METHODIST

Estd:2008

COLLEGE OF ENGINEERING AND TECHNOLOGY

(Affiliated to Osmania University & Approved by AICTE, New Delhi)

LABORATORY MANUAL

ADVANCED COMPUTER SKILLS LAB

BE III Semester (AICTE Model Curriculum): 2020-21

NAME: ____

ROLL NO:____

BRANCH:________________ **SEM:**__________________________

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERNG

Empower youth- Architects of Future World

METHODIST

Estd:2008 COLLEGE OF ENGINEERING AND TECHNOLOGY

VISION

To produce ethical, socially conscious and innovative professionals who would contribute to sustainable technological development of the society.

MISSION

To impart quality engineering education with latest technological developments and interdisciplinary skills to make students succeed in professional practice.

To encourage research culture among faculty and students by establishing state of art laboratories and exposing them to modern industrial and organizational practices.

To inculcate humane qualities like environmental consciousness, leadership, social values, professional ethics and engage in independent and lifelong learning for sustainable contribution to the society.

METHODIST

Estd:2008 COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LABORATORY MANUAL

ADVANCED COMPUTER SKILLS LAB

Prepared
By
Er. Sandeep Ravikanti,

Assistant Professor.

Estd:2008

METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION & MISSION

VISION

To become a leader in providing Computer Science & Engineering education with emphasis on knowledge and innovation.

MISSION

- To offer flexible programs of study with collaborations to suit industry needs.
- To provide quality education and training through novel pedagogical practices.
- To expedite high performance of excellence in teaching, research and innovations.
- To impart moral, ethical values and education with social responsibility.

METHODIST

Estd:2008 COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM EDUCATIONAL OBJECTIVES

After 3-5 years of graduation, the graduates will be able to

PEO1: Apply technical concepts, Analyze, Synthesize data to Design and create novel products and solutions for the real life problems.

PEO2: Apply the knowledge of Computer Science Engineering to pursue higher education with due consideration to environment and society.

PEO3: Promote collaborative learning and spirit of team work through multidisciplinary projects

PEO4: Engage in life-long learning and develop entrepreneurial skills.

Estd:2008

METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM OUTCOMES

Engineering graduates will be able to:

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the Engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES

At the end of 4 years, Computer Science and Engineering graduates at MCET will be able to:

PSO1: Apply the knowledge of Computer Science and Engineering in various domains like networking and data mining to manage projects in multidisciplinary environments.

PSO2: Develop software applications with open-ended programming environments**.**

PSO3: Design and develop solutions by following standard software engineering principles and implement by using suitable programming languages and platforms

Course Code	Course Title						Core / Elective
PC 253 CS	**Advanced Computer Skills Lab**						**Core**
Prerequisite	Contact Hours per Week				CIE	SEE	Credits
	L	T	D	P			
-	**-**	**-**	**-**	**2**	**25**	**50**	**1**

Course Objectives

- Introducing a new object oriented programming
- Enabling students to learn Big Data, Machine Learning etc.
- Preparing students to cope up with new Market tendencies
- To learn programs in MATLAB environment
- To handle Functions, Polynomials by using MATLAB commands
- Ability to solve any Mathematical functions
- To learn Mathematical Modelling in a new approach
- To plot Graphics (2-D) easily and effectively

Course Outcomes

After completing this course, the student will be able to

1. Implement basic syntax in python.
2. Analyse and implement different kinds of OOP concept in real world problems.
3. Implement MATLAB operations and graphic functions.

List of Programming Exercises:

1. Python Variables, Executing Python from the Command Line, Editing Python Files, Python Reserved Words.
2. Comments, Strings and Numeric Data Types, Simple Input and Output.
3. Control Flow and Syntax, Indenting, if Statement, Relational Operators, Logical Operators, Bit Wise Operators, while Loop, break and continue, for Loop, Lists, Tuples, Sets, Dictionaries.
4. Functions: Passing parameters to a Function, Variable Number of Arguments, Scope, Passing Functions to a Function, Mapping Functions in a Dictionary, Lambda, Modules, Standard Modules.
5. OOP concepts: Classes, File Organization, Special Methods, Inheritance, Polymorphism, Special Characters, Character Classes, Quantifiers, Dot Character, Greedy Matches, Matching at Beginning or End, Match Objects, Compiling Regular Expressions.
6. MATLAB Menus, Toolbars, Computing with MATLAB, Script Files and the Editor/Debugger, MATLAB help System.
7. MATLAB controls: Relational Logical Variables. Conditional Statements: if – else – elseif, switch 2
   10. Loops: for – while – break, continue. User-Defined Functions.
8. Arrays, Matrices and Matrix Operations Debugging MATLAB Programs. Working with Data Files, and Graphing Functions: XY Plots – Sub-plots.

Suggested Readings:

1. Mark Summerfield," Programming in Python
2. A Complete introduction to the Python Language", Addison-Wesley Professional, 2009.
3. Martin C. Brown," PYTHON: The Complete Reference", McGraw-Hill, 2001.
4. W.J. Palm III, Introduction to MATLAB 7 for Engineers, McGraw-Hill International Edition, 2005.
5. Wesley J Chun," Core Python Applications Programming", Prentice Hall, 2012.
6. Allen B Downey," Think Python", O'Reilly, 2012.
7. Stormy Attaway, "MATLAB: A Practical Introduction to Programming and Problem Solving".3rd Edition.

Estd:2008

METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course Outcomes (CO's):

SUBJECT NAME: ADVANCED COMPUTER SKILLS LAB **CODE: PC253CS**

SEMESTER: III

CO No.	Course Outcomes	Taxonomy Level
PC253CS.1	Implement basic syntax in python.	Creating
PC253CS.2	Analyse and implement different kinds of OOP concept in python	Analyzing
PC253CS.3	Implement MATLAB operations and graphic functions	Creating
PC253CS.4	Understand the Numbers, Math functions, Strings, List, Tuples and Dictionaries in Python	Understanding
PC253CS.5	Able to implement Decision Making statements and Functions in python and MATLAB	Creating
PC253CS.6	Able to use understand Object oriented Principles in Python	Understanding

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to starting time), those who come after 5 minutes will not be allowed into the lab.
2. Plan your task properly much before to the commencement, come prepared to the lab with the program / experiment details.
3. Student should enter into the laboratory with:
   a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.
   b. Laboratory Record updated up to the last session experiments.
   c. Formal dress code and Identity card.
4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.
5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.
6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.
7. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.
8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviours with the staff and systems etc., will attract severe punishment.
9. Students must take the permission of the faculty in case of any urgency to go out. If anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.
10. Students should SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

Head of the Department **Principal**

CODE OF CONDUCT FOR THE LABORATORY

- All students must observe the dress code while in the laboratory
- Footwear is NOT allowed
- Foods, drinks and smoking are NOT allowed
- All bags must be left at the indicated place
- The lab timetable must be strictly followed
- Be PUNCTUAL for your laboratory session
- All programs must be completed within the given time
- Noise must be kept to a minimum
- Workspace must be kept clean and tidy at all time
- All students are liable for any damage to system due to their own negligence
- Students are strictly PROHIBITED from taking out any items from the laboratory
- Report immediately to the lab programmer if any damages to equipment

BEFORE LEAVING LAB:

- Arrange all the equipment and chairs properly.
- Turn off / shut down the systems before leaving.
- Please check the laboratory notice board regularly for updates.

Lab In – charge

Estd:2008

METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LIST OF EXPERIMENTS

SI.No.	Name of the Experiment	Date of Experiment	Date of Submission	Page No	Faculty Signature
1.	Python Variables, Executing Python from the Command Line, Editing Python Files, Python Reserved Words			7	
2.	Comments, Strings and Numeric Data Types, Simple Input and output			11	
3.	Control Flow and Syntax, Indenting, if Statement, Relational Operators, Logical Operators, Bit Wise Operators, while Loop, break and continue, for Loop, Lists, Tuples, Sets, Dictionaries			17	
4.	Functions: Passing parameters to a Function, Variable Number of Arguments, Scope, Passing Functions to a Function, Mapping Functions in a Dictionary, Lambda, Modules, Standard Modules			32	

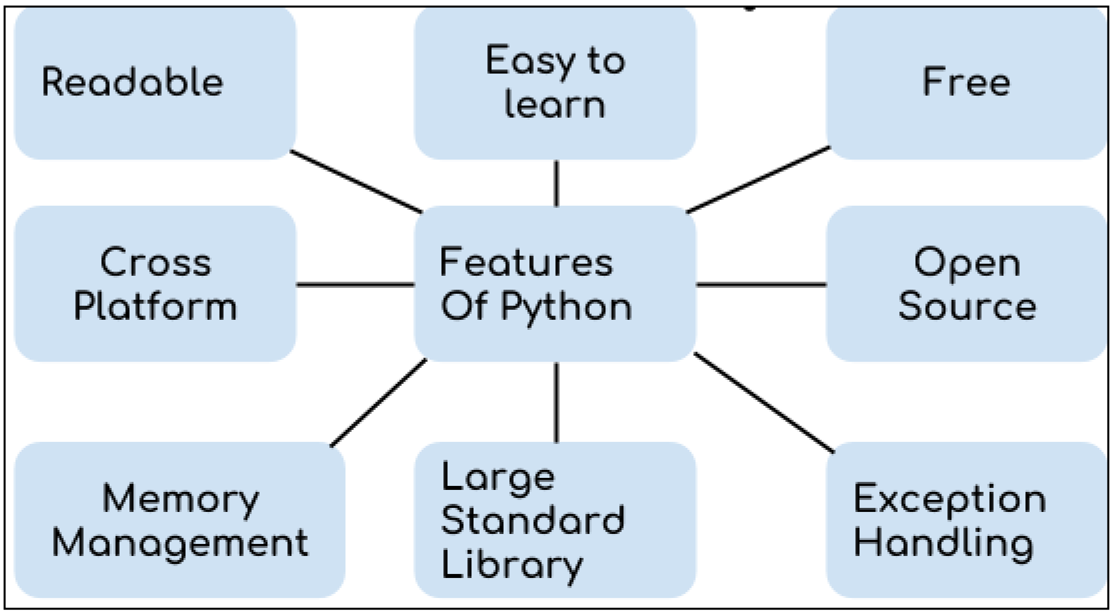
ADDITIONAL EXPERIMENTS

Introduction to Python Programming language

Python is developed by **Guido van Rossum**. Guido van Rossum started implementing Python in 1989. Python is a very simple programming language so even if you are new to programming, you can learn python without facing any issues.

Interesting fact: Python is named after the comedy television show Monty Python's Flying Circus. It is not named after the Python snake.

Features of Python programming language



1. Readable: Python is a very readable language.

2. **Easy to Learn:** Learning python is easy as this is a expressive and high level programming language, which means it is easy to understand the language and thus easy to learn.

3. Cross platform: Python is available and can run on various operating systems such as Mac, Windows, Linux, Unix etc. This makes it a cross platform and portable language.

4. Open Source: Python is a open source programming language.

5. Large standard library: Python comes with a large standard library that has some handy codes and functions which we can use while writing code in Python.

6. Free: Python is free to download and use. This means you can download it for free and use it in your application. See: Open Source Python License. Python is an example of a FLOSS (Free/Libre Open Source Software), which means you can freely distribute copies of this software, read its source code and modify it.

7. Supports exception handling: If you are new, you may wonder what is an exception? An exception is an event that can occur during program exception and can disrupt the normal flow of program. Python supports exception handling which means we can write less error prone code and can test various scenarios that can cause an exception later on.

8. Advanced features: Supports generators and list comprehensions. We will cover these features later.

9. Automatic memory management: Python supports automatic memory management which means the memory is cleared and freed automatically. You do not have to bother clearing the memory.

What Can You Do with Python?

You may be wondering what all are the applications of Python. There are so many applications of Python, here are some of the them.

1. Web development – Web framework like Django and Flask are based on Python. They help you write server side code which helps you manage database, write backend programming logic, mapping urls etc.
2. Machine learning – There are many machine learning applications written in Python. Machine learning is a way to write a logic so that a machine can learn and solve a particular problem on its own. For example, products recommendation in websites like Amazon, Flipkart, eBay etc. is a machine learning algorithm that recognises user's interest. Face recognition and Voice recognition in your phone is another example of machine learning.
3. Data Analysis – Data analysis and data visualisation in form of charts can also be developed using Python.
4. Scripting – Scripting is writing small programs to automate simple tasks such as sending automated response emails etc. Such type of applications can also be written in Python programming language.
5. Game development – You can develop games using Python.
6. You can develop Embedded applications in Python.
7. Desktop applications – You can develop desktop application in Python using library like TKinter or QT.

Python Programming Characteristics

- It provides rich data types and easier to read syntax than any other programming languages
- It is a platform independent scripted language with full access to operating system API's
- Compared to other programming languages, it allows more run-time flexibility
- It includes the basic text manipulation facilities of Perl and Awk

- A module in Python may have one or more classes and free functions
- Libraries in Pythons are cross-platform compatible with Linux, Macintosh, and Windows
- For building large applications, Python can be compiled to byte-code
- Python supports functional and structured programming as well as OOP
- It supports interactive mode that allows interacting Testing and debugging of snippets of code
- In Python, since there is no compilation step, editing, debugging and testing is fast.

