



MRCET CAMPUS

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS INSTITUTION –UGC,GOVT. OF INDIA)

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

PYTHON PROGRAMMING DIGITAL NOTES

**B.TECH I YEAR – I SEM (R22)
(2023-2024)**

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

I Year B.Tech-I Sem

L /T/P/C

3/-/-/3

(R22A0582) PYTHON PROGRAMMING

COURSE OBJECTIVES:

The students will be able to

1. To read and write simple Python programs.
2. To develop Python programs with conditionals and loops.
3. To develop Python programs with using arrays and functions.
4. To use Python data structures— lists, tuples, dictionaries.
5. To do input/output with files in Python.

UNIT- I

Introduction to Computing – Computer Systems, Functional Units of a Computer, Computer Languages, Language Translators, Representation of Algorithms and Flowcharts with examples.

Introduction to Python Programming Language: Introduction to Python Language, Features of Python, Python Installation, Python Input and Output Statements, Numeric Data Types: int, float, boolean, complex and string and its operations, Standard Data Types: List, Tuples, Sets and Dictionaries, Data Type conversions, Comments in Python.

UNIT- II

Variables and Operators: Understanding Python variables, multiple variable declarations, Operators in Python: Arithmetic operators, Assignment operators, Comparison operators, Logical operators, Identity operators, Membership operators, Bitwise operators, Precedence of operators, Expressions.

Control Flow and Loops: Indentation, if statement, if-else statement, chained conditional if-elif -else statement, Loops: While loop, for loop using ranges, Loop manipulation using break, continue and pass.

UNIT- III

Arrays: Advantages of Arrays, Creating an Array, Importing the Array Module, Indexing and Slicing on Arrays, Types of arrays, working with arrays using numpy.

UNIT- IV

Functions: Defining a function, Calling Functions, Passing parameters and arguments, Python Function arguments: Positional Arguments, Keyword Arguments, Default Arguments, Variable-length arguments, Scope of the Variables in a Function – Local and Global Variables, Fruitful Functions, Anonymous functions or Lambda functions, Powerful Lambda functions in Python.

UNIT- V

File Handling in Python: Introduction to files, Text files and Binary files, Access Modes, Writing Data to a File, Reading Data from a File, File input / output functions.

Error Handling in Python: Introduction to Errors and Exceptions: Compile-Time Errors, Runtime Errors, Logical Errors, Types of Exceptions, Exception Handling, Handling Multiple Exceptions.

COURSE OUTCOMES:

Upon completion of the course, students will be able to

1. Read, write, execute by hand simple Python programs.
2. Structure simple Python programs for solving problems.
3. Decompose a Python program into arrays and functions.
4. Represent compound data using Python lists, tuples, dictionaries.
5. Read and write data from/to files in Python programs.

TEXT BOOKS

1. R. Nageswara Rao, “Core Python Programming”, dreamtech.
2. Allen B. Downey, “Think Python: How to Think Like a Computer Scientist,,,,, 2nd edition, Updated for Python 3, Shroff/O,,Reilly Publishers, 2016.
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	Introduction to Computing – Computer Systems	1-4
	Functional Units of a Computer	4-5
	Computer Languages, Language Translators,	6-8
	Algorithms and Flowcharts	9-10
	Introduction to Python Programming Language: Introduction to Python Language and installation and Features	11-13
	Input Output Functions	14-16
	Numeric Data Types: int, float, Boolean, complex and string and its operations,	16-28
	Standard Data Types: List, tuples, set and Dictionaries,	29-52
	Data Type conversions	52-53
	commenting in python.	54
II	Understanding Python variables, Multiple variable declarations	55-57
	Python basic statements	57
	Python basic operators: Arithmetic operators	58-60
	Assignment operators, Comparison operators	
	Logical operators, Identity operators	
	Membership operators, Bitwise operators	
	Precedence of operators	61
	Expressions.	62-65
	CONTROL FLOW AND LOOPS	66-70
	conditional (if)	
	alternative (if-else)	
	chained conditional (if- elif -else)	
	Loops: For loop using ranges, string, Use of while loops in python	71-79
	Loop manipulation using pass, continue and break	79-84

III	Arrays: Advantages of Arrays	85-86
	Creating an Array	86-88
	Importing the Array Module	88
	Indexing and Slicing on Arrays	89
	Types of arrays	90
	working with arrays using numpy	91-109
IV	Functions Defining Your Own Functions	110
	Calling Functions, passing parameters and arguments	111-117
	Python Function arguments: Keyword Arguments, Default Arguments, Variable-length arguments	118-124
	Anonymous Functions, Powerful Lambda functions in python	125-129
	Fruitful Functions	130-131
	Local and Global Scope	132-136
V	File Handling in Python: Introduction to files, Text files and Binary files, Access Modes	137 - 138
	Writing Data to a File, Reading Data from a File, File input / output functions.	139-142
	Introduction to Errors and Exceptions, Types of Errors	143-145
	Exception Handling, Handling Multiple Exceptions.	145-146

UNIT I

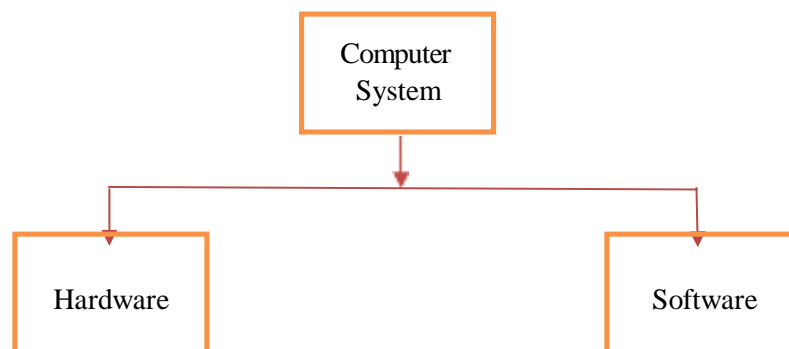
Introduction to Computing – Computer Systems, Functional Units of a Computer, Computer Languages, Language Translators, Representation of Algorithms and Flowcharts with examples.

Introduction to Python Programming Language: Introduction to Python Language, Features of Python, Python Installation, Python Input and Output Statements, Numeric Data Types: int, float, boolean, complex and string and its operations, Standard Data Types: List, Tuples, Sets and Dictionaries, Data Type conversions, Comments in Python.

Introduction to Computing:

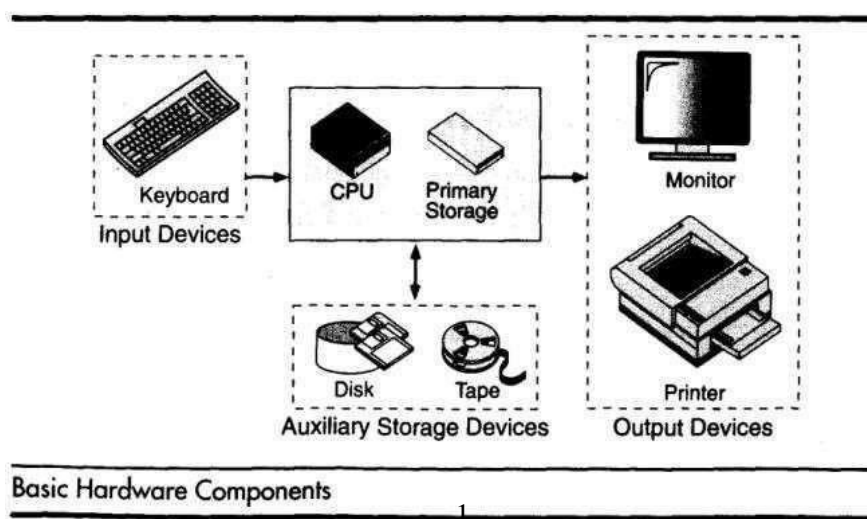
Computer Systems:

A computer is a system made of two major components: hardware and software. The computer hardware is the physical equipment. The software is the collection of programs (instructions) that allow the hardware to do its job.



Computer Hardware

The hardware component of the computer system consists of five parts: input devices, central processing unit (CPU), primary storage, output devices, and auxiliary storage devices.



The input device is usually a keyboard where programs and data are entered into the computers. Examples of other input devices include a mouse, a pen or stylus, a touch screen, or an audio input unit.

The central processing unit (CPU) is responsible for executing instructions such as arithmetic calculations, comparisons among data, and movement of data inside the system. Today's computers may have one, two, or more CPUs. Primary storage, also known as main memory, is a place where the programs and data are stored temporarily during processing. The data in primary storage are erased when we turn off a personal computer or when we log off from a time-sharing system.

The output device is usually a monitor or a printer to show output. If the output is shown on the monitor, we say we have a soft copy. If it is printed on the printer, we say we have a hard copy.

Auxiliary storage, also known as secondary storage, is used for both input and output. It is the place where the programs and data are stored permanently. When we turn off the computer, or programs and data remain in the secondary storage, ready for the next time we need them.

Computer Software

Computer software is divided into two broad categories: system software and application software. System software manages the computer resources. It provides the interface between the hardware and the users. Application software, on the other hand, is directly responsible for helping users solve their problems.

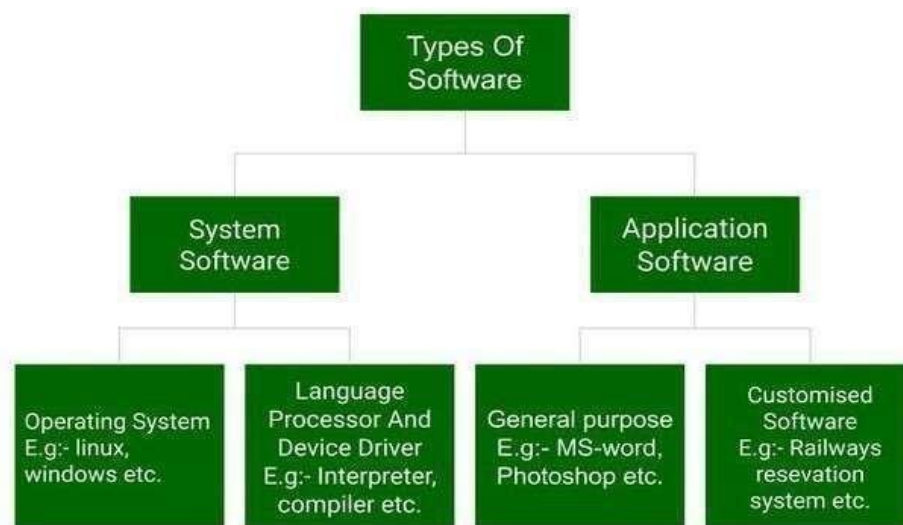


Fig: Types of software

System Software basically controls a computer's internal functioning and also controls hardware devices such as monitors, printers, and storage devices, etc. It is like an interface between hardware and user applications, it helps them to communicate with each other because hardware understands machine language (i.e. 1 or 0) whereas user applications are work in human-readable languages like English, Hindi, German, etc. so system software converts the human-readable language into machine language and vice versa.

