Good bye!")
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/wh.py
=The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!

For loop:

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

Syntax: for var in sequence:

        Statement(s)   A sequence of values assigned to var in each iteration

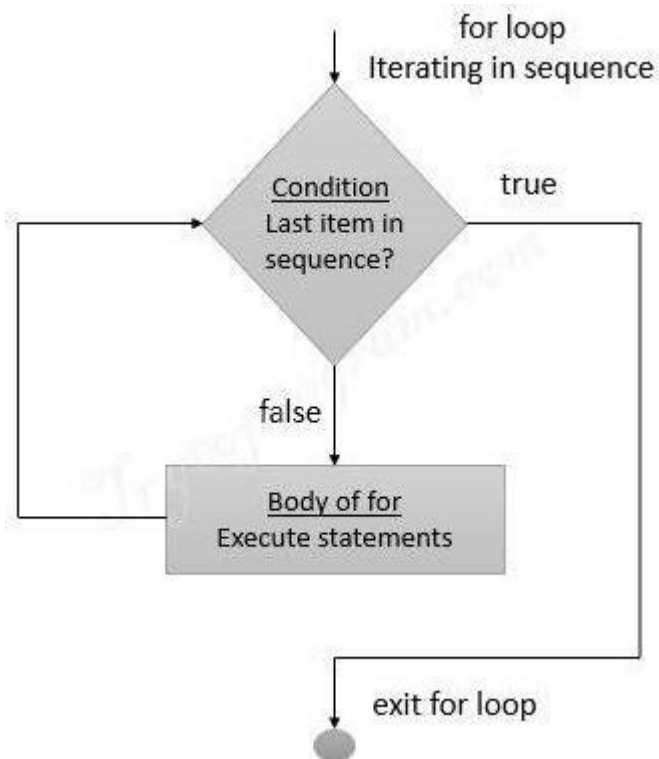Holds the value of item
in sequence in each iteration

Sample Program:

numbers = [1, 2, 4, 6, 11, 20]

seq=0

for val in numbers:

   seq=val*val

   print(seq)

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/fr.py1
4
16
36
121
400

Flowchart:

Iterating over a list:

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

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

Output:

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

Iterating over a Tuple:

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

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

Iterating over a dictionary:

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

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

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

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

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/dic.py Keys are:
ces
it
ece
Values are:
block1
block2
block3

Iterating over a String:

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

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

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/strr.py M
R
C
E
T

Nested For loop:

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

Syntax:

```
for val in sequence:

      for val in sequence:
```

statements

statements

Example 1 of Nested For Loops (Pattern Programs)

for i in range(1,6):

  for j in range(0,i):

    print(i, end=" ")

  print('')

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/nesforr.py 1

2 2

3 3 3

4 4 4 4

5 5 5 5 5

- - - - - - - - - - - - - - - -

Example 2 of Nested For Loops (Pattern Programs)

for i in range(1,6):

  for j in range(5,i-1,-1):

    print(i, end=" ")

  print('')

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

Output:

1 1 1 1 1

2 2 2 2

3 3 3

4 4

Break and continue:

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

Break:

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

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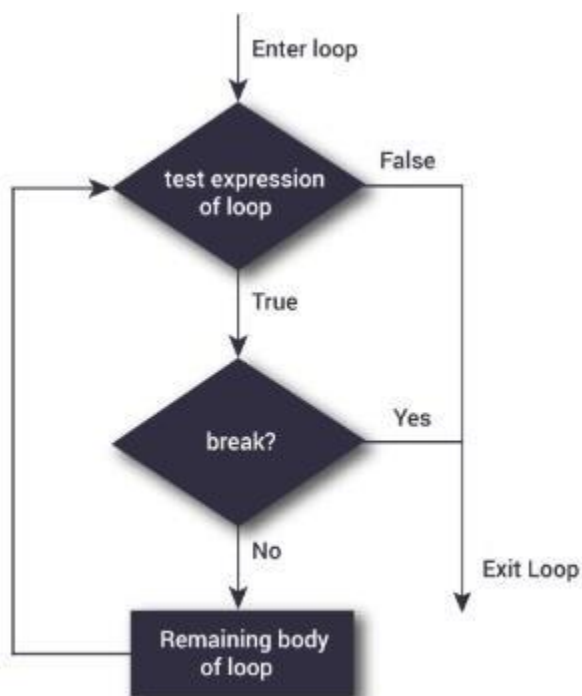
The following shows the working of break statement in for and while loop:

for var in sequence:
     # code inside for loop
     If condition:
          break (if break condition satisfies it jumps to outside loop)
     # code inside for loop
code outside for loop

while test expression

      # code inside while loop
      If condition:
          break (if break condition satisfies it jumps to outside loop)
      # code inside while loop
code outside while loop

Example:

```
for val in "ISLAMIAH
  COLLEGE":if val == " ":
    break
  print(val)

print("The end")
```

Output:

I

S

L

M

I

A

H
The end

 # Program to display all the elements before number 88

```
for num in [11, 9, 88, 10, 90, 3, 19]:
  print(num)
  if(num==88):
        print("The number 88 is found")
        print("Terminating the loop")
        break
```

Output:
11
9
88
The number 88 is found

Terminating the loop

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

Output:

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

Current Letter : P

Current Letter : y

Current Letter : t

Continue:

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

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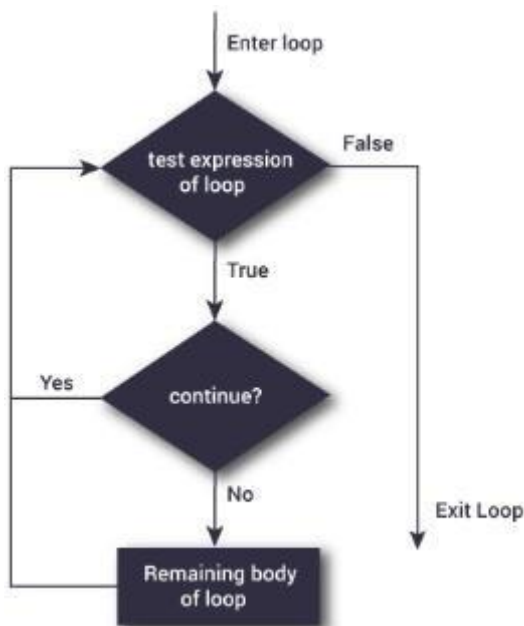
The following shows the working of break statement in for and while loop:

```
for var in sequence:
        # code inside for loop
        If condition:
                continue (if break condition satisfies it jumps to outside loop)
        # code inside for loop
# code outside for loop


while test expression

        # code inside while loop
        If condition:
                continue(if break condition satisfies it jumps to outside loop)
        # code inside while loop
# code outside while loop
```

Example:

```
# Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

Output:

```
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/cont.py s
t
r
n
g
The end
```

```
# program to display only odd numbers

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

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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/cont2.py 11
9
89

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

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/con1.py Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n

Pass:

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored.
pass is just a placeholder for functionality to be added later.

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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/f1.y.py

>>>

Similarily we can also write,

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

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

FUNCTIONS, ARRAYS

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

Functions, Arrays:

Fruitful functions:

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

<div align="center">(or)</div>

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

Return values:

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

 # returns the area of a circle with the given radius:

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

<div align="center">**(or)**</div>

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

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

```
def absolute_value(x):
```

```python
if x < 0:
```

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

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

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

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

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

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

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

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

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

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

Output:
C:\Users\ISLAMIAH\AppData\Local\Programs\Python\Python38-32\pyyy\fu1.py

True
True
None

#‾‾‾‾‾‾‾‾‾‾

def area(radius):

   b = 3.14159 * radius**2

   return b

Parameters:

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

Example:
#here a and b are parameters

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

#12 and 13 are arguments
#function call
result=add(12,13)
print(result)

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/paraarg.py 25

Some examples on functions:

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

#function defination
def display():
   print("vandemataram")
print("i am in main")

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

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

#Type1 : No parameters and no return type

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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/fu1.py function 1

#Type 2: with param with out return type

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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/fu1.py Hello

#Type 3: without param with return type

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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/fu1.py welcome to python

#Type 4: with param with return type

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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/fu1.py python is better then c

Local and Global scope:

Local Scope:

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

Global Scope:

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

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

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

 # create a global variable

```
x = "global"

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

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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/fu1.py x inside : global

x outside: global

create a local variable

```
def f1():

    y = "local"

    print(y)

f1()
```

Output:
local

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

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

f2()
print(y)
```

Then when we try to run it shows an error,

```
Traceback (most recent call last):
  File "C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py",
line6, in <module>
```

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

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

use local and global variables in same code

```
x = "global"

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

f3()
```

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

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

use Global variable and Local variable with same name

```
x = 5

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

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

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/fu1.py local x: 10
global x: 5

Function Composition:

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

FunctionB(FunctionA(x))

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

Example 1:

#create a function compose2

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

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

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

>>> a=compose2(d,e)   #  FunctionC = compose(FunctionB,FunctionA)
>>> a(5)              # FunctionC(x)
12
```


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

Example 2:

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

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

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

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

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

Recursion:

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

Python Recursive Function

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

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

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

```python
# Write a program to factorial using recursion
def fact(x):
    if x==0:
        result = 1
    else :
        result = x * fact(x-1)
    return result
print("zero factorial",fact(0))
print("five factorial",fact(5))
```

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/rec.py zero factorial 1
five factorial 120

- - - - - - - - - - - - - - - -
```python
def calc_factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * calc_factorial(x-1))
```

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

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/rec.py The factorial of 4 is 24

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

```
>>> type("name")
<class 'str'>
>>> name=str()
>>> name
"
>>> a=str('Islamiah')
>>> a
'Islamiah'
>>> a=str(Islamiah)
>>> a[2]
'c'
>>> fruit = 'banana'
>>> letter = fruit[1]
```
The second statement selects character number 1 from fruit and assigns it to letter. The expression in brackets is called an index. The index indicates which character in the sequence we want

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

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

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

`Syntax:`**[Start: stop: steps]**

- Slicing will start from index and will go up to **stop** in **step** of steps.

- Default value of start is 0,

- Stop is last index of list

- And for step default is 1

For example 1−

str = 'Hello World!'

print str   # Prints complete string

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

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

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

str * 2 # Prints string two times

print str + "TEST" # Prints concatenated string

Output:

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

Example 2:

>>> x='computer'

>>> x[1:4]

'omp'

>>> x[1:6:2]

'opt'

>>> x[3:]

'puter'

>>> x[:5]

'compu'

>>> x[-1]

'r'

>>> x[-3:]

'ter'

>>> x[:-2]

'comput'

>>> x[::-2]

'rtpo'

>>> x[::-1]

'retupmoc'

Immutability:

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

>>> greeting='Islamiah college!'
>>> greeting[0]='n'

TypeError: 'str' object does not support item assignment

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

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

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

String functions and methods:

There are many methods to operate on String.

