

**PYTHON PROGRAMMING**  
**(R18A0513)**  
**LECTURE NOTES**  
**B.TECH III YEAR – I SEM**  
**(R18) (2021-2022)**



**MALLA REDDY COLLEGE OF ENGINEERING &  
TECHNOLOGY**

**(Autonomous Institution – UGC, Govt. of India)**

Recognized under 2(f) and 12 (B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)  
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India



# SYLLABUS

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

III Year B. Tech CSE -I SEM

L	T/P/D	C
3	-/-	3

(R18A0513) PYTHON PROGRAMMING

## OBJECTIVES:

- To read and write simple Python programs.
- To develop Python programs with conditionals and loops.
- To define Python functions and call them.
- To use Python data structures — lists, tuples, dictionaries.
- To do input/output with files in Python.

## UNIT I

### INTRODUCTION DATA, EXPRESSIONS, STATEMENTS

Introduction to Python and installation, variables, expressions, statements, Numeric datatypes: Int, float, Boolean, string. Basic data types: list--- list operations, list slices, list methods,list loop, mutability, aliasing, cloning lists, list parameters. Tuple --- tuple assignment, tuple asreturn value, tuple methods. Dictionaries: operations and methods.

## UNIT II

### CONTROL FLOW, LOOPS

**Conditionals:** Boolean values and operators, conditional (if), alternative (if-else), chainedconditional (if-elif-else); Iteration: statements break, continue.

**Functions---** function and its use, pass keyword, flow of execution, parameters and arguments.

## UNIT III

### ADVANCED FUNCTIONS, ARRAYS

**Fruitful functions:** return values, parameters, local and global scope, function composition,recursion; Advanced Functions: lambda, map, filter, reduce, basic data type comprehensions.

**Python arrays:** create an array, Access the Elements of an Array, array methods.

## UNIT IV

### FILES, EXEPTIONS

File I/O, Exception Handling, introduction to basic standard libraries, Installation of pip,Demonstrate Modules: Turtle, pandas, numpy, pdb, Explore packages.

## UNIT V

### OOPS , FRAMEWORK

**Oops concepts:** Object, Class, Method, Inheritance, Polymorphism, Data abstraction,Encapsulation,

**Python Frameworks:** Explore django framework with an example

**OUTCOMES:** Upon completion of the course, students will be able to

- Read, write, execute by hand simple Python programs.
- Structure simple Python programs for solving problems.

- Decompose a Python program into functions.
- Represent compound data using Python lists, tuples, dictionaries.
- Read and write data from/to files in Python Programs

**TEXT BOOKS**

1. Allen B. Downey, ``Think Python: How to Think Like a Computer Scientist``, 2nd edition, Updated for Python 3, Shroff/O'Reilly Publishers, 2016.
2. R. Nageswara Rao, "Core Python Programming", dreamtech
3. Python Programming: A Modern Approach, Vamsi Kurama, Pearson

**REFERENCE BOOKS:**

1. Core Python Programming, W.Chun, Pearson.
2. Introduction to Python, Kenneth A. Lambert, Cengage
3. Learning Python, Mark Lutz, Orielly

# INDEX

UNIT	TOPIC	PAGE NO
I	<b>INTRODUCTION DATA, EXPRESSIONS, STATEMENTS</b>	1
	<b>Introduction to Python and installation</b>	1-6
	Variables	6-8
	Expressions	8-11
	Statements	11-12
	Numeric data types: Int, float, Boolean, string.	12-24
	Basic data types:	24
	list--- list operations, list slices, list methods, list loop, mutability	24-34
	aliasing, cloning lists, list parameters.	34-36
	Tuple --- tuple assignment, tuple as return value, tuple methods	36-44
	Dictionaries: operations and methods	44-48
II	<b>CONTROL FLOW, LOOPS</b>	49
	<b>Conditionals:</b> Boolean values and operators,	
	conditional (if)	50-51
	alternative (if-else)	51-52
	chained conditional (if-elif-else)	53-54
	Iteration: statements, break, continue.	55-67
	Functions--- function and its use	67-68
	pass keyword	68
	flow of execution	69-70
	parameters and arguments	70-82
III	<b>ADVANCED FUNCTIONS, ARRAYS</b>	83
	<b>Fruitful functions:</b> return values	83-85
	Parameters	85-87
	local and global scope	87-90
	function composition	90-91
	Recursion	91-92
	Advanced Functions: lambda, map, filter, reduce	92-96
	basic data type comprehensions	96-99
	Python arrays: <b>create an array</b>	100-101
	Access the Elements of an Array	102
	Array methods	102-104
	<b>FILES, EXEPTIONS</b>	105
	File I/O	105-110

IV	Exception Handling	111-120
	introduction to basic standard libraries	121-133
	Installation of pip	133-137
	Demonstrate Modules: Turtle, pandas, numpy, pdb	137-146
	Explore packages.	146-149
V	<b>OOPS , FRAMEWORK</b>	150
	Oops concepts: Object, Class, Method,	150-155
	Inheritance, Polymorphism	155-159
	Data abstraction, Encapsulation	159-162
	Python Frameworks: Explore django framework with an example	162-165

**UNIT – I****INTRODUCTION DATA, EXPRESSIONS, STATEMENTS**

Introduction to Python and installation, variables, expressions, statements, Numeric datatypes: Int, float, Boolean, string. Basic data types: list--- list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters. Tuple --- tuple assignment, tuple as return value, tuple methods. Dictionaries: operations and methods.

**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 compatible 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.

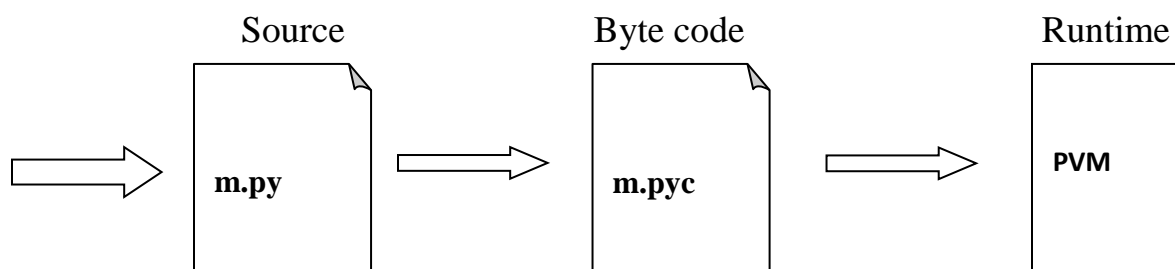
Step 6: Add Python Path to Environment Variables (Optional)



## 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.

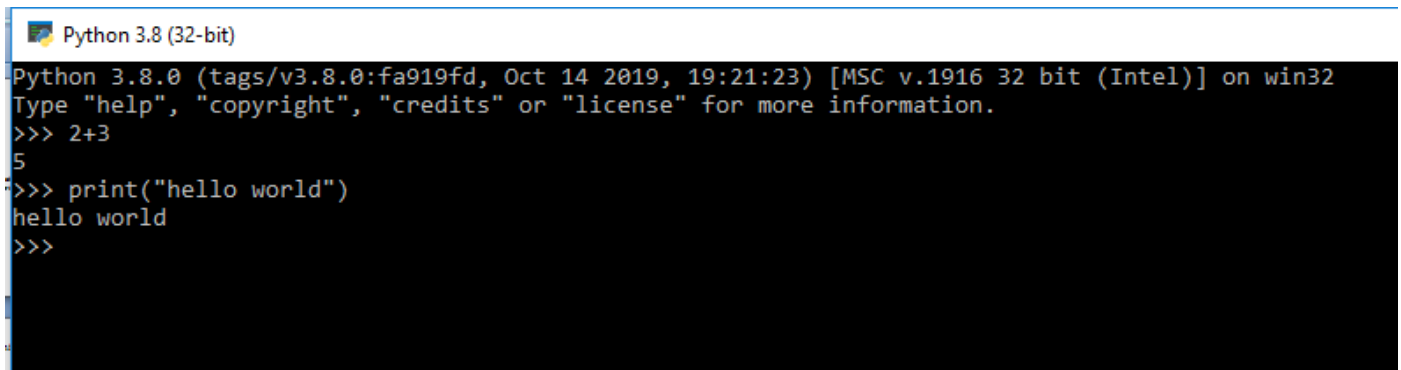
```
>>> x
```

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

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

```
>>> 2+3
```

```
5
```

A screenshot of a Python 3.8 (32-bit) interpreter window. The window title is "Python 3.8 (32-bit)". The command prompt shows the following interactions: the user enters "2+3" and the interpreter outputs "5"; the user enters "print('hello world')" and the interpreter outputs "hello world". The prompt ">>>" is visible at the end of the last line.