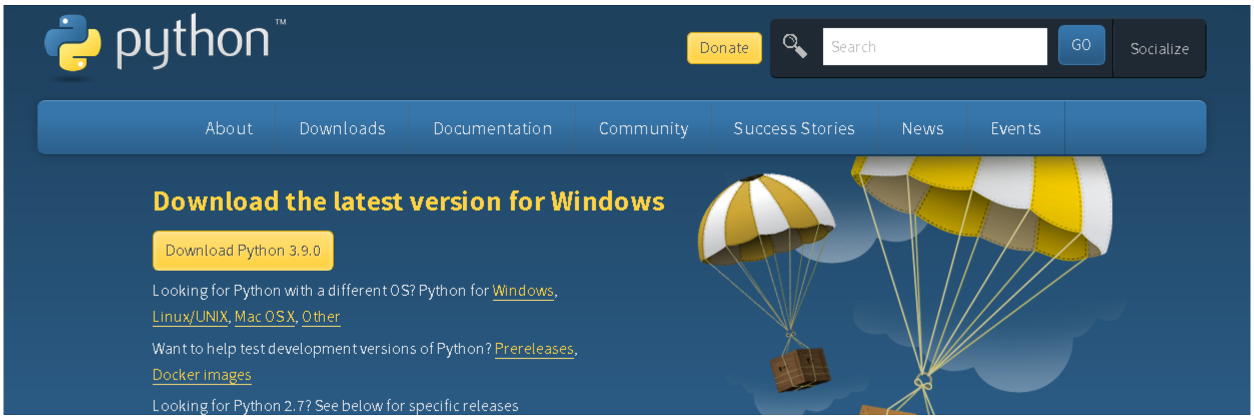
Python is commonly used by programmers to:

- Program video games
- Build Artificial Intelligence algorithms
- Program various scientific programs such as statistical models

How to install Python

Python installation is pretty simple, you can install it on any operating system such as Windows, Mac OS X, Ubuntu etc.

To install the Python on your operating system, go to this link: https://www.python.org/downloads/. You will see a screen like this.



Download the latest version for Windows

Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other

Want to help test development versions of Python? Prereleases, Docker images

Looking for Python 2.7? See below for specific releases

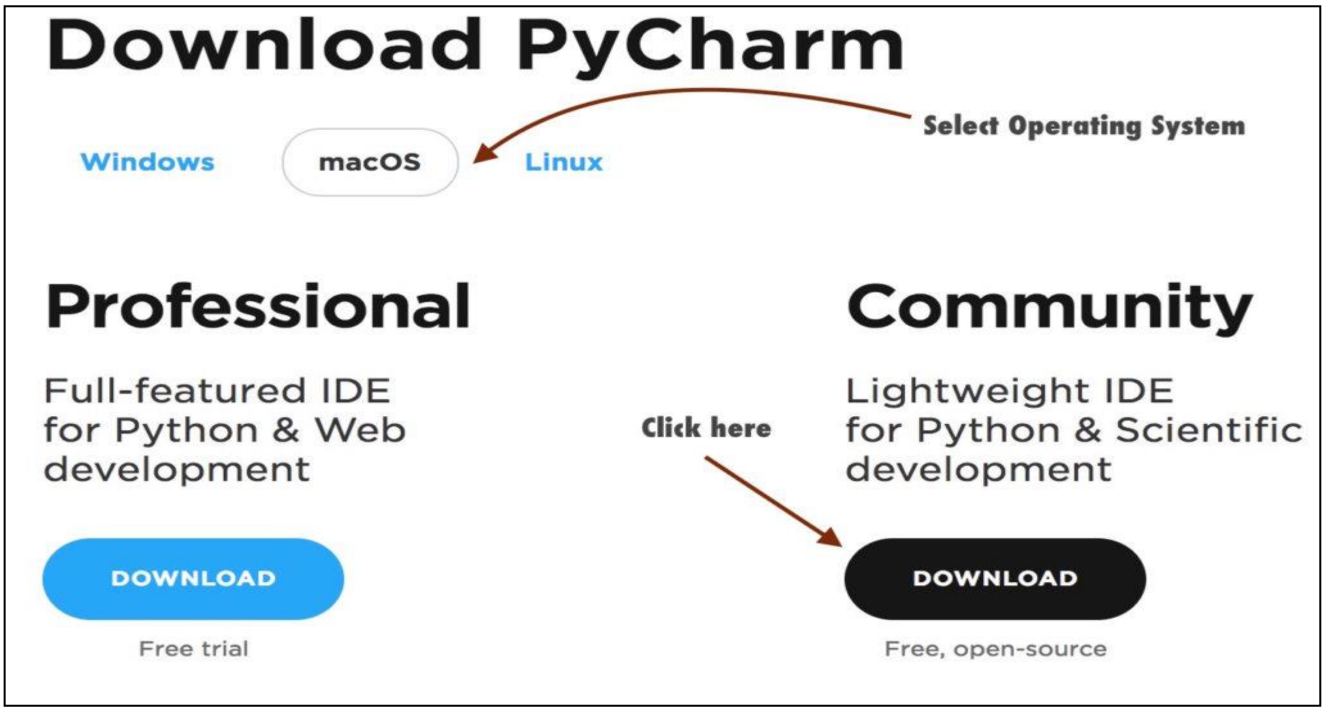
This is the **official Python website** and it will detect the operating system and based on that it would recommend you to download Python. Here I am using Mac OS X so it gave me the download options for Python 2 and Python 3 for Mac OS X. I would recommend you to download the **latest version of Python 3** (Python 3.9.0 in the screenshot).

Installation steps are pretty simple. You just have to accept the agreement and finish the installation. If you are using Mac OS X then you may notice that Python is already installed in your system,however it would most likely be a Python 2 so you can download the latest Python 3 version from here and install it in your system.

Install PyCharm Python IDE in Windows, Mac OS X, Linux/Unix

PyCharm Installation

1. Go to this link: https://www.jetbrains.com/pycharm/download/ and download the community edition.



2. Install the downloaded file.

Mac: Double click the .dmg file and drag PyCharm to the Application Folder.

Windows: Double click the .exe file and follow the installation steps for the default PyCharm installation.

Launch PyCharm

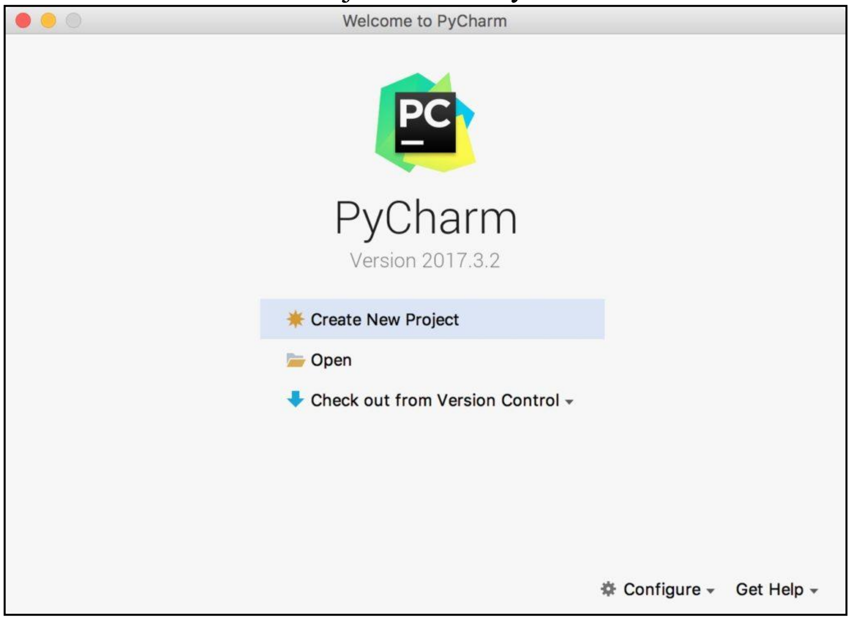
Mac: Go to the **Applications** folder and click on the PyCharm icon. Alternatively, you can drag the icon to your dock to open the IDE quickly by clicking on the icon in dock.

Windows: If you have followed the default installation process then you can see the PyCharm icon on your desktop. If you don't find the icon then go to the PyCharm folder – C:\Program Files (x86)\JetBrains\PyCharm 2017.1\bin (the path may be different for your system) and click on the PyCharm.exe file to launch the IDE

First Python Project in PyCharm IDE

Creating Python Project in PyCharm

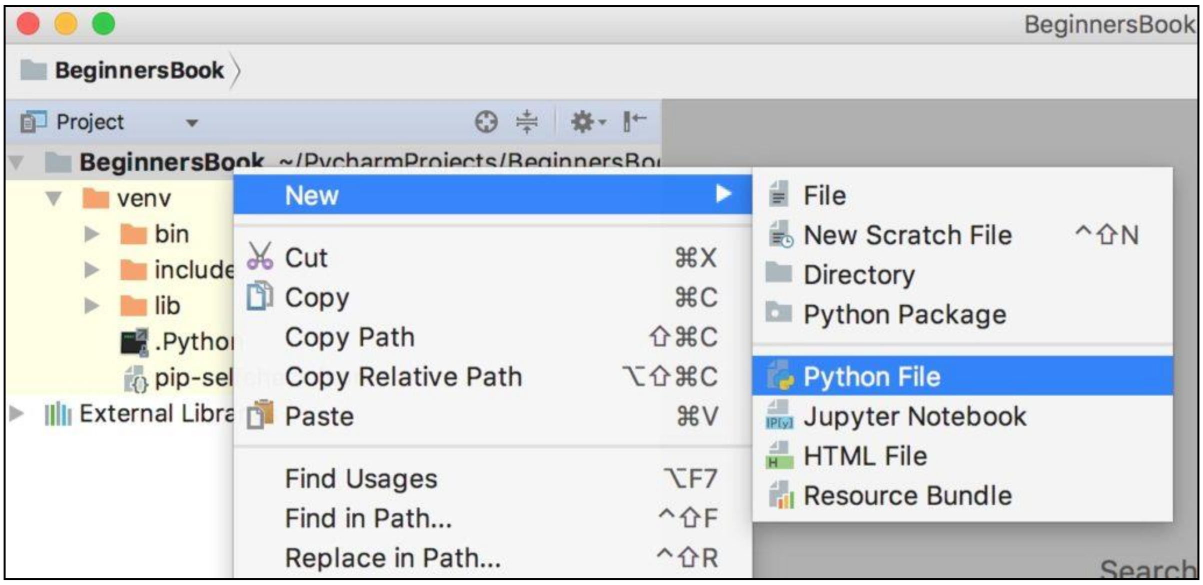
1. Click “Create New Project” in the PyCharm welcome screen.



2. Give a meaningful project name.

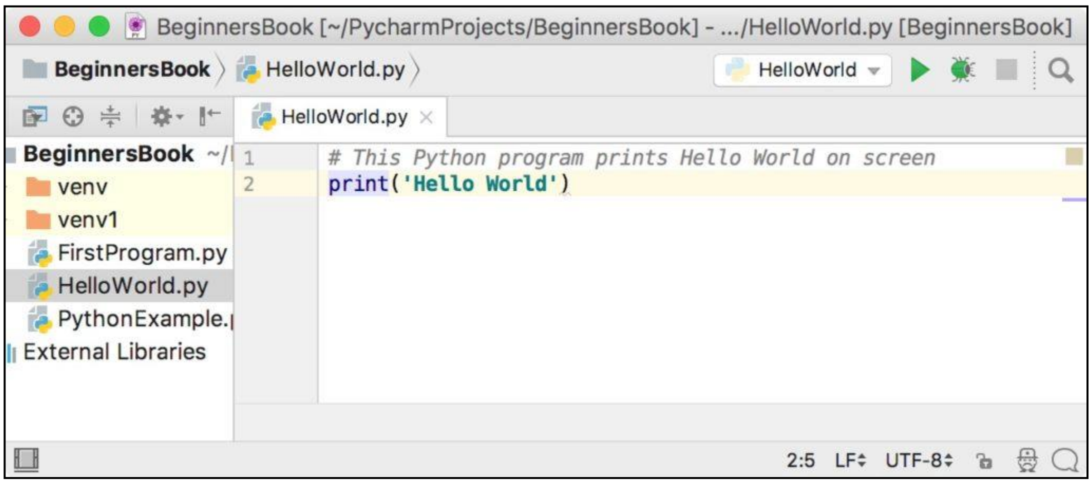
Writing and running your first Python Program

1. Now that we have created a Python project, it’s time to create a Python program file to write and run our first Python program. To create a file, right click on the folder name > New > Python File (as shown in the screenshot). Give the file name as “HelloWorld” and click ok.