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

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

1. isalnum():

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

Syntax:
String.isalnum()

Example:
>>> string="123alpha"
>>> string.isalnum() True

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

Syntax:
String.isalpha()

 Example:
>>> string="nikhil"
>>> string.isalpha()
True

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

Syntax:
String.isdigit()

Example:
>>> string="123456789"
>>> string.isdigit()
True

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

<u>Syntax:</u>

String.islower()

Example:
>>> string="nikhil"
>>> string.islower()
True

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

Syntax:
String.isnumeric()

Example:
>>> string="123456789"
>>> string.isnumeric()
True

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

Syntax:
String.isspace()

Example:
>>> string=" "
>>> string.isspace()
True

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

Syntax:
String.istitle()

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

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

Syntax:
String.isupper()

Example:
>>> string="HELLO"
>>> string.isupper()
True

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

Syntax:
String.replace()

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

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

Syntax:
String.split()

Example:
>>> string="Nikhil Is Learning"

```
>>> string.split()
```

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

    11.count()
count() method counts the occurrence of a string in another string <u>Syntax:</u>
String.count()

 <u>Example:</u>
>>> string='Nikhil Is Learning'
>>> string.count('i')
3

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

<u>Syntax:</u>
String.find(,,string")

<u>Example:</u>
>>> string="Nikhil Is Learning"
>>> string.find('k')
 2
    13.swapcase()
converts lowercase letters in a string to uppercase and viceversa

<u>Syntax:</u>
String.find(,,string")

<u>Example:</u>
>>> string="HELLO"
>>> string.swapcase()
 'hello'

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

Syntax:
String.startswith(„string")

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

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

Syntax:
String.endswith(„string")

 Example:
>>> string="Nikhil Is Learning"
>>> string.startswith('g')
True

String module:

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

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

Syntax: import string

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

Example:
import string
print(string.ascii_letters)
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.digits)

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

Output:

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

==

abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789

0123456789abcdefABCDEF

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

Python String Module Classes

Python string module contains two classes – Formatter and Template.

Formatter

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

Syntax: from string import Formatter

Template

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

Syntax: from string import Template

Python arrays:

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

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

Array Representation

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

Elements

Int array [10] = {10, 20, 30, 40, 50, 60, 70, 80, 85, 90}

Type Name Size   Index 0
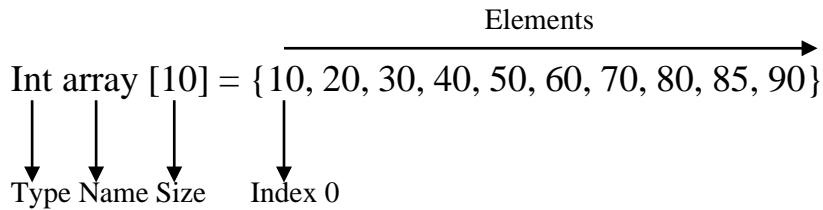
As per the above illustration, following are the important points to be considered.

- Index starts with 0.

- Array length is 10 which means it can store 10 elements.

- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 70

Basic Operations

Following are the basic operations supported by an array.

- Traverse − print all the array elements one by one.

- Insertion − Adds an element at the given index.

- Deletion − Deletes an element at the given index.

- Search − Searches an element using the given index or by the value.

- Update − Updates an element at the given index.

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

```
from array import *

arrayName=array(typecode, [initializers])
```

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

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

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

Creating an array:

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

Output:

>>>

 RESTART: C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/arr.py10
20
30
40
50

Access the elements of an Array:

Accessing Array Element

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

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

Output:

RESTART: C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/arr2.py10
30

Array methods:

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

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

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

Example:

>>> college=["Islamiah","it","cse"]

>>> college.append("autonomous")

>>> college

['Islamiah', 'it', 'cse', 'autonomous']

>>> college.append("eee")

>>> college.append("ece")

>>> college

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

>>> college.pop()

'ece'

>>> college

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

>>> college.pop(4)

'eee'

>>> college

['Islamiah', 'it', 'cse', 'autonomous']

>>> college.remove("it")

>>> college

['Islamiah', 'cse', 'autonomous']

LISTS, TUPLES, DICTIONARIES

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

<u>Lists, Tuples, Dictionaries:</u>

List:

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

Ex:

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

>>> print(list1)

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

- - - - - - - - - - - - - -

>>> x=list()

>>> x

[]

- - - - - - - - - - - - - - - -

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

>>> x=list(tuple1)

>>> x

[1, 2, 3, 4]

List operations:

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

Basic List Operations:

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

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

Indexing, Slicing, and Matrixes

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

Assuming following input −

L = ['Islamiah', 'college', 'ISLAMIAH!']

Python Expression	Results	Description
L[2]	ISLAMIAH	Offsets start at zero

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

List slices:

>>> list1=range(1,6)

>>> list1

range(1, 6)

>>> print(list1)

range(1, 6)

>>> list1=[1,2,3,4,5,6,7,8,9,10]

>>> list1[1:]

[2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> list1[:1]

[1]

>>> list1[2:5]

[3, 4, 5]

>>> list1[:6]

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

>>> list1[1:2:4]

[2]

>>> list1[1:8:2]

[2, 4, 6, 8]

List methods:

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

- Del()

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

Delete: Delete a list or an item from a list

>>> x=[5,3,8,6]

>>> del(x[1])   #deletes the index position 1 in a list

>>> x

[5, 8, 6]

- - - - - - - -

>>> del(x)

>>> x   # complete list gets deleted

Append: Append an item to a list

>>> x=[1,5,8,4]

>>> x.append(10)

>>> x

[1, 5, 8, 4, 10]

Extend: Append a sequence to a list.

>>> x=[1,2,3,4]

>>> y=[3,6,9,1]

>>> x.extend(y)

>>> x

[1, 2, 3, 4, 3, 6, 9, 1]

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

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

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

>>> x

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

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

>>> x

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

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

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

>>> x.pop()

7

>>> x

[1, 2, 10, 4, 6]

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

>>> x.pop(2)

10

>>> x

[1, 2, 4, 6]

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

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

>>> x.remove(33)

>>> x

[1, 2, 10, 4, 6]

>>> x.remove(4)

>>> x

[1, 2, 10, 6]

Reverse: Reverse the order of a given list.

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

>>> x.reverse()

>>> x

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

Sort: Sorts the elements in ascending order

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

>>> x.sort()

>>> x

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

- - - - - - - - - - - - - -

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

>>> x.sort()

>>> x

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

List loop:

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

Method #1: For loop

#list of items

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

```
i = 1

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

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

Method #2: For loop and range()
In case we want to use the traditional for loop which iterates from number x to number y.
```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]

# getting length of list
length = len(list)

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

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy/listlooop.py 1
3
5
7
9
Method #3: using while loop

```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]
```

Getting length of list

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

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

Mutability:

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

Append: Append an item to a list

>>> x=[1,5,8,4]

>>> x.append(10)

>>> x

[1, 5, 8, 4, 10]

Extend: Append a sequence to a list.

>>> x=[1,2,3,4]

>>> y=[3,6,9,1]

>>> x.extend(y)

>>> x

Delete: Delete a list or an item from a list

>>> x=[5,3,8,6]

>>> del(x[1])   #deletes the index position 1 in a list

>>> x

[5, 8, 6]

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

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

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

```
>>> x
```

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

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

>>> x
```

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

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

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

>>> x.pop()
```

7

```
>>> x
```

[1, 2, 10, 4, 6]

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

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

10

```
>>> x
```

[1, 2, 4, 6]

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

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

>>> x.remove(33)

>>> x
```

[1, 2, 10, 4, 6]

>>> x.remove(4)

>>> x

[1, 2, 10, 6]

Reverse: Reverse the order of a given list.

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

>>> x.reverse()

>>> x

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

Sort: Sorts the elements in ascending order

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

>>> x.sort()

>>> x

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

- - - - - - - - - - - - - -

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

>>> x.sort()

>>> x

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

Aliasing:

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

For ex:
a = [81, 82, 83]

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

Output:

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

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

Cloning Lists:

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

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

Example:

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

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

print(a == b)

print(a is b)

b[0] = 5

print(a)

print(b)
```

Output:

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

True
False
[81, 82, 83]
[5, 82, 83]
Now we are free to make changes to b without worrying about a

List parameters:

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

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

```python
def doubleStuff(List):

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

    for position in range(len(List)):

        List[position] = 2 * List[position]


things = [2, 5, 9]

print(things)

doubleStuff(things)

print(things)
```

Output:

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

==[2, 5, 9]

[4, 10, 18]

List comprehension:

List:

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

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

>>> list1=[]

>>> for x in range(10):

    list1.append(x**2)

>>> list1

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

(or)

This is also equivalent to

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

>>> list1

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

(or)

Which is more concise and redable.

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

>>> list1

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

<u>Similarily some examples:</u>

```
>>> x=[m for m in range(8)]
>>> print(x)
[0, 1, 2, 3, 4, 5, 6, 7]


>>> x=[z**2 for z in range(10) if z>4]
>>> print(x)
[25, 36, 49, 64, 81]


>>> x=[x ** 2  for x in range (1, 11)   if x % 2 == 1]
>>> print(x)
[1, 9, 25, 49, 81]


>>> a=5
>>> table = [[a, b, a * b] for b in range(1, 11)]
>>> for i in table:
        print(i)


[5, 1, 5]
[5, 2, 10]
[5, 3, 15]
[5, 4, 20]
[5, 5, 25]
[5, 6, 30]
[5, 7, 35]
[5, 8, 40]
[5, 9, 45]
[5, 10, 50]
```

Tuples:

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

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

accidently being added, changed, or deleted.