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> print("hello world")
hello world
>>>
```

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 divisibile zero division by zero
resource close
finished
```

**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,"mrcet"
```

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 = "mrcet "   # x is now of type str
print(x)
```

**Output:**mrcet

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="mrcet"
```

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

```
>>> print("mrcet colege")
```

```
mrcet college
```

### **Numeric 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'>
```

### **Float:**

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("mrcet college")

mrcet college

>>> type("mrcet college")

<class 'str'>

>>> print('mrcet college')

mrcet college

>>> " "

''
```

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('mrcet')
>>> a
'mrcet'
>>> a=str(mrcet)
>>> 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='mrcet 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.

Syntax:

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 Syntax:

String.count()

#### Example:

```
>>> 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

Syntax:

String.find(„string“)

Example:

```
>>> string="Nikhil Is Learning"
```

```
>>> string.find('k')
```

```
2
```

13.swapcase()

converts lowercase letters in a string to uppercase and viceversa

Syntax:

String.find(„string“)

Example:

```
>>> 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
```

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("mrcet is an autonomous (') college")
```

```
mrcet is an autonomous (') college
```

```
>>> print('mrcet is an autonomous (") college')
```

```
mrcet 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("mrcet is an autonomous (\') college")
```

```
mrcet is an autonomous (') college
```

```
>>> print('mrcet is an autonomous (\") college')
```

```
mrcet 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("mrcet \n college")  
mrcet  
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
```

### **Basic Data types:**

#### **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 = ['mrcet', 'college', 'MRCET!']
```

Python Expression	Results	Description
L[2]	MRCET	Offsets start at zero
L[-2]	college	Negative: count from the right
L[1:]	['college', 'MRCET!']	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/MRCET/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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/listloop.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/MRCET/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/MRCET/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:

```
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/MRCET/AppData/Local/Programs/Python/Python38-32/lipar.py ==
```

```
[2, 5, 9]
[4, 10, 18]
```

**Tuple:**

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 accidentally 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)
```

### 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 = ('mrcet', 'eng college','2004','cse', 'it','csit');`
- `>>> tup2 = (1,2,3,4,5,6,7);`
- `>>> print(tup1[0])`
- `mrcet`
- `>>> print(tup2[1:4])`
- `(2, 3, 4)`

Tuple 1 includes list of information of mrcet

Tuple 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 mrcet for first tuple while for second tuple it gives number (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 = "mrcet 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/MRCET/AppData/Local/Programs/Python/Python38-32/tupretval.py
mrcet 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/MRCET/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 methods:**

**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
```

### Set:

A set is a collection which is unordered and unindexed with no duplicate elements. In Python sets are written with curly brackets.

- To create an empty set we use **set()**
- Curly braces ‘{}’ or the **set()** function can be used to create sets

We can construct tuple in many ways:

```
X=set()
X={3,5,6,8}
X=set(list1)
```

Example:

```
>>> x={1,3,5,6}
>>> x
{1, 3, 5, 6}
```

```
-----
>>> x=set()
>>> x
set()
```

```
-----
>>> list1=[4,6,"dd",7]
>>> x=set(list1)
>>> x
{4, 'dd', 6, 7}
```

- We cannot access items in a set by referring to an index, since sets are unordered the items has no index.
- But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Some of the basic set operations are:

- Add()
- Remove()
- Len()
- Item in x
- Pop
- Clear

**Add ():** To add one item to a set use the add() method. To add more than one item to a set use the update() method.

```
>>> x={"mrcet","college","cse","dept"}
>>> x.add("autonomous")
>>> x
{'mrcet', 'dept', 'autonomous', 'cse', 'college'}
```

```
----
>>> x={1,2,3}
>>> x.update("a","b")
>>> x
{1, 2, 3, 'a', 'b'}
```

```
-----
>>> x={1,2,3}
>>> x.update([4,5],[6,7,8])
>>> x
{1, 2, 3, 4, 5, 6, 7, 8}
```

**Remove ():** To remove an item from the set we use remove or discard methods.

```
>>> x={1, 2, 3, 'a', 'b'}
>>> x.remove(3)
>>> x
{1, 2, 'a', 'b'}
```

**Len ():** To know the number of items present in a set, we use len().

```
>>> z={'mrcet', 'dept', 'autonomous', 'cse', 'college'}
>>> len(z)
```

5

**Item in X:** you can loop through the set items using a for loop.

```
>>> x={'a','b','c','d'}
>>> for item in x:
    print(item)
```

```
c
d
a
b
```

**pop ():** This method is used to remove an item, but this method will remove the **last** item. Remember that sets are unordered, so you will not know what item that gets removed.

```
>>> x={1, 2, 3, 4, 5, 6, 7, 8}
>>> x.pop()
1
>>> x
{2, 3, 4, 5, 6, 7, 8}
```

**Clear ():** This method will the set as empty.

```
>>> x={2, 3, 4, 5, 6, 7, 8}
>>> x.clear()
>>> x
set()
```

The set also consist of some mathematical operations like:

Intersection	AND	&
Union	OR	
Symmetric Diff	XOR	^
Diff	In set1 but not in set2	set1-set2
Subset	set2 contains set1	set1<=set2
Superset	set1 contains set2	set1>=set2

Some examples:

```
>>> x={1,2,3,4}
>>> y={4,5,6,7}
>>> print(x|y)
{1, 2, 3, 4, 5, 6, 7}
```

```
-----
>>> x={1,2,3,4}
```

```
>>> y={4,5,6,7}
>>> print(x&y)
{4}
-----
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> print(A-B)
{1, 2, 3}
-----
>>> B = {4, 5, 6, 7, 8}
>>> A = {1, 2, 3, 4, 5}
>>> print(B^A)
{1, 2, 3, 6, 7, 8}
```

### 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)
```

### Examples:

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1
{'brand': 'mrcet', 'model': 'college', 'year': 2004}
-----
```

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

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

**To access keys and values and items of dictionary:**

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1.keys()
```

```
dict_keys(['brand', 'model', 'year'])
>>> dict1.values()
dict_values(['mrcet', 'college', 2004])
>>> dict1.items()
dict_items([('brand', 'mrcet'), ('model', 'college'), ('year', 2004)])
```

```
-----
>>> for items in dict1.values():
    print(items)
```

```
mrcet
college
2004
```

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

```
brand
model
year
```

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

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

Some of the operations are:

- 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": "mrcet", "model": "college", "year": 2004}
>>> dict1["year"] = 2005
```

```
>>> dict1
{'brand': 'mrcet', 'model': 'college', 'year': 2005}
```

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

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

**UNIT – II****CONTROL FLOW, LOOPS**

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

**Functions---** function and its use, pass keyword, flow of execution, parameters and arguments.

**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.

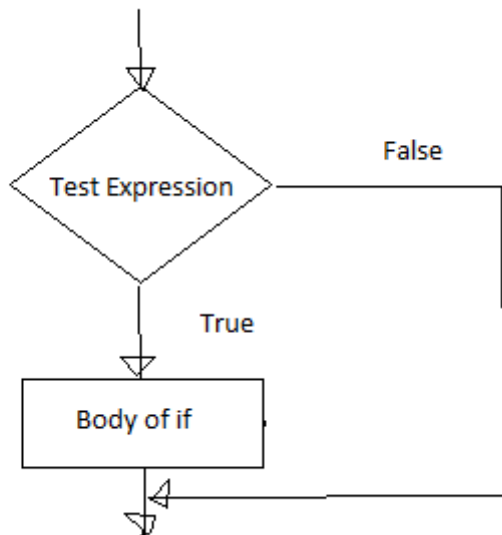
**if Statement Flowchart:**

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/MRCET/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/MRCET/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 elseStatement following if.

**Syntax of if - else :**

```
if test expression:
```

```
    Body of ifstmts
```

```
else:
```

```
    Body of elsestmts
```

**If - else Flowchart :**

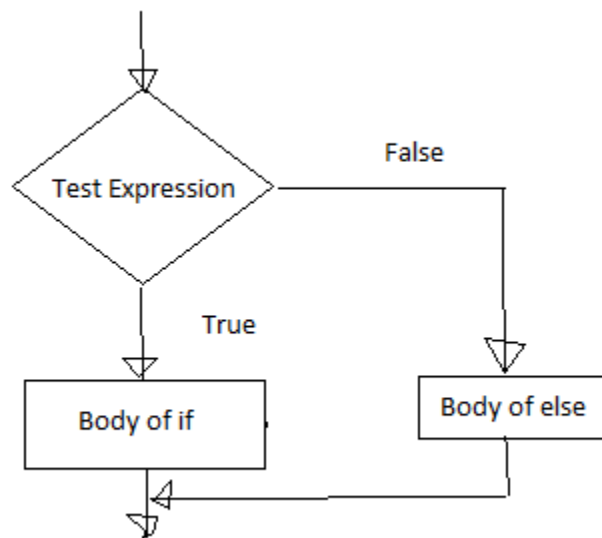


Fig: Operation of if – else statement

### 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/MRCET/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/MRCET/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
```

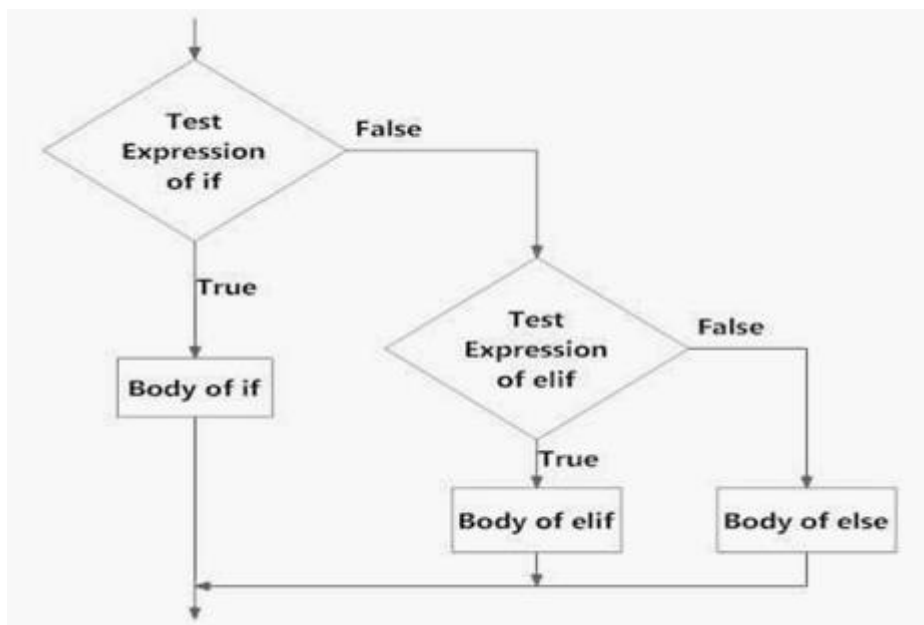
**Flowchart of if – elif - else:**

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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py

enter the number5

enter the number2

enter the number9

a is greater

>>>

C:/Users/MRCET/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/MRCET/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:

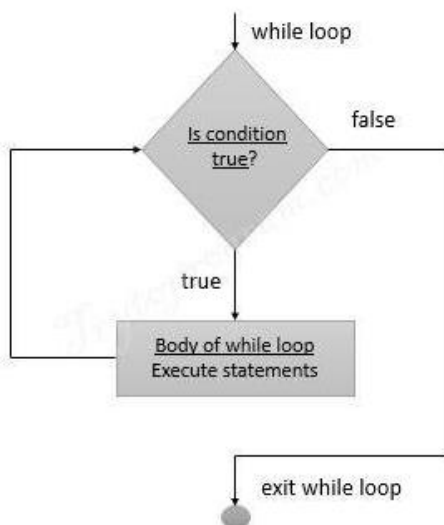
1. While Loop
2. For Loop
3. Nested For Loops

**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("Mrcet college")  
    i=i+1
```