Types of system software:

It has two subtypes which are:

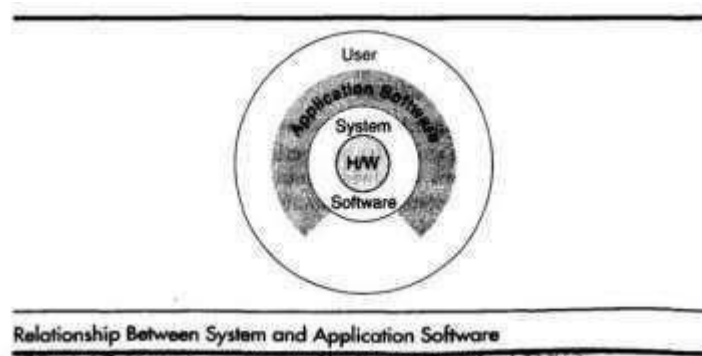
1. **Operating System:** It is the main program of a computer system. When the computer system ON it is the first software that loads into the computer's memory. Basically, it manages all the resources such as memory, CPU, printer, hard disk, etc., and provides an interface to the user, which helps the user to interact with the computer system. It also provides various services to other computer software. Examples of operating systems are Linux, Apple macOS, Microsoft Windows, etc.
2. **Language Processor:** As we know that system software converts the human-readable language into a machine language and vice versa. So, the conversion is done by the language processor. It converts programs written in high-level programming languages like Java, C, C++, Python, etc(known as source code), into sets of instructions that are easily readable by machines(known as object code or machine code).
3. **Device Driver:** A device driver is a program or software that controls a device and helps that device to perform its functions. Every device like a printer, mouse, modem, etc. needs a driver to connect with the computer system eternally. So, when you connect a new device with your computer system, first you need to install the driver of that device so that your operating system knows how to control or manage that device.

Application software is designed to perform a specific task for end-users. It is a product or a program that is designed only to fulfill end-users' requirements. It includes word processors, spreadsheets, database management, inventory, payroll programs, etc.

There are different types of application software and those are:

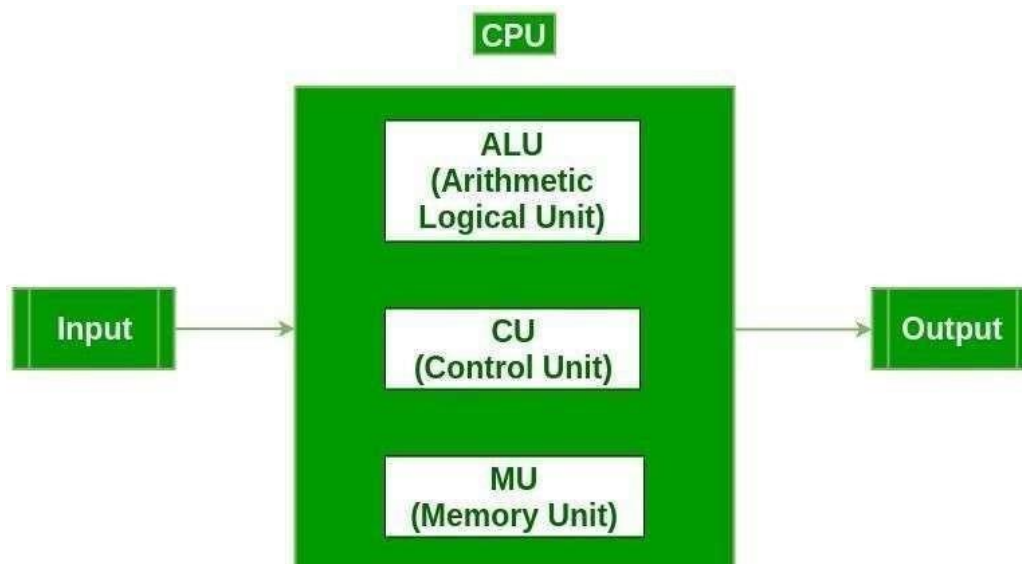
1. **General Purpose Software:** This type of application software is used for a variety of tasks and it is not limited to performing a specific task only. For example, MS-Word, MS-Excel, PowerPoint, etc.
2. **Customized Software:** This type of application software is used or designed to perform specific tasks or functions or designed for specific organizations. For example, railway reservation system, airline reservation system, invoice management system, etc.

3. **Utility Software:** This type of application software is used to support the computer infrastructure. It is designed to analyze, configure, optimize and maintains the system, and take care of its requirements as well. For example, antivirus, disk memory tester, disk repair, disk cleaners, disk space analyzer, etc.



Functional Units of a Computer

- **Input Unit :** The input unit consists of input devices that are attached to the computer. These devices take input and convert it into binary language that the computer understands. Some of the common input devices are keyboard, mouse, joystick, scanner etc



- **Central Processing Unit (CPU) :** Once the information is entered into the computer by the input device, the processor processes it. The CPU is called the brain of the computer because it is the control center of the computer. It first fetches instructions from memory and then interprets them so as to know what is to be done. If required, data is fetched from memory or input device. Thereafter CPU executes or performs the required computation and then either stores the output or displays on the output device.
- The CPU has three main components which are responsible for different functions – **Arithmetic Logic Unit (ALU), Control Unit (CU) and Memory registers**

Arithmetic and Logic Unit (ALU) : The ALU, as its name suggests performs mathematical calculations and takes logical decisions. Arithmetic calculations include addition, subtraction, multiplication and division. Logical decisions involve comparison of two data items to see which one is larger or smaller or equal.

Control Unit : The Control unit coordinates and controls the data flow in and out of CPU and also controls all the operations of ALU, memory registers and also input/output units. It is also responsible for carrying out all the instructions stored in the program. It decodes the fetched instruction, interprets it and sends control signals to input/output devices until the required operation is done properly by ALU and memory.

Memory Registers : A register is a temporary unit of memory in the CPU. These are used to store the data which is directly used by the processor. Registers can be of different sizes(16 bit, 32 bit, 64 bit and so on) and each register inside the CPU has a specific function like storing data, storing an instruction, storing address of a location in memory etc.

The user registers can be used by an assembly language programmer for storing operands, intermediate results etc. Accumulator (ACC) is the main register in the ALU and contains one of the operands of an operation to be performed in the ALU.

Memory : when a program is executed, it's data is copied to the internal memory and is stored in the memory till the end of the execution. The internal memory is also called the Primary memory or Main memory. This memory is also called as RAM, i.e. Random Access Memory.

Output Unit : The output unit consists of output devices that are attached with the computer. It converts the binary data coming from CPU to human understandable form. The common output devices are monitor, printer, plotter etc.

Computer Languages:

To write a program for a computer, we must use a computer language. Over the years computer languages have evolved from machine languages to natural languages.

1940's Machine level Languages

1950's Symbolic Languages

1960's High-Level Languages

Machine Languages

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language, which is made of streams of 0's and 1's.

Instructions in machine language must be in streams of 0's and 1's because the internal circuits of a computer are made of switches transistors and other electronic devices that can be in one of two states: off or on. The off state is represented by 0 , the on state is represented by 1.

The only language understood by computer hardware is machine language.

Symbolic Languages:

In early 1950's Admiral Grace Hopper, A mathematician and naval officer developed the concept of a special computer program that would convert programs into machine language.

The early programming languages simply mirror to the machine languages using symbols of mnemonics to represent the various machine language instructions because they used symbols, these languages were known as symbolic languages.

Computer does not understand symbolic language it must be translated to the machine language. A special program called assembler translates symbolic code into machine language. Because symbolic languages had to be assembled into machine language they soon became known as assembly languages.

Symbolic language uses symbols or mnemonics to represent the various ,machine language instructions.

High Level Languages:

Symbolic languages greatly improved programming efficiency; they still required programmers to concentrate on the hardware that they were using. Working with symbolic languages was also very tedious because each machine instruction has to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of high-level language.

High level languages are portable to many different computers, allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer. High-level languages are designed to relieve the programmer from the details of the assembly language. High level languages share one thing with symbolic languages, They must be converted into machine language. The process of converting them is known as compilation.

The first widely used high-level languages, FORTRAN (FORMula TRANslation) was created by John Backus and an IBM team in 1957; it is still widely used today in scientific and engineering applications. After FORTRAN was COBOL (Common Business-Oriented Language). Admiral Hopper was played a key role in the development of the COBOL Business language.

C is a high-level language used for system software and new application code.

Language Translators

There are mainly three Types of translators which are used to translate different programming languages into machine equivalent code:

1. Assembler
2. Compiler
3. Interpreter

Assembler

An assembler translates assembly language into machine code.

Assembly language consists of Mnemonics for machine Op-codes, so assemblers perform a 1:1 translation from mnemonic to a direct instruction. For example, LDA #4 converts to 0001001000100100.

Compiler

Compiler is a computer program that translates code written in a high level language to a low level language, object/machine code.

The most common reason for translating source code is to create an executable program (converting from high level language into machine language).

Interpreter

An interpreter is a translator, which converts High Level programs into machine code. This process is known as interpretation. Note that the purpose of interpreter is same as that of a compiler. Interpreter is used to convert the High Level program into computer understandable form. Interpreter performs line-by-line execution of the source program, while compiler executes entire program at a time.