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

We can construct tuple in many ways:

X=()   #no item tuple

X=(1,2,3)

X=tuple(list1)

X=1,2,3,4

Example:

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

>>> print(x)

(1, 2, 3)

>>> x

(1, 2, 3)

>>> x=()

>>> x

()

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

>>> y=tuple(x)

>>> y

(4, 5, 66, 9)

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

>>> x

(1, 2, 3, 4)

Some of the operations of tuple are:

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

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

```
>>> x=('a','b','c','g')
>>> print(x[2])
c
```

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

```
>>> x=(2,5,7,'4',8)
>>> x[1]=10
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    x[1]=10
TypeError: 'tuple' object does not support item assignment
>>> x
(2, 5, 7, '4', 8)     # the value is still the same
```

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

```
>>> x=4,5,6,7,2,'aa'
>>> for i in x:
        print(i)


4
5
6
7
2
aa
```

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

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

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

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

(Or)

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

Length (): To know the number of items or values present in a tuple, we use len().
```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=len(x)
>>> print(y)
12
```

Tuple Assignment

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

For   Example,

Here is the code,

- >>> tup1 = ('Islamiah', 'eng college','2004','cse', 'it','csit');

- >>> tup2 = (1,2,3,4,5,6,7);

- >>> print(tup1[0])

- Islamiah

- >>> print(tup2[1:4])

- (2, 3, 4)

Tuple 1 includes list of information of

IslamiahTuple 2 includes list of numbers in

it

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

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

Tuple as return values:

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

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

```
# This function returns a tuple
def fun():
    str = "Islamiah
    college"x    = 20
    return str, x; # Return tuple, we could also
                    # write (str, x)
 # Driver code to test above method
str, x = fun() # Assign returned tuple
print(str)
print(x)
```

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/tupretval.py Islamiah college
20

- Functions can return tuples as return values.

```
def circleInfo(r):
    """ Return (circumference, area) of a circle of radius r """
    c = 2 * 3.14159 * r
    a = 3.14159 * r * r
    return (c, a)
print(circleInfo(10))
```
Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/functupretval.py (62.8318, 314.159)

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

Tuple comprehension:

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

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

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

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

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

a
b
c
```

Create a list of 2-tuples like (number, square):
```
>>> z=[(x, x**2) for x in range(6)]
>>> z
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

<u>Set:</u>

Similarly to list comprehensions, set comprehensions are also supported:

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

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

Dictionaries:

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

We can construct or create dictionary like:

X={1:'A',2:'B',3:'c'}
X=dict([('a',3) ('b',4)]
X=dict('A'=1,'B' =2)

Example:
>>> dict1 = {"brand":"Islamiah","model":"college","year":2004}
>>> dict1
{'brand': 'Islamiah', 'model': 'college', 'year': 2004}

Operations and methods:

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

Method	Description

clear()	Remove all items form the dictionary.

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

Below are some dictionary operations:

To access specific value of a dictionary, we must pass its key,
```
>>> dict1 = {"brand":"Islamiah","model":"college","year":2004}
>>> x=dict1["brand"]
>>> x
'Islamiah'
```
- - - - - - - - - - - - -
To access keys and values and items of dictionary:
```
>>> dict1 = {"brand":"Islamiah","model":"college","year":2004}
>>> dict1.keys()
dict_keys(['brand', 'model', 'year'])
>>> dict1.values()
dict_values(['Islamiah', 'college',
2004])
>>> dict1.items()
dict_items([('brand', 'Islamiah'), ('model', 'college'), ('year', 2004)])
```
- -
```
>>> for items in dict1.values():
        print(items)


Islamia
h
college
2004

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


brand
model
year

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


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

Some more operations like:
- Add/change

- Add/change

- Remove
- Length
- Delete

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

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

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

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

Delete: Deletes a particular item.

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

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

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

Iterating over (key, value) pairs:
```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> for key in x:
        print(key, x[key])
```

```
1 1
2 4
3 9
```

```
4 16
5 25
>>> for k,v in x.items():
        print(k,v)
```

```
1 1
2 4
3 9
4 16
5 25
```

List of Dictionaries:

```
>>> customers = [{"uid":1,"name":"John"},
    {"uid":2,"name":"Smith"},
        {"uid":3,"name":"Andersson"},
        ]
>>> >>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'Andersson'}]
```

```
## Print the uid and name of each customer
>>> for x in customers:
        print(x["uid"], x["name"])
```

```
1 John
2 Smith
3 Andersson
```

```
## Modify an entry, This will change the name of customer 2 from Smith to Charlie
>>> customers[2]["name"]="charlie"
>>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'charlie'}]
```

```
## Add a new field to each entry
```

```
>>> for x in customers:
        x["password"]="123456" # any initial value
```

```
>>> print(customers)
```

[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 2, 'name': 'Smith', 'password': '123456'}, {'uid': 3, 'name': 'charlie', 'password': '123456'}]

Delete a field
```
>>> del customers[1]
>>> print(customers)
```
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 3, 'name': 'charlie', 'password': '123456'}]

```
>>> del customers[1]
>>> print(customers)
```
[{'uid': 1, 'name': 'John', 'password': '123456'}]

Delete all fields

```
>>> for x in customers:
        del x["uid"]
```


```
>>> x
```
{'name': 'John', 'password': '123456'}

Comprehension:

Dictionary comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> z={x: x**2 for x in (2,4,6)}
>>> z
```
{2: 4, 4: 16, 6: 36}

```
>>> dict11 = {x: x*x for x in range(6)}
>>> dict11
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

FILES, EXCEPTIONS, MODULES, PACKAGES

Files and exception: text files, reading and writing files, command line arguments, errors and exceptions, handling exceptions, modules (datetime, time, OS , calendar, math module), Explore packages.

Files, Exceptions, Modules, Packages:

Files and exception:

A file is some information or data which stays in the computer storage devices. Python gives you easy ways to manipulate these files. Generally files divide in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.

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

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

Text files:

We can create the text files by using the syntax:

 Variable name=open ("file.txt", file mode)

For ex: f= open ("hello.txt","w+")

- We declared the variable f to open a file named hello.txt. **Open** takes 2 arguments, the file that we want to open and a string that represents the kinds of permission or operation we want to do on the file
- Here we used "w" letter in our argument, which indicates write and the plus sign that means it will create a file if it does not exist in library

- The available option beside "w" are "r" for read and "a" for append and plus sign means if it is not there then create it

File Modes in Python:

Mode	Description
'r'	This is the default mode. It Opens file for reading.
'w'	This Mode Opens file for writing. If file does not exist, it creates a new file. If file exists it truncates the file.
'x'	Creates a new file. If file already exists, the operation fails.
'a'	Open file in append mode. If file does not exist, it creates a new file.
't'	This is the default mode. It opens in text mode.
'b'	This opens in binary mode.
'+'	This will open a file for reading and writing (updating)

Reading and Writing files:

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

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

C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>start notepad hello.txt

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

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>notepad 111.txt
```

Hit on enter then it shows the following whether to open or not?

Untitled - Notepad

File   Edit   Format   View   Help

Notepad   ×

!   Cannot find the 111.txt file.

Do you want to create a new file?

Yes   No   Cancel
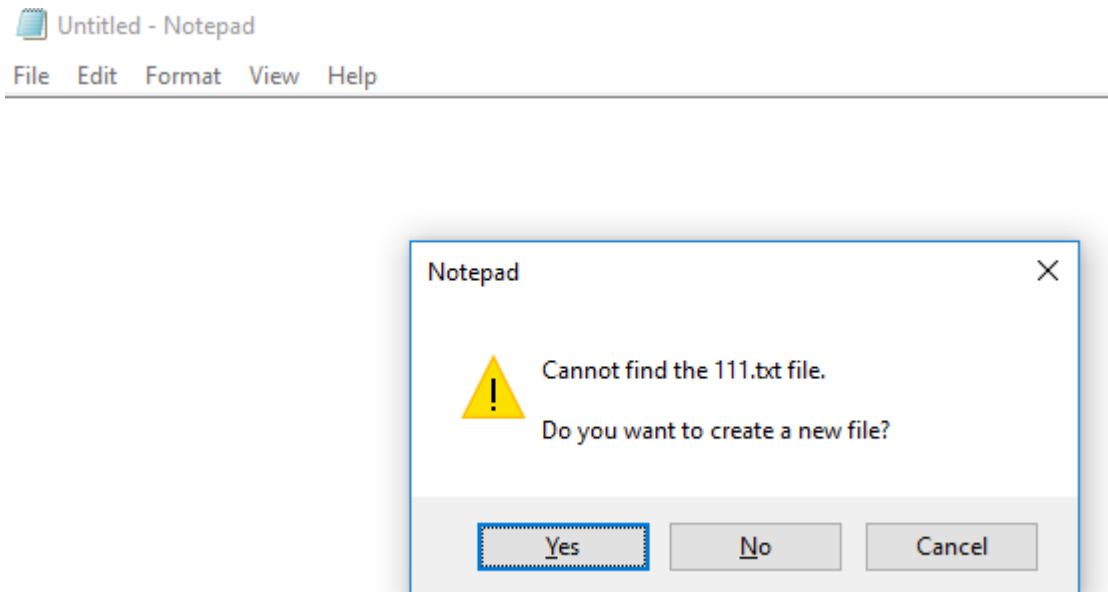
Click on "yes" to open else "no" to cancel

Write a python program to open and read a file

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

print(a.read())

a.close()

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

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>python f1.py
welcome to python programming
```

Note: All the program files and text files need to saved together in a particular file then only the program performs the operations in the given file mode
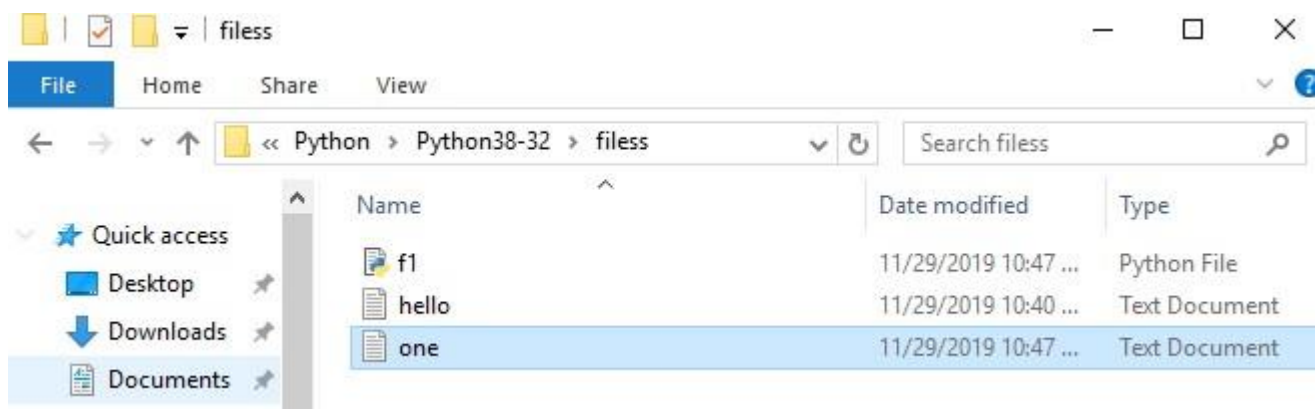
f.close() ---- This will close the instance of the file somefile.txt stored

 # Write a python program to open and write "hello world" into a file?

f=open("1.txt","a")

f.write("hello world")

f.close()

Output:

1 - Notepad

File   Edit   Format   View   Help

hello world

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type 1.txt
hello world
```

Note: In the above program the 1.txt file is created automatically and adds hello world into txt file

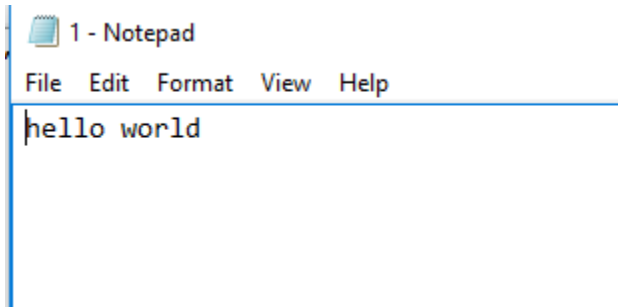If we keep on executing the same program for more than one time then it append the data that many times

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type 1.txt
hello worldhello world
```

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

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

f.write("hi python programming")

f.close()

Output:

hello - Notepad

File   Edit   Format   View   Help

hi python programming
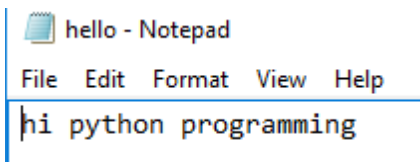
In the above program the hello txt file consist of data like

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

But when we try to write some data on to the same file it overwrites and saves with the current data (check output)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type hello.txt
hi python programming
```

Write a python program to open and write the content to file and read it.