**output:**

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh1.py

Mrcet college

Mrcet college

Mrcet college

Mrcet college

Mrcet college

Mrcet college

```
2. -----  
i=1  
  
while i<=3:  
    print("MRCET",end=" ")  
    j=1  
    while j<=1:  
        print("CSE DEPT",end="")  
        j=j+1  
    i=i+1  
    print()
```

**Output:**

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh2.py

MRCET CSE DEPT

MRCET CSE DEPT

MRCET CSE DEPT

```
3. -----  
  
i=1
```

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

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

**Output:**

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

```
MRCET CSE DEPT
MRCET
MRCET
```

4. -----

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

**Output:**

C:/Users/MRCET/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/MRCET/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/MRCET/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)

Holds the value of item  
in sequence in each iteration

↓ A sequence of values assigned to var in each iteration

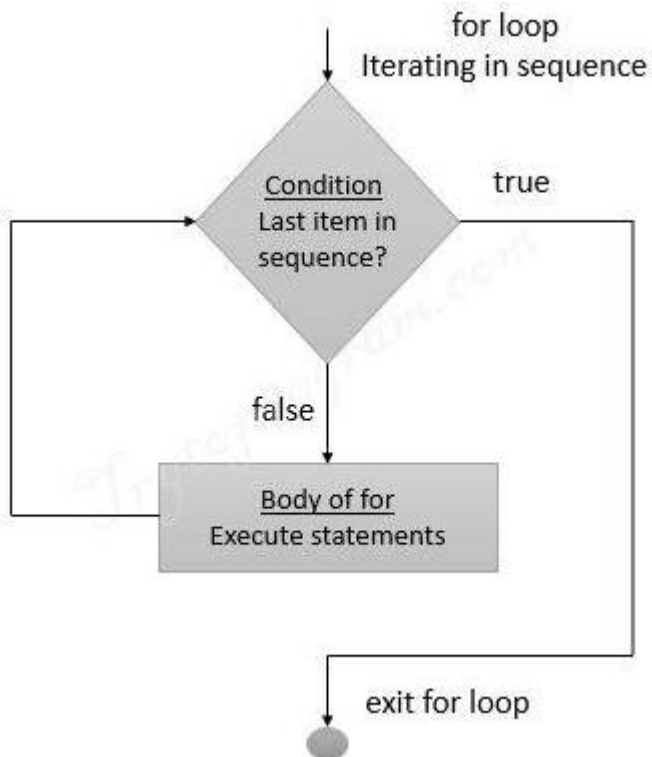
**Sample Program:**

```
numbers = [1, 2, 4, 6, 11, 20]
seq=0
for val in numbers:
    seq=val*val
    print(seq)
```

**Output:**

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/fr.py

```
1
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/MRCET/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/MRCET/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/MRCET/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 = 'MRCET'
```

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

Output:

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

#### Output:

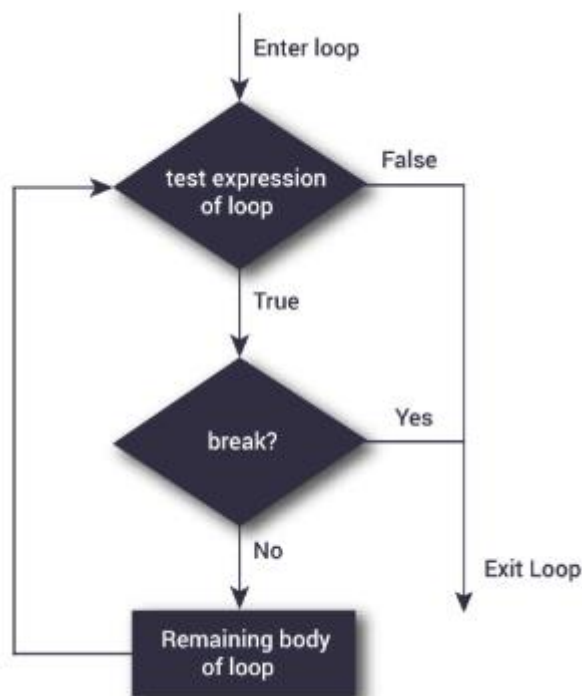
```
1 1 1 1 1  
2 2 2 2  
3 3 3  
4 4
```

**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 "MRCET COLLEGE":
```

```
    if val == " ":
```

```
        break
```

```
    print(val)
```

```
print("The end")
```

**Output:**

M

R

C

E

T

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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/br.py =

Current Letter : P

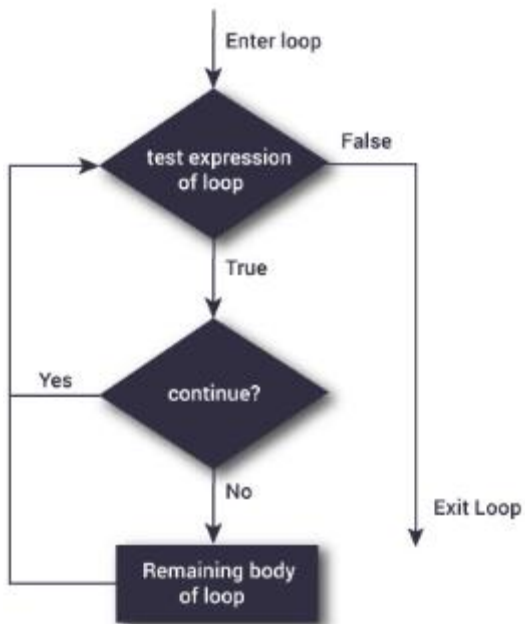
Current Letter : y

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/MRCET/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]:
```

```
# 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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/cont2.py

11

9

89

#-----

for letter in "Python": # First Example

if letter == "h":

continue

print("Current Letter :", letter)

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/con1.py

Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : o

Current Letter : n

**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. **Built-in functions** - 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. User-defined functions** - 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

**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:**

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

**Output:**

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

**Similarly we can also write,**

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

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

**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**

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

**Output:**

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

welcome

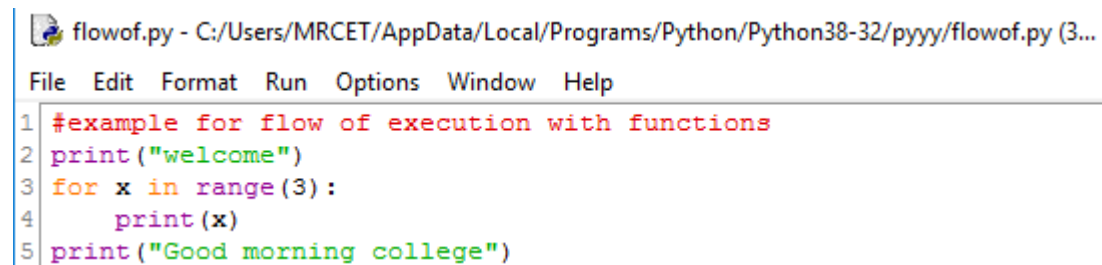
0

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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py

hi

hello

Good morning

mrcet

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/MRCET/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
```

```
def hf():
```

```
    print("hw")
```

```
    print("gh kfjg 66666")
```

```
hf()
```

```
hf()
```

```
hf()
```

**Output:**

```
hw
```

```
gh kfjg 66666
```

```
hw
```

```
gh kfjg 66666
```

```
hw
```

```
gh kfjg 66666
```

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

```
    c=x+y
```

```
    print(c)
```

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

**Output:**

```
9
```

```
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/MRCET/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("mrcet"))
```

**Output:**

```
mrcet
```

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("MRCET","Good morning!")
```

**Output:**

Hello MRCET Good morning!

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

```
>>>wish("MRCET") # 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("mrcet","college"))
```

**Output:**

himrcet,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 tuple 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 dictionary 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("MRCET","CSE","SIR","MADAM")
```

**Output:**

```
Hello MRCET
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/MRCET/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 = calculate(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/MRCET/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/MRCET/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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

Enter a value 5  
Enter b value -6  
Enter c value 7  
big number= 7

**#Write 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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py  
Enter one subject marks 35  
fail
```

**#Write a program to display mrcet cse dept 10 times on the screen.(while loop)**

```
def usingFunctions():  
    count =0  
    while count<10:  
        print("mrcet cse dept",count)  
        count=count+1  
  
usingFunctions()
```