Interpreter reads source program one line at a time, converts into machine language, executes the line, and then proceeds to the next line. If an error occurs during execution, error must be solved before it can proceed to the next line.

ALGORITHM:

Algorithms was developed by an Arab mathematician. Algorithm is a step – by – step procedure which is helpful in solving a problem. If, it is written in English like sentences then, it is called as ‘PSEUDO CODE’.

Properties of an Algorithm

An algorithm must possess the following five properties –

- Input
- Output
- Finiteness
- Definiteness
- Effectiveness

Example: Algorithm/pseudo code to add two numbers

Step 1: Start

Step 2: Read the two numbers in to a, b

Step 3: $c=a+b$

Step 4: write/print c






Step 5: Stop.

FLOW CHART:

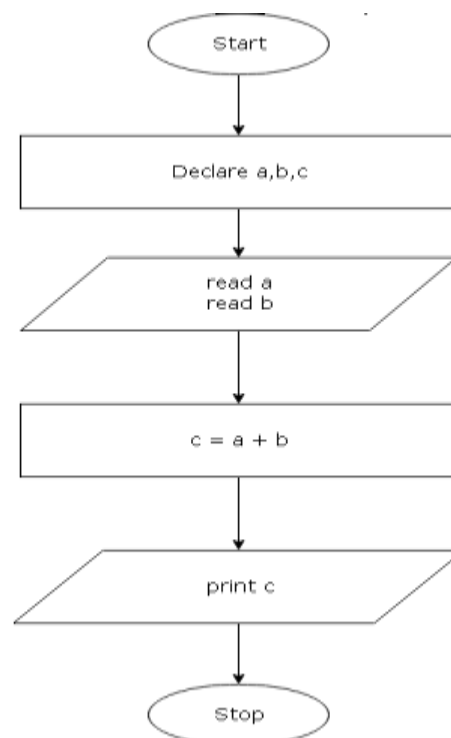
A Flow chart is a Graphical representation of an Algorithm or a portion of an Algorithm. Flow charts are drawn using certain special purpose symbols such as Rectangles, Diamonds, Ovals and small circles. These symbols are connected by arrows called flow lines.

(or)

The diagrammatic representation of way to solve the given problem is called flow chart.

Symbol	Name	Function
	Start/end	An oval represents a start or end point.
	Arrows	A line is a connector that shows relationships between the representative shapes.
	Input/Output	A parallelogram represents input or output.
	Process	A rectangle represents a process.
	Decision	A diamond indicates a decision.

Example: Flowchart to add two numbers



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
```


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.

4. 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.

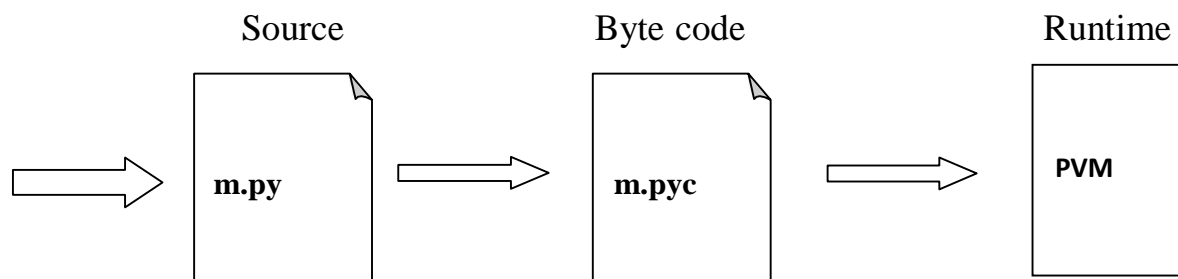
Step 6: Add Python Path to Environment Variables (Optional)



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.

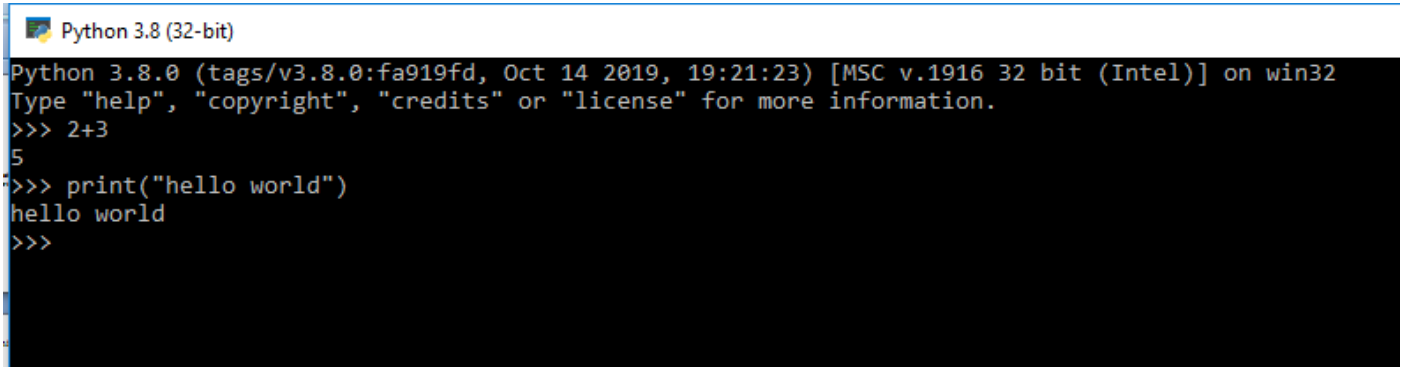
```
>>> x
```

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 title bar reads "Python 3.8 (32-bit)". The window content shows the following text:

```
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
```

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()
```

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.endswith('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

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
<code>len([1, 2, 3])</code>	3	Length
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>	Concatenation
<code>['Hi!'] * 4</code>	<code>['Hi!', 'Hi!', 'Hi!', 'Hi!']</code>	Repetition
<code>3 in [1, 2, 3]</code>	True	Membership
<code>for x in [1, 2, 3]: print x,</code>	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
<code>L[2]</code>	MRCET	Offsets start at zero
<code>L[-2]</code>	college	Negative: count from the right
<code>L[1:]</code>	<code>['college', 'MRCET!']</code>	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
```

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)
12
```

Tuple Assignment

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

For Example,

Here is the code,

- >>> tup1 = ('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:

Union	OR	&
Symmetric Diff	XOR	
Diff	In set1 but not in set2	^
Subset	set2 contains set1	set1-set2
Superset	set1 contains set2	set1<=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')
```

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'}
```

Data Type conversions:

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types. Casting in python is therefore done using constructor functions:

int() - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)

float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)

str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Examples:**Integers:**

```
x = int(1) # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

```
Print(x)
```

```
Print(y)
```

```
Print(z)
```

Output:

```
1
2
3
```

Floats:

```
x = float(1) # x will be 1.0
y = float(2.8) # y will be 2.8
z = float("3") # z will be 3.0
w = float("4.2") # w will be 4.2
```

```
Print(x)
```

```
Print(y)
```

```
Print(z)
```

```
Print(w)
```

Output:

```
1.0
2.8
3.0
4.2
```

Strings:

```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0) # z will be '3.0'Print(x)
```

```
Print(y)
```

```
Print(z)
```

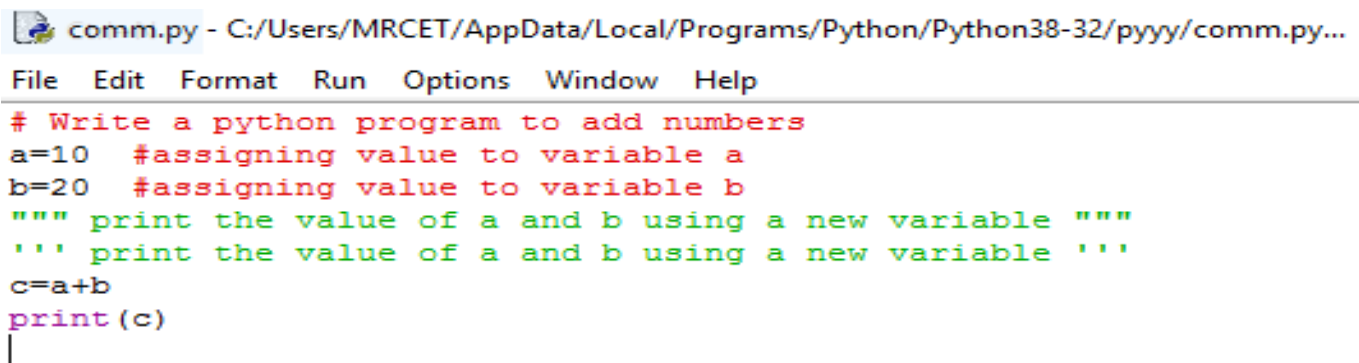
Output:

```
s1
2
3.0
```

Commenting in python:

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

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



```
comm.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/comm.py...
File Edit Format Run Options Window Help
# Write a python program to add numbers
a=10 #assigning value to variable a
b=20 #assigning value to variable b
""" print the value of a and b using a new variable """
''' print the value of a and b using a new variable '''
c=a+b
print(c)
|
```

UNIT II

Variables and Operators: Understanding Python variables, multiple variable declarations, Operators in Python: Arithmetic operators, Assignment operators, Comparison operators, Logical operators, Identity operators, Membership operators, Bitwise operators, Precedence of operators, Expressions.

Control Flow and Loops: Indentation, if statement, if-else statement, chained conditional if-elif -else statement, Loops: While loop, for loop using ranges, Loop manipulation using break, continue and pass.

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:

- Variable name must start with a letter or the underscore character
- 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)    #100
print (b)    #1000.0
print (c)    #John
```