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

fo.write("Python Programming")

print(fo.read())

fo.close()

Output:

abc - Notepad

File   Edit   Format   View   Help

Python Programming introduced for III years

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type abc.txt
Python Programming introduced for III years
```

Note: It creates the abc.txt file automatically and writes the data into it

filess

File   Home   Share   View

« Python > Python38-32 > filess   Search filess

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

Quick access
Desktop
Downloads
Documents
Pictures
ANE4
mlr17
mlr18
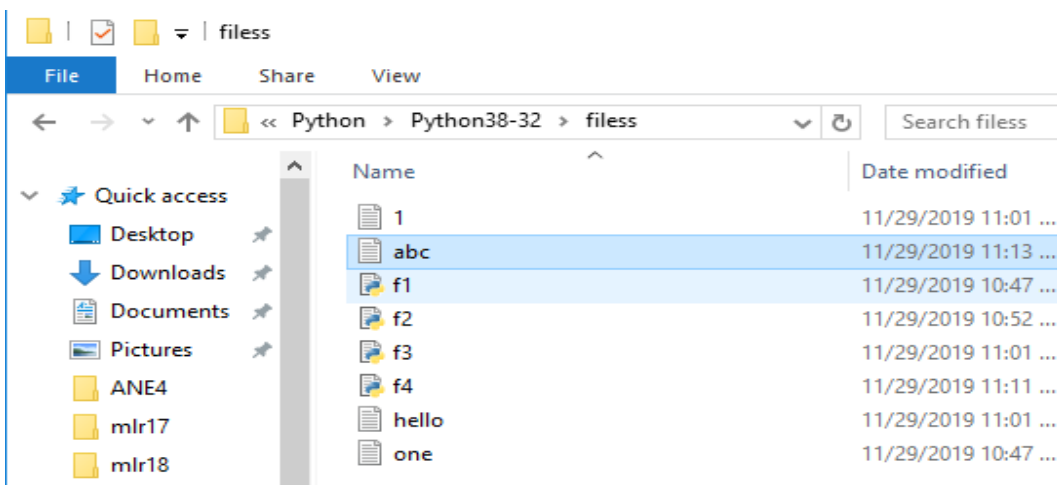
Command line arguments:

The command line arguments must be given whenever we want to give the input before the start of the script, while on the other hand, raw_input() is used to get the input while the python program / script is running.

The command line arguments in python can be processed by using either 'sys' module, 'argparse' module and 'getopt' module.

'sys' module :

Python sys module stores the command line arguments into a list, we can access it using sys.argv. This is very useful and simple way to read command line arguments as String.
sys.argv is the list of commandline arguments passed to the Python program. argv represents all the items that come along via the command line input, it's basically an array holding the command line arguments of our program

>>> sys.modules.keys() -- this prints so many dict elements in the form of list.

Python code to demonstrate the use of 'sys' module for command line arguments

import sys

# command line arguments are stored in the form
# of list in sys.argv
argumentList = sys.argv
print(argumentList)

# Print the name of file
print(sys.argv[0])

Print the first argument after the name of file
#print(sys.argv[1])

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/cmndlinarg.py

['C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/cmndlinarg.py']
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/cmndlinarg.py

Note: Since my list consist of only one element at '0' index so it prints only that list element, if we try to access at index position '1' then it shows the error like,

IndexError: list index out of range
- - - - - - - - - - - -

import sys

```python
print(type(sys.argv))
print('The command line arguments are:')
for i in sys.argv:
    print(i)
```

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/symod.py ==
<class 'list'>
The command line arguments are:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/symod.py

 # write a python program to get python version.
```python
import sys
print("System version is:")
print(sys.version)
print("Version Information is:")
print(sys.version_info)
```

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/s1.py =
System version is:
3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)]
Version Information is:
sys.version_info(major=3, minor=8, micro=0, releaselevel='final', serial=0)

'argparse' module :

Python getopt module is very similar in working as the C getopt() function for parsing command-line parameters. Python getopt module is useful in parsing command line arguments where we want user to enter some options too.

>>> parser = argparse.ArgumentParser(description='Process some integers.')

#

```
import argparse

parser = argparse.ArgumentParser()
print(parser.parse_args())
```

'getopt' module :

Python argparse module is the preferred way to parse command line arguments. It provides a lot of option such as positional arguments, default value for arguments, help message, specifying data type of argument etc

It parses the command line options and parameter list. The signature of this function is mentioned below:

getopt.getopt(args, shortopts, longopts=[ ])

- args are the arguments to be passed.
- shortopts is the options this script accepts.
- Optional parameter, longopts is the list of String parameters this function accepts which should be supported. Note that the -- should not be prepended with option names.

-h ----------- print help and usage message
-m----------- accept custom option value
-d ----------- run the script in debug mode

```
import getopt
import sys

argv = sys.argv[0:]
try:
    opts, args = getopt.getopt(argv, 'hm:d', ['help', 'my_file='])
    #print(opts)
    print(args)
except getopt.GetoptError:
    # Print a message or do something useful
    print('Something went wrong!')
    sys.exit(2)
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/gtopt.py

==['C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/gtopt.py']

Errors and Exceptions:

Python Errors and Built-in Exceptions: Python (interpreter) raises exceptions when it encounters **errors.** When writing a program, we, more often than not, will encounter errors. Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error

ZeroDivisionError:

ZeroDivisionError in Python indicates that the second argument used in a division (or modulo) operation was zero**.**

OverflowError:

OverflowError in Python indicates that an arithmetic operation has exceeded the limits of the current Python runtime. This is typically due to excessively large float values, as integer values that are too big will opt to raise memory errors instead.

ImportError:

It is raised when you try to import a module which does not exist. This may happen if you made a typing mistake in the module name or the module doesn't exist in its standard path. In the example below, a module named "non_existing_module" is being imported but it doesn't exist, hence an import error exception is raised.

IndexError:

An IndexError exception is raised when you refer a sequence which is out of range. In the example below, the list abc contains only 3 entries, but the 4th index is being accessed, which will result an IndexError exception.

TypeError:

When two unrelated type of objects are combined, TypeErrorexception is raised.In example below, an int and a string is added, which will result in TypeError exception.

IndentationError:

Unexpected indent. As mentioned in the "expected an indentedblock" section, Python not only insists on indentation, it insists on consistentindentation. You are free to choose the number of spaces of indentation to use, but you then need to stick with it.

Syntax errors:

These are the most basic type of error. They arise when the Python parser is unable to understand a line of code. Syntax errors are almost always fatal, i.e. there is almost never a way to successfully execute a piece of code containing syntax errors.

Run-time error:

A run-time error happens when Python understands what you are saying, but runs into trouble when following your instructions.

Key Error :

Python raises a KeyError whenever a dict() object is requested (using the format a = adict[key]) and the key is not in the dictionary.

Value Error:

In Python, a value is the information that is stored within a certain object. To encounter a ValueError in Python means that is a problem with the content of the object you tried to assign the value to.

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong. In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from Exception class.

Different types of exceptions:

- ArrayIndexOutOfBoundException.
- ClassNotFoundException.
- FileNotFoundException.
- IOException.
- InterruptedException.
- NoSuchFieldException.
- NoSuchMethodException

Handling Exceptions:

The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc. Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files. Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.

Python uses try and except keywords to handle exceptions. Both keywords are followed by indented blocks.

Syntax:

try :

   #statements in try block

except :

   #executed when error in try block

Typically we see, most of the times

- **Syntactical errors** (wrong spelling, colon (:) missing ….),
  At developer level and compile level it gives errors.

- **Logical errors** (2+2=4, instead if we get output as 3 i.e., wrong output …..,),
  As a developer we test the application, during that time logical error may obtained.

- **Run time error** (In this case, if the user doesn't know to give input, 5/6 is ok but if the user say 6 and 0 i.e.,6/0 (shows error a number cannot be divided by zero))
  This is not easy compared to the above two errors because it is not done by the system, it is (mistake) done by the user.

The things we need to observe are:

1. You should be able to understand the mistakes; the error might be done by user, DB connection or server.
2. Whenever there is an error execution should not stop.
   Ex: Banking Transaction
3. The aim is execution should not stop even though an error occurs.

For ex:

a=5

b=2

print(a/b)

print("Bye")

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/ex1.py 2.5

Bye

- **The above is normal execution with no error, but if we say when b=0, it is a critical and gives error, see below**

a=5

b=0

print(a/b)

print("bye") #this has to be printed, but abnormal termination

Output:

Traceback (most recent call last):

  File "C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/ex2.py", line3, in <module>

    print(a/b)

ZeroDivisionError: division by zero

- **To overcome this we handle exceptions using except keyword**

a=5

b=0

```
try:

    print(a/b)

except Exception:

    print("number can not be divided by zero")

    print("bye")
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/ex3.py number can not be divided by zero

bye

- **The except block executes only when try block has an error, check it below**

```
a=5

b=2

try:

    print(a/b)

except Exception:

    print("number can not be divided by zero")

    print("bye")
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/ex4.py 2.5

- **For example if you want to print the message like what is an error in a program then we use "e" which is the representation or object of an exception.**

```
a=5

b=0

try:
```

```
    print(a/b)

except Exception as e:

    print("number can not be divided by zero",e)

print("bye")
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/ex5.py number can not be divided by zero **division by zero**

bye

(Type of error)

Let us see some more examples:

I don't want to print bye but I want to close the file whenever it is opened.