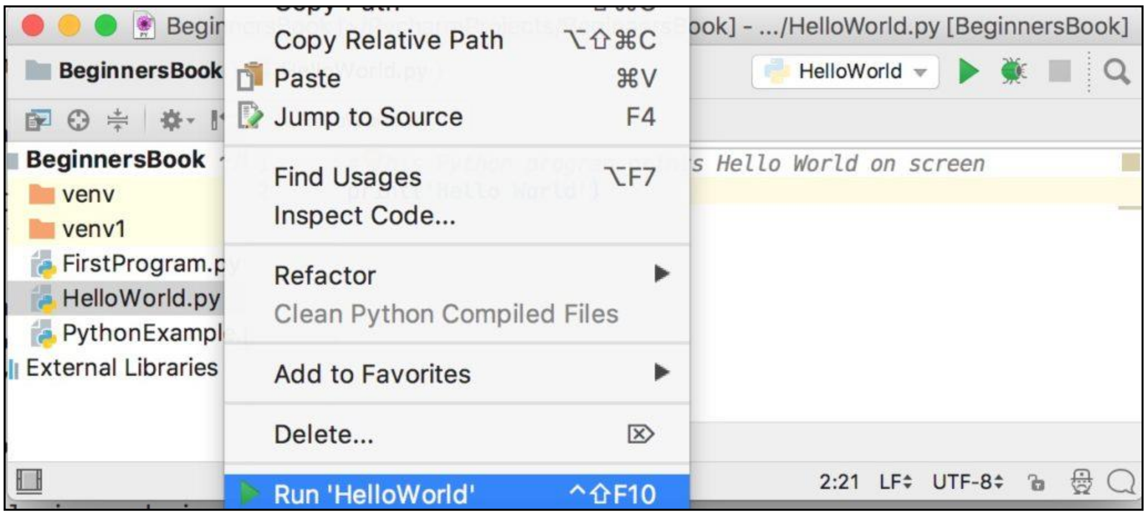
2. Write the following code in the file.

```
# This Python program prints Hello World on screen
print('Hello World')
```

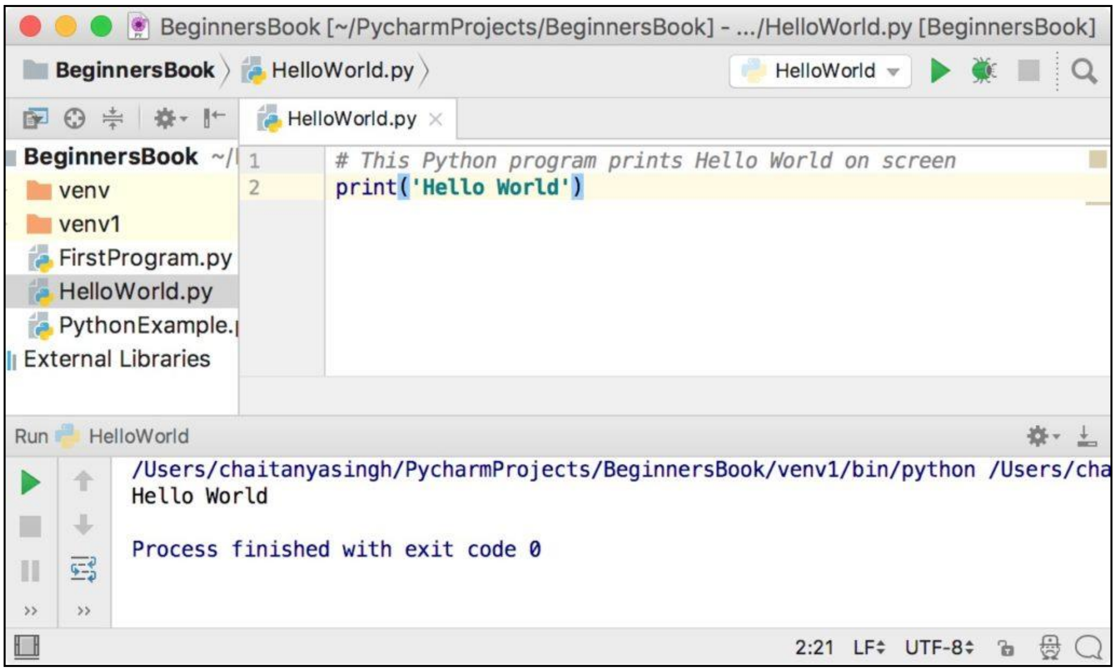


```
# This Python program prints Hello World on screen
print('Hello World')
```

3. Lets run the code. Right click on the HelloWorld.py file (or the name you have given while creating Python file) in the left sidebar and click on 'Run HelloWorld'.



4. You can see the output of the program at the bottom of the screen.


BeginnersBook [~/PycharmProjects/BeginnersBook] - .../HelloWorld.py [BeginnersBook]
Run HelloWorld
2:21 LF UTF-8

```
# This Python program prints Hello World on screen
print('Hello World')
```

```
/Users/chaitanyasingh/PycharmProjects/BeginnersBook/venv1/bin/python /Users/cha
Hello World

Process finished with exit code 0
```

PROGRAM 1

AIM : Python Variables, Executing Python from the Command Line, Editing Python Files, Python Reserved Words.

A Python variable is a reserved memory location to store values. In other words, a variable in a python program gives data to the computer for processing.

How to Declare and use a Variable

We will declare variable "a" and print it.

```
a=100
print a
```

Re-declare a Variable

You can re-declare the variable even after you have declared it once.

Here we have variable initialized to f=0.

Later, we re-assign the variable f to value "hello"

example:

```
# Declare a variable and initialize it
f = 0
print f
# re-declaring the variable works
f = 'hello'
print f
```

Delete a variable

You can also delete variable using the command **del** "variable name".

Example:

```
f = 11;
print(f)
del f
print(f)
```

Multiple Assignment

Python allows us to assign a value to multiple variables in a single statement which is also known as multiple assignment.

We can apply multiple assignments in two ways either by assigning a single value to multiple variables or assigning multiple values to multiple variables.

Assigning single value to multiple variables

```
    x=y=z=50
    print iple
```

```
print y
print z
```

Assigning multiple values to multiple variables:

```
a,b,c=5,10,15
print a
print b
print c
```

Running Python on your OS

Run the Python command-line interpreter, under your OS of choice,

Windows

Open Command line: Start menu -> Run and type cmd

Type: C:\python27\python.exe

Note: This is the default path for Python 2.7. If you are using a computer where Python is not installed in this path, change the path accordingly.

Mac OS X

Open Command line: Finder -> Go menu -> Applications -> Terminal

Type: python

Linux

Open a command prompt (e.g. xterm)

Type: python

Using the IDLE interpreter:

Windows

Start IDLE: Start menu -> Programming -> Python 2.7 -> IDLE (Python GUI)

(alternatively, press Start menu and start typing "idle" and click when it appears.)

Mac OS X

Start IDLE: Finder -> Go menu -> Applications -> IDLE

(alternatively, it may be found under MacPorts. Or use SpotLight to search for "IDLE".)

Linux

Open a command prompt (e.g. xterm)

Type: idle

Run any plain text editor

Windows

Start Notepad: Start menu -> Accessories -> Notepad

Mac OS X

Start TextEdit: Finder -> Go menu -> Applications -> TextEdit

Alternative: Start TextWrangler: Finder -> Go menu -> Applications -> TextWrangler

Linux

Start nano: open a command prompt and type nano

Alternative: Start gedit: open a command prompt and type gedit

---

Run your script

Suppose your script is in "Z:\code\hw01\script.py" or ~/code/hw01/script.py on Unix-like systems.

Windows

Open Command line: Start menu -> Run and type cmd

Type: C:\python27\python.exe Z:\code\hw01\script.py

Or if your system is configured correctly, you can drag and drop your script from Explorer onto the Command Line window and press enter.

Mac OS X

Open Command line: Finder -> Go menu -> Applications -> Terminal

Type: python ~/code/hw01/script.py

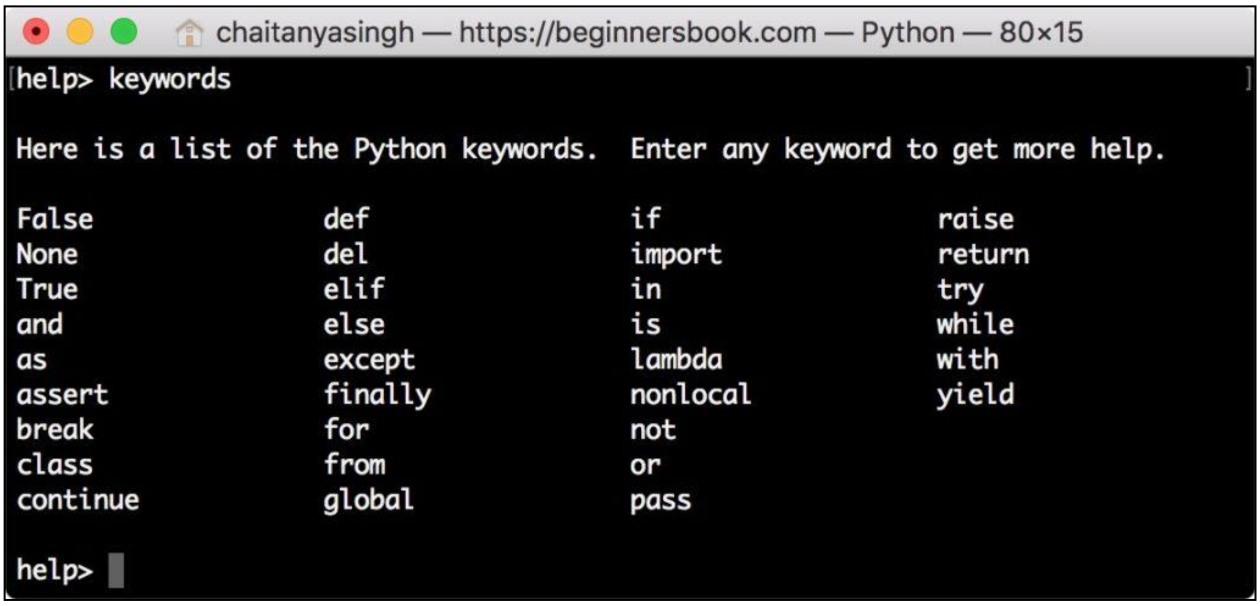
Linux

Open a command prompt (e.g. xterm)

Type: python ~/code/hw01/script.py

What is a Python Keyword?

A python keyword is a reserved word which you can’t use as a name of your variable, class, function etc. These keywords have a special meaning and they are used for special purposes in Python programming language. For example – Python keyword “while” is used for while loop thus you can’t name a variable with the name “while” else it may cause compilation error. There are total 33 keywords in Python 3.7. To get the keywords list on your operating system, open command prompt (terminal on Mac OS) and type “Python” and hit enter. After that type help() and hit enter. Type keywords to get the list of the keywords for the current python version running on your operating system.



```
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

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

help> 
```

PROGRAM 2

AIM : Comments, Strings and Numeric Data Types, Simple Input and Output.

Python comments

A comment is text that doesn’t affect the outcome of a code, it is just a **piece of text** to let someone know what you have done in a program or what is being done in a block of code. This is especially helpful when someone else has written a code and you are analysing it for bug fixing or making a change in logic, **by reading a comment you can understand the purpose of code much faster then by just going through the actual code**.

Types of Comments in Python

There are two types of comments in Python.

1. Single line comment
2. Multiple line comment

Single line comment

In python we use # special character to start the comment. Lets take few examples to understand the usage.

```
# This is just a comment. Anything written here is ignored by Python
```

Multi-line comment:

To have a multi-line comment in Python, we use triple single quotes at the beginning and at the end of the comment, as shown below.

```
'''
This is a
multi-line
comment
'''
```

# character inside quotes

When # character is encountered inside quotes, it is not considered as comment. For example:

```
print("#this is not a comment")
```

Output:

```
#this is not a comment
```

Python Data Types

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

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

1. <u>Numbers</u>

2. String
3. List
4. Tuple
5. Dictionary

In this section of the tutorial, we will give a brief introduction of the above data types. We will discuss each one of them in detail later in this tutorial.

Numbers

Number stores numeric values. Python creates Number objects when a number is assigned to a variable. For example;

```
a = 3 , b = 5  #a and b are number objects
```

Python supports 4 types of numeric data.

1. int (signed integers like 10, 2, 29, etc.)
2. long (long integers used for a higher range of values like 908090800L, -0x1929292L, etc.)
3. float (float is used to store floating point numbers like 1.9, 9.902, 15.2, etc.)
4. complex (complex numbers like 2.14j, 2.0 + 2.3j, etc.)

Python allows us to use a lower-case L to be used with long integers. However, we must always use an upper-case L to avoid confusion.

A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts respectively).

String

The string can be defined as the sequence of characters represented in the quotation marks. In python, we can use single, double, or triple quotes to define a string.

String handling in python is a straightforward task since there are various inbuilt functions and operators provided.

In the case of string handling, the operator + is used to concatenate two strings as the operation *"hello"+" python"* returns *"hello python"*.

The operator * is known as repetition operator as the operation "Python " *2 returns "Python Python ".

The following example illustrates the string handling in python.

```
str1 = 'hello javatpoint' #string str1
str2 = ' how are you' #string str2
print (str1[0:2]) #printing first two character using slice operator
print (str1[4]) #printing 4th character of the string
print (str1*2) #printing the string twice
print (str1 + str2) #printing the concatenation of str1 and str2
```

Output:

```
he
o
hello javatpointhello javatpoint
hello javatpoint how are you
```

List

Lists are similar to arrays in C. However; the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].

We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

Consider the following example.

```
l  = [1, "hi", "python", 2]
print (l[3:]);
print (l[0:2]);
print (l);
print (l + l);
print (l * 3);
```

Output:

```
[2]
[1, 'hi']
[1, 'hi', 'python', 2]
[1, 'hi', 'python', 2, 1, 'hi', 'python', 2]
[1, 'hi', 'python', 2, 1, 'hi', 'python', 2, 1, 'hi', 'python', 2]
```

Tuple

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Let's see a simple example of the tuple.

```
t  = ("hi", "python", 2)
print (t[1:]);
print (t[0:1]);
print (t);
print (t + t);
print (t * 3);
print (type(t))
t[2] = "hi";
```

Output:

```
('python', 2)
('hi',)
('hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2, 'hi', 'python', 2)
<type 'tuple'>
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

Dictionary

Dictionary is an ordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type whereas value is an arbitrary Python object.

The items in the dictionary are separated with the comma and enclosed in the curly braces {}.

Consider the following example.