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

Python basic 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("mrcetcollege")  
  
mrcet college
```

Python basic Operators:

Operators are used to perform operations on variables and values. Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Arithmetic operators

Operator	Name	Example
+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	x / y

Assignment operators

Operator	Example	Same As
=	$x = 5$	$x = 5$
+=	$x += 3$	$x = x + 3$
-=	$x -= 3$	$x = x - 3$

<code>*=</code>	<code>x *= 3</code>	<code>x = x * 3</code>
<code>/=</code>	<code>x /= 3</code>	<code>x = x / 3</code>

Comparison operators

Operator	Name	Example
<code>==</code>	Equal	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

Logical operators

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

Identity operators

Operator	Description	Example
----------	-------------	---------

is	Returns true if both variables are the same object	x is y
is not	Returns true if both variables are not the same object	x is not y

Membership operators

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

Bitwise operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Precedence of Operators:

Operator precedence affects how an expression is evaluated.

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

Example 1:

```
>>> 3+4*2
```

```
11
```

Multiplication gets evaluated before the addition operation

```
>>> (10+10)*2
```

```
40
```

Parentheses () overriding the precedence of the arithmetic operators

Example 2:

```
a = 20
```

```
b = 10
```

```
c = 15
```

```
d = 5
```

```
e = 0
```

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

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

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

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

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

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

```
e = a + (b * c) / d;     # 20 + (150/5)
```

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

Output:

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

```
Value of (a + b) * c / d is 90.0
```

```
Value of ((a + b) * c) / d is 90.0
```

```
Value of (a + b) * (c / d) is 90.0
```

Value of $a + (b * c) / d$ is 50.0

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'
```

CONTROLFLOWAND LOOPS

conditional (if), alternative (if-else), chained conditional (if- elif -else), Loops: For loop using ranges, string, use of while loops in python, Loop manipulation using pass, continue and break

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:

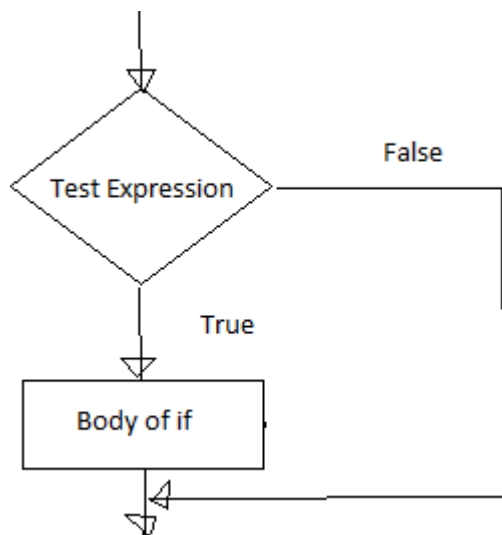


Fig: Operation of if statement

Example: Python if Statement

```
a = 3
if a > 2:
    print(a, "isgreater")
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

Ais 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 if stmts
else:
    Body of else stmts
```

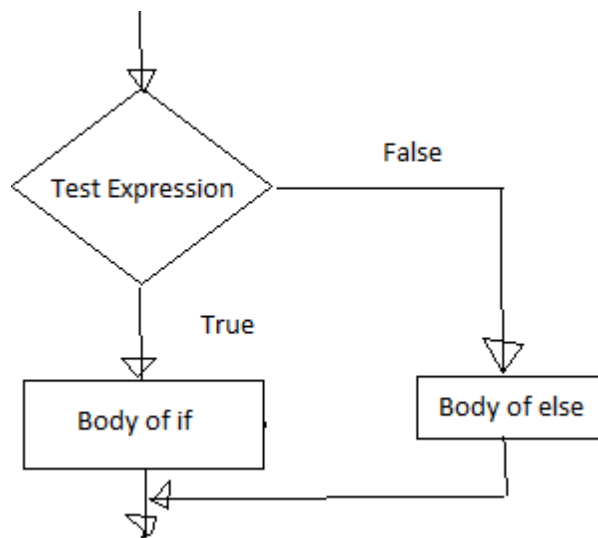

If - else Flowchart :

Fig: Operation of if – else statement

Example of if - else:

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

Output:

C:/Users/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 elifstmts

else:

 Body of else stmts

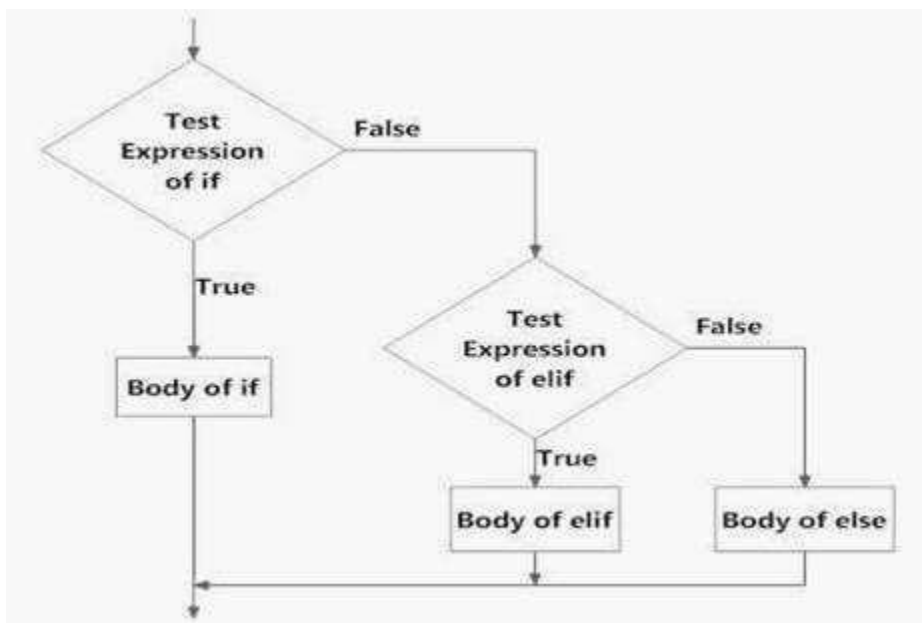
Flowchart of if – elif - else:

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

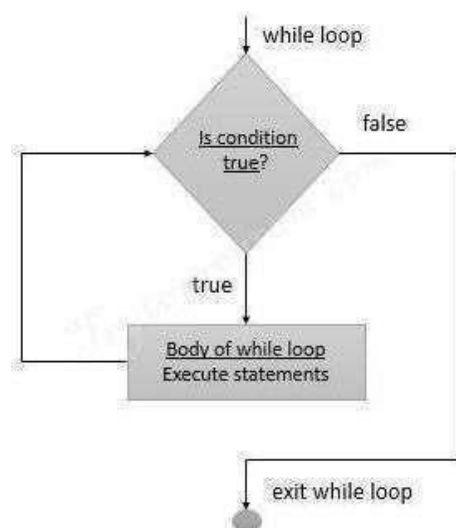
While loop:

- 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
while (a<10):
    print ('Iteration',a)
    a = a + 1
    if (a == 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

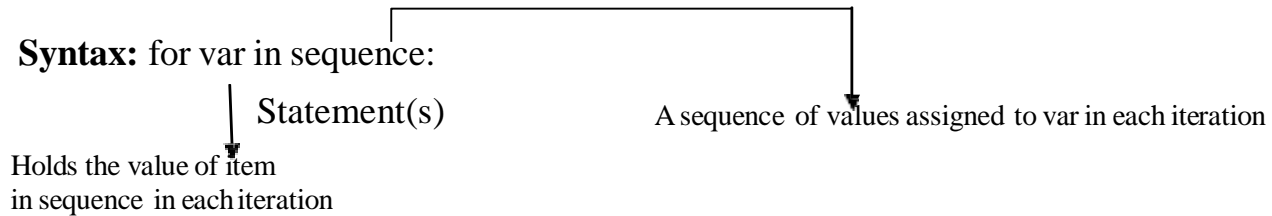
The count is: 7

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

**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
```

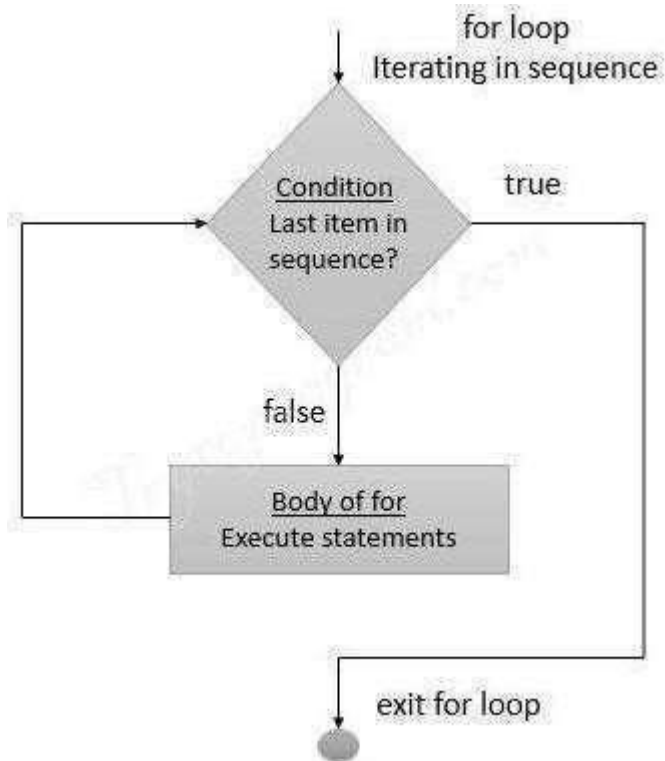
```
16
```

```
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

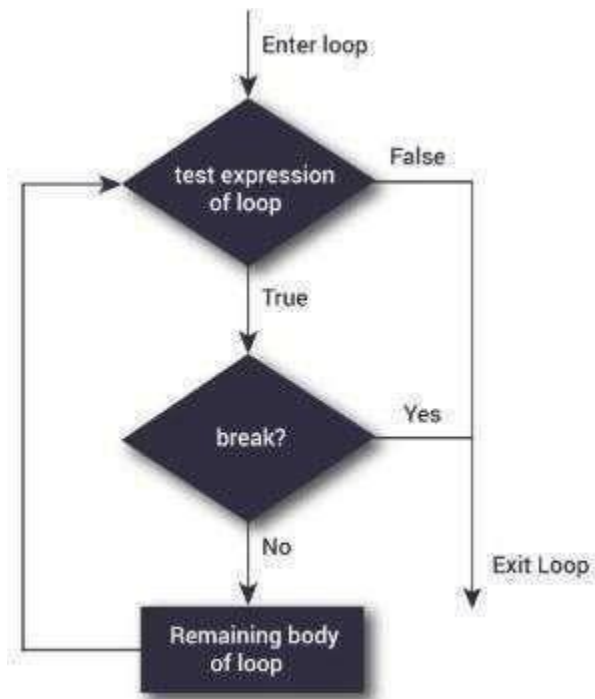
Break and continue:

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:

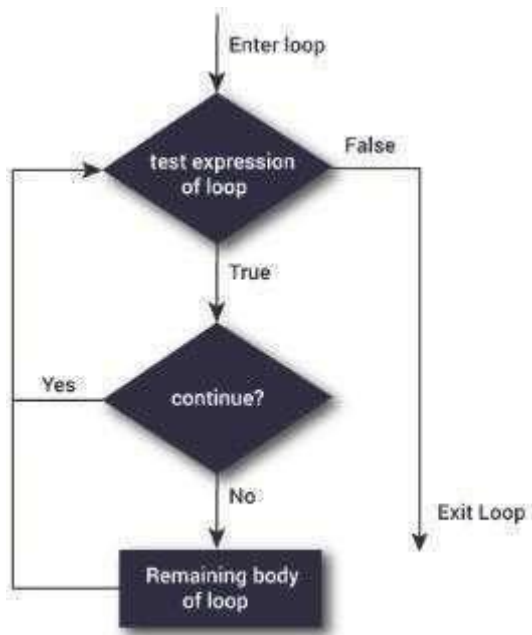
```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/br.py =
Current Letter : P
Current Letter : y
```

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
```

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)
```

Unit III

Arrays are a fundamental data structure, and an important part of most programming languages. In Python, they are containers which are able to store more than one item at the same time.

Specifically, they are an ordered collection of elements with every value being of the same data type. That is the most important thing to remember about Python arrays - the fact that they can only hold a sequence of multiple items that are of the same type.

Difference between Python Lists and Python Arrays

Lists are one of the most common data structures in Python, and a core part of the language.

Lists and arrays behave similarly.

Just like arrays, lists are an ordered sequence of elements.

They are also mutable and not fixed in size, which means they can grow and shrink throughout the life of the program. Items can be added and removed, making them very flexible to work with.

However, lists and arrays are **not** the same thing.

Lists store items that are of **various data types**. This means that a list can contain integers, floating point numbers, strings, or any other Python data type, at the same time. That is not the case with arrays.

Arrays store only items that are of the **same single data type**. There are arrays that contain only integers, or only floating point numbers, or only any other Python data type you want to use.

Advantages Python Arrays

Lists are built into the Python programming language, whereas arrays aren't. Arrays are not a built-in data structure, and therefore need to be imported via the array module in order to be used.

Arrays of the array module are a thin wrapper over C arrays, and are useful when you want to work with homogeneous data.

They are also more compact and take up less memory and space which makes them more size efficient compared to lists.

Creating Python arrays

In order to create Python arrays, you'll first have to import the array module which contains all the necessary functions.

There are three ways you can import the array module:

1. By using import array at the top of the file. This includes the module array. You would then go on to create an array using array.array()

```
import array
```

```
#how you would create an array  
array.array()
```

2. Instead of having to type `array.array()` all the time, you could use `import array as arr` at the top of the file, instead of `import array` alone. You would then create an array by typing `arr.array()`. The `arr` acts as an alias name, with the array constructor then immediately following it.

```
import array as arr
```

```
#how you would create an array  
arr.array()
```

3. Lastly, you could also use `from array import *`, with `*` importing all the functionalities available. You would then create an array by writing the `array()` constructor alone.

```
from array import *
```

```
#how you would create an array  
array()
```

Define Arrays in Python

Once the array module is imported, we can then go on to define a Python array.

The general syntax for creating an array looks like this:

```
variable_name = array(typecode,[elements])
```

- `variable_name` would be the name of the array.
- The typecode specifies what kind of elements would be stored in the array. Whether it would be an array of integers, an array of floats or an array of any other Python data type. Remember that all elements should be of the same data type.
- Inside square brackets you mention the elements that would be stored in the array, with each element being separated by a comma. You can also create an *empty* array by just writing `variable_name = array(typecode)` alone, without any elements.

The different typecodes that can be used with the different data types when defining Python arrays:

TYPECODE	C TYPE	PYTHON TYPE	SIZE
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	wchar_t	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Example of how to define an array in Python:

```
import array as arr
numbers = arr.array('i',[10,20,30])
print(numbers)
```

```
#output
#array('i', [10, 20, 30])
```

Example of how to create an array numbers of float data type.

```
from array import *
#an array of floating point values
numbers = array('d',[10.0,20.0,30.0])
print(numbers)
```

```
#output
#array('d', [10.0, 20.0, 30.0])
```

Array Indexing and How to Access Individual Items in an Array in Python

Each item in an array has a specific address. Individual items are accessed by referencing their *index number*.

Indexing in Python, and in all programming languages and computing in general, starts at 0.

To access an element, we first write the name of the array followed by square brackets. Inside the square brackets you include the item's index number.

The index value of the last element of an array is always one less than the length of the array.

Where n is the length of the array, n - 1 will be the index value of the last item.

The general syntax would look something like this:

```
array_name[index_value_of_item]
```

Example:

```
import array as arr
numbers = arr.array('i',[10,20,30])
print(numbers[0]) # gets the 1st element
print(numbers[1]) # gets the 2nd element
print(numbers[2]) # gets the 3rd element
```

```
#output
#10
#20
#30
```

We can also access each individual element using negative indexing.

With negative indexing, the last element would have an index of -1, the second to last element would have an index of -2, and so on.

Example

```
import array as arr
numbers = arr.array('i',[10,20,30])
print(numbers[-1]) #gets last item
print(numbers[-2]) #gets second to last item
print(numbers[-3]) #gets first item
```

```
#output
#30
#20
#10
```

How to Slice an Array in Python

To access a specific range of values inside the array, use the slicing operator, which is a colon :.

When using the slicing operator and you only include one value, the counting starts from 0 by default. It gets the first item, and goes up to but not including the index number you specify.

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#get the values 10 and 20 only
print(numbers[:2]) #first to second position
```

```
#output
#array('i', [10, 20])
```

When you pass two numbers as arguments, you specify a range of numbers. In this case, the counting starts at the position of the first number in the range, and up to but not including the second one:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#get the values 20 and 30 only
print(numbers[1:3]) #second to third position
```

```
#output

#array('i', [20, 30])
```

Array Types in Python

When talking about arrays, any programming language like C or Java offers two types of arrays. They are:

Single dimensional arrays: These arrays represent only one row or one column of elements. For example, marks obtained by a student in 5 subjects can be written as 'marks' array, as:

```
marks = array('i', [50, 60, 70, 66, 72])
```

The above array contains only one row of elements. Hence it is called single dimensional array or **one dimensional array**.

Multi-dimensional arrays: These arrays represent more than one row and more than one column of elements. For example, marks obtained by 3 students each one in 5 subjects can be written as 'marks' array as:

```
marks = [[50, 60, 70, 66, 72], [60, 62, 71, 56, 70], [55, 59, 80, 68, 65]]
```

The first student's marks are written in first row. The second student's marks are in second row and the third student's marks are in third row. In each row, the marks in 5 subjects are mentioned. Thus this array contains 3 rows and 5 columns and hence it is called **multi-dimensional array**.

```
marks = [[50, 60, 70, 66, 72],  
         [60, 62, 71, 56, 70],  
         [55, 59, 80, 68, 65]]
```

Each row of the above array can be again represented as a single dimensional array. Thus the above array contains 3 single dimensional arrays. Hence, it is called a two dimensional array. A two dimensional array is a combination of several single dimensional arrays. Similarly, a three dimensional array is a combination of several two dimensional arrays.

In Python, we can create and work with single dimensional arrays only. So far, the examples and methods discussed by us are applicable to single dimensional arrays.

Python does not support multi-dimensional arrays. We can construct multidimensional arrays using third party packages like **numpy (numerical python)**

Unit III

Arrays are a fundamental data structure, and an important part of most programming languages. In Python, they are containers which are able to store more than one item at the same time.

Specifically, they are an ordered collection of elements with every value being of the same data type. That is the most important thing to remember about Python arrays - the fact that they can only hold a sequence of multiple items that are of the same type.

Difference between Python Lists and Python Arrays

Lists are one of the most common data structures in Python, and a core part of the language.

Lists and arrays behave similarly.

Just like arrays, lists are an ordered sequence of elements.

They are also mutable and not fixed in size, which means they can grow and shrink throughout the life of the program. Items can be added and removed, making them very flexible to work with.

However, lists and arrays are **not** the same thing.

Lists store items that are of **various data types**. This means that a list can contain integers, floating point numbers, strings, or any other Python data type, at the same time. That is not the case with arrays.

Arrays store only items that are of the **same single data type**. There are arrays that contain only integers, or only floating point numbers, or only any other Python data type you want to use.

Advantages Python Arrays

Lists are built into the Python programming language, whereas arrays aren't. Arrays are not a built-in data structure, and therefore need to be imported via the array module in order to be used.

Arrays of the array module are a thin wrapper over C arrays, and are useful when you want to work with homogeneous data.

They are also more compact and take up less memory and space which makes them more size efficient compared to lists.

Creating Python arrays

In order to create Python arrays, you'll first have to import the array module which contains all the necessary functions.

There are three ways you can import the array module:

1. By using `import array` at the top of the file. This includes the module array. You would then go on to create an array using `array.array()`


```
import array
```

```
#how you would create an array  
array.array()
```

2. Instead of having to type `array.array()` all the time, you could use `import array as arr` at the top of the file, instead of `import array` alone. You would then create an array by typing `arr.array()`. The `arr` acts as an alias name, with the array constructor then immediately following it.

```
import array as arr
```

```
#how you would create an array  
arr.array()
```

3. Lastly, you could also use `from array import *`, with `*` importing all the functionalities available. You would then create an array by writing the `array()` constructor alone.

```
from array import *
```

```
#how you would create an array  
array()
```

Define Arrays in Python

Once the array module is imported, we can then go on to define a Python array.