```
a=5

b=2

try:

    print("resource opened")

    print(a/b)

    print("resource closed")

except Exception as e:

    print("number can not be divided by zero",e)
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/ex6.py resource opened

2.5

resource closed

- **Note: the file is opened and closed well, but see by changing the value of b to 0,**

a=5

b=0

try:

   print("resource opened")

   print(a/b)

   print("resource closed")

except Exception as e:

   print("number can not be divided by zero",e)

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/ex7.py resource opened

number can not be divided by zero division by zero

- **Note: resource not closed**
- **To overcome this, keep print("resource closed") in except block, see it**

a=5

b=0

try:

   print("resource opened")

   print(a/b)

except Exception as e:

     print("number can not be divided by zero",e)

     print("resource closed")

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/ex8.py resource opened

number can not be divided by zero division by zero

resource closed

- **The result is fine that the file is opened and closed, but again change the value of b to back (i.e., value 2 or other than zero)**

```
a=5
b=2
try:
    print("resource opened")
    print(a/b)
except Exception as e:
        print("number can not be divided by zero",e)
        print("resource closed")
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/ex9.py resource opened

2.5

- **But again the same problem file/resource is not closed**
- **To overcome this python has a feature called finally:**

This block gets executed though we get an error or not

Note: Except block executes, only when try block has an error, but finally block executes, even though you get an exception.

```
a=5
b=0

try:
```

```
        print("resource open")
        print(a/b)
        k=int(input("enter a number"))
        print(k)
except ZeroDivisionError as e:
        print("the value can not be divided by zero",e)
finally:
        print("resource closed")
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py

resource open

the value can not be divided by zero division by zero

resource closed

- **change the value of b to 2 for above program, you see the output like**

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py

resource open

2.5

enter a number 6

6

resource closed

- **Instead give input as some character or string for above program, check the output**

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py

resource open

2.5

enter a number p

resource closed

Traceback (most recent call last):

  File "C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py",
line7, in <module>

    k=int(input("enter a number"))

ValueError: invalid literal for int() with base 10: ' p'

```
#_____
a=5
b=0

try:
        print("resource open")
        print(a/b)
        k=int(input("enter a number"))
        print(k)
except ZeroDivisionError as e:
        print("the value can not be divided by zero",e)
except ValueError as e:
        print("invalid input")
except Exception as e:
        print("something went wrong...",e)

finally:
        print("resource closed")
```

Output:
C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/ex11.py
resource open
the value can not be divided by zero division by zero
resource closed

- **Change the value of b to 2 and give the input as some character or string (other than int)**

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/ex12.py
resource open
2.5
enter a number p
invalid input
resource closed

Modules (Date, Time, os, calendar, math):

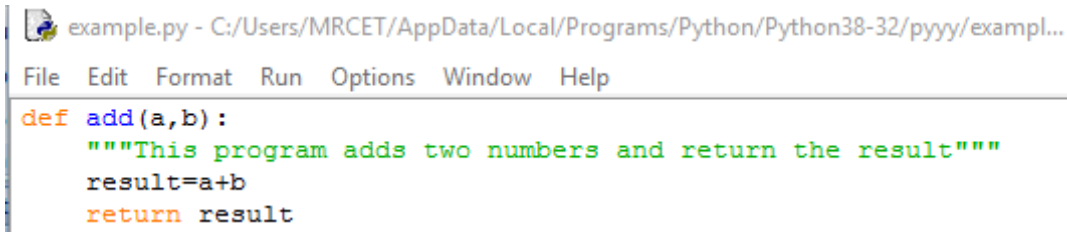- Modules refer to a file containing Python statements and definitions.

- We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

- **Modular programming** refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or **modules**.

Advantages :

- **Simplicity:** Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem. If you're working on a single module, you'll have a smaller problem domain to wrap your head around. This makes development easier and less error-prone.

- **Maintainability:** Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program. This makes it more viable for a team of many programmers to work collaboratively on a large application.

- **Reusability:** Functionality defined in a single module can be easily reused (through an appropriately defined interface) by other parts of the application. This eliminates the need to recreate duplicate code.

- **Scoping:** Modules typically define a separate **namespace**, which helps avoid collisions between identifiers in different areas of a program.

- **Functions**, **modules** and **packages** are all constructs in Python that promote code modularization.

A file containing Python code, for e.g.: example.py, is called a module and its module name would be example.

```
>>> def add(a,b):
        result=a+b
        return result
        """This program adds two numbers and return the result"""
```

```
example.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/exampl...
File  Edit  Format  Run  Options  Window  Help
def add(a,b):
    """This program adds two numbers and return the result"""
    result=a+b
    return result
```

Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

How to import the module is:

- We can import the definitions inside a module to another module or the Interactive interpreter in Python.

- We use the import keyword to do this. To import our previously defined module example we type the following in the Python prompt.

- Using the module name we can access the function using dot (.) operation. For Eg:

>>> import example

>>> example.add(5,5)

10

- Python has a ton of standard modules available.   Standard modules can be imported the same way as we import our user-defined modules.

Reloading a module:

def hi(a,b):

   print(a+b)

hi(4,4)

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/add.py 8

>>> import add

8

>>> import add

>>> import add

>>>

Python provides a neat way of doing this. We can use the reload() function inside the imp module to reload a module. This is how its done.

- >>> import imp

- >>> import my_module

- This code got executed >>> import my_module >>> imp.reload(my_module) This code got executed <module 'my_module' from '.\\my_module.py'>how its done.

>>> import imp

>>> import add

>>> imp.reload(add)

8

<module 'add' from
            'C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-
32/pyyy\\add.py'>

The dir() built-in function

>>> import example

>>> dir(example)

['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'add']

>>> dir()

['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '__warningregistry__', 'add', 'example', 'hi', 'imp']

It shows all built-in and user-defined modules.

For ex:

```
>>> example.__name__
'example'
```

Datetime module:

Write a python program to display date, time

```
>>> import datetime
>>> a=datetime.datetime(2019,5,27,6,35,40)
>>> a
datetime.datetime(2019, 5, 27, 6, 35, 40)
```

write a python program to display date

```
import  datetime
a=datetime.date(2000,9,18)
print(a)
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/d1.py =2000-09-18

write a python program to display time

```
import datetime
a=datetime.time(5,3)
print(a)
```

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =

05:03:00

#write a python program to print date, time for today and now.

```
import datetime
```

a=datetime.datetime.today()

b=datetime.datetime.now()

print(a)

print(b)

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/d1.py =2019-11-29 12:49:52.235581

2019-11-29 12:49:52.235581

#write a python program to add some days to your present date and print the date added.

import datetime

a=datetime.date.today()

b=datetime.timedelta(days=7)

print(a+b)

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/d1.py =2019-12-06

#write a python program to print the no. of days to write to reach your birthday

import datetime

a=datetime.date.today()

b=datetime.date(2020,5,27)

c=b-a

print(c)

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py

=180 days, 0:00:00

#**write an python program to print date, time using date and time functions**

import datetime

t=datetime.datetime.today()

print(t.date())

print(t.time())

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/d1.py =2019-11-29

12:53:39.226763

Time module:

#**write a python program to display time.**

import time

print(time.time())

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =

1575012547.1584706

#**write a python program to get structure of time stamp**.

import time

print(time.localtime(time.time()))

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =

time.struct_time(tm_year=2019, tm_mon=11, tm_mday=29, tm_hour=13, tm_min=1, tm_sec=15, tm_wday=4, tm_yday=333, tm_isdst=0)

#write a python program to make a time stamp.

import time

a=(1999,5,27,7,20,15,1,27,0)

print(time.mktime(a))

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =

927769815.0

#write a python program using sleep().

import time

time.sleep(6)   #prints after 6 seconds

print("Python Lab")

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =

Python Lab (#prints after 6 seconds)

os module:

>>> import os

>>> os.name

'nt'

>>> os.getcwd()

'C:\\Users\\ISLAMIAH\\AppData\\Local\\Programs\\Python\\Python38-

32\\pyyy'

>>> os.mkdir("temp1")
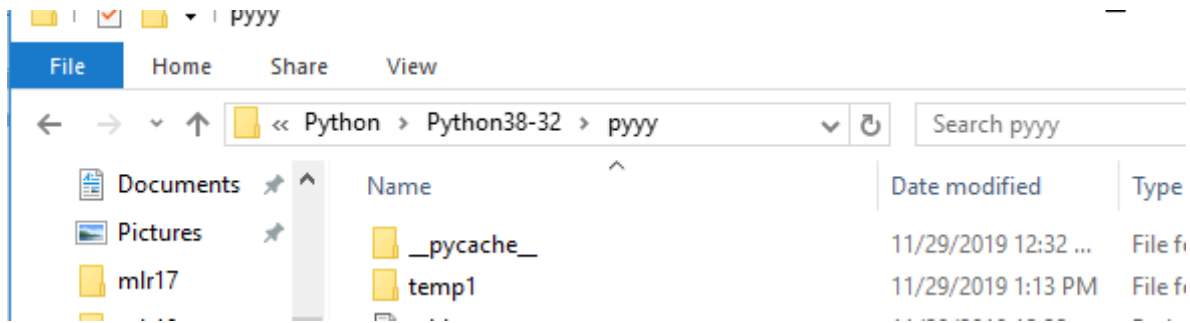
Note: temp1 dir is created

```
>>> os.getcwd()
'C:\\Users\\ISLAMIAH\\AppData\\Local\\Programs\\Python\\Python38-32\\pyyy'
>>> open("t1.py","a")
<_io.TextIOWrapper name='t1.py' mode='a' encoding='cp1252'>
>>> os.access("t1.py",os.F_OK)
True
>>> os.access("t1.py",os.W_OK)
True
>>> os.rename("t1.py","t3.py")
>>> os.access("t1.py",os.F_OK)
False
>>> os.access("t3.py",os.F_OK)
True
>>> os.rmdir('temp1')
(or)
os.rmdir('C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/temp1')
```
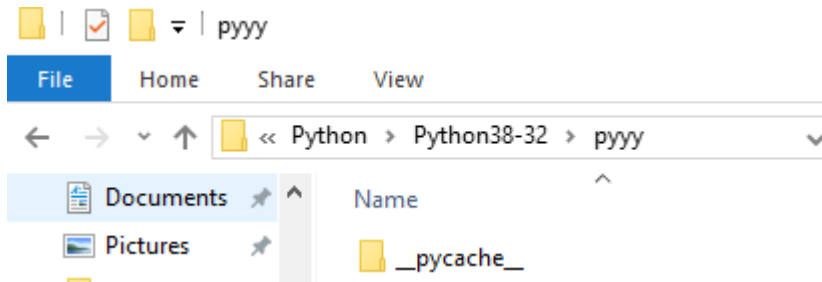
Note: Temp1dir is removed

>>> os.remove("t3.py")

Note: We can check with the following cmd whether removed or not

>>> os.access("t3.py",os.F_OK)

False

>>> os.listdir()

['add.py', 'ali.py', 'alia.py', 'arr.py', 'arr2.py', 'arr3.py', 'arr4.py', 'arr5.py', 'arr6.py', 'br.py', 'br2.py', 'bubb.py', 'bubb2.py', 'bubb3.py', 'bubb4.py', 'bubbdesc.py', 'clo.py', 'cmndlinarg.py', 'comm.py', 'con1.py', 'cont.py', 'cont2.py', 'd1.py', 'dic.py', 'e1.py', 'example.py', 'f1.y.py', 'flowof.py', 'fr.py', 'fr2.py', 'fr3.py', 'fu.py', 'fu1.py', 'if1.py', 'if2.py', 'ifelif.py', 'ifelse.py', 'iff.py', 'insertdesc.py', 'inserti.py', 'k1.py', 'l1.py', 'l2.py', 'link1.py', 'linklisttt.py', 'lis.py', 'listlooop.py', 'm1.py', 'merg.py', 'nesforr.py', 'nestedif.py', 'opprec.py', 'paraarg.py', 'qucksort.py', 'qukdesc.py', 'quu.py', 'r.py', 'rec.py', 'ret.py', 'rn.py', 's1.py', 'scoglo.py', 'selecasce.py', 'selectdecs.py', 'stk.py', 'strmodl.py', 'strr.py', 'strr1.py', 'strr2.py', 'strr3.py', 'strr4.py', 'strrmodl.py', 'wh.py', 'wh1.py', 'wh2.py', 'wh3.py', 'wh4.py', 'wh5.py', '__pycache__']

>>> os.listdir('C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32')

['argpar.py', 'br.py', 'bu.py', 'cmndlinarg.py', 'DLLs', 'Doc', 'f1.py', 'f1.txt', 'filess', 'functupretval.py', 'funturet.py', 'gtopt.py', 'include', 'Lib', 'libs', 'LICENSE.txt', 'lisparam.py', 'mysite', 'NEWS.txt', 'niru', 'python.exe', 'python3.dll', 'python38.dll', 'pythonw.exe', 'pyyy', 'Scripts', 'srp.py', 'sy.py', 'symod.py', 'tcl', 'the_weather', 'Tools', 'tupretval.py', 'vcruntime140.dll']

Calendar module:

#write a python program to display a particular month of a year using calendar module.

import calendar

print(calendar.month(2020,1))

Output:

```
>>>
= RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/cl1.py
    January 2020
Mo Tu We Th Fr Sa Su
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

 # write a python program to check whether the given year is leap or not.

import calendar

print(calendar.isleap(2021))

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/pyyy/cl1.py False

#write a python program to print all the months of given year.

import calendar

print(calendar.calendar(2020,1,1,1))

Output:

```
>>>
= RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32,
                                2020

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

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

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

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

Activate Windows

math module:

 # write a python program which accepts the radius of a circle from user and computes the area.

import math

r=int(input("Enter radius:"))

area=math.pi*r*r

print("Area of circle is:",area)

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-32/pyyy/m.py =

Enter radius:4

Area of circle is: 50.26548245743669

>>> import math

>>> print("The value of pi is", math.pi)

O/P: The value of pi is 3.141592653589793

Import with renaming:

- We can import a module by renaming it as follows.

- **For Eg:**

    >>> import math as m

    >>> print("The value of pi is", m.pi)

O/P: The value of pi is 3.141592653589793

- We have renamed the math module as m. This can save us typing time in some cases.

- Note that the name math is not recognized in our scope. Hence, math.pi is invalid, m.pi is the correct implementation.

Python from...import statement:

- We can import specific names from a module without importing the module as a whole. Here is an example.

    >>> from math import pi

    >>> print("The value of pi is", pi)

O/P: The value of pi is 3.141592653589793

- We imported only the attribute pi from the module.

- In such case we don't use the dot operator. We could have imported multiple attributes as follows.

>>> from math import pi, e

>>> pi

3.141592653589793

>>> e

2.718281828459045

Import all names:

- We can import all names(definitions) from a module using the following construct.

    >>>from math import *

    >>>print("The value of pi is", pi)

- We imported all the definitions from the math module. This makes all names except those beginnig with an underscore, visible in our scope.

Explore packages:

- We don't usually store all of our files in our computer in the same location. We use a well-organized hierarchy of directories for easier access.

- Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.

- As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.

- Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.

- A directory must contain a file named __init__.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

- Here is an example. Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.
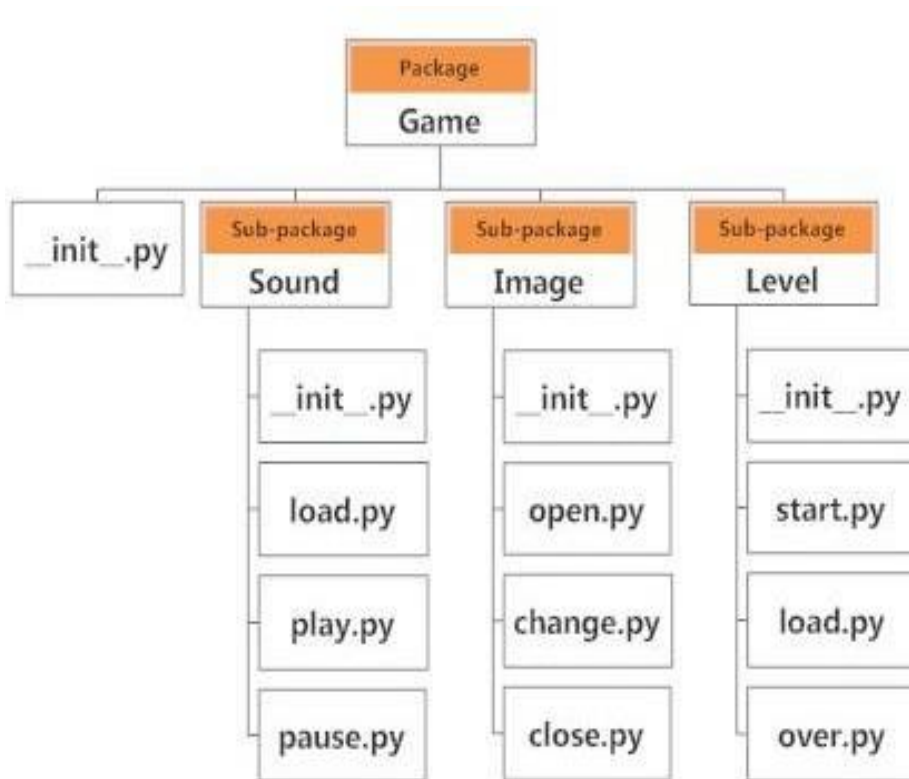
- If a file named_init_.py is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.

- For example   __init__.py

- A **module** in the package can access the global by importing it in turn

- We can import modules from packages using the dot (.) operator.

- For example, if want to import the start module in the above example, it is done as follows.

- import Game.Level.start

- Now if this module contains a function named **select_difficulty()**, we must use the full name to reference it.

- Game.Level.start.select_difficulty(2)

- If this construct seems lengthy, we can import the module without the package prefix as follows.

- from Game.Level import start

- We can now call the function simply as follows.

- **start.select_difficulty(2)**

- Yet another way of importing just the required function (or class or variable) form a module within a package would be as follows.

- **from Game.Level.start import select_difficulty**

- Now we can directly call this function.

- **select_difficulty(2)**

Examples:

#Write a python program to create a package (II YEAR),sub-package(CSE),modules(student) and create read and write function to module

def read():

   print("Department")

def write():

   print("Student")

Output:

>>> from IIYEAR.CSE import student

>>> student.read()

Department

>>> student.write()

Student

>>> from IIYEAR.CSE.student import read

>>> read

>>> read()

Department

>>> from IIYEAR.CSE.student import write

>>> write()

Student

# **Write a program to create and import module?**
def add(a=4,b=6):

   c=a+b

   return c

Output:

C:\Users\ISLAMIAH\AppData\Local\Programs\Python\Python38-32\IIYEAR\modu1.py

>>> from IIYEAR import modu1

>>> modu1.add()

10

Write a program to create and rename the existing module.

def a():

   print("hello world")

a()

Output:

C:/Users/ISLAMIAH/AppData/Local/Programs/Python/Python38-

32/IIYEAR/exam.py hello world

>>> import exam as ex

hello world

Unit - 4

Python Classes and Objects

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together.

.

Syntax: Class Definition

```
class ClassName:

    # Statement
```

Syntax: Object Definition

```
obj = ClassName()

print(obj.atrr)
```

Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Some points on Python class:
• Classes are created by keyword class.
• Attributes are the variables that belong to a class.
• Attributes are always public and can be accessed using the dot (.) operator.

Class Objects

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with *actual values*.
An object consists of :
• **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
• **Behaviour:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
• **Identity:** It gives a unique name to an object and enables one object to interact with other objects.
 Example:
class Dog:

        attr1 = "mammal"
        attr2 = "dog"

        def fun(self):
                print("I'm a", self.attr1)
                print("I'm a", self.attr2)
Rodger = Dog()
print(Rodger.attr1)
Rodger.fun()
Output

```
mammal
```

```
I'm a mammal
```

```
I'm a dog
```

In the above example, an object is created which is basically a dog named Rodger. This class only has two class attributes that tell us that Rodger is a dog and a mammal.

The self

- Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it.
- If we have a method that takes no arguments, we still have one argument.
- This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

__init__ method

The __init__ method is similar to constructors in C++ and Java. Constructors are used to initialize the object's state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Example:
```
class Person:
        # init method or constructor
        def __init__(self, name):
                self.name = name
        # Sample Method
        def say_hi(self):
                print('Hello, my name is', self.name)
p = Person('Nikhil')
p.say_hi()
```
Output:
```
Hello, my name is Nikhil
```

Class and Instance Variables

Instance variables are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.

Defining instance variables using a constructor.