```
d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'};
print("1st name is "+d[1]);
print("2nd name is "+ d[4]);
print (d);
print (d.keys());
print (d.values());
```

Output:

```
1st name is Jimmy
2nd name is mike
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
[1, 2, 3, 4]
['Jimmy', 'Alex', 'john', 'mike']
```

A Program needs to interact with the user to accomplish the desired task; this is done using

Input-Output facility. Input means the data entered by the user of the program. In python, we have input() and raw_input ( ) function available for **Input**.

1) input()

Syntax:

input (expression)

If prompt is present, it is displayed on monitor, after which the user can provide data from keyboard. Input takes whatever is typed from the keyboard and evaluates it. As the input provided is evaluated, it expects valid python expression. If the input provided is not correct then either syntax error or exception is raised by python.

Example:

```
>>>x= input ("Enter data:")
Enter data:  34.78
>>>print(x)


34.78
```

2) raw_input()

Syntax:

raw_input (expression)

This input method fairly works in older versions (like 2.x).

If prompt is present, it is displayed on the monitor after which user can provide the data from keyboard. The function takes exactly what is typed from keyboard, convert it to string and then return it to the variable on LHS of '='.

Example: In interactive mode

```
>>>x=raw_input ('Enter your name: ')


Enter your name: ABC
```

x is a variable which will get the string (ABC), typed by user during the execution of program. Typing of data for the raw_input function is terminated by enter key.

We can use raw_input() to enter numeric data also. In that case we typecast, i.e., change the data type using function, the string data accepted from user to appropriate Numeric type.

Example:

```
>>>y=int(raw_input("Enter your roll no."))

Enter your roll no. 5
```

It will convert the accepted string i.e., 5 to integer before assigning it to 'y'.

Print statement

Now how can we display the input values on screen? You might think that all we have to do is just type the variable and press the Enter key. Well, it is true that we have been doing this the whole time, but this only works when you are working on IDLE.

While creating real world python programs you have to write statements that outputs the strings, or numbers explicitly.

We use the print statement to do so

Syntax:

```
print (expression/constant/variable)
```

Print evaluates the expression before printing it on the monitor. Print statement outputs an entire (complete) line and then goes to next line for subsequent output (s). To print more than one item on a single line, comma (,) may be used.

Example:

```
>>> print ("Hello")
Hello
>>> print (5.5)
5.5
>>> print (4+6)
10
```

PROGRAM 3:

AIM: Control Flow and Syntax, Indenting, if Statement, Relational Operators, Logical Operators, Bit Wise Operators, while Loop, break and continue, for Loop

Python Indentation

Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:
  print("Five is greater than two!")
```

Python will give you an error if you skip the indentation:

Example

Syntax Error:

```
if 5 > 2:
print("Five is greater than two!")
```

The number of spaces is up to you as a programmer, but it has to be at least one.

Example

```
if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
        print("Five is greater than two!")
```

Python 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

Relational Operators: Relational operators compares the values. It either returns True or False according to the condition.

```
# Examples of Relational Operators
a = 13
b = 33

# a > b is False
print(a > b)

# a < b is True
print(a < b)

# a == b is False
print(a == b)

# a != b is True
print(a != b)

# a >= b is False
print(a >= b)

# a <= b is True
print(a <= b)
```

Output:

```
False
True
False
True
False
True
```

Logical Operators

Logical operators perform Logical AND**,** Logical OR **and** Logical NOT **operations.**

```
# Examples of Logical Operator
a = True
b = False

# Print a and b is False
```

```
print(a and b)

# Print a or b is True
print(a or b)

# Print not a is False
print(not a)
```

Output:

```
False
True
False
```

Bit Wise Operators

Bitwise operators acts on bits and performs bit by bit operation

```
# Examples of Bitwise operators
a = 10
b = 4

# Print bitwise AND operation
print(a & b)

# Print bitwise OR operation
print(a | b)

# Print bitwise NOT operation
print(~a)

# print bitwise XOR operation
print(a ^ b)

# print bitwise right shift operation
print(a >> 2)

# print bitwise left shift operation
print(a << 2)
```

Output:

```
0
14
-11
14
2
40
```

3.A] Python Program to Reverse a Given Number

AIM: This is a Python Program to reverse a given number.

Problem Description

The program takes a number and reverses it.

Problem Solution

1. Take the value of the integer and store in a variable.
2. Using a while loop, get each digit of the number and store the reversed number in another variable.
3. Print the reverse of the number.
4. Exit.

Program/Source Code

Here is the source code of the Python Program to reverse a given number.

```
 n=int(input("Enter number: "))
rev=0
while(n>0):
    dig=n%10
    rev=rev*10+dig
    n=n//10
print("Reverse of the number:",rev)
```

Program Explanation

1. User must first enter the value and store it in a variable n
2. The while loop is used and the last digit of the number is obtained by using the modulus operator.
3. The last digit is then stored at the one's place, second last at the ten's place and so on.
4. The last digit is then removed by truly dividing the number with 10.
5. This loop terminates when the value of the number is 0.
6. The reverse of the number is then printed.

Runtime Test Cases

```
 Case 1:
Enter number: 124
Reverse of the number: 421
 Case 2:
Enter number: 4538
Reverse of the number: 8354
```

3.B] Python Program to Exchange the Values of Two Numbers Without Using a Temporary Variable

AIM: This is a Python Program to exchange the values of two numbers without using a temporary variable.

Problem Description

The program takes both the values from the user and swaps them without using temporary variable.

Problem Solution

1. Take the values of both the elements from the user.
2. Store the values in separate variables.
3. Add both the variables and store it in the first variable.
4. Subtract the second variable from the first and store it in the second variable.
5. Then, subtract the first variable from the second variable and store it in the first variable.
6. Print the swapped values.
7. Exit.

Program/Source Code

Here is source code of the Python Program to exchange the values of two numbers without using a temporary variable. The program output is also shown below.

```
a=int(input("Enter value of first variable: "))
b=int(input("Enter value of second variable: "))
a=a+b
b=a-b
a=a-b
print("a is:",a," b is:",b)
```

Program Explanation

1. User must first enter the values for both the elements.
2. The first element is assigned the sum of the first two elements.

3. Second element is assigned the difference between the sum in the first variable and the second variable, which is basically the first element.
4. Later the first element is assigned the difference between the sum in the variable and the second variable, which is the second element.
5. Then the swapped values are printed.

Runtime Test Cases

```
 Case 1
Enter value of first variable: 3
Enter value of second variable: 5
a is: 5  b is: 3
 Case 2
Enter value of first variable: 56
Enter value of second variable: 25
a is: 25  b is: 56
```

3.C] Python Program to Find the Sum of Digits in a Number

AIM: This is a Python Program to find the sum of digits in a number.

Problem Description

The program takes in a number and finds the sum of digits in a number.

Problem Solution

1. Take the value of the integer and store in a variable.
2. Using a while loop, get each digit of the number and add the digits to a variable.
3. Print the sum of the digits of the number.
4. Exit.

Program/Source Code

Here is the source code of the Python Program to find the sum of digits in a number. The program output is also shown below.

```
 n=int(input("Enter a number:"))
tot=0
while(n>0):
    dig=n%10
    tot=tot+dig
    n=n//10
print("The total sum of digits is:",tot)
```

Program Explanation

1. User must first enter the value and store it in a variable.
2. The while loop is used and the last digit of the number is obtained by using the modulus operator.
3. The digit is added to another variable each time the loop is executed.
4. This loop terminates when the value of the number is 0.
5. The total sum of the number is then printed.

Runtime Test Cases

```
 Case 1:
Enter a number:1892
The total sum of digits is: 20
 Case 2:
Enter a number:157
The total sum of digits is: 13
```

3.D] Python Program to check if a Number is a Palindrome

AIM: This is a Python Program to check whether a given number is a palindrome.

Problem Description

The program takes a number and checks whether it is a palindrome or not.

Problem Solution

1. Take the value of the integer and store in a variable.
2. Transfer the value of the integer into another temporary variable.
3. Using a while loop, get each digit of the number and store the reversed number in another variable.
4. Check if the reverse of the number is equal to the one in the temporary variable.
5. Print the final result.
6. Exit.

Program/Source Code

Here is source code of the Python Program to check whether a given number is a palindrome. The program output is also shown below.

```
n=int(input("Enter number:"))
temp=n
rev=0
while(n>0):
    dig=n%10
    rev=rev*10+dig
    n=n//10
if(temp==rev):
    print("The number is a palindrome!")
else:
    print("The number isn't a palindrome!")
```

Program Explanation

1. User must first enter the value of the integer and store it in a variable.
2. The value of the integer is then stored in another temporary variable.
3. The while loop is used and the last digit of the number is obtained by using the modulus operator.
4. The last digit is then stored at the one's place, second last at the ten's place and so on.
5. The last digit is then removed by truly dividing the number with 10.
6. This loop terminates when the value of the number is 0.
7. The reverse of the number is then compared with the integer value stored in the temporary variable.
8. If both are equal, the number is a palindrome.
9. If both aren't equal, the number isn't a palindrome.
10. The final result is then printed.

Runtime Test Cases

```
 Case 1
Enter number: 121
The number is a palindrome!
 Case 2
Enter number: 567
The number isn't a palindrome!
```

3.E] Python Program to Find the LCM of Two Numbers

This is a Python Program to find the LCM of two numbers.

Problem Description

The program takes two numbers and prints the LCM of two numbers.

Problem Solution

1. Take in both the integers and store it in separate variables.
2. Find which of the integer among the two is smaller and store it in a separate variable.
3. Use a while loop whose condition is always True until break is used.
4. Use an if statement to check if both the numbers are divisible by the minimum number and increment otherwise.

5. Print the final LCM.
6. Exit.

Program/Source Code

Here is source code of the Python Program to find the LCM of two numbers. The program output is also shown below.

```
a=int(input("Enter the first number:"))
b=int(input("Enter the second number:"))
if(a>b):
    min1=a
else:
    min1=b
while(1):
    if(min1%a==0 and min1%b==0):
        print("LCM is:",min1)
        break
    min1=min1+1
```

Program Explanation

1. User must enter both the numbers and store it in separate variables.
2. An if statement is used to find out which of the numbers is smaller and store in a minimum variable.
3. Then a while loop is used whose condition is always true (or 1) unless break is used.
4. Then an if statement within the loop is used to check whether the value in the minimum variable is divisible by both the numbers.
5. If it is divisible, break statement breaks out of the loop.
6. If it is not divisible, the value in the minimum variable is incremented.
7. The final LCM is printed.

Runtime Test Cases

```
 Case 1:
Enter the first number:5
Enter the second number:3
LCM is: 15
 Case 2:
Enter the first number:15
Enter the second number:20
LCM is: 60
```

3.F] Python Program to Find the GCD of Two Numbers

This is a Python Program to find the GCD of two numbers.

Problem Description

The program takes two numbers and prints the GCD of two numbers.

Problem Solution

1. Import the fractions module.
2. Take in both the integers and store it in separate variables.
3. Use the in-built function to find the GCD of both the numbers.
4. Print the GCD.
5. Exit.

Program/Source Code

Here is source code of the Python Program to find the GCD of two numbers. The program output is also shown below.

```python
import fractions
a=int(input("Enter the first number:"))
b=int(input("Enter the second number:"))
print("The GCD of the two numbers is",fractions.gcd(a,b))
```

Program Explanation

1. Import the fractions module.
2. User must enter both the numbers and store it in separate variables.
3. The in-built function of fractions.gcd(a,b) is used where a and b are the variables containing the integer values.
4. The final GCD is printed.

Runtime Test Cases

```
 Case 1:
Enter the first number:15
Enter the second number:5
The GCD of the two numbers is 5
 Case 2:
Enter the first number:105
Enter the second number:222
The GCD of the two numbers is 3
```

3.G]Python Program to Check if a Number is a Prime Number

This is a Python Program to check if a number is a prime number.

Problem Description

The program takes in a number and checks if it is a prime number.

Problem Solution

1. Take in the number to be checked and store it in a variable.
2. Initialize the count variable to 0.
3. Let the for loop range from 2 to half of the number (excluding 1 and the number itself).
4. Then find the number of divisors using the if statement and increment the count variable each time.
5. If the number of divisors is lesser than or equal to 0, the number is prime.
6. Print the final result.
5. Exit.

Program/Source Code

Here is source code of the Python Program to check if a number is a prime number. The program output is also shown below.

```
a=int(input("Enter number: "))
k=0
for i in range(2,a//2+1):
    if(a%i==0):
        k=k+1
if(k<=0):
    print("Number is prime")
else:
    print("Number isn't prime")
```

Program Explanation

1. User must enter the number to be checked and store it in a different variable.
2. The count variable is first initialized to 0.
3. The for loop ranges from 2 to the half of the number so 1 and the number itself aren't counted as divisors.
4. The if statement then checks for the divisors of the number if the remainder is equal to 0.
5. The count variable counts the number of divisors and if the count is lesser or equal to 0, the number is a prime number.
6. If the count is greater than 0, the number isn't prime.
7. The final result is printed.