**Output:**

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py  
mrcet cse dept 0  
mrcet cse dept 1  
mrcet cse dept 2  
mrcet cse dept 3  
mrcet cse dept 4  
mrcet cse dept 5  
mrcet cse dept 6  
mrcet cse dept 7  
mrcet cse dept 8  
mrcet cse dept 9
```

**UNIT – III****FUNCTIONS, ARRAYS**

**Fruitful functions:** return values, parameters, local and global scope, function composition, recursion; Advanced Functions: lambda, map, filter, reduce, basic data type comprehensions.

**Python arrays:** Create an array, 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."

(or)

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))
```

(or)

```
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):
```

```
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], [1,7,8,9,10]))
print(common_data([1,2,3,4,5], [6,7,8,9,10]))
```

**Output:**

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy\ful.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/MRCET/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 definition
```

```
def display():
```

```
    print("vandemataram")
```

```
print("i am in main")
```

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

**Output:**

```
C:/Users/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py", line 6, 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/MRCET/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/MRCET/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)
```

```
<zip object at 0x03DAC6C8>
```

```
>>> 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.

```
# 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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/rec.py
```

```
zero factorial 1
```

```
five factorial 120
```

```
-----  
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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/rec.py  
The factorial of 4 is 24

**Advanced Functions:**

**Anonymous function** is a function i.e. defined without name.

While normal functions are defined using the **def keyword**.

**Anonymous functions** are defined using **lambda keyword** hence anonymous functions are also called **lambda functions**.

**Syntax:** lambda arguments: expression

- Lambda function can have any no. of arguments for any one expression.
- The expression is evaluated and returns.

**Use of Lambda functions:**

- Lambda functions are used as nameless functions for a short period of time.
- In python lambda functions are an argument to higher order functions.
- Lambda functions are used along with built-in functions like filter(),map() and reduce()etc....

**# Write a program to double a given number**

```
double = lambda x:2*x
print(double(5))
```

**Output:**

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py  
10

**#Write a program to sum of two numbers**

```
add = lambda x,y:x+y
```

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

**Output:**

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
```

```
9
```

**#Write a program to find biggest of two numbers**

```
biggest = lambda x,y: a if x>y else y
```

```
print(biggest(20,30))
```

**Output:**

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
```

```
30
```

**Powerful Lamda function in python:**

Lambda functions are used along with built-in functions like filter(),map() and reduce()etc....

**Filter():**

- The filter functions takes list as argument.
- The filter() is called when new list is returned which contains items for which the function evaluates to true.
- Filter:The filter() function returns an iterator were the items are filteredthrough a function to test if the item is accepted or not.

**Syntax:** filter(function, iterable)

**#Write a program to filter() function to filter out only even numbers from the given list**

```
myList =[1,2,3,4,5,6]
```

```
newList = list(filter(lambda x: x%2 ==0,myList ))
```

```
print(newList)
```

**Output:**

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

[2, 4, 6]

**#Write a program for filter() function to print the items greater than 4**

```
list1 = [10,2,8,7,5,4,3,11,0, 1]
```

```
result = filter (lambda x: x > 4, list1)
```

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

**Output:**

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ml.py =

[10, 8, 7, 5, 11]

**Map() :**

- Map() function in python takes a function & list.
- The function is called with all items in the list and a new list is returned which contains items returned by that function for each item.
- Map applies a function to all the items in an list.
- The advantage of the lambda operator can be seen when it is used in combination with the map() function.
- map() is a function with two arguments:

**Syntax:** r = map(func, seq)

**#Write a program for map() function to double all the items in the list**

```
myList =[1,2,3,4,5,6,7,8,9,10]
```

```
newList = list(map(lambda x: x*2,myList))
```

```
print(newList)
```

**Output:**

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

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

**# Write a program to separate the letters of the word "hello" and add the letters as items of the list.**

```
letters = []  
letters = list(map(lambda x:x,"hello"))  
print(letters)
```

**Output:**

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy\fu1.py
```

```
['h', 'e', 'l', 'l', 'o']
```

**#Write a program for map() function to double all the items in the list?**

```
def addition(n):
```

```
    return n + n
```

```
numbers = (1, 2, 3, 4)
```

```
result = map(addition, numbers)
```

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

**Output:**

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/m1.py =
```

```
[2, 4, 6, 8]
```

**Reduce():**

- Applies the same operation to items of sequence.
- Use the result of the first operation for the next operation
- Returns an item, not a list.
- Reduce: The `reduce(fun,seq)` function is used to apply a particular
- function passed in its argument to all of the list elements mentioned in the sequence passed along. This function is defined in “functools” module.



**#Write a program to find some of the numbers for the elements of the list by using reduce()**

```
import functools
myList=[1,2,3,4,5,6,7,8,9,10]
print(functools.reduce(lambda x,y: x+y,myList))
```

**Output:**

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy\fu1.py
```

```
55
```

**#Write a program for reduce() function to print the product of items in a list**

```
from functools import reduce

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

product = reduce (lambda x, y: x*y, list1)

print(product)
```

**Output:**

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

```
120
```

### **Basic data type comprehensions:**

#### **List comprehension:**

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 readable.

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

```
>>> list1  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

**Similarly some examples:**

```
>>> 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]
```

### **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)]
```

### **Setcomprehension:**

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}
```

### **Dictionary 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}
```

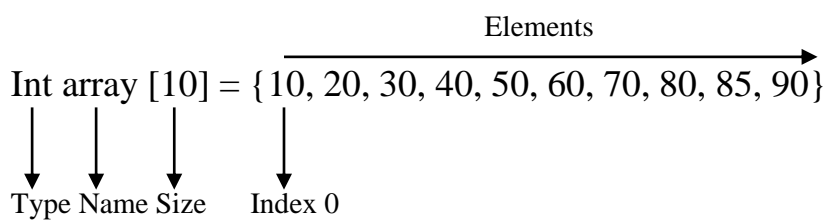
## Python arrays:

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.



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
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/MRCET/AppData/Local/Programs/Python/Python38-32/arr.py

```
10  
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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/arr2.py  
10  
30
```

### Array methods:

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

Method	Description
<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position

<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the first item with the specified value
<u>reverse()</u>	Reverses the order of the list
<u>sort()</u>	Sorts the list

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

**Example:**

```
>>> college=["mrcet","it","cse"]
>>> college.append("autonomous")
>>> college
['mrcet', 'it', 'cse', 'autonomous']
>>> college.append("eee")
>>> college.append("ece")
>>> college
['mrcet', 'it', 'cse', 'autonomous', 'eee', 'ece']
>>> college.pop()
'ece'
>>> college
['mrcet', 'it', 'cse', 'autonomous', 'eee']
>>> college.pop(4)
'eee'
>>> college
```



```
['mrcet', 'it', 'cse', 'autonomous']
```

```
>>> college.remove("it")
```

```
>>> college
```

```
['mrcet', 'cse', 'autonomous']
```

**UNIT –IV****FILES, EXEPTIONS**

File I/O, Exception Handling, introduction to basic standard libraries, Installation of pip, Demonstrate Modules: Turtle, pandas, numpy, pdb, Explore packages.

**File I/O:**

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.

**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.

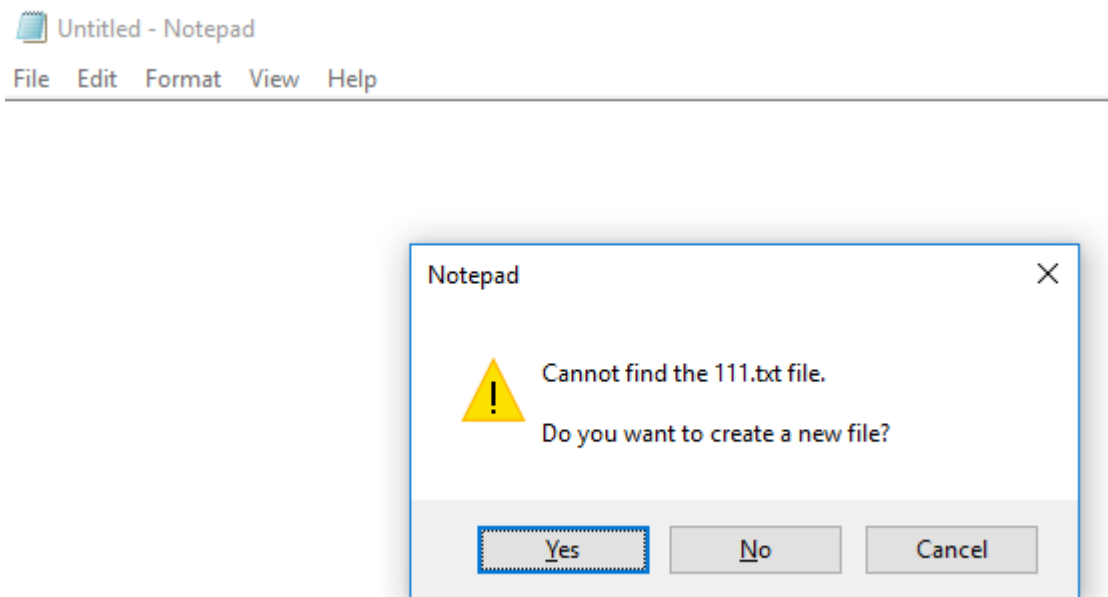
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>start notepad hello.txt

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?