```
# Python3 program to show that the variables with a value  assigned in the class declaration, are class variables and  variables inside methods and constructors are instance  variables.
class Dog:

        # Class Variable
        animal = 'dog'
        # The init method or constructor
        def __init__(self, breed, color):
                # Instance Variable
                self.breed = breed
                self.color = color
# Objects of Dog class
Rodger = Dog("Pug", "brown")
Buzo = Dog("Bulldog", "black")
print('Rodger details:')
print('Rodger is a', Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)
print('\nBuzo details:')
print('Buzo is a', Buzo.animal)
print('Breed: ', Buzo.breed)
```

```
print('Color: ', Buzo.color)
# Class variables can be accessed using class  name also
print("\nAccessing class variable using class name")
print(Dog.animal)

Example:
# program to show that we can create  instance variables inside methods
class Dog:
        # Class Variable
        animal = 'dog'
        # The init method or constructor
        def __init__(self, breed):
                # Instance Variable
                self.breed = breed
        # Adds an instance variable
        def setColor(self, color):
                self.color = color
        # Retrieves instance variable
        def getColor(self):
                return self.color
# Driver Code
Rodger = Dog("pug")
Rodger.setColor("brown")
print(Rodger.getColor())
```

Output
```
brown
```

Constructors in Python

Constructors are generally used for instantiating an object. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the __init__() method is called the constructor and is always called when an object is created.

Syntax of constructor declaration :
```
def __init__(self):

    # body of the constructor
```

Types of constructors :
• **default constructor:** The default constructor is a simple constructor which doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.
• **parameterized constructor:** constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Example 1:
```
class College:

        # default constructor
        def __init__(self):
                self.x = "Islamiah"

        # a method for printing data members
```

```python
        def print_college(self):
                print(self.x)


# creating object of the class
obj = College()

# calling the instance method using the object obj
obj.print_college()
```
Output :
```
Islamiah
```


Example of the parameterized constructor :
```python
class Addition:
        first = 0

        second = 0

        answer = 0


        # parameterized constructor
        def __init__(self, f, s):
                self.first = f
                self.second = s


        def display(self):
                print("First number = " + str(self.first))
                print("Second number = " + str(self.second))
                print("Addition of two numbers = " + str(self.answer))


        def calculate(self):
                self.answer = self.first + self.second


# creating object of the class
# this will invoke parameterized constructor
obj = Addition(1000, 2000)


# perform Addition
obj.calculate()
```

```
# display result
obj.display()
```

Output :
```
First number = 1000

Second number = 2000

Addition of two numbers = 3000
```

Destructors in Python

Destructors are called when an object gets destroyed.
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.
Syntax of destructor declaration :

```
def __del__(self):
   # body of destructor
```

Note : A reference to objects is also deleted when the object goes out of reference or when the program ends.
Example 1 : Here is the simple example of destructor. By using del keyword we deleted the all references of object 'obj', therefore destructor invoked automatically.

Example:
```
# Python program to illustrate destructor
class Employee:

        # Initializing
        def __init__(self):
                print('Employee created.')

        # Deleting (Calling destructor)
        def __del__(self):
                print('Destructor called, Employee deleted.')

obj = Employee()
del obj
```

Output:
```
Employee created.

Destructor called, Employee deleted.
```

Example:
```
# Python program to illustrate destructor

class Employee:

        # Initializing
        def __init__(self):
```

```python
                print('Employee created')

        # Calling destructor
        def __del__(self):
                print("Destructor called")

def Create_obj():
        print('Making Object...')
        obj = Employee()
        print('function end...')
        return obj

print('Calling Create_obj() function...')
obj = Create_obj()
print('Program End...')
```

Output:
```
Calling Create_obj() function...

Making Object...

Employee created

function end...

Program End...

Destructor called
```

Inheritance in Python

Inheritance is the capability of one class to derive or inherit the properties from another class.

Benefits of inheritance are:

• It represents real-world relationships well.
• It provides the **reusability** of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
• It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Python Inheritance Syntax

```
Class BaseClass:

    {Body}
Class DerivedClass(BaseClass):

    {Body}
```

Example:
A Python program to demonstrate inheritance

Base or Super class. Note object in bracket.
(Generally, object is made ancestor of all classes)
In Python 3.x "class Person" is
equivalent to "class Person(object)"

class Person(object):

        # Constructor
        def __init__(self, name):
                self.name = name

        # To get name
        def getName(self):
                return self.name

        # To check if this person is an employee
        def isEmployee(self):
                return False

# Inherited or Subclass (Note Person in bracket)
class Employee(Person):

        # Here we return true
        def isEmployee(self):
                return True

Driver code

```
emp = Person("Bsc1") # An Object of Person
print(emp.getName(), emp.isEmployee())

emp = Employee("Bsc2") # An Object of Employee
print(emp.getName(), emp.isEmployee())
```
Output:
```
Bsc1 False

Bsc2 True
```

Subclassing (Calling constructor of parent class)

A child class needs to identify which class is its parent class. This can be done by mentioning the parent class name in the definition of the child class.

Eg: class **subclass_name (superclass_name)**:

```
# Python code to demonstrate how parent constructors
# are called.

# parent class
class Person(object):

        # __init__ is known as the constructor
        def __init__(self, name, idnumber):
                self.name = name
                self.idnumber = idnumber

        def display(self):
                print(self.name)
                print(self.idnumber)

# child class
class Employee(Person):
        def __init__(self, name, idnumber, salary, post):
                self.salary = salary
                self.post = post

                # invoking the __init__ of the parent class
                Person.__init__(self, name, idnumber)


# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using its instance
a.display()
```
Output:
```
Rahul

886012
```

Different types of Inheritance:

- **Single inheritance**: When a child class inherits from only one parent class, it is called single inheritance. We saw an example above.
- **Multiple inheritances**: When a child class inherits from multiple parent classes, it is called multiple inheritances.
- **Multi level** : When we have a child and grandchild relationship.
- **Hierarchical inheritance:** More than one derived class are created from a single base.
- **Hybrid inheritance:** This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.
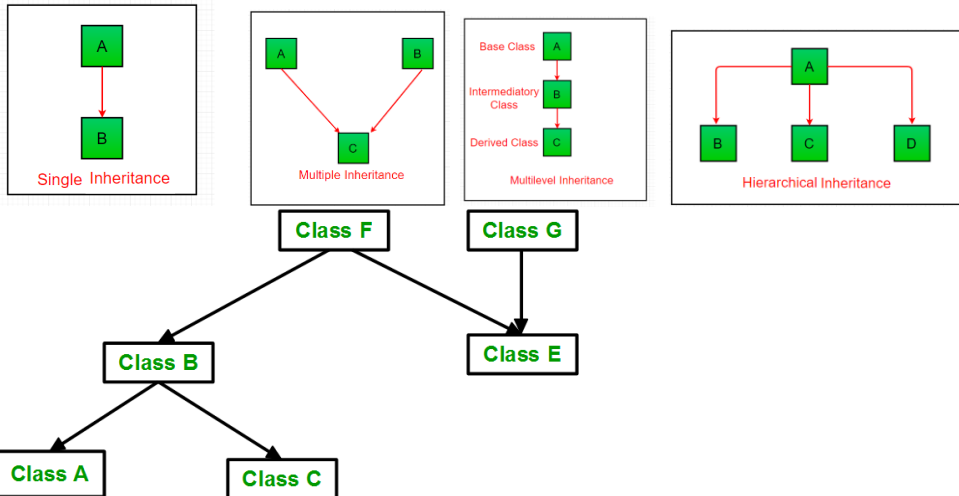
Unlike java, python shows multiple inheritances.

Python example to show the working of multiple inheritance

```python
class Base1(object):

        def __init__(self):

                self.str1 = "BSc1"

                print("Base1")

class Base2(object):

        def __init__(self):

                self.str2 = "BSc2"

                print("Base2")

class Derived(Base1, Base2):

        def __init__(self):
```

```
                # Calling constructors of Base1

                # and Base2 classes

                Base1.__init__(self)

                Base2.__init__(self)

                print("Derived")



        def printStrs(self):

                print(self.str1, self.str2)

ob = Derived()

ob.printStrs()
```

Output:
```
Base1

Base2

Derived

Bsc1 BSc2
```

Example:
```python
# A Python program to demonstrate multilevel  inheritance
class Base(object):
        def __init__(self, name):
                self.name = name
        # To get name
        def getName(self):
                return self.name


# Inherited or Sub class (Note Person in bracket)
class Child(Base):
        def __init__(self, name, age):
                Base.__init__(self, name)
                self.age = age
        def getAge(self):
                return self.age
class GrandChild(Child):
        def __init__(self, name, age, address):
                Child.__init__(self, name, age)
                self.address = address
        def getAddress(self):
                return self.address
# Driver code
```

```
g = GrandChild("BSc1", 23, "Vaniyambadi")
print(g.getName(), g.getAge(), g.getAddress())
```

Output:
```
BSc1 23 Vaniyambadi
```

Example:
```
# Python program to demonstrate Hierarchical inheritance

class Parent:
        def func1(self):
                print("This function is in parent class.")
# Derived class1
class Child1(Parent):
        def func2(self):
                print("This function is in child 1.")
# Derivied class2
class Child2(Parent):
        def func3(self):
                print("This function is in child 2.")
# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output:
```
This function is in parent class.

This function is in child 1.

This function is in parent class.

This function is in child 2.
```

Example:
```
# Python program to demonstrate  hybrid inheritance
class School:
        def func1(self):
                print("This function is in school.")
class Student1(School):
        def func2(self):
                print("This function is in student 1. ")
class Student2(School):
        def func3(self):
                print("This function is in student 2.")
class Student3(Student1, School):
        def func4(self):
                print("This function is in student 3.")
# Driver's code
object = Student3()
```

```
object.func1()
object.func2()
```

<div align="center">Private members of the parent class</div>

We can make some of the instance variables of the parent class private, which won't be available to the child class. We can make an instance variable private by adding double underscores before its name. For example,

```
# Python program to demonstrate private members of the parent class
class C(object):
        def __init__(self):
                self.c = 21
                # d is private instance variable
                self.__d = 42
class D(C):
        def __init__(self):
                self.e = 84
                C.__init__(self)
object1 = D()
# produces an error as d is private instance variable
print(object1.d)
```
Output :
```
  File "/home/993bb61c3e76cda5bb67bd9ea05956a1.py", line 16, in

    print (object1.d)

AttributeError: type object 'D' has no attribute 'd'
```

Protected members

Protected members are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses. To accomplish this in Python, just follow **the convention** by prefixing the name of the member by a **single underscore "_"**.

```
# Python program to  demonstrate protected members
class Base:
        def __init__(self):


                # Protected member
                self._a = 2
class Derived(Base):
        def __init__(self):
                # Calling constructor of  Base class
                Base.__init__(self)
                print("Calling protected member of base class: ",self._a)
                # Modify the protected variable:
                self._a = 3
                print("Calling modified protected member outside class: ",self._a)
obj1 = Derived()
obj2 = Base()
# Calling protected member
# Can be accessed but should not be done due to convention
print("Accessing protected member of obj1: ", obj1._a)

# Accessing the protected variable outside
print("Accessing protected member of obj2: ", obj2._a)
```

Polymorphism in Python

Polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

Example of inbuilt polymorphic functions:
```
# Python program to demonstrate in-built polymorphic functions
# len() being used for a string
print(len("Alpha"))

# len() being used for a list
print(len([10, 20, 30]))
```
Output:
```
5

3
```

Example
```
# A simple Python function to demonstrate Polymorphism
def add(x, y, z = 0):
        return x + y+z
# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```
Output:
```
5

9
```

Polymorphism with class methods:
The below code shows how Python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

 Example:
```
class India():
        def capital(self):
                print("New Delhi is the capital of India.")
        def language(self):
                print("Hindi is the most widely spoken language of India.")
        def type(self):
                print("India is a developing country.")

class USA():
        def capital(self):
                print("Washington, D.C. is the capital of USA.")
        def language(self):
                print("English is the primary language of USA.")
        def type(self):
                print("USA is a developed country.")
obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
        country.capital()
```

```
        country.language()
        country.type()
```
Output:
New Delhi is the capital of India.

Hindi is the most widely spoken language of India.

India is a developing country.

Washington, D.C. is the capital of USA.

English is the primary language of USA.

USA is a developed country.

Polymorphism with Inheritance:
In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This process of re-implementing a method in the child class is known as **Method Overriding**.
Example:
```
class Bird:
def intro(self):
        print("There are many types of birds.")

def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):
def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```
Output:
There are many types of birds.