Runtime Test Cases

```
 Case 1:
Enter number: 7
Number is prime
 Case 2:
Enter number: 35
Number isn't prime
```

3.H] Python Program to Find the Largest Number in a List

This is a Python Program to find the largest number in a list.

Problem Description

The program takes a list and prints the largest number in the list.

Problem Solution

1. Take in the number of elements and store it in a variable.
2. Take in the elements of the list one by one.
3. Sort the list in ascending order.
4. Print the last element of the list.
5. Exit.

Program/Source Code

Here is source code of the Python Program to find the largest number in a list. The program output is also shown below.

```
a=[]
n=int(input("Enter number of elements:"))
for i in range(1,n+1):
    b=int(input("Enter element:"))
    a.append(b)
a.sort()
print("Largest element is:",a[n-1])
```

Program Explanation

1. User must enter the number of elements and store it in a variable.
2. User must then enter the elements of the list one by one using a for loop and store it in a list.
3. The list should then be sorted.
4. Then the last element of the list is printed which is also the largest element of the list.

Runtime Test Cases

```
 Case 1:
```

```
Enter number of elements:3
Enter element:23
Enter element:567
Enter element:3
Largest element is: 567
 Case 2:
Enter number of elements:4
Enter element:34
Enter element:56
Enter element:24
Enter element:54
Largest element is: 56
```

3.I] Python Program to find the factorial of a number

This is a Python Program to find the factorial of a number without using recursion.

Problem Description

The program takes a number and finds the factorial of that number without using recursion.

Problem Solution

1. Take a number from the user.
2. Initialize a factorial variable to 1.
3. Use a while loop to multiply the number to the factorial variable and then decrement the number.
4. Continue this till the value of the number is greater than 0.
5. Print the factorial of the number.
6. Exit.

Program/Source Code

Here is source code of the Python Program to find the factorial of a number without using recursion. The program output is also shown below.

```
n=int(input("Enter number:"))
fact=1
while(n>0):
    fact=fact*n
    n=n-1
print("Factorial of the number is: ")
print(fact)
```

Program Explanation

1. User must enter a number.
2. A factorial variable is initialized to 1.
3. A while loop is used to multiply the number to the factorial variable and then the number is decremented each time.
4. This continues till the value of the number is greater than 0.
5. The factorial of the number is printed.

Runtime Test Cases

```
 Case 1:
Enter number:5
Factorial of the number is:
120
```

3.J] Python Program to Create a List of Tuples with the First Element as the Number and Second Element as the Square of the Number

This is a Python Program to create a list of tuples with the first element as the number and the second element as the square of the number.

Problem Description

The program takes a range and creates a list of tuples within that range with the first element as the number and the second element as the square of the number.

Problem Solution

1. Take the upper and lower range for the numbers from the user.
2. A list of tuples must be created using list comprehension where the first element is the number within the range and the second element is the square of the number.
3. This list of tuples is then printed.
4. Exit.

Program/Source Code

Here is source code of the Python Program to create a list of tuples with the first element as the number and the second element as the square of the number. The program output is also shown below.

```
l_range=int(input("Enter the lower range:"))
u_range=int(input("Enter the upper range:"))
a=[(x,x**2) for x in range(l_range,u_range+1)]
print(a)
```

Program Explanation

1. User must enter the upper and lower range for the numbers.
2. List comprehension must be used to create a list of tuples where the first number is the number itself from the given range and the second element is a square of the first number.
3. The list of tuples which is created is printed.

Runtime Test Cases

Case 1:

```
Enter the lower range:1
Enter the upper range:4
[(1, 1), (2, 4), (3, 9), (4, 16)]
```

Case 2:

```
Enter the lower range:45
Enter the upper range:49
[(45, 2025), (46, 2116), (47, 2209), (48, 2304), (49, 2401)]
```

PROGRAM 4

AIM : Functions: Passing parameters to a Function, Variable Number of Arguments, Scope, Passing Functions to a Function, Mapping Functions in a Dictionary, Lambda, Modules, and Standard Modules.

Python Functions

Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the python program.

In other words, we can say that the collection of functions creates a program. The function is also known as procedure or subroutine in other programming languages.

Python provide us various inbuilt functions like range() or print(). Although, the user can create its functions which can be called user-defined functions.

Advantage of functions in python

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call python functions any number of times in a program and from any place in a program.
- We can track a large python program easily when it is divided into multiple functions.
- Reusability is the main achievement of python functions.
- However, Function calling is always overhead in a python program.

Creating a function

In python, we can use **def** keyword to define the function. The syntax to define a function in python is given below.

```
def my_function():
    function-suite
    return <expression>
```

The function block is started with the colon (:) and all the same level block statements remain at the same indentation.

A function can accept any number of parameters that must be the same in the definition and function calling.

Function calling

In python, a function must be defined before the function calling otherwise the python interpreter gives an error. Once the function is defined, we can call it from another function or the python prompt. To call the function, use the function name followed by the parentheses.

A simple function that prints the message "Hello Word" is given below.

```
def hello_world():
    print("hello world")

hello_world()
```

Output:

```
hello world
```

Parameters in function

The information into the functions can be passed as the parameters. The parameters are specified in the parentheses. We can give any number of parameters, but we have to separate them with a comma.

```
#python function to calculate the sum of two variables
#defining the function
def sum (a,b):
return a+b;

#taking values from the user
a = int(input("Enter a: "))
b = int(input("Enter b: "))

#printing the sum of a and b
print("Sum = ",sum(a,b))
```

Output:

```
Enter a: 10
Enter b: 20
Sum =  30
```

Call by reference in Python

In python, all the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.

However, there is an exception in the case of mutable objects since the changes made to the mutable objects like string do not revert to the original string rather, a new string object is made, and therefore the two different objects are printed.

Example 1 Passing Immutable Object (List)

```
#defining the function
def change_list(list1):
list1.append(20);
list1.append(30);
print("list inside function = ",list1)

#defining the list
list1 = [10,30,40,50]

#calling the function
change_list(list1);
print("list outside function = ",list1);
```

Output:

```
list inside function =  [10, 30, 40, 50, 20, 30]
list outside function =  [10, 30, 40, 50, 20, 30]
```

Example 2 Passing Mutable Object (String)

```
#defining the function
def change_string (str):
str = str + " Hows you";
print("printing the string inside function :",str);

string1 = "Hi I am there"

#calling the function
change_string(string1)

print("printing the string outside function :",string1)
```

Output:

```
printing the string inside function : Hi I am there Hows you
printing the string outside function : Hi I am there
```

Types of arguments

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

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

Required Arguments

Till now, we have learned about function calling in python. However, we can provide the arguments at the time of function calling. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, then the python interpreter will show the error.

Consider the following example.

```
#the function simple_interest accepts three arguments and returns the simple interest
accordingly
def simple_interest(p,t,r):
return (p*t*r)/100
p = float(input("Enter the principle amount? "))
r = float(input("Enter the rate of interest? "))
t = float(input("Enter the time in years? "))
print("Simple Interest: ",simple_interest(p,r,t))
```

Output:

```
Enter the principle amount? 10000
Enter the rate of interest? 5
Enter the time in years? 2
Simple Interest:  1000.0
```

Keyword arguments

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

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

Consider the following example.

```
#The function simple_interest(p, t, r) is called with the keyword arguments the order of argume
nts doesn't matter in this case
def simple_interest(p,t,r):
return (p*t*r)/100
print("Simple Interest: ",simple_interest(t=10,r=10,p=1900))
```

Default Arguments

Python allows us to initialize the arguments at the function definition. If the value of any of the argument is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

```
def printme(name,age=22):
print("My name is",name,"and age is",age)
printme(name = "john") #the variable age is not passed into the function however the default val
ue of age is considered in the function
printme(age = 10,name="David") #the value of age is overwritten here, 10 will be printed as age
```

Output:

```
My name is john and age is 22
My name is David and age is 10
```

Variable length Arguments

In the large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to provide the comma separated values which are internally treated as tuples at the function call.

However, at the function definition, we have to define the variable with * (star) as *<variable - name >.

Consider the following example.

```
def printme(*names):
print("type of passed argument is ",type(names))
print("printing the passed arguments...")
for name in names:
print(name)
printme("john","David","smith","nick")
```

Output:

```
type of passed argument is  <class 'tuple'>
printing the passed arguments...
john
```

```
David
smith
nick
```

Scope of variables

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.

1. Global variables
2. Local variables

The variable defined outside any function is known to have a global scope whereas the variable defined inside a function is known to have a local scope.

Consider the following example.

```
def calculate(*args):
sum=0
for arg in args:
sum = sum +arg
print("The sum is",sum)
sum=0
calculate(10,20,30) #60 will be printed as the sum
print("Value of sum outside the function:",sum) # 0 will be printed
```

Output:

```
The sum is 60
Value of sum outside the function: 0
```

Passing Functions to a Function

Python implements the following method where the first parameter is a function:

map(function, iterable, ...) - Apply function to every item of iterable and return a list of the results.

We can also write custom functions where we can pass a function as an argument.

We rewrite given code to pass the function sqr(x) as function argument using map method.

```
s = [1, 3, 5, 7, 9]
def sqr(x): return x ** 2
print(map(sqr, s))
```

We can as well use lambda function to get same output

```
s = [1, 3, 5, 7, 9]
print(map((lambda x: x**2), s))
```

OUTPUT

```
C:/Users/TutorialsPoint1/~.py
[1, 9, 25, 49, 81]
```

Lambda

In Python, anonymous function means that a function is without a name. As we already know that *def* keyword is used to define the normal functions and the *lambda* keyword is used to create anonymous functions. It has the following syntax:

lambda arguments: expression

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming besides other types of expressions in functions.

Example:

```
# Python code to illustrate cube of a number
# showing difference between def() and lambda().
def cube(y):
   return y*y*y;

g = lambda x: x*x*x
print(g(7))

print(cube(5))
```

output:

```
343
125
```

Use of lambda () with filter ()

The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence “sequence”, for which the function returns True. Here is a small program that returns the odd numbers from an input list:

```
# Python code to illustrate
# filter() with lambda()
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(filter(lambda x: (x%2 != 0) , li))
print(final_list)
```

Output:

```
[5, 7, 97, 77, 23, 73, 61]
```

Use of lambda () with map ()

The map() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item.

Example:

```
# Python code to illustrate
# map() with lambda()
# to get double of a list.
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(map(lambda x: x*2 , li))
print(final_list)
```

Output:

```
[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]
```

Use of lambda () with reduce ()

The reduce() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new reduced result is returned. This performs a repetitive operation over the pairs of the list. This is a part of functools module.

Example:

```
# Python code to illustrate
# reduce() with lambda()
# to get sum of a list
from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
```

print (sum)

Output:

193

Python Modules

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

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

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

Example

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

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

- #displayMsg prints a message to the name being passed.
- **def** displayMsg(name)
- **print**("Hi "+name);

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

Loading the module in our python code

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

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

The import statement

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

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

The syntax to use the import statement is given below.

- **import** module1,module2,........ module n

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

Example:

```
import file;
name = input("Enter the name?")
file.displayMsg(name)
```

Output:

```
Enter the name?John
Hi John
```

The from-import statement

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

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

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

calculation.py:

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

Main.py:

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

Output:

```
Enter the first number10
Enter the second number20
Sum =  30
```

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

Consider the following syntax.

- **from** <module> **import** *

Renaming a module

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

The syntax to rename a module is given below.

- **import** <module-name> as <specific-name>

Example

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

Output:

```
Enter a?10
Enter b?20
Sum =  30
```

Using dir() function

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

Consider the following example.

Example

```
import json
List = dir(json)
print(List)
```

Output:

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

The reload() function

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

- reload(<module-name>)

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

- reload(calculation)

Scope of variables

In Python, variables are associated with two types of scopes. All the variables defined in a module contain the global scope unless or until it is defined within a function.

All the variables defined inside a function contain a local scope that is limited to this function itself. We can not access a local variable globally.

If two variables are defined with the same name with the two different scopes, i.e., local and global, then the priority will always be given to the local variable.

Consider the following example.