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()
```

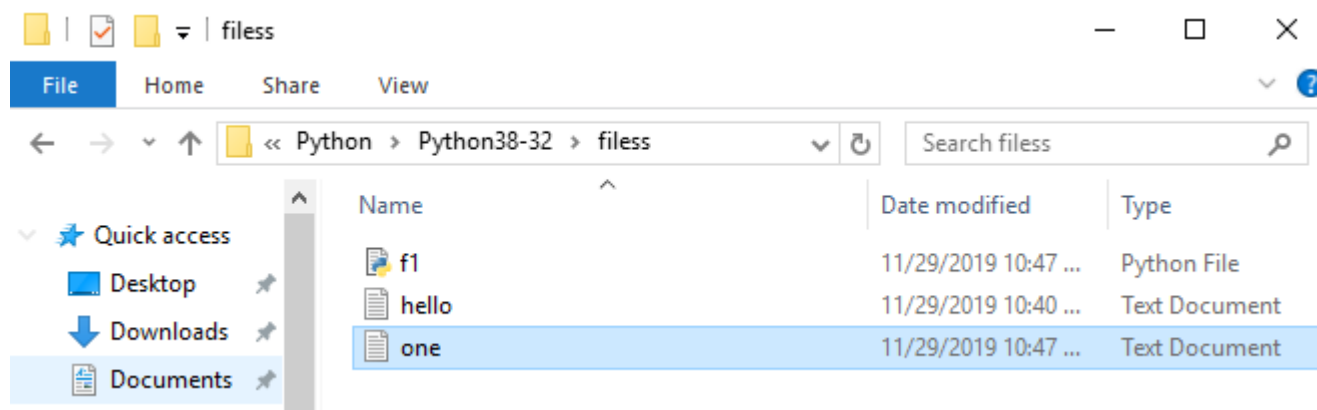
**Output:**

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/files/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 be 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

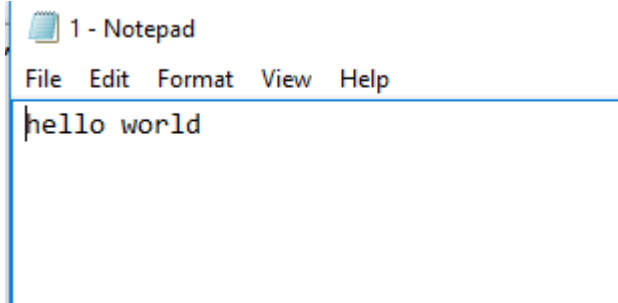
**# Write a python program to open and write “hello world” into a file?**

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

```
f.write("hello world")
```

```
f.close()
```

**Output:**



(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
```

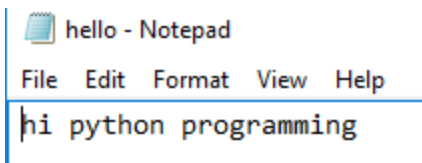
**# Write a python program to write the content “hi python programming” for the existing file.**

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

```
f.write("hi python programming")
```

```
f.close()
```

**Output:**



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

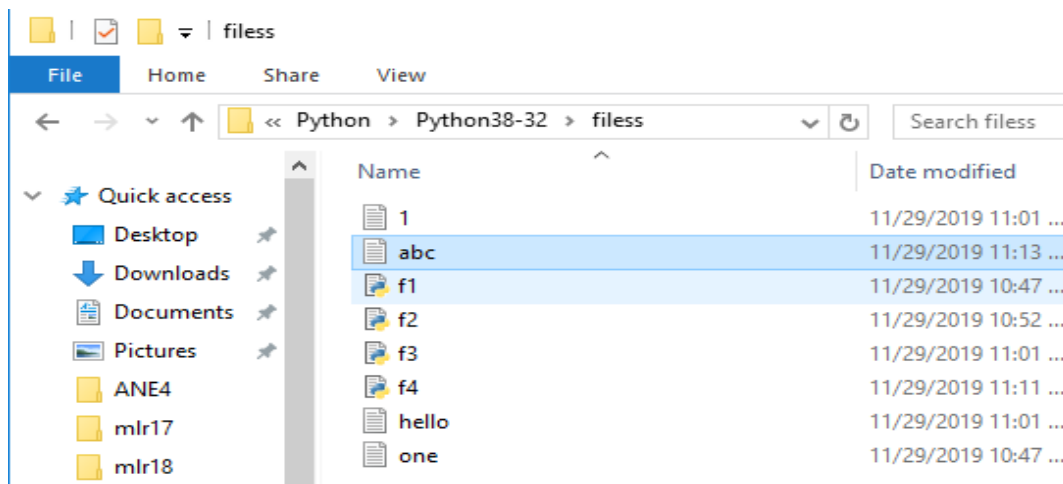
File Edit Format View Help

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**



## Exception Handling:

### Errors and Exceptions:

**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.

**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, `TypeError` exception 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 indented block" section, Python not only insists on indentation, it insists on consistent indentation. 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:**

- `ArrayIndexOutOfBoundsException`.
- `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/MRCET/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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex2.py", line 3, 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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py", line 7, 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/MRCET/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/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex12.py  
resource open  
2.5  
enter a number p  
invalid input  
resource closed
```

**Introduction to basic standard libraries:**

- 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.

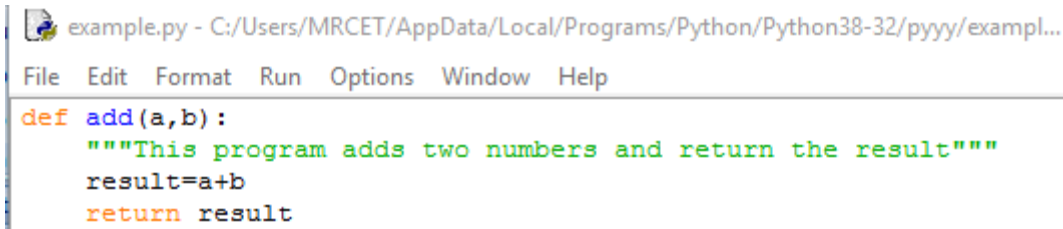
A file containing Python code, for e.g.: example.py, is called a module and its module name would be example.

```
>>> 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/MRCET/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/MRCET/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'
```

**Modules (Datetime, Time, os, calendar, math):**

**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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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/MRCET/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:**

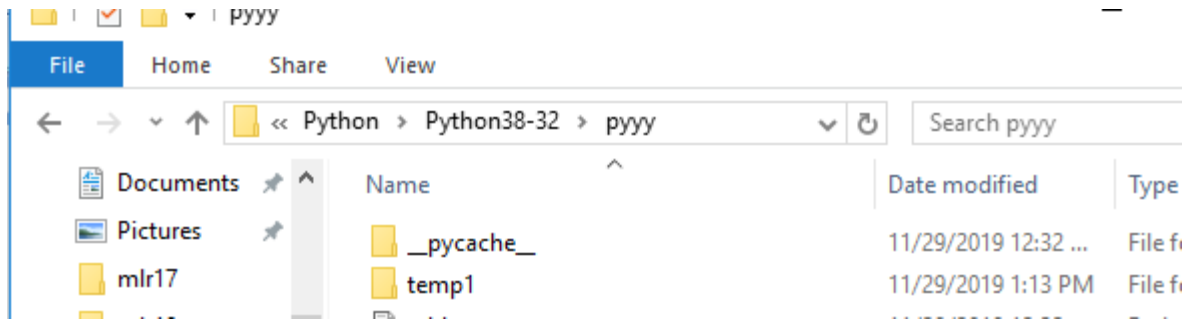
```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =  
Python Lab (#prints after 6 seconds)
```

**os module:**

```
>>> import os  
  
>>> os.name  
  
'nt'  
  
>>> os.getcwd()  
  
'C:\\Users\\MRCET\\AppData\\Local\\Programs\\Python\\Python38-32\\pyyy'
```



```
>>> os.mkdir("temp1")
```



**Note:** temp1 dir is created

```
>>> os.getcwd()
```

```
'C:\\Users\\MRCET\\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
```

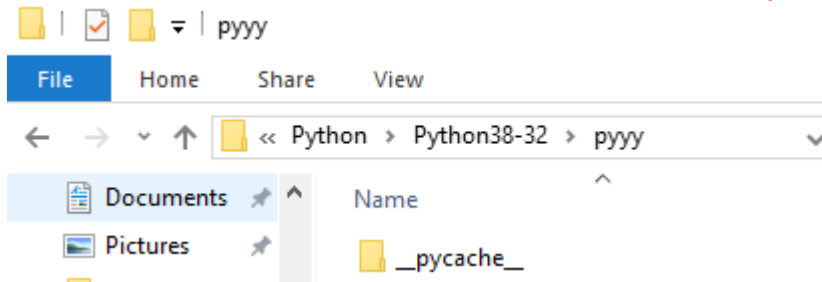
```
>>> os.access("t3.py",os.F_OK)
```

```
True
```

```
>>> os.rmdir('temp1')
```

(or)

```
os.rmdir('C:/Users/MRCET/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', 'cmdnlinarg.py', 'comm.py', 'con1.py', 'cont.py', 'cont2.py', 'dl.py', 'dic.py', 'el.py', 'example.py', 'fl.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', 'listloop.py', 'm1.py', 'merg.py', 'nesforr.py', 'nestedif.py', 'opprec.py', 'paraarg.py', 'qucksort.py', 'qukdsc.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/MRCET/AppData/Local/Programs/Python/Python38-32')
```

```
['argpar.py', 'br.py', 'bu.py', 'cmdnlinarg.py', 'DLLs', 'Doc', 'fl.py', 'fl.txt', 'filess', 'functupretval.py', 'funture.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/c11.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  
,
```

Activate Windows

**#write a python program to check whether the given year is leap or not.**

```
import calendar  
  
print(calendar.isleap(2021))
```

**Output:**

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/c11.py  
  
False
```

**#write a python program to print all the months of given year.**

```
import calendar  
  
print(calendar.calendar(2020,1,1,1))
```