The general syntax for creating an array looks like this:

```
variable_name = array(typecode,[elements])
```

- `variable_name` would be the name of the array.
- The typecode specifies what kind of elements would be stored in the array. Whether it would be an array of integers, an array of floats or an array of any other Python data type. Remember that all elements should be of the same data type.
- Inside square brackets you mention the elements that would be stored in the array, with each element being separated by a comma. You can also create an *empty* array by just writing `variable_name = array(typecode)` alone, without any elements.

The different typecodes that can be used with the different data types when defining Python arrays:

TYPECODE	C TYPE	PYTHON TYPE	SIZE
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	wchar_t	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Example of how to define an array in Python:

```
import array as arr
numbers = arr.array('i',[10,20,30])
print(numbers)
```

```
#output
#array('i', [10, 20, 30])
```

Example of how to create an array numbers of float data type.

```
from array import *  
#an array of floating point values  
numbers = array('d',[10.0,20.0,30.0])  
print(numbers)
```

```
#output  
#array('d', [10.0, 20.0, 30.0])
```

Array Indexing and How to Access Individual Items in an Array in Python

Each item in an array has a specific address. Individual items are accessed by referencing their *index number*.

Indexing in Python, and in all programming languages and computing in general, starts at 0.

To access an element, we first write the name of the array followed by square brackets. Inside the square brackets you include the item's index number.

The index value of the last element of an array is always one less than the length of the array.

Where n is the length of the array, n - 1 will be the index value of the last item.

The general syntax would look something like this:

```
array_name[index_value_of_item]
```

Example:

```
import array as arr  
numbers = arr.array('i',[10,20,30])  
print(numbers[0]) # gets the 1st element  
print(numbers[1]) # gets the 2nd element  
print(numbers[2]) # gets the 3rd element
```

```
#output  
#10  
#20  
#30
```

We can also access each individual element using negative indexing.

With negative indexing, the last element would have an index of -1, the second to last element would have an index of -2, and so on.

Example

```
import array as arr
numbers = arr.array('i',[10,20,30])
print(numbers[-1]) #gets last item
print(numbers[-2]) #gets second to last item
print(numbers[-3]) #gets first item
```

```
#output
#30
#20
#10
```

How to Slice an Array in Python

To access a specific range of values inside the array, use the slicing operator, which is a colon :.

When using the slicing operator and you only include one value, the counting starts from 0 by default. It gets the first item, and goes up to but not including the index number you specify.

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#get the values 10 and 20 only
print(numbers[:2]) #first to second position
```

```
#output
#array('i', [10, 20])
```

When you pass two numbers as arguments, you specify a range of numbers. In this case, the counting starts at the position of the first number in the range, and up to but not including the second one:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#get the values 20 and 30 only
print(numbers[1:3]) #second to third position
```

```
#output

#array('i', [20, 30])
```

Array Types in Python

When talking about arrays, any programming language like C or Java offers two types of arrays. They are:

Single dimensional arrays: These arrays represent only one row or one column of elements. For example, marks obtained by a student in 5 subjects can be written as 'marks' array, as:

```
marks = array('i', [50, 60, 70, 66, 72])
```

The above array contains only one row of elements. Hence it is called single dimensional array or **one dimensional array**.

Multi-dimensional arrays: These arrays represent more than one row and more than one column of elements. For example, marks obtained by 3 students each one in 5 subjects can be written as 'marks' array as:

```
marks = [[50, 60, 70, 66, 72], [60, 62, 71, 56, 70], [55, 59, 80, 68, 65]]
```

The first student's marks are written in first row. The second student's marks are in second row and the third student's marks are in third row. In each row, the marks in 5 subjects are mentioned. Thus this array contains 3 rows and 5 columns and hence it is called **multi-dimensional array**.

```
marks = [[50, 60, 70, 66, 72],  
         [60, 62, 71, 56, 70],  
         [55, 59, 80, 68, 65]]
```

Each row of the above array can be again represented as a single dimensional array. Thus the above array contains 3 single dimensional arrays. Hence, it is called a two dimensional array. A two dimensional array is a combination of several single dimensional arrays. Similarly, a three dimensional array is a combination of several two dimensional arrays.

In Python, we can create and work with single dimensional arrays only. So far, the examples and methods discussed by us are applicable to single dimensional arrays.

Python does not support multi-dimensional arrays. We can construct multidimensional arrays using third party packages like **numpy (numerical python)**

Numpy

- Numerical python popularly known as numpy. Numpy is designed to carry out mathematical computations at a faster and easier rate.

- In command line
 - pip install numpy

In [1]

```
: pip install numpy numpy
Requirement already satisfied: numpy in c:\users\vaikya\anaconda3\lib\site-pac
kages (1.16.2)
Note: you may need to restart the
```

In [2]

In [3]: `import numpy as np`

Out[3]: `np.__version__`

```
: '1.16.2'
```

- 1D - array
- 2D - array
- 3D - array

In [10]

In [11]: `a = np.array([1,2,3])`

Out[11]: `a`
 In [12]: `array([1, 2, 3])`

Out[12]: `a.ndim`

```
:
```

In [13]: `1`
 Out[13]: `a.dtype`

```
:
```

In [14]: `dtype('int32')`
 Out[14]: `b = np.array([1,2,3],float)`
 Out[14]: `b.dtype`
 Out[14]: `dtype('float64')`

```
In [15]: c = np.array([1,2.3,"apssdc"])
c
```

```
Out[15]: array(['1', '2.3', 'apssdc'], dtype='<U32')
```

```
In [16]: c.dtype
```

```
Out[16]: dtype('<U32')
```

```
In [18]: d = np.array([1,2.3,2])
d.dtype
```

```
Out[18]: dtype('float64')
```

```
In [19]: e = np.array("06-10-2020")
e.dtype
```

```
Out[19]: dtype('<U10')
```

```
In [21]: s = np.array(2+7j)
s.dtype
```

```
Out[21]
```

```
In [22]: dtype('complex128')
```

```
Out[22]: range(1,10)
```

```
In [23]: range(1, 10)
```

```
Out[23]: np.arange(1,10)
```

```
In [24]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
: range(1,10,0.2)
```

in <module>

TypeError

Traceback (most recent call last)

<ipython-input-24-30a9bd0f9fb4>

----> 1 range(1,10,0.2)

'+loat'

```
In [26]: TypeError: object cannot be interpreted as an integer
```

```
Out[26]: a = np.arange(1,10,0.2)
a
          3.2, 3.4,
          5. ,      5.6, 5.8, 6. ,
          7.6,      8.2, 8.4, 8.6,
: array([1. , 1.2, 1.4, 1.6, 1.8, 2. , 2.2, 2.4, 2.6, 2.8, 3. ,
        3.6, 3.8, 4. , 4.2, 4.4, 4.6, 4.8, 5.2, 5.4,
        6.2, 6.4, 6.6, 6.8, 7. , 7.2, 7.4, 7.8, 8. ,
        8.8, 9. , 9.2, 9.4, 9.6, 9.8])
```

Out[27]: 45

```
In [28]: a.shape
```

Out[28]: (45,)

```
In [29]: f = np.arange(1,51)
f
```

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

```
In [30]: f.reshape(5,10)
```

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

```
In [31]: f.reshape(10,5)
```

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

```
In [32]: f.reshape(2,-1) # 2,25 or 25,2 or 5,10 or 10,5
```

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

```
In [33]: f.reshape(-1,25)
```

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


```
In [34]: f.reshape(-1,-1)
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-34-af6f5b90af7b> in <module>  
----> 1 f.reshape(-1,-1)  
  
ValueError: can only specify one unknown dimension
```

```
In [35]: a = np.arange(1,50)  
a
```

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

```
In [39]: a1 = a.reshape(7,7)  
a1
```

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

```
In [37]: a.reshape(8,-1)
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-37-2f6f96af1ea8> in <module>  
----> 1 a.reshape(8,-1)  
  
ValueError: cannot reshape array of size 49 into shape (8,newaxis)
```

```
In [40]: a1.ndim  
Out[40]: 2
```

```
np.array((1,2,3))
```

```
In [41]: array([1, 2, 3])  
Out[41]: np.array({1:2})
```

```
In [42]: array({1: 2}, dtype=object)
```

```
Out[42]
```

```
In [43]: arr = np.arange(1,51).reshape(5,1,10)
arr
```

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

```
In [44]: arr.ndim
```

```
Out[44]: 3
```

```
In [45]: arr.shape
```

```
Out[45]: (5, 1, 10)
```

```
In [46]: arr.reshape(2,-1)
```