Example

```
name = "john"
def print_name(name):
print("Hi",name) #prints the name that is local to this function only.
name = input("Enter the name?")
print_name(name)
```

Output:

```
Hi David
```

Python packages

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

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

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

ITEmployees.py

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

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

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

__init__.py

```
from ITEmployees import getITNames
from BPOEmployees import getBPONames
```

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

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

Test.py

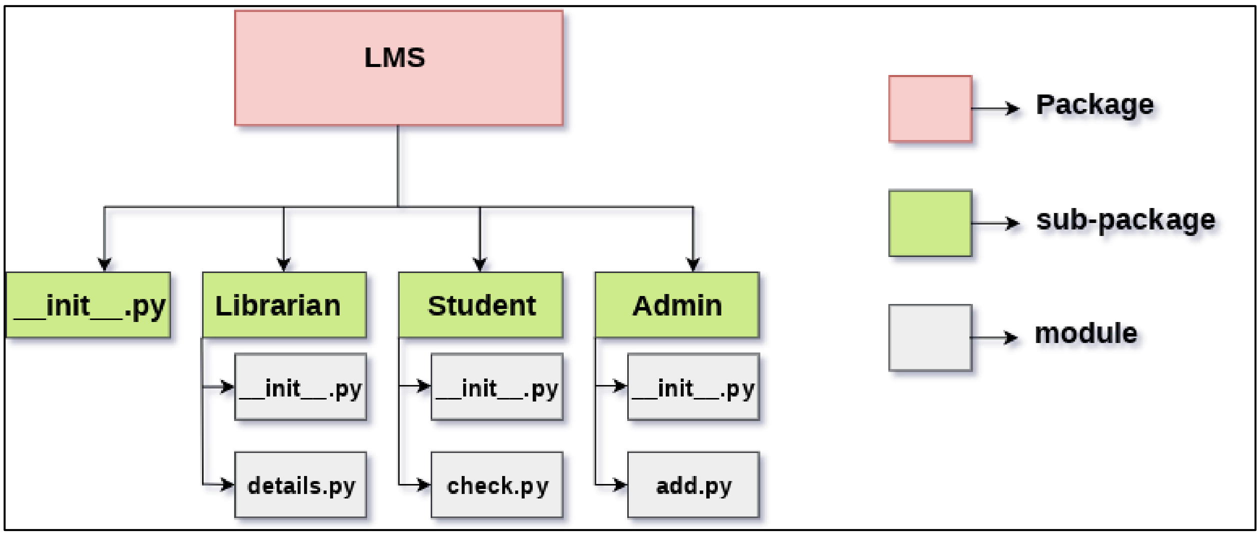
- **import** Employees
- **print**(Employees.getNames())

Output:

```
['John', 'David', 'Nick', 'Martin']
```

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

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



PROGRAM 5

AIM:OOP concepts: Classes, File Organization, Special Methods, Inheritance, Polymorphism, Special Characters, Character Classes, Quantifiers, Dot Character, Greedy Matches, Matching at Beginning or End, Match Objects, Compiling Regular Expressions.

Like other general purpose languages, python is also an object-oriented language since its beginning. Python is an object-oriented programming language. It allows us to develop applications using an Object Oriented approach. In Python, we can easily create and use classes and objects.

Major principles of object-oriented programming system are given below.

- Object
- Class
- Method
- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation

Python Class and Objects

Creating classes in python

In python, a class can be created by using the keyword class followed by the class name. The syntax to create a class is given below.

Syntax

```
class ClassName:
    #statement_suite
```

In python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.__doc__**. A class contains a statement suite including fields, constructor, function, etc. definition.

Consider the following example to create a class Employee which contains two fields as Employee id, and name.

The class also contains a function display() which is used to display the information of the Employee.

Example

```
class Employee:
    id = 10;
    name = "ayush"
    def display (self):
        print(self.id,self.name)
```

Here, the self is used as a reference variable which refers to the current class object. It is always the first argument in the function definition. However, using self is optional in the function call.

Creating an instance of the class

A class needs to be instantiated if we want to use the class attributes in another class or method. A class can be instantiated by calling the class using the class name.

The syntax to create the instance of the class is given below.

```
<object-name> = <class-name>(<arguments>)
```

The following example creates the instance of the class Employee defined in the above example.

Example

```
class Employee:
    id = 10;
    name = "John"
    def display (self):
        print("ID: %d \nName: %s"%(self.id,self.name))
emp = Employee()
emp.display()
```

Output:

```
ID: 10
Name: ayush
```

Python Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

Example 1

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
```

```
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output:

```
dog barking
Animal Speaking
```

Polymorphism

Sometimes an object comes in many types or forms. If we have a button, there are many different draw outputs (round button, check button, square button, button with image) but they do share the same logic: onClick(). We access them using the same method . This idea is called *Polymorphism*. Polymorphism is based on the greek words Poly (many) and morphism (forms). We will create a structure that can take or use many forms of objects.

Polymorphism with a function:

We create two classes: Bear and Dog, both can make a distinct sound. We then make two instances and call their action using the same method.

```
class Bear(object):
    def sound(self):
        print("Groarrr")

class Dog(object):
    def sound(self):
        print("Woof woof!")

    def makeSound(animalType):
        animalType.sound()



bearObj = Bear()
dogObj = Dog()
```

```
makeSound(bearObj)
makeSound(dogObj)
```

Output:

```
Groarrr
Woof woof!
```

Regular Expression

A regular expression is a compact notation for representing a collection of strings. What makes regular expressions so powerful is that a single regular expression can represent an unlimited number of strings—provided that they meet the regular expression's requirements. Regular expressions (we'll refer to them mostly as "regexes" from now on) are defined using a mini-language that's completely different from Python—but Python includes the re module, through which we can seamlessly create and use regexes.

Regexes are used for four main purposes:

- Validation. Checking whether a piece of text meets some criteria; for example, a currency symbol followed by digits.
- Searching. Locating substrings that can have more than one form; for example, finding any of *pet.png*, *pet.jpg*, *pet.jpeg*, or *pet.svg* while avoiding *carpet.png* and similar.
- Searching and replacing. Replacing everywhere the regex matches with a string; for example, finding *bicycle* or *human powered vehicle* and replacing either with *bike*.
- Splitting strings. Splitting a string at each place the regex matches; for example, splitting everywhere a colon (:) or equal sign (=) is encountered.

Python's Regular Expression Language

This section looks at the regular expression language in four subsections. The first subsection shows how to match individual characters or groups of characters; for example, match *a*, or match *b*, or match either *a* or *b*. The second subsection shows how to quantify matches; for example, match once, or match at least once, or match as many times as possible. The third subsection shows how to group subexpressions and how to capture matching text, and the final subsection shows how to use the language's assertions and flags to affect how regular expressions work.

Characters and Character Classes

The simplest expressions are just literal characters, such as **a** or **5**, and if no quantifier is explicitly given the expression is taken to be "match one occurrence." For example, the regex **tune** consists of four expressions, each implicitly quantified to match once, so it matches one *t* followed by one *u* followed by one *n* followed by one *e*, and hence matches the strings **tune** and at**tune**d.

Although most characters can be used as literals, some are special characters—symbols in the regex language that must be escaped by preceding them with a backslash (\) to use them as literals. The *special characters* are \.^$?+*{}[]()|. Most of Python's standard string escapes can also be used within regexes; for example, \n for newline and \t for tab, as well as hexadecimal escapes for characters using the \x*HH*, \u*HHHH*, and \U*HHHHHHHH* syntaxes.

In many cases, rather than matching one particular character we want to match any one of a set of characters. This can be achieved by using a *character class*—one or more characters enclosed in square brackets. (This has nothing to do with a Python class, and is simply the regex term for "set of characters.") A character class is an expression. Like any other expression, if not explicitly quantified it matches exactly one character (which can be any of the characters in the character class). For example, the regex **r[ea]d** matches both **red** and **rad**ar, but not read. Similarly, to match a single digit we can use the regex **[0123456789]**. For convenience we can specify a range of characters using a hyphen, so the regex **[0-9]** also matches a digit. It's possible to negate the meaning of a character class by following the opening bracket with a caret, so **[^0-9]** matches any character that is *not* a digit.

Note that inside a character class, apart from the backslash (\), the special characters lose their special meaning, although the caret (^) acquires a new meaning (negation) if it's the first character in the character class, and otherwise is simply a literal caret. Also, the hyphen (-) signifies a character range unless it's the first character, in which case it's a literal hyphen. Since some sets of characters are required so frequently, several have shorthand forms, which are shown in Table 1. With one exception, the shorthands can be used inside character sets; for example, the regex **[\dA-Fa-f]** matches any hexadecimal digit. The exception is the period (**.**) which is a shorthand outside a character class but matches a literal period inside a character class.

Table 1 Character Class Shorthands

Symbol	**Meaning**
.	Matches any character except newline, any character at all with the re.DOTALL flag, or inside a character class matches a literal period.
\d	Matches a Unicode digit, or **[0-9]** with the re.ASCII flag.
\D	Matches a Unicode nondigit, or **[^0-9]** with the re.ASCII flag.
\s	Matches a Unicode whitespace, or **[\t\n\r\f\v]** with the re.ASCII flag.
\S	Matches a Unicode non-whitespace, or **[^ \t\n\r\f\v]** with the re.ASCII flag.

\w	Matches a Unicode "word" character, or **[a-zA-Z0-9_]** with the re.ASCII flag.
\W	Matches a Unicode non-"word" character, or **[^a-zA-Z0-9_]** with the re.ASCII flag.

Quantifiers

A quantifier has the form **{*m*,*n*}** where *m* and *n* are the minimum and maximum times the expression to which the quantifier applies must match. For example, both **e{1,1}e{1,1}** and **e{2,2}** match f**ee**l, but neither matches felt.

Writing a quantifier after every expression would soon become tedious, and is certainly difficult to read. Fortunately, the regex language supports several convenient shorthands. If only one number is given in the quantifier, it's taken to be both the minimum and the maximum, so **e{2}** is the same as **e{2,2}**. As noted in the preceding section, if no quantifier is explicitly given, it's assumed to be **1** (that is, **{1,1}** or **{1}**); therefore, **ee** is the same as **e{1,1}e{1,1}** and **e{1}e{1}**, so both **e{2}** and **ee** match f**ee**l but not felt.

Having a different minimum and maximum is often convenient. For example, to match travelled and traveled (both legitimate spellings),we could use either **travel{1,2}ed** or **travell{0,1}ed**. The **{0,1}** quantification is used so often that it has its own shorthand form, ?, so another way of writing the regex (and the one most likely to be used in practice) is **travell?ed**.

Two other quantification shorthands are provided: A plus sign (+) stands for **{1**,*n***}** ("at least one") and an asterisk (*****) stands for **{0,***n*} ("any number of"). In both cases, *n* is the maximum possible number allowed for a quantifier, usually at least 32767. Table 2 shows all the quantifiers.

The **+** quantifier is very useful. For example, to match integers, we could use **\d+** to match one or more digits. This regex could match in two places in the string 4588.91, for example: **4588**.91 and 4588.**91**. Sometimes typos are the result of pressing a key too long. We could use the regex **bevel+ed** to match the legitimate **beveled** and **bevelled**, and the incorrect **bevellled**. If we wanted to standardize on the single-*l* spelling, and match only occurrences that had two or more *l*'s, we could use **bevell+ed** to find them.

The ***** quantifier is less useful, simply because it can lead so often to unexpected results. For example, supposing that we want to find lines that contain comments in Python files, we might try searching for **#***. But this regex will match any line whatsoever, including blank lines, because the meaning is "match any number of pound signs"—and that includes none. As a rule for those new to regexes, avoid using ***** at all, and if you do use it (or if you use **?**), make sure that at least one other expression in the regex has a nonzero quantifier. Use at least one quantifier other than ***** or **?**, that is, since both of these can match their expression zero times.

Often it's possible to convert ***** uses to **+** uses and vice versa. For example, we could match "tasselled" with at least one *l* using **tassell*ed** or **tassel+ed**, and match those with two or more *l*'s using **tasselll*ed** or **tassell+ed**.

If we use the regex **\d+** it will match **136**. But why does it match all the digits, rather than just the first one? By default, all quantifiers are *greedy*—they match as many characters as they can. We can make any quantifier nongreedy (also called *minimal*) by following it with a question mark (**?**) symbol. (The question mark has two different meanings—on its own it's a shorthand for the **{0,1}** quantifier, and when it follows a quantifier it tells the quantifier to be nongreedy.) For example, **\d+?** can match the string 136 in three different places: **1**36, 1**3**6, and 13**6**. Here's another example: **\d??** matches zero or one digits, but prefers to match none since it's nongreedy; on its own it suffers the same problem as ***** in that it will match nothing—that is, any text at all.

Table 2 Regular Expression Quantifiers

Syntax	**Meaning**
***e*? or *e*{0,1}**	Greedily match zero occurrences or one occurrence of expression *e*.
***e*?? or *e*{0,1}?**	Nongreedily match zero occurrences or one occurrence of expression *e*.
***e*+ or *e*{1,}**	Greedily match one or more occurrences of expression *e*.
***e*+? or *e*{1,}?**	Nongreedily match one or more occurrences of expression *e*.
***e** or *e*{0,}**	Greedily match zero or more occurrences of expression *e*.
***e**? or *e*{0,}?**	Nongreedily match zero or more occurrences of expression *e*.
***e*{*m*}**	Match exactly *m* occurrences of expression *e*.
***e*{*m*,}**	Greedily match at least *m* occurrences of expression *e*.
***e*{*m*,}?**	Nongreedily match at least *m* occurrences of expression *e*.
***e*{,*n*}**	Greedily match at most *n* occurrences of expression *e*.
***e*{,*n*}?**	Nongreedily match at most *n* occurrences of expression *e*.
***e*{*m*,*n*}**	Greedily match at least *m* and at most *n* occurrences of expression *e*.
***e*{*m*,*n*}?**	Nongreedily match at least *m* and at most *n* occurrences of expression *e*.

Nongreedy quantifiers can be useful for quick-and-dirty XML and HTML parsing. For example, to match all the image tags, writing **<img.*>** (match one each in order of <, i, m, and g, and then zero or more of any character apart from newline, and then one >) will not work because the **.*** part is greedy and will match everything including the tag's closing >, and will keep going until it reaches the last > in the entire text.

Three solutions present themselves (apart from using a proper parser):

- **<img[^>]*>** matches <img, any number of non-> characters, and then the tag's closing > character.
- **<img.*?>** matches <img, any number of characters (but nongreedily, so it will stop immediately before the tag's closing >), and then the >.
- **<img[^>]*?>** combines both of the preceding options.