Most of the birds can fly but some cannot.

There are many types of birds.

Sparrows can fly.

There are many types of birds.

Ostriches cannot fly.

Polymorphism with a Function and objects:
It is also possible to create a function that can take any object, allowing for polymorphism.

```python
class India():
        def capital(self):
                print("New Delhi is the capital of India.")
        def language(self):
                print("Hindi is the most widely spoken language of India.")
        def type(self):
                print("India is a developing country.")
class USA():
        def capital(self):
                print("Washington, D.C. is the capital of USA.")
        def language(self):
                print("English is the primary language of USA.")
        def type(self):
                print("USA is a developed country.")
def func(obj):
        obj.capital()
        obj.language()
        obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

Output:
New Delhi is the capital of India.

Hindi is the most widely spoken language of India.

India is a developing country.

Washington, D.C. is the capital of USA.

English is the primary language of USA.

USA is a developed country.

Class or Static Variables in Python

All objects share class or static variables.

Features of Static Variables

- Static variables are allocated memory ones when the object for the class is created for the first time.
- Static variables are created outside of methods but inside a class
- Static variables can be accessed through a class but not directly with a instance.
- Static variables behavior doesn't change for every object.

The Python approach is simple; it doesn't require a static keyword.

Note: All variables which are assigned a value in the *class declaration are class variables. And variables* that *are assigned values inside methods are instance variables.*

Example:

```python
# Python program to show that the variables with a value

# assigned in class declaration, are class variables


# Class for Computer Science Student
class CSStudent:
        stream = 'cse'                          # Class Variable
        def __init__(self,name,roll):
                self.name = name                # Instance Variable
                self.roll = roll            # Instance Variable


# Objects of CSStudent class
a = CSStudent('Geek', 1)
b = CSStudent('Nerd', 2)

print(a.stream) # prints "cse"
print(b.stream) # prints "cse"
print(a.name) # prints "Geek"
print(b.name) # prints "Nerd"
print(a.roll) # prints "1"
print(b.roll) # prints "2"


# Class variables can be accessed using class
# name also
print(CSStudent.stream) # prints "cse"


# Now if we change the stream for just a it won't be changed for b
```

a.stream = 'ece'

print(a.stream) # prints 'ece'

print(b.stream) # prints 'cse'

To change the stream for all instances of the class we can change it

directly from the class

CSStudent.stream = 'mech'

print(a.stream) # prints 'ece'

print(b.stream) # prints 'mech'

Output:
```
cse
cse
Geek
Nerd
1
2
cse
ece
cse
ece
mech
```


Class method vs Static method in Python

What is Class Method in Python?

The @classmethod decorator is a built-in [function decorator](#) that is an expression that gets evaluated after your function is defined. The result of that evaluation shadows your function definition. A [class method](#) receives the class as an implicit first argument, just like an instance method receives the instance

Syntax Python Class Method:
```
class C(object):
    @classmethod
    def fun(cls, arg1, arg2, ...):
        ....
```
`fun`: function that needs to be converted into a class method

`returns`: a class method for function.

• A class method is a method that is bound to the class and not the object of the class.

• They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.

• It can modify a class state that would apply across all the instances of the class. For example, it can modify a class variable that will be applicable to all the instances.

What is the Static Method in Python?

A [static method](#) does not receive an implicit first argument. A static method is also a method that is bound to the class and not the object of the class. This method can't access or modify the class state. It is present in a class because it makes sense for the method to be present in class.

Syntax Python Static Method:
```
class C(object):
    @staticmethod
    def fun(arg1, arg2, ...):
        ...
returns: a static method for function fun.
```

Class method vs Static Method

The difference between the [Class method and the static method](#) is:
- A class method takes cls as the first parameter while a static method needs no specific parameters.
- A class method can access or modify the class state while a static method can't access or modify it.
- In general, static methods know nothing about the class state. They are utility-type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as a parameter.
- We use @classmethod decorator in python to create a class method and we use @staticmethod decorator to create a static method in python.

When to use the class or static method?

- We generally use the class method to create factory methods. Factory methods return class objects (similar to a constructor) for different use cases.
- We generally use static methods to create utility functions.

How to define a class method and a static method?

To define a class method in python, we use @classmethod decorator, and to define a static method we use @staticmethod decorator.

Let us look at an example to understand the difference between both of them. Let us say we want to create a class Person. Now, python doesn't support method overloading like C++ or Java so we use class methods to create factory methods. In the below example we use a class method to create a person object from birth year.

As explained above we use static methods to create utility functions. In the below example we use a static method to check if a person is an adult or not.

Example:

Python program to demonstrate

use of class method and static method.

from datetime import date

class Person:

        def __init__(self, name, age):

                self.name = name

                self.age = age

```python
        # a class method to create a Person object by birth year.

        @classmethod

        def fromBirthYear(cls, name, year):

                return cls(name, date.today().year - year)



        # a static method to check if a Person is adult or not.

        @staticmethod

        def isAdult(age):

                return age > 18



person1 = Person('mayank', 21)

person2 = Person.fromBirthYear('mayank', 1996)


print(person1.age)

print(person2.age)


# print the result

print(Person.isAdult(22))
```

Output:
```
21
25
True
```

Python Exception Handling

Try and Except Statement – Catching Exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

 # Python program to handle simple runtime error

```
a = [1, 2, 3]
try:
        print ("Second element = %d" %(a[1]))


        # Throws error since there are only 3 elements in array
        print ("Fourth element = %d" %(a[3]))


except:
        print ("An error occurred")
```

Output
```
Second element = 2

An error occurred
```

Catching Specific Exception

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed. For example, we can add IndexError in the above code. The general syntax for adding specific exceptions are –

```
try:
    # statement(s)
except IndexError:
    # statement(s)
except ValueError:
    # statement(s)
```

 # Program to handle multiple errors with one except statement

```
def fun(a):
        if a < 4:

                # throws ZeroDivisionError for a = 3
                b = a/(a-3)
```

```
        # throws NameError if a >= 4

        print("Value of b = ", b)


try:

        fun(3)

        fun(5)

except ZeroDivisionError:

        print("ZeroDivisionError Occurred and Handled")

except NameError:

        print("NameError Occurred and Handled")
```

Output
```
ZeroDivisionError Occurred and Handled

NameError Occurred and Handled
```


Try with Else Clause

In python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

Example: Try with else clause

```
# Program to depict else clause with try-except

def AbyB(a , b):

        try:

                c = ((a+b) / (a-b))

        except ZeroDivisionError:

                print ("a/b result in 0")

        else:

                print (c)


# Driver program to test above function

AbyB(2.0, 3.0)

AbyB(3.0, 3.0)
```

Output:
```
-5.0

a/b result in 0
```

Finally Keyword in Python

Python provides a keyword finally, which is always executed after the try and except blocks. The final block always executes after normal termination of try block or after try block terminates due to some exception.

Syntax:

```
try:
    # Some Code....

except:
    # optional block
    # Handling of exception (if required)

else:
    # execute if no exception

finally:
    # Some code .....(always executed)
```

Example:

 # Python program to demonstrate finally

 # No exception Exception raised in try block

try:

    k = 5//0 # raises divide by zero exception.

    print(k)

 # handles zerodivision exception

except ZeroDivisionError:

    print("Can't divide by zero")

finally:

    # this block is always executed

    # regardless of exception generation.

    print('This is always executed')

Output:

```
Can't divide by zero

This is always executed
```

Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

Some of the common Exception Errors are :

- **IOError:** if the file can't be opened
- **KeyboardInterrupt:** when an unrequired key is pressed by the user
- **ValueError:** when built-in function receives a wrong argument
- **EOFError:** if End-Of-File is hit without reading any data
- **ImportError:** if it is unable to find the module

Try and Except statement is used to handle these errors within our code in Python. The try block is used to check some code for errors i.e the code inside the try block will execute when there is no error in the program. Whereas the code inside the except block will execute whenever the program encounters some error in the preceding try block.

Syntax:

```
try:
    # Some Code
except:
    # Executed if error in the
    # try block
```

Ex 1:

```
# Python code to illustrate
# working of try()
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional Part as Answer
        result = x // y
        print("Yeah ! Your answer is :", result)
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")


# Look at parameters and note the working of Program
divide(3, 2)
```

Output :

```
('Yeah ! Your answer is :', 1)
```

Ex 2:

```
# Python code to illustrate  working of try() with except
def divide(x, y):
    try:
```

```
            # Floor Division : Gives only Fractional Part as Answer
            result = x // y
            print("Yeah ! Your answer is :", result)
        except ZeroDivisionError:
            print("Sorry ! You are dividing by zero ")
```

# Look at parameters and note the working of Program
divide(3, 0)
Output :

```
Sorry ! You are dividing by zero
```

Else Clause

In python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

Syntax:
```
try:
    # Some Code
except:
    # Executed if error in the
    # try block
else:
    # execute if no exception
```
Ex 3:
Program to depict else clause with try-except

# Function which returns a/b
def AbyB(a , b):
    try:
        c = ((a+b) // (a-b))
    except ZeroDivisionError:
        print ("a/b result in 0")
    else:
        print (c)

Driver program to test above function

AbyB(2.0, 3.0)

AbyB(3.0, 3.0)

Output:
```
-5.0
a/b result in 0
```

Finally Keyword in Python

Python provides a keyword finally, which is always executed after the try and except blocks. The final block always executes after normal termination of try block or after try block terminates due to some exceptions.

Syntax:
```
try:
    # Some Code
except:
    # Executed if error in the
    # try block
else:
    # execute if no exception
finally:
    # Some code .....(always executed)
```

Ex 4:

 # Python program to demonstrate finally

 # No exception Exception raised in try block

try:

    k = 5//0 # raises divide by zero exception.

    print(k)

 # handles zerodivision exception

except ZeroDivisionError:

    print("Can't divide by zero")

finally:

    # this block is always executed

    # regardless of exception generation.

    print('This is always executed')

Output:
```
Can't divide by zero

This is always executed
```

User-Defined Exception in Python

Exceptions need to be derived from the Exception class, either directly or indirectly. Although not mandatory, most of the exceptions are named as names that end in **"Error"** similar to the naming of the standard exceptions in python. **For example,**

 # A python program to create user-defined exception

 # class MyError is derived from super class Exception

class MyError(Exception):

    # Constructor or Initializer

    def __init__(self, value):

        self.value = value

    # __str__ is to print() the value

    def __str__(self):

        return(repr(self.value))

try:

    raise(MyError(3*2))

 # Value of Exception is stored in error

except MyError as error:

    print('A New Exception occurred: ', error.value)

Output
```
('A New Exception occurred: ', 6)
```