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

```
In [47]: np.array([[1,2,3],[4,5,6]])
```

```
Out[47]: array([[1, 2, 3],
                [4, 5, 6]])
```

```
In [49]: a =np.array([[[1,2,3],[4,5,6],[1,2,3],[4,5,6]])
a
```

```
Out[49]: array([[[1, 2, 3],
                  [4, 5, 6],
                  [1, 2, 3],
                  [4, 5, 6]])
```

```
Out[57]: array([ 1.          ,  1.04522613,  1.09045226,  1.13567839,  1.18090452,
 1.22613065,  1.27135678,  1.31658291,  1.36180905,  1.40703518,
 1.45226131,  1.49748744,  1.54271357,  1.5879397 ,  1.63316583,
 1.67839196,  1.72361809,  1.76884422,  1.81407035,  1.85929648,
 1.90452261,  1.94974874,  1.99497487,  2.04020101,  2.08542714,
 2.13065327,  2.1758794 ,  2.22110553,  2.26633166,  2.31155779,
 2.35678392,  2.40201005,  2.44723618,  2.49246231,  2.53768844,
 2.58291457,  2.6281407 ,  2.67336683,  2.71859296,  2.7638191 ,
 2.80904523,  2.85427136,  2.89949749,  2.94472362,  2.98994975,
 3.03517588,  3.08040201,  3.12562814,  3.17085427,  3.2160804 ,
 3.26130653,  3.30653266,  3.35175879,  3.39698492,  3.44221106,
 3.48743719,  3.53266332,  3.57788945,  3.62311558,  3.66834171,
 3.71356784,  3.75879397,  3.8040201 ,  3.84924623,  3.89447236,
 3.93969849,  3.98492462,  4.03015075,  4.07537688,  4.12060302,
 4.16582915,  4.21105528,  4.25628141,  4.30150754,  4.34673367,
 4.3919598 ,  4.43718593,  4.48241206,  4.52763819,  4.57286432,
 4.61809045,  4.66331658,  4.70854271,  4.75376884,  4.79899497,
 4.84422111,  4.88944724,  4.93467337,  4.9798995 ,  5.02512563,
 5.07035176,  5.11557789,  5.16080402,  5.20603015,  5.25125628,
 5.29648241,  5.34170854,  5.38693467,  5.4321608 ,  5.47738693,
 5.52261307,  5.5678392 ,  5.61306533,  5.65829146,  5.70351759,
 5.74874372,  5.79396985,  5.83919598,  5.88442211,  5.92964824,
 5.97487437,  6.0201005 ,  6.06532663,  6.11055276,  6.15577889,
 6.20100503,  6.24623116,  6.29145729,  6.33668342,  6.38190955,
 6.42713568,  6.47236181,  6.51758794,  6.56281407,  6.6080402 ,
 6.65326633,  6.69849246,  6.74371859,  6.78894472,  6.83417085,
 6.87939698,  6.92462312,  6.96984925,  7.01507538,  7.06030151,
 7.10552764,  7.15075377,  7.1959799 ,  7.24120603,  7.28643216,
 7.33165829,  7.37688442,  7.42211055,  7.46733668,  7.51256281,
 7.55778894,  7.60301508,  7.64824121,  7.69346734,  7.73869347,
 7.7839196 ,  7.82914573,  7.87437186,  7.91959799,  7.96482412,
 8.01005025,  8.05527638,  8.10050251,  8.14572864,  8.19095477,
 8.2361809 ,  8.28140704,  8.32663317,  8.3718593 ,  8.41708543,
 8.46231156,  8.50753769,  8.55276382,  8.59798995,  8.64321608,
 8.68844221,  8.73366834,  8.77889447,  8.8241206 ,  8.86934673,
 8.91457286,  8.95979899,  9.00502513,  9.05025126,  9.09547739,
 9.14070352,  9.18592965,  9.23115578,  9.27638191,  9.32160804,
 9.36683417,  9.4120603 ,  9.45728643,  9.50251256,  9.54773869,
 9.59296482,  9.63819095,  9.68341709,  9.72864322,  9.77386935,
 9.81909548,  9.86432161,  9.90954774,  9.95477387, 10.          ])
```

```
In [55]: a1 = a.flatten()
a1
```

```
Out[55]: array([1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6])
```

```
Out[62]: array([[1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1],
               [1, 1]])
```

```
In [63]: np.ones((3,2,2),dtype = int)
```

```
Out[63]: array([[[1, 1],
                 [1, 1]],

                [[1, 1],
                 [1, 1]],

                [[1, 1],
                 [1, 1]]])
```

```
In [65]: np.zeros((2,3,1),object)
```

```
Out[65]: array([[[0],
                 [0],
                 [0]],

                [[0],
                 [0],
                 [0]]], dtype=object)
```

```
In [66]: np.eye(4,3) # identity matrix
```

```
Out[66]: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.],
               [0., 0., 0.]])
```

Sorting

```
In [67]: arr1 = np.array([2,34,12,67,23,90,78])
         arr1
```

```
Out[67]: array([ 2, 34, 12, 67, 23, 90, 78])
```

```
In [68]: np.sort(arr1)
```

```
Out[68]: array([ 2, 12, 23, 34, 67, 78, 90])
```

```
In [70]: places = ("vijayawada","tirupathi","kalahasthri")
         temples = ("durgama","tirumala","shiva")
         a = np.lexsort((temples,places))
         a
```

```
Out[70]: array([2, 1, 0], dtype=int64)
```

```
In [72]: [places[i]+" "+temples[i] for i in a]
```

```
Out[72]: ['kalahasthri shiva', 'tirupathi tirumala', 'vijayawada durgama']
```

Indexing

```
In [73]: d = np.arange(1,100,2)
         d
```

```
Out[73]: array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33,
                35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67,
                69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99])
```

```
In [74]: d[6]
```

```
Out[74]: 13
```

```
In [75]: d[2:10]
```

```
Out[75]: array([ 5,  7,  9, 11, 13, 15, 17, 19])
```

```
In [76]: arr2 = np.arange(1,51).reshape(5,10)
         arr2
```

```
Out[76]: array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10],
                [11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
                [21, 22, 23, 24, 25, 26, 27, 28, 29, 30],
                [31, 32, 33, 34, 35, 36, 37, 38, 39, 40],
                [41, 42, 43, 44, 45, 46, 47, 48, 49, 50]])
```

```
In [77]: arr2[2,2]
```

```
Out[77]: 23
```

```
In [78]: arr2[3,7]
```

```
Out[78]: 38
```

```
In [79]: arr2[2:3,2:3]
```

```
Out[79]: array([[23]])
```

```
Out[80]: array([[[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]],
               [[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]],
               [[21, 22, 23, 24, 25, 26, 27, 28, 29, 30]],
               [[31, 32, 33, 34, 35, 36, 37, 38, 39, 40]],
               [[41, 42, 43, 44, 45, 46, 47, 48, 49, 50]])
```

```
In [ ]: #how acces the particular element in 3d
```

```
In [81]: # concatenate
a = np.array([[2,3],[7,8]])
b = np.array([[6,3],[8,12]])
a
```

```
Out[81]: array([[2, 3],
               [7, 8]])
```

```
In [82]: b
```

```
Out[82]: array([[ 6,  3],
               [ 8, 12]])
```

```
In [83]: np.concatenate((a,b))
Out[83]: array([[ 2,  3],
               [ 7,  8],
               [ 6,  3],
               [ 8, 12]])
```

```
In [84]: np.concatenate((a,b),axis=1) # axis=1--columns,axis=0---rows
Out[84]: array([[ 2,  3,  6,  3],
               [ 7,  8,  8, 12]])
```

```
In [85]: np.vstack((a,b))
Out[85]: array([[ 2,  3],
               [ 7,  8],
               [ 6,  3],
               [ 8, 12]])
```

```
In [86]: np.hstack((a,b))
Out[86]: array([[ 2,  3,  6,  3],
               [ 7,  8,  8, 12]])
```

```
Out[87]: array([[2, 3],
               [7, 8]])
```

```
In [88]: # transpose
a.T
```

```
Out[88]: array([[2, 7],
               [3, 8]])
```

```
In [91]: a.swapaxes(0,1)
```

```
Out[91]: array([[2, 7],
               [3, 8]])
```

Math

- Arithmetic functions

```
In [92]: a
```

```
Out[92]: array([[2, 3],
               [7, 8]])
```

```
In [93]: b
```

```
Out[93]: array([[ 6,  3],
               [ 8, 12]])
```

```
In [94]: print(a+b)
[[ 8  6]
 [15 20]]
```

```
In [95]: print(a-b)
[[-4  0]
 [-1 -4]]
```

```
In [96]: print(a*b)
[[2 0]
 [7 8]]
```

In [97]:

```
[[0 1]
 [0 0]]
```

In [98]: `print(a/b)`

```
[[0.33333333 1.          ]
 [0.875      0.66666667]]
```

In [99]: `np.divide(a,b)`

Out[99]: `array([[0.33333333, 1.],
 [0.875 , 0.66666667]])`

In [100]: `np.divmod(a,b)`

Out[100]: `(array([[0, 1],
 [0, 0]], dtype=int32), array([[2, 0],
 [7, 8]], dtype=int32))`

In [101]: `a%b`

Out[101]: `array([[2, 0],
 [7, 8]], dtype=int32)`

In [102]: `np.floor_divide(a,b)`

Out[102]: `array([[0, 1],
 [0, 0]], dtype=int32)`

In [103]: `a//b`

Out[103]: `array([[0, 1],
 [0, 0]], dtype=int32)`

In [104]: `a`

Out[104]: `array([[2, 3],
 [7, 8]])`

In [105]: `np.log(a)`

Out[105]: `array([[0.69314718, 1.09861229],
 [1.94591015, 2.07944154]])`

In [106]: `np.exp(a)`

Out[106]: `array([[7.3890561 , 20.08553692],
 [1096.63315843, 2980.95798704]])`


```
Out[107]: array([[0.30103    , 0.47712125],
                 [0.84509804, 0.90308999]])
```

```
In [108]: np.exp2(a)
```

```
Out[108]: array([[ 4.,  8.],
                 [128., 256.]])
```

Random concept

```
In [109]: from numpy import random
```

```
In [132]: random.randint(100) # generate a random integer from 0 to 100
```

```
Out[132]: 32
```

```
In [120]: random.randint(10,100,size = 11)
```

```
Out[120]: array([11, 37, 74, 15, 32, 19, 62, 24, 77, 88, 39])
```

```
In [133]: random.rand() # 0 to 1
```

```
Out[133]: 0.07132085658417153
```

```
In [121]: a = random.rand(10)
a
```

```
Out[121]: array([0.31442079, 0.42479701, 0.46666873, 0.91499173, 0.34568887,
                 0.16933889, 0.14767002, 0.18532065, 0.89948857, 0.60589583])
```

```
In [122]: a*10
```

```
Out[122]: array([3.14420792, 4.24797012, 4.66668729, 9.14991733, 3.45688868,
                 1.69338887, 1.47670016, 1.85320646, 8.99488568, 6.05895833])
```

```
In [123]: a+10
```

```
Out[123]: array([10.31442079, 10.42479701, 10.46666873, 10.91499173, 10.34568887,
                 10.16933889, 10.14767002, 10.18532065, 10.89948857, 10.60589583])
```

```
In [124]: a+a
```

```
Out[124]: array([0.62884158, 0.84959402, 0.93333746, 1.82998347, 0.69137774,
                 0.33867777, 0.29534003, 0.37064129, 1.79897714, 1.21179167])
```



```
Out[135]: array([[12, 12,  5,  5],
                 [ 3, 12, 12,  5]])
```

```
In [131]: random.choice([3,5,12],p=[0.2,0.1,0.7],size=(4,20))
```

```
Out[131]: array([[ 5,  3,  3, 12, 12,  3,  3, 12, 12, 12,  3, 12,  5,  3, 12,  3,
                  5, 12, 12, 12],
                 [12, 12, 12,  3,  5, 12, 12, 12, 12, 12, 12, 12, 12, 12,  5, 12, 12,
                  12, 12, 12, 12],
                 [12, 12, 12, 12, 12,  3, 12,  3, 12, 12, 12,  3,  3, 12,  3,  3,
                  5,  3,  3, 12],
                 [12, 12,  5, 12,  5, 12, 12,  3, 12, 12, 12, 12, 12,  3,  5,  3,  5,
                  5, 12, 12, 12]])
```

```
In [ ]:
```

UNIT IV**Functions:**

Defining Your Own Functions, Calling Functions, passing parameters and arguments, Python Function arguments: Keyword Arguments, Default Arguments, Variable-length arguments, Anonymous Functions, Fruitful Functions (Function Returning Values), Scope of the Variables in a Function - Global and Local Variables. Powerful Lambda functions in python.