None of these is correct, though, since they can all match **<img>**, which is not valid. Since we know that an image tag must have a src attribute, a more accurate regex is as follows:

```
<img\s+[^>]*?src=\w+[^>]*?>
```

This regex matches the literal characters <img, one or more whitespace characters, nongreedily zero or more of anything except > (to skip any other attributes such as alt), the src attribute (the literal characters src= and then at least one "word" character), and then any other non-> characters (including none) to account for any other attributes, and finally the closing >.

Grouping and Capturing

In practical applications, we often need regexes that can match any one of two or more alternatives, and we often need to capture the match or some part of the match for further processing. Also, we sometimes want a quantifier to apply to several expressions. All of these goals can be achieved by grouping with parentheses; and, in the case of alternatives, using alternation with the vertical bar (|).

Alternation is especially useful when we want to match any one of several quite different alternatives. For example, the regex **aircraft|airplane|jet** will match any text that contains *aircraft* or *airplane* or *jet*. The same objective can be achieved using the regex **air(craft|plane)|jet**. Here, the parentheses are used to group expressions, so we have two outer expressions, **air(craft|plane)** and **jet**. The first of these has an inner expression, **craft|plane**, and because this is preceded by **air**, the first outer expression can match only *aircraft* or *airplane*.

Parentheses serve two different purposes: grouping expressions and capturing the text that matches an expression. We'll use the term group to refer to a grouped expression whether it captures or not, and *capture* and *capture group* to refer to a captured group. If we used the regex **(aircraft|airplane|jet)**, it not only would match any of the three expressions, but would capture whichever one was matched for later reference. Compare this with the regex **(air(craft|plane)|jet)**, which has two captures if the first expression matches (*aircraft* or *airplane* as the first capture and *craft* or *plane* as the second capture), and one capture if the second expression matches (*jet*). We can switch off the capturing effect by following an opening parenthesis with **?:** like this:

```
(air(?:craft|plane)|jet)
```

This will have only one capture if it matches (*aircraft* or *airplane* or *jet*).

A grouped expression is an expression, and therefore can be quantified. As with any other expression, the quantity is assumed to be 1 unless explicitly given. For example, if we've read a text file with lines of the form *key=value*, where each *key* is alphanumeric, the regex **(\w+)=(.+)** will match every line that has a nonempty key and a nonempty value. (Recall that . matches anything except newlines.) And for every line that matches, two captures are made, the first being the key and the second being the value.

For example, the *key=value* regular expression will match the entire line **topic**= **physical geography** with the two captures shown shaded. Notice that the second capture includes some whitespace, and that whitespace before the equal sign is not accepted. We could refine the regex to be more flexible in accepting whitespace, and to strip off unwanted whitespace, by using a somewhat longer version:

```
[ \t]*(\w+)[ \t]*=[ \t]*(.+)
```

This example matches the same line as before, as well as lines that have whitespace around the equal sign, but with the first capture having no leading or trailing whitespace, and the second capture having no leading whitespace (for example, **topic** = **physical geography**).

We've been careful to keep the whitespace-matching parts outside the capturing parentheses, and to allow for lines that have no whitespace at all. We didn't use **\s** to match whitespace because that matches newlines (\n), which could lead to incorrect matches that span lines (for instance, if the re.MULTILINE flag is used). And for the value we didn't use **\S** to match non-whitespace because we want to allow for values that contain whitespace (English sentences, for example). To avoid the second capture having trailing whitespace, we would need a more sophisticated regex; we'll see this in the next subsection.

We can refer to captures by using *backreferences*; that is, by referring back to an earlier capture group. Note that backreferences cannot be used inside character classes; that is, inside brackets ([]).

One syntax for backreferences inside regexes themselves is *i*, where *i* is the capture number. Captures are numbered starting from one and increasing by one going from left to right as each new (capturing) left parenthesis is encountered. For example, to match duplicated words simplistically, we can use the regex **(\w+)\s+\1** to match a "word," followed by at least one whitespace, and then the same word as was captured. (Capture number 0 is created automatically without the need for parentheses; it holds the entire match—that is, what we show underlined.) We'll see a more sophisticated way to match duplicate words later.

In long or complicated regexes, it's often more convenient to use names rather than numbers for captures. This approach can also make maintenance easier, because adding or removing capturing parentheses may change the numbers but won't affect names. To name a capture, follow the opening parenthesis with **?P<*name*>**.For example,

```
(?P<key>\w+)=(?P<value>.+)
```

has two captures called key and value. The syntax for backreferences to named captures inside a regex is **(?P=*name*)**. For example,

```
(?P<word>\w+)\s+(?P=word)
```

Matches duplicate words using a capture called word.

MATLAB

PROGRAM 6,7,8

6. AIM : MATLAB Menus, Toolbars, Computing with MATLAB, Script Files and the Editor/Debugger, MATLAB help System.

7. AIM: MATLAB controls: Relational Logical Variables. Conditional Statements: if – else – else if, switch Loops: for – while – break, continue. User-Defined Functions.

8. AIM : Arrays, Matrices and Matrix Operations Debugging MATLAB Programs. Working with Data Files, and Graphing Functions: XY Plots – Sub-plots.

(THEORY REALTED TO PROGRAM 6, 7 and 8)

MATLAB (matrix laboratory) is a fourth-generation high-level programming language and interactive environment for numerical computation, visualization and programming. MATLAB is developed by MathWorks.

MATLAB's Power of Computational Mathematics

MATLAB is used in every facet of computational mathematics. Following are some commonly used mathematical calculations where it is used most commonly −

- Dealing with Matrices and Arrays
- 2-D and 3-D Plotting and graphics
- Linear Algebra
- Algebraic Equations
- Non-linear Functions
- Statistics
- Data Analysis
- Calculus and Differential Equations
- Numerical Calculations
- Integration
- Transforms
- Curve Fitting
- Various other special functions

Features of MATLAB

Following are the basic features of MATLAB −

- It is a high-level language for numerical computation, visualization and application development.
- It also provides an interactive environment for iterative exploration, design and problem solving.

- It provides vast library of mathematical functions for linear algebra, statistics, Fourier analysis, filtering, optimization, numerical integration and solving ordinary differential equations.
- It provides built-in graphics for visualizing data and tools for creating custom plots.
- MATLAB's programming interface gives development tools for improving code quality maintainability and maximizing performance.
- It provides tools for building applications with custom graphical interfaces.
- It provides functions for integrating MATLAB based algorithms with external applications and languages such as C, Java, .NET and Microsoft Excel.

Uses of MATLAB

MATLAB is widely used as a computational tool in science and engineering encompassing the fields of physics, chemistry, math and all engineering streams. It is used in a range of applications including −

- Signal Processing and Communications
- Image and Video Processing
- Control Systems
- Test and Measurement
- Computational Finance
- Computational Biology

6. MATLAB Menus, Toolbars, Computing with MATLAB, Script Files and the Editor/Debugger, MATLAB help System.

MATLAB Menus

Create multiple-choice dialog box

Syntax

```
choice = menu(message,opt1,opt2,...,optn)
choice = menu(message, options)
```

Description

choice = menu(message,opt1,opt2,...,optn) displays a modal menu dialog box containing the text in message and the choices specified by opt1, opt2,... optn. The menu function returns the number of the selected menu item, or 0 if the user clicks the close button on the window.

Specify message as a character vector or string scalar. Specify opt1, opt2,... optn as character vectors or string scalars.

choice = menu(message,options) specifies the choices as a cell array of character vectors or string array.

If the user's terminal provides a graphics capability, menu displays the menu items as push buttons in a figure window (Example 1). Otherwise. they will be given as a numbered list in the Command Window (Example 2).

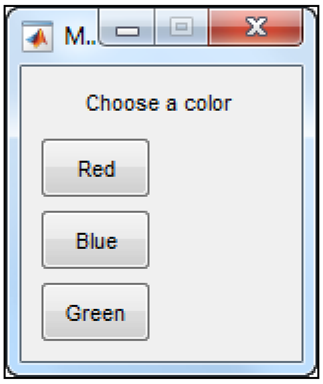
Examples

Example 1

On a system with a display, menu displays choices as buttons in a dialog box:

```
choice = menu('Choose a color','Red','Blue','Green')
```

displays the following dialog box.



The number entered by the user in response to the prompt is returned as choice (i.e., choice = 2 implies that the user selected Blue).

After input is accepted, the dialog box closes, returning the output in choice. You can use choice to control the color of a graph:

```
t = 0:.1:60;
s = sin(t);
color = ['r','b','g']
plot(t,s,color(choice))
```

Example 2

On a system without a display, menu displays choices in the Command Window:

```
choice = menu('Choose a color','Red','Blue','Green')
```

displays the following text.

```
----- Choose a color -----
1) Red
2) Blue
3) Green
Select a menu number:
```

TOOL BAR

Create menu or menu items

collapse all in page

Syntax

m = uimenu

m = uimenu(Name,Value)

m = uimenu(parent)

m = uimenu(parent,Name,Value)

Description

m = uimenu creates a menu in the current figure and returns the Menu object. If there is no figure available, MATLAB® calls the figure function to create one.

Example

m = uimenu(Name,Value) specifies menu property values using one or more name-value pair arguments.

m = uimenu(parent) creates the menu in the specified parent container. The parent container can be a figure created with either the figure or uifigure function, or another Menu object. Property values for uimenu vary slightly depending on whether the app is created with
the figure or uifigure function. For more information, see Name-Value Pair Arguments.

Example

m = uimenu(parent,Name,Value) specifies the parent container and one or more property values.

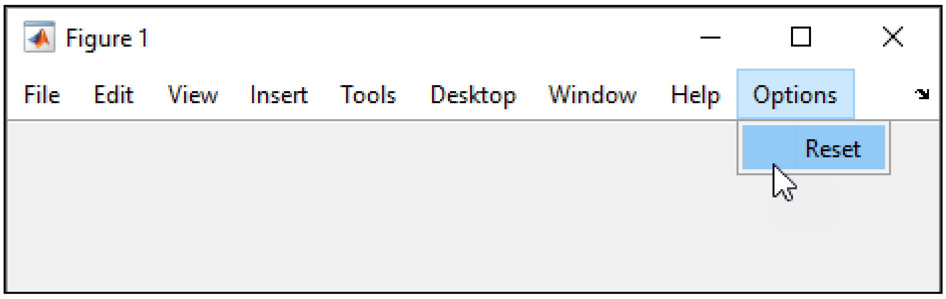
Examples

collapse all

Menu in Default Menu Bar

Create a figure that displays the default menu bar. Add a menu and a menu item.

```
f = figure('Toolbar','none');
m = uimenu('Text','Options');
mitem = uimenu(m,'Text','Reset');
```



Menu Item with Keyboard Shortcuts and Callback

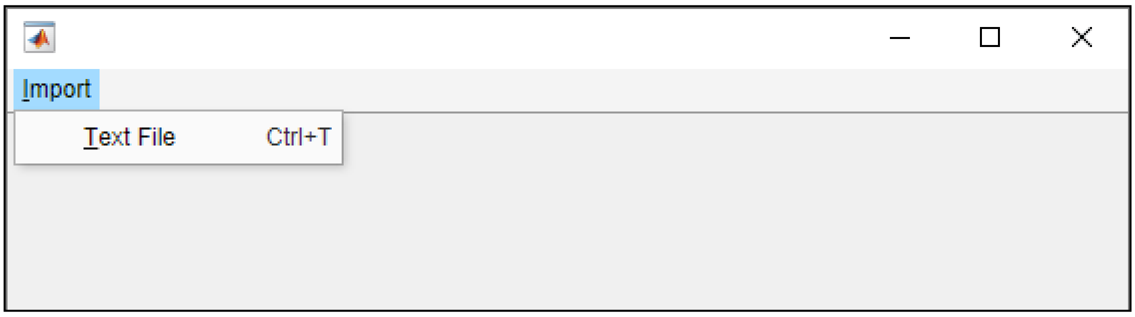
Add a menu item with keyboard shortcuts to the menu bar and define a callback that executes when the menu item is selected.

First, create a program file called importmenu.m. Within the program file:

- Create a figure.
- Add a menu called **Import**. Create a mnemonic keyboard shortcut for the menu by specifying '&Import' as the text label.
- Create a menu item and specify mnemonic and accelerator keyboard shortcuts.
- Define a MenuSelectedFcn callback that executes when the user clicks the menu item or uses the mnemonic or accelerator keyboard shortcuts.

Run the program file.