**Output:**

&gt;&gt;&gt;

```
= 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/MRCET/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 beginning with an underscore, visible in our scope.

## Installation of pip

### Introduction

**PIP** is a package management system used to install and manage software packages written in Python. It stands for “preferred installer program” or “Pip Installs Packages.”

**PIP for Python** is a utility to manage PyPI package installations from the command line.

If you are using an older version of Python on Windows, you may need to install PIP. You can easily install PIP on Windows by downloading the installation package, opening the command line, and launching the installer.

**Note:** The latest versions of Python come with PIP pre-installed, but older versions require manual installation. The following guide is for version 3.4 and above. If you are using an older version of Python, you can upgrade Python via the Python website.

### Step 1: Check if PIP is Already Installed

Before you **install PIP on Windows**, check if PIP is already installed.

Type in the following command at the command prompt:

### **pip help**

If PIP responds, then PIP is installed. Otherwise, there will be an error saying the program could not be found.

PIP is automatically installed with Python 2.7.9+ and Python 3.4+.

PIP also comes with the *virtualenv* and *pyvenv* virtual environments.

### **Step 2: Verify Python Installation**

As a Python utility, **PIP requires an active Python installation**. In newer versions of Python and Python-enabled virtual environments, PIP is already installed, and you do not need to reinstall it.

To determine whether you have Python installed:

- Open the Command Prompt window.
- When the console window opens, type in:

### **python**

If this command is not unrecognized, you need to install Python before you can install PIP.

If the command is recognized, Python responds with its version and a list of commands.

When Python is installed correctly, you should see:

**Python 3.7.0 (v3.7.0:1bf9cc5093, Jan 25 2019, 07:44:31) [MSC v.1914 64 bit (AMD64)]  
on win32**

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

## **Installing PIP on Windows**

### **Step 1: Download PIP get-pip.py**

Before installing PIP, download the get-pip.py file: get-pip.py on [pypi.io](https://pypi.org/project/get-pip/).

Download the file to the desired folder in Windows. You can save the file to any location, but remember the path so you can use it later.

## Step 2: Launch Windows Command Line

PIP is a command-line program. When you install PIP, the PIP command is added to your system.

To launch the Command Prompt window:

- Press **Windows Key + X**.
- Click **Run**.
- Type in **cmd.exe** and hit enter.

Alternatively, type **cmd** in the Windows search bar and click the “Command Prompt” icon.

Both options open the Command Prompt window. However, note that you may need to run the Command Prompt “As Administrator.” If you get an error at any point stating that you don’t have the necessary permissions to perform a task, you will need to open the app as admin.

To run the Command Prompt window “As Administrator,” right-click “Command Prompt” and then select the “Run as...” option.

## Step 3: Installing PIP on Windows

Open the Command Prompt if it isn’t already open. Use the **cd** command followed by a folder name to navigate to the location of the *get-pip.py* file. This is the folder you previously used as the download location.

To install PIP type in the following:

```
python get-pip.py
```

PIP installation should start. If the file isn’t found, double-check the path to the folder where you saved the file.

You can view the contents of your current directory using the following command:

```
dir
```

The **dir** command returns a full listing of the contents of a directory.

## Step 4: How to Check PIP Version

To check the current version of PIP, type the following command:

```
pip --version
```

This command returns the current version of the platform.



### Step 5: Verify Installation

Once you've installed PIP, you can test whether the installation has been successful by typing the following:

#### **pip help**

If PIP has been installed, the program runs, and you should see:

```
pip 18.0 from c:\users\administrator\appdata\local\programs\python\python37\lib\site-packages\pip (python 3.7)
```

If you receive an error, repeat the installation process.

### Step 6: Configuration

In Windows, the PIP configuration file is **%HOME%\pip\pip.ini**.

There is also a legacy per-user configuration file. The file is located at **%APPDATA%\pip\pip.ini**.

You can set a custom path location for this config file using the environment variable **PIP\_CONFIG\_FILE**.

### Upgrading PIP for Python on Windows

New versions of PIP are released occasionally. These versions may improve the functionality or be obligatory for security purposes.

You can upgrade PIP on Windows using the Command Prompt window.

To upgrade PIP on Windows, enter the following in the command prompt:

```
python -m pip install --upgrade pip
```

This command first uninstalls the old version of PIP and then installs the most current version of PIP.

### Downgrade PIP Version

This may be necessary if a new version of PIP starts performing undesirably.

If you want to downgrade PIP to a prior version, you can do so by specifying the version.

To downgrade PIP, enter:

```
python -m pip install pip==18.1
```

You should now see the version of PIP that you specified.

## Conclusion

Congratulations, you have **installed PIP for Python on Windows**.

Now that you have PIP up and running, you are ready to manage your Python packages.

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices. Check out our guide and learn how to install NumPy using PIP.

**Note:** ex: `pip install numpy`

## Demonstrate Modules:

### Turtle:

“Turtle” is a Python feature like a drawing board, which lets us command a turtle to draw all over it! We can use functions like `turtle.forward(...)` and `turtle.right(...)` which can move the turtle around. Commonly used turtle methods are :

METHOD	PARAMETER	DESCRIPTION
<code>Turtle()</code>	None	Creates and returns a new turtle object
<code>forward()</code>	amount	Moves the turtle forward by the specified amount
<code>backward()</code>	amount	Moves the turtle backward by the specified amount
<code>right()</code>	angle	Turns the turtle clockwise
<code>left()</code>	angle	Turns the turtle counter clockwise
<code>penup()</code>	None	Picks up the turtle's Pen
<code>pendown()</code>	None	Puts down the turtle's Pen
<code>up()</code>	None	Picks up the turtle's Pen

METHOD	PARAMETER	DESCRIPTION
down()	None	Puts down the turtle's Pen
color()	Color name	Changes the color of the turtle's pen
fillcolor()	Color name	Changes the color of the turtle will use to fill a polygon
heading()	None	Returns the current heading
position()	None	Returns the current position
goto()	x, y	Move the turtle to position x,y
begin_fill()	None	Remember the starting point for a filled polygon
end_fill()	None	Close the polygon and fill with the current fill color
dot()	None	Leave the dot at the current position
stamp()	None	Leaves an impression of a turtle shape at the current location
shape()	shapename	Should be 'arrow', 'classic', 'turtle' or 'circle'

### Plotting using Turtle

To make use of the turtle methods and functionalities, we need to import turtle.”turtle” comes packed with the standard Python package and need not be installed externally.The roadmap for executing a turtle program follows 4 steps:

1. Import the turtle module
2. Create a turtle to control.
3. Draw around using the turtle methods.
4. Run turtle.done().

### Example:

**Write a python code to set background color and pic and draw a circle using turtle graphics**

**PROGRAM:-**

```
import turtle
t=turtle.Turtle()
t.circle(50)
s=turtle.Screen()
s.bgcolor("pink")
s.bgpic("pic.gif")
```

**OUTPUT:-**



**Pandas**

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data.

Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data — load, prepare, manipulate, model, and analyze.

Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

**Key Features of Pandas**

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.

- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

Pandas deals with the following three data structures –

- Series
- DataFrame
- Panel

### Dimension & Description

The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, DataFrame is a container of Series, Panel is a container of DataFrame.

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, sizeimmutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

### Examples:

**Write a python program to create a Data Frame with dictionary.**

### PROGRAM:-

```
import pandas as pd
one=pd.DataFrame({'Name':["Yoshitha","Anisha","Thejaswini"],
                  'sub_id':['sub1','sub2','sub3'],
```

```
'Marks':[92,88,76]},  
index=[1,2,3])  
print(one)
```

**OUTPUT:-**

	Name	sub_id	Marks
1	Yoshitha	sub1	92
2	Anisha	sub2	88
3	Thejaswini	sub3	76

**Write a python program to concatenate the dataframes with three different objects.**

**PROGRAM:-**

```
import pandas as pd  
  
one=pd.DataFrame({'Name':["anisha","yoshitha","john"],  
                  'sub_id':['sub1','sub2','sub3'],  
                  'Marks':[82,98,70]},  
                  index=[1,2,3])  
  
two=pd.DataFrame({'Name':["yukta","manoj","amar"],  
                  'sub_id':['sub2','sub4','sub1'],  
                  'Marks':[72,69,50]},  
                  index=[4,5,6])  
  
three=pd.DataFrame({'Name':["tarun","harsha","steve"],  
                    'sub_id':['sub1','sub3','sub4'],  
                    'Marks':[67,54,40]},  
                    index=[7,8,9])  
  
print(pd.concat([one,two,three]))
```

**OUTPUT:-**

```
===== KESIAK
      Name sub_id Marks
1   anisha  sub1    82
2 yoshitha  sub2    98
3    john   sub3    70
4    yukta  sub2    72
5   manoj   sub4    69
6    amar   sub1    50
7   tarun   sub1    67
8   harsha  sub3    54
9    steve  sub4    40
```

**Numpy**

NumPy, which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed.

**Operations using NumPy**

Using NumPy, a developer can perform the following operations –

- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

**Examples:**

**Using a numpy module create array and check the following:**

- 1. Reshape 3X4 array to 2X2X3 array**
- 2. Sequence of integers from 0 to 30 with steps of 5**
- 3. Flatten array**
- 4. constant value array of complex type**

**PROGRAM:-**

```
import numpy as np
arr = np.array([[1, 2, 3, 4],
                [5, 2, 4, 2],
                [1, 2, 0, 1]])

newarr = arr.reshape(2, 2, 3)
print ("\nOriginal array:\n", arr)
```

```
print ("Reshaped array:\n", newarr)

f = np.arange(0, 30, 5)
print ("\nA sequential array with steps of 5:\n", f)

arr = np.array([[1, 2, 3], [4, 5, 6]])
flarr = arr.flatten()
print ("\nOriginal array:\n", arr)
print ("Flattened array:\n", flarr)

d = np.full((3, 3), 6, dtype = 'complex')
print ("\nAn array initialized with all 6s."
        "Array type is complex:\n", d)
```

### **OUTPUT:-**