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:

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

Some examples on functions:

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

#function defination

```
def display():
```

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

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

#function call

```
display()
```

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

Output:

C:/Users/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

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:

```
defhf():
```

```
    print("hello world")
```

```
hf()
```

Output:

```
hello world
```

```
defhf():
```

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

```
    print("ghkfjg 66666")
```

```
hf()
```

```
hf()
```

```
hf()
```

Output:

```
hw
```

```
ghkfjg 66666
```

```
hw
```

```
ghkfjg 66666
```

```
hw
```

```
ghkfjg 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.

```
defhf():
```

```
    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:

```
defhello(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 function. using arguments with return value

```
def calculate(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

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

```
def result(a):
    if a>40:
        return "pass"
    else:
        return "fail"
a=int(input("Enter one subject marks"))

print(result(a))
```

Output:

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

Enter one subject marks 35

fail

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

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

Output:

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

```
mrcetcse dept 0
mrcetcse dept 1
mrcetcse dept 2
mrcetcse dept 3
mrcetcse dept 4
mrcetcse dept 5
mrcetcse dept 6
mrcetcse dept 7
mrcetcse dept 8
mrcetcse dept 9
```

Anonymous 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 seperate 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\python\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/python/ml.py =

120

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], [1,7,8,9,510]))  
print(common_data([1,2,3,4,5], [6,7,8,9,10]))
```

Output:

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy\fu1.py  
True  
True  
None
```

```
def area(radius):  
  
    b = 3.14159 * radius**2  
  
    return b
```

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
```

Brief on other functions like sort, sorted and range:

The **sort()** method sorts the elements of a given list in a specific ascending or descending order.

The syntax of the sort() method is:

```
list.sort(key=..., reverse=...)
```

Example:

```
L1=[2,4,6,8,1,3,5]
```

```
L1.sort()
```

```
L2=[9,11,13,10,12,15,14]
```

```
L2.sort()
```

The sorted() function returns a sorted list of the specified iterable object.

You can specify ascending or descending order. Strings are sorted alphabetically, and numbers are sorted numerically.

Note: You cannot sort a list that contains BOTH string values AND numeric values.

Syntax:

```
sorted(iterable, key=key, reverse=reverse)
```

Example:

```
a = (1, 11, 2)
```

```
x = sorted(a)
```

```
print(x)
```

The built-in function range() generates the integer numbers between the given start integer to the stop integer, i.e., It returns a range object. Using for loop, we can iterate over a sequence of numbers produced by the range() function

range() function in for loop to iterate over numbers defined by range().

How to use range():

range(n) : will generate numbers from 0 to (n-1)

For example: range(8) is equivalent to [0, 1, 2, 3, 4, 5, 6, 7]

range(x, y) : will generate numbers from x to (y-1)

For example: range(5, 9) is equivalent to [5, 6, 7, 8]

range(start, end, step_size) : will generate numbers from start to end with step_size as incremental factor in each iteration. step_size is default if not explicitly mentioned.

For example: range(1, 10, 2) is equivalent to [1, 3, 5, 7, 9]

Example:

```
x=10
```

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

```
    print(i)
```

Output:

```
6
```

```
7
```

```
8
```

```
9
```

UNIT – V

File Handling in Python: Introduction to files, Text files and Binary files, Access Modes, Writing Data to a File, Reading Data from a File, File input / output functions.

Error Handling in Python: Introduction to Errors and Exceptions: Compile-Time Errors, RuntimeErrors, Logical Errors, Types of Exceptions, Exception Handling, Handling Multiple Exceptions.

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>
```

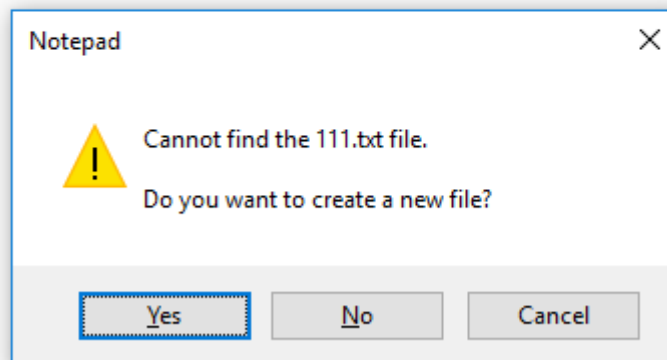
(or)

```
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?

Untitled - Notepad

File Edit Format View Help



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()
```

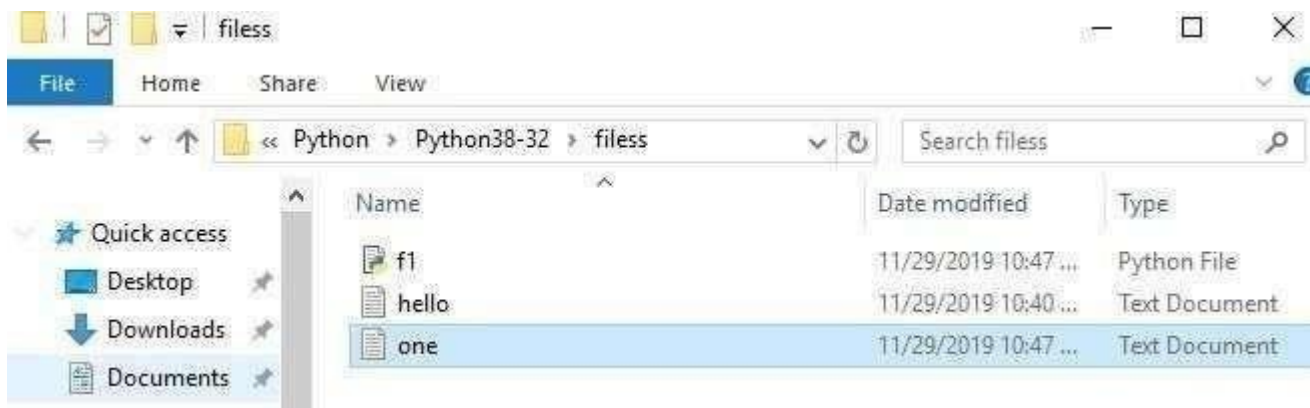
Output:

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

(or)

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

Note: All the program files and text files need to 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

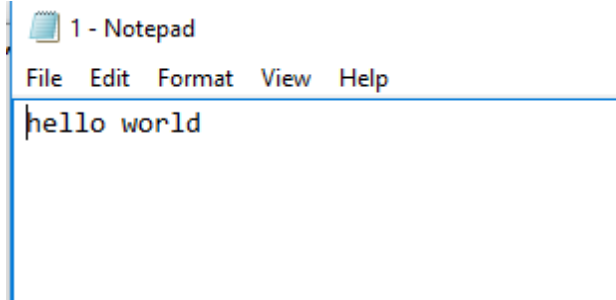
Write a python program to open and write “hello world” into a file?

```
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
```

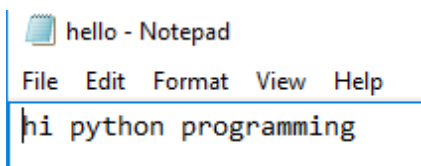
Write a python program to write the content “hi python programming” for the existing file.

```
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

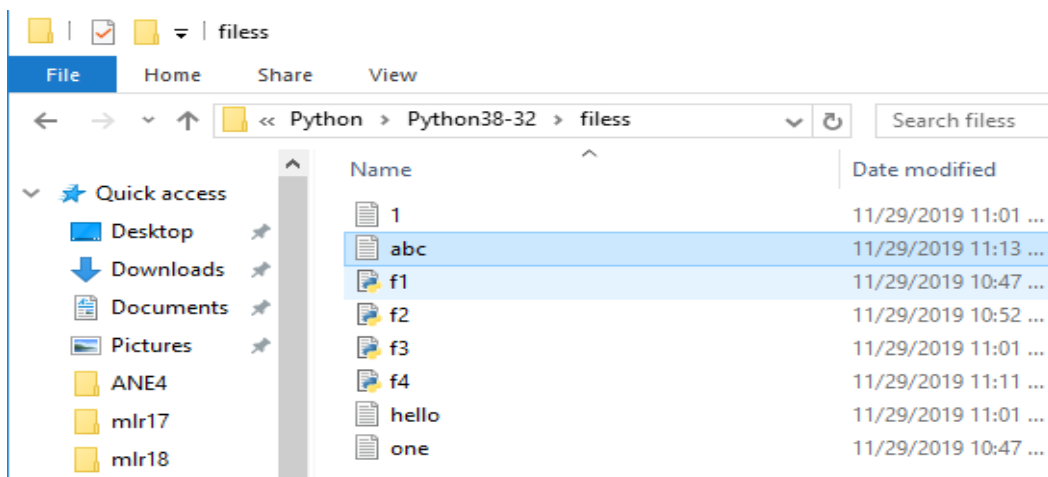
File Edit Format View Help

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 = dict[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