```
Original array:
[[1 2 3 4]
 [5 2 4 2]
 [1 2 0 1]]
Reshaped array:
[[[1 2 3]
  [4 5 2]]

 [[4 2 1]
  [2 0 1]]]

A sequential array with steps of 5:
[ 0  5 10 15 20 25]

Original array:
[[1 2 3]
 [4 5 6]]
Flattened array:
[1 2 3 4 5 6]

An array initialized with all 6s.Array type is complex:
[[6.+0.j 6.+0.j 6.+0.j]
 [6.+0.j 6.+0.j 6.+0.j]
 [6.+0.j 6.+0.j 6.+0.j]]
```

### **Pdb:**



'Debugging' term is popularly used to process of locating and rectifying errors in a program. Python's standard library contains pdb module which is a set of utilities for debugging of Python programs.

The debugging functionality is defined in a Pdb class. The module internally makes use of bdb and cmd modules.

The pdb module has a very convenient command line interface. It is imported at the time of execution of Python script by using `-m` switch

**`python -m pdb script.py`**

### **Examples:**

**Write a python code to perform addition using functions with pdb module.**

### **PROGRAM:-**

```
import pdb

def add(x,y):
    pdb.set_trace()
    sum=x+y
    return sum

def main():
    x=int(input("num1: "))
    y=int(input("num2: "))
    z=add(x,y)
    print(z)

main()
```

### **OUTPUT:-**

### **Python shell-**

```
num1: 7
num2: 3
> c:\17r21a1217\w16.py(4) add()
-> sum=x+y
(Pdb) c
10
```

**Write a python Program to Add five different elements to the list dynamically using pdb module.**

**PROGRAM:-**

```
import pdb

l2=[]

pdb.set_trace()

a=int(input("Enter first element"));
b=int(input("Enter second element"));
c=int(input("Enter third element"));
d=int(input("Enter fourth element"));
e=int(input("Enter five element"));

l2.append(a)
l2.append(b)
l2.append(c)
l2.append(d)
l2.append(e)

print(l2)
```

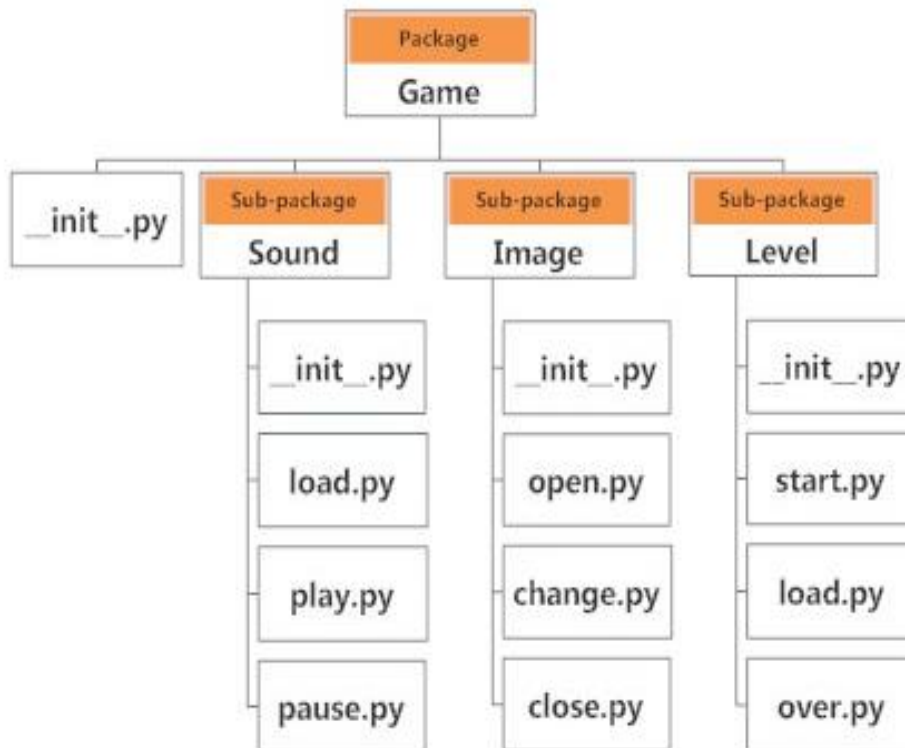
**OUTPUT:-**

**Python shell-**

```
-> a=int(input("Enter first elemet"));  
(Pdb) c  
Enter first elemet 6  
Enter second elemet 8  
Enter third elemet 1  
Enter fourth elemet 3  
Enter five elemet 9  
[6, 8, 1, 3, 9]
```

### 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) from 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
```

```
<function read at 0x03BD1070>
```

```
>>> 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\MRCET\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/MRCET/AppData/Local/Programs/Python/Python38-32/IIYEAR/exam.py

hello world

```
>>> import exam as ex
```

hello world

**UNIT –V****OOPS , FRAMEWORK**

**Oops concepts:** Object, Class, Method, Inheritance, Polymorphism, Data abstraction, Encapsulation,

**Python Frameworks:** Explore django framework with an example

**Oops concepts:**

OOP uses the concept of objects and classes. A class can be thought of as a 'blueprint' for objects. These can have their own attributes (characteristics they possess), and methods (actions they perform).

Python is a great programming language that supports OOP. You will use it to define a class with attributes and methods, which you will then call. Python offers a number of benefits compared to other programming languages like Java, C++ or R. It's a dynamic language, with high-level data types. This means that development happens much faster than with Java or C++. It does not require the programmer to declare types of variables and arguments. This also makes Python easier to understand and learn for beginners, its code being more readable and intuitive.

**Class and object:**

Class is a collection of data members and member functions.

Class is a blue print of an object.

Object is a real time entity. Instance of a class is called object.

**Instantiating objects**

To instantiate an object, type the class name, followed by two brackets. You can assign this to a variable to keep track of the object and then print it

The process of creating object is called instantiation (allocating memory to the data members and member function of the class)

**Class syntax:**

Class <class name> :

**Object creation:**

Variable name=class name()

```
def displayMethod(self):  
self represents like this in java
```

A class contains static variables and instance variables and methods as well.

**Programming to display mrcet by using classes and objects**

```
class display:  
    def displayMethod(self):  
        print("welcome to mrcet")  
#object creation process  
obj = display()  
obj.displayMethod()
```

**Output:**

welcome to mlrit

**Defining method in a class:**

**To define a method in class we use def keyword.**

def keyword is used again, as well as the self argument.

**calling variable and method:****#Programming to call data member and function using classes and objects**

```
class display:  
    a="hello"  
    def displayMethod(self):  
        print("welcome to mrcet")  
#object creation process  
obj = display()  
obj.displayMethod()  
print(obj.a)
```

**Output:**



welcome to mrcet  
hello

**#write a program to find sum of two numbers using class and methods**

```
class Addition:
    a=0
    b=0
    c=0
    def getData(self):
        self.a=int(input("Enter a value"))
        self.b=int(input("Enter b Value"))
    def add(self):
        self.c=self.a+self.b
    def display(self):
        print("addition is",self.c)

obj=Addition()
obj.getData()
obj.add()
obj.display()
```

### **Output:**

Enter a value3  
Enter b Value4  
addition is 7

### **Constructors:**

Constructor is special kind of method that will be called at the time of object creation. Python depends on constructor to perform initialization i.e. assigning values to any instance variables

### **Createing a constructor:**

Constructor in python is a special method that begins with (\_\_) double underscore. Name of the constructor is always \_\_init\_\_(parameters)

### **Types of Constructors:**

0 arguments constructor

Arguments constructor

Default constructor

### Zero args constructor:

The constructor present with out arguments

### Example:

1. Write a program with serial number, name of the student, through zero args constructors assign the values and assign the same.

2. Instance 2 objects and observe the output

**#Write a program with serial number, name of the student, through zero args constructors assign the values and assign the same.**

```
class student:
    def __init__(self):
        self.serialNumber=1
        self.nameOfStudent="nirosha"
    def display(self):
        print("serial number ",self.serialNumber)
        print("da: ",self.nameOfStudent)
obj1=student()
obj1.display()
obj2=student()
obj2.display()
```

### Output:

serial number 1

da: nirosha

serial number 1

da: nirosha

**#write a program to find factorial of the given number by using classes and objects using zero args constructor**

```
class factorial:
    def __init__(self):
        self.first=1
        self.fact=1
```

```
def getNumber(self):
    self.n=int(input("Enter a number"))
def factorial1(self):
    while self.first<=self.n:
        self.fact=self.fact*self.first
        self.first=self.first+1
def display(self):
    print("factorial ",self.fact)
obj1=factorial()
obj1.getNumber()
obj1.factorial1()
obj1.display()
```

**Output:**

```
Enter a number5
factorial 120
```

**parameterized constructor:**

a constructor with parameters is know as parameterized constructor. Used to provide different values to different objects.

**Example:**

**# Program to create student class serial no and name, Write a parameterized constructor to provide values to serial no and name, Instance two objects display serial no and name observe the output**

```
'''
class Student:
    def __init__(self,sno,name):
        self.sno=sno
        self.name=name

    def display(self):
        print("serial number ",self.sno)
        print("name ",self.name)
obj1=Student(1,"nirosha")
```

```
obj1.display()
```

```
obj2=Student(2,"mrce")  
obj2.display()
```

**Output:**

```
serial number 1  
name nirosha  
serial number 2  
name mrcet
```

**Inheritance:**

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

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

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

**Syntax:**

Class Base:

Class derived(base):

**Example:**

''' Write a program to create a parent class and declare a method called parent name and declare a child class declare a method child name, instance a object for child class and invoke super class and sub class methods'''

```
class Parent:  
    def parentName(self):  
        print("parent name")  
class Child(Parent):  
    def childName(self):  
        print("child name")
```

```
o=Child()
o.childName();
o.parentName();
```

**output:**

```
child name
parent name
>>>
```

**''' constructors in parent and child'''**

```
class Parent:
    def __init__(self):
        print("parent class constructor is called")
    def parentName(self):
        print("parent name")
class Child(Parent):
    def __init__(self):
        print("child class constructor is called")
    def childName(self):
        print("child name")
o=Child()
o.childName();
o.parentName();
```

**output:**

```
child class constructor is called
child name
parent name
>>>
```

**“How to call parent class constructor from child constructor.”**

```
Parent.__init__(self)
```

**''' constructors in parent and child'''**

```
class Parent:
    def __init__(self):
        print("parent class constructor is called")
    def parentName(self):
        print("parent name")
```

```
class Child(Parent):
    def __init__(self):
        Parent.__init__(self)
        print("child class constructor is called")
    def childName(self):
        print("child name")
o=Child()
o.childName();
o.parentName();
```

**output:**

```
parent class constructor is called
child class constructor is called
child name
parent name
>>>
```

**Polymorphism:**

**Polymorphism in python** defines 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. Also, it is possible to modify a method in a child class that it has inherited from the parent class.

We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.

**Polymorphism in Class Methods**

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

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")
```

```
def make_sound(self):
    print("Meow")

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

    def info(self):
        print(f"I am a dog. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Bark")

cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

**Output:**

```
Meow
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
Bark
I am a dog. My name is Fluffy. I am 4 years old.
Bark
```

Here, we have created two classes Cat and Dog. They share a similar structure and have the same method names info() and make\_sound().

However, notice that we have not created a common superclass or linked the classes together in any way. Even then, we can pack these two different objects into a tuple and iterate through it using a common animal variable. It is possible due to polymorphism.

### **Data Abstraction:**

Abstraction in Python is the process of hiding the real implementation of an application from the user and emphasizing only on usage of it.

**Need:** Through the process of abstraction in Python, a programmer can hide all the irrelevant data/process of an application in order to reduce complexity and increase efficiency.

### **Abstract Classes In Python**

A class containing one or more abstract methods is called an abstract class.

Abstract methods do not contain any implementation. Instead, all the implementations can be defined in the methods of sub-classes that inherit the abstract class. An abstract class is created by importing a class named 'ABC' from the 'abc' module and inheriting the 'ABC' class. Below is the syntax for creating the abstract class.

#### **Syntax**

```
from abc import ABC
Class ClassName(ABC):
```

#### **Example:**

```
from abc import ABC, abstractmethod
class Animal(ABC):
```

```
    def move(self):
        pass
```

```
class Human(Animal):
```

```
    def move(self):
        print("I can walk and run")
```



```
class Snake(Animal):  
  
    def move(self):  
        print("I can crawl")  
  
class Dog(Animal):  
  
    def move(self):  
        print("I can bark")  
  
class Lion(Animal):  
  
    def move(self):  
        print("I can roar")
```

# Driver code

```
R = Human()  
R.move()
```

```
K = Snake()  
K.move()
```

```
R = Dog()  
R.move()
```

```
K = Lion()  
K.move()
```

**Output:**

```
I can walk and run  
I can crawl  
I can bark  
I can roar
```

**Encapsulation:**

The concept of Encapsulation is to keep together the implementation (code) and the data it manipulates (variables). Having proper encapsulation ensures that the code and data both are safe from misuse by outside entity.

### Encapsulation in Python

In any object oriented language first step towards encapsulation is the class and encapsulation in Python also starts from a class as the class encapsulates the methods and variables.

When a Python class is created it contains the methods and the variables. Since it's the code in the methods that operates on the variables, in a properly encapsulated Python class, methods should define how member variables can be used.

But that's where the things differ a bit in Python from a language like Java where we have access modifiers like public, private. In Python there are no explicit access modifiers and everything written within the class (methods and variables) are public by default.

**For example in the class Person there are two variables as you can see those variables are accessed through a method as well as directly.**

```
class Person:
    def __init__(self, name, age=0):
        self.name = name
        self.age = age

    def display(self):
        print(self.name)
        print(self.age)
```

```
person = Person('John', 40)
#accessing using class method
person.display()
#accessing directly from outside
print(person.name)
print(person.age)
```

### Output:

John

40

John

40

**Explore django framework with an example:**

**Django** is an open-source python web framework **used** for rapid development, pragmatic, maintainable, clean design, and secures websites. ... The main goal of the **Django** framework is to allow developers to focus on components of the application that are new instead of spending time on already developed components.

**To get the version of Django :**

```
Python -m django --version
```

**To start a project , go to python path, from there open command prompt and start creating a project**

```
$ django-admin startproject mysite
```

**Move to dir mysite**

```
cd mysite
```

```
python manage.py server
```

```
.....
```

```
.....
```

```
.....
```

Performing system check

```
....
```

```
.....
```

Strating development server at

<http://127.0.0.1:8000/>

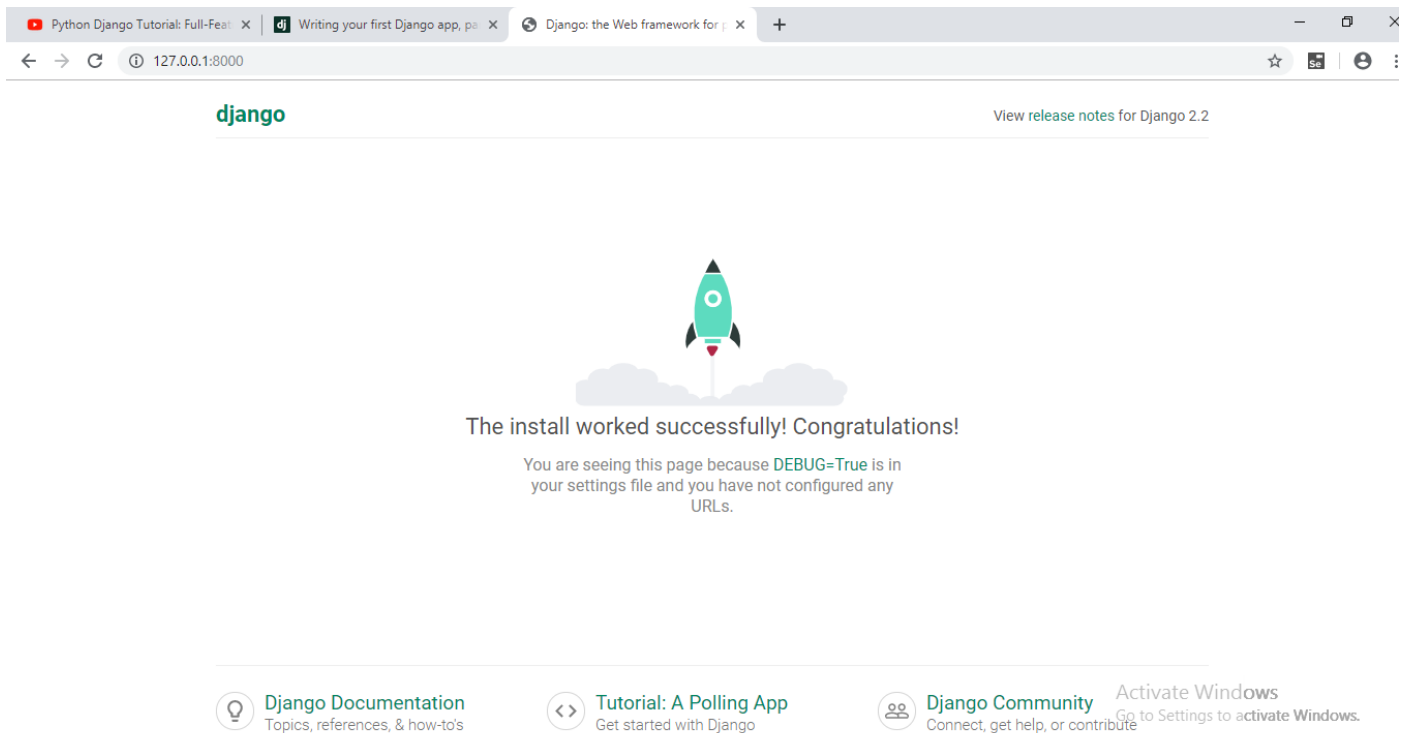
copy thr url and paste it then it shows **Django app successfully installed**

To quit press ctrl+c

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32>django-admin startproject niru
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32>cd niru
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\niru>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin,
auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
October 25, 2019 - 10:47:03
Django version 2.2.6, using settings 'niru.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
[25/Oct/2019 10:47:11] "GET / HTTP/1.1" 200 16348
```



**Mysite> python manage.py startapp polls**

Creating a Weather App in Django x | M Inbox (1,429) - nirosha.kunduru@ x | Log in | Django site admin x +

127.0.0.1:8000/admin/login/?next=/admin/

Django administration

Username:

Password:

Log in

Creating a Weather App in Django x | M Inbox (1,429) - nirosha.kunduru@ x | Site administration | Django site x +

127.0.0.1:8000/admin/

Django administration WELCOME, MRCET. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION	
Groups	+ Add Change
Users	+ Add Change

Recent actions

My actions

None available

**From here we create our views in polls/views.py**

**To create URLconf in polls dir create a file urls.py i.e., polls/urls.py**

**Next step to point URLconf at polls.urls**

**Mysite> python manage.py runserver**

Next go to website and paste the url (127.0.0.0:8000/polls/) which you have defined in index view to show the output