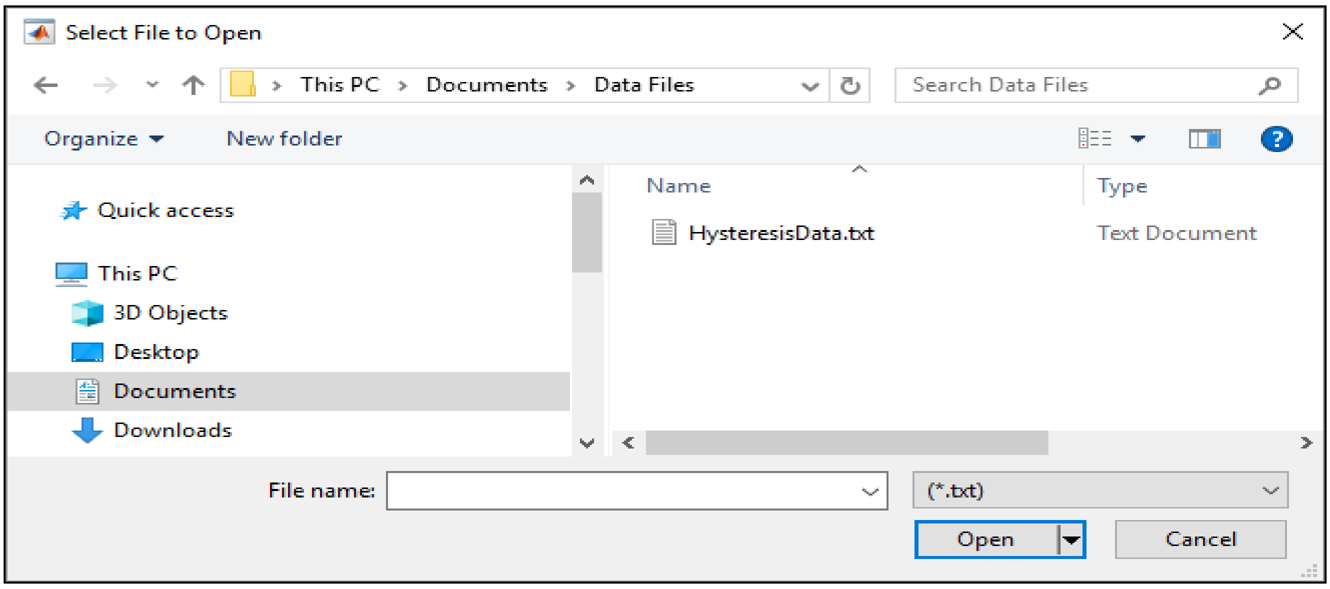
```
function importmenu
f = uifigure;
m = uimenu(f,'Text','&Import');
 mitem = uimenu(m,'Text','&Text File');
mitem.Accelerator = 'T';
mitem.MenuSelectedFcn = @MenuSelected;
     function MenuSelected(src,event)
        file = uigetfile('*.txt');
    end
 end
```



You can interact with the menu and menu item, using the keyboard, in the following ways:

- Select the **Import** menu by pressing **Alt+I**.
- Select the **Text File** menu item and execute the callback by pressing **Alt+I+T**.
- Select the **Text File** menu item and execute the callback by using the accelerator **Ctrl+T**.

When you select the **Text File** menu item, the Select File to Open dialog box opens with the extension field filtered to text files.



Menu with Checked Menu Item and Shared Callback

Create a checked menu item that can be selected or cleared to show a grid in axes. Share the callback with a push button so that pushing it also shows or hides the grid.

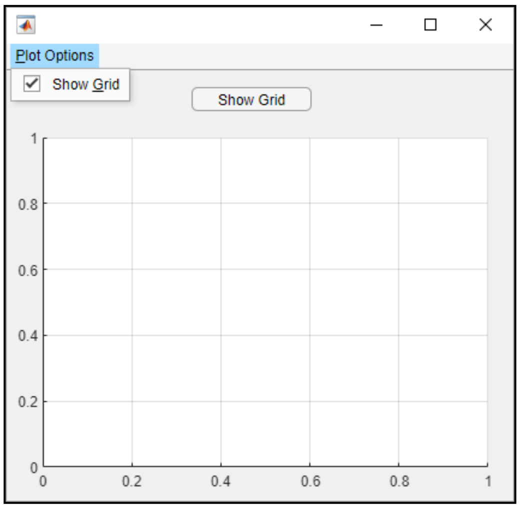
First, create a program file called plotOptions.m. Within the program file:

- Create a figure with a push button, and axes that display a grid.
- Add a menu and a menu item with mnemonics. Specify that the menu item is checked.
- Define a MenuSelectedFcn callback that hides or shows the grid when the user interacts with the menu item.
- Define a ButtonPushedFcn that uses the same callback function as the menu item.

Run the program file.

```
function plotOptions
f = uifigure;
ax = uiaxes(f);
grid(ax);
btn = uibutton(f,'Text','Show Grid');
btn.Position = [155 325 100 20];
m = uimenu(f,'Text','&Plot Options');
mitem = uimenu(m,'Text','Show &Grid','Checked','on');
mitem.MenuSelectedFcn = @ShowGrid;
btn.ButtonPushedFcn = @ShowGrid;
   function ShowGrid(src,event)
      grid(ax);
      if strcmp(mitem.Checked,'on')
         mitem.Checked = 'off';
```

```
        else
            mitem.Checked = 'on';
        end
    end
end
```



Input Arguments

parent — Parent container

Figure object | Menu object

Parent container, specified as a Figure object created with either the figure or uifigure function, or another Menu object. If you do not specify a parent container, then MATLAB calls figure to create one, and places the menu in the menu bar of that figure. Specify the parent as an existing Menu object to add menu items to a menu, or to nest menu items.

To add menu items to a context menu in GUIDE, or context menus in a figure created with the figure function, specify the parent as a ContextMenu object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: m = uimenu('Text','Open') creates a menu and sets its label to 'Open'.

- For a list of properties available for App Designer, or for creating apps with the uifigure function, see Menu Properties for App Designer.
- For a list of properties available for GUIDE, or for creating apps with the figure function, see Men

Script Files and the Editor/Debugger, MATLAB help System.

You can perform operations in MATLAB in two ways:

In the interactive mode, in which all commands are entered directly in the Command window, or

By running a MATLAB program stored in script file. This type of file contains MATLAB commands, so running it is equivalent to typing all the commands-one at a time-at the Command window prompt. You can run the file by typing its name at the Command window prompt.

Using the interactive mode is similar to using a calculator, but is convenient only for simpler problems. When the problem requires many commands, a repeated set of commands, or has arrays with many elements, the interactive mode is inconvenient. Fortunately, MATLAB allows you to write your own programs to avoid this difficulty. You write and save MATLAB programs in M-files, which have the extension. m; for example, program1. m

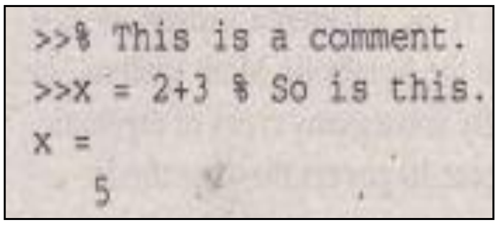
MATLAB uses two types of M-files: script files and function files. You can use the Editor Debugger built into MATLAB to create M-files. A script file contains a sequence of MATLAB commands, and is useful when to use many commands or arrays with many elements. Because they con tam commands, script files are sometimes called command files. You execute a script-file at the Command window prompt by typing its name without the extension.m.

Another type of M-file is a function file, which is useful when you need to repeat the operation of a set of commands. You can create your own function files.

Script files may contain any valid MATLAB commands or functions, including user-written functions. When you type the name of a script file at the Command window prompt, you get the same result as if you had typed at the Command window prompt all. the commands stored in the script file, one at a time. When you type the name of the script file, we say that you are "running the file" or "executing the file." The values of variables produced by running a script file are available in the workspace; thus, we say the variables created by a script file are global variables.

Creating and Using a Script File

The symbol %designates a comment, which is not executed by MATLAB. Comments are not that useful for interactive sessions, and are used mainly in script files for the purpose of documenting the file. The comment symbol may be put anywhere in the line. MATLAB ignores everything to the right of the %symbol. For example, consider the following session.

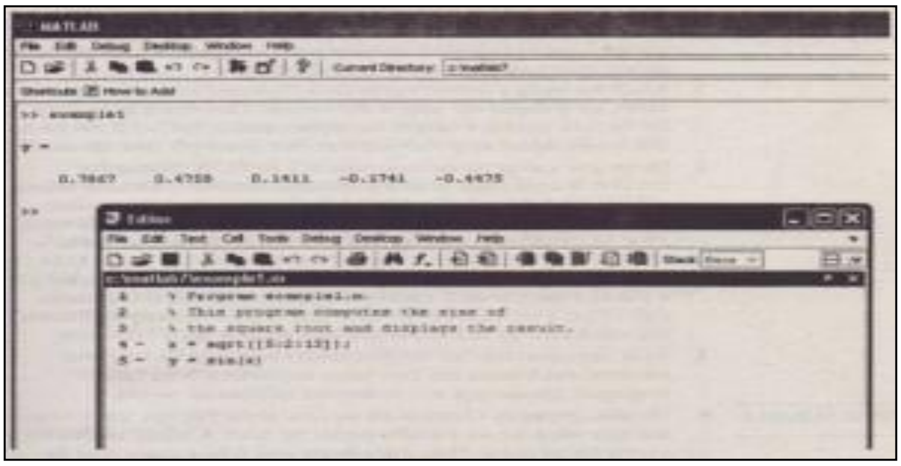


```
>>% This is a comment.
>>x = 2+3 % So is this.
x =
   5
```

Note that the portion of the line before the %sign is executed to compute x. Here is a simple example that illustrates how to create, ‘save, and run a script file, using the Editor/Debugger built into MATLAB. However, you may use another text editor to create the file. The sample file is shown below. It computes the sine of the square root of several numbers and displays the results on the screen.

```
% Program example1.m
% This program computes the sine of
% the square root and displays the result.
x = sqrt ([5:2 :13] );
Y = sin(x)
```

To create this new M-file (in the MS Windows environment), in the Command window select New from the File menu, then select M-file. You will then see a new edit window. This is the Editor/Debugger window as shown in Figure 1.4-1. Type in the file as shown above. You can use the keyboard and the Edit menu in the Editor/Debugger as you would in most word processors to create and edit the file. When finished, select Save from the File menu in the Editor/Debugger. In the dialog box that appears, replace the default name provided (usually named Untitled) with the name example1 , and click on Save. The Editor/Debugger will automatically provide the extension . m and save the file in the MATLAB current directory, which for now we will assume to be on the hard drive.



```
>> example1

y =

    0.7867    0.4758    0.1411   -0.1741   -0.4475

>>
```

```
% Program example1.m
% This program computes the sine of
% the square root and displays the result.
x = sqrt([5:2:13]);
y = sin(x)
```

Figure :The MATLAB Command window with the Editor/Debugger open.

Once the file has been saved, in the MATLAB Command window type the script file’s name example1 to execute the program. You should see the result displayed in the Command window. The session looks like the following.

```
»example1
Y =
0.7867 0.4758 0.1411 0.1741 -0.4475
```

Figure 1.4-1 shows a MATLAB MS Windows screen containing the resulting Command window display and the Editor/Debugger opened to display the script file.

Effective Use of Script Files

Create script fiIes to avoid the need to retype commonly used procedures. The above file, example1 implements a very simple procedure, for which we ordinarily would not create a script file. However, it illustrates how a script file is useful. For example, to change the numbers evaluated from [3: 2 : 11] to [2 : 5 : 27] , simply edit tile corresponding line and save the file again (a common oversight is to forget to resave the file after making changes to it).

Here are some other things to keep in mind when using script files:

1. The name of a script file must follow the MATLAB convention for naming variables; that is, the name must begin with a letter. and may include dig and the underscore character. up to 31 characters.
2. Recall that typing a variable's name at the Command window prompt causes MATLAB to display the value of that variable. Thus, do not give a script file the same name as a variable it computes because MATLAB will not be able to execute that script file more than once. unless you clear the variable.
3. Do not give a script file the same name as a MATLAB command or function. You can check to see if a command. function or file name already exists by using the exi s t command. For example, to see if a variable example1 already exists, type exist ('example1'); this will return a 0 if the variable does not exist, and a I if it does. To see if an M-file example1 m already exists, type exist (' example1 .m file before creating the file; this will return a 0 if the file does not exist, and a if it does. Finally, to see if a built-in function example1 already exists, type exist ('example1′ builtin') before creating the file will return a 0 if the built-in function does not exist, and a 5 if it does.
4. As in interactive mode, all variables created by a script file are global variables, which means that their values are available in the basic workspace. You can type who to see what variables are present.
5. Variables created by a function file are local to that function, which mean that their values are not available outside the function. All the variables in a script file are global. Thus, if you do not need to have access to an the variables in a script file, consider using a function file instead. This will avoid "cluttering" the workspace with variable names, and will reduce memory requirements. Creation of user-defined function files is discussed.
6. You can use the type command to view an M-file without opening it with a text editor. For example, to view the file examplel, the command is type examplel.

Note that not all functions supplied with MATLAB are "built-in" function. For example, the function mean.m is supplied but is not a built-in function. The command exist (' mean. m', 'file') will return a 2, but the command exist (‘mean’, ‘builtin’) will return a O.You may think of builtin functions as primitives that form the basis for other MATLAB functions. You cannot view the entire file of a built-in function in a text editor. only the comments.

Effective Use of the Command and Editor/Debugger Window

Here are some tips on using the Command and Editor/Debugger window effectively.

1. You can use the mouse to resize and move windows so they can be viewed simultaneously. Or you can dock the Editor/Debugger window inside the Desktop by selecting Dock from the view menu of the Editor/Debugger. To activate a window, click on it.
2. If the Editor/Debugger is not docked, use the Alt- Tab key combination to switch back and forth quickly between the Editor/Debugger window and the Command window. In the Command window. use the up-arrow key to retrieve the previously typed script-file name, and press Enter to execute the script file. This technique allows you to check and correct your program quickly. After making changes in the script file, be sure to save it before switching to the Command window.
3. You can use the Editor/Debugger as a basic word processor to write a short report that includes your script file, results, and discussion, perhaps present your solution to one of the problems. First use the mouse to highlight the results shown in the Command window, then copy and paste them to the Editor/Debugger window above or below your script file (use Copy and Paste on the Edit menu). Then, to save space, delete any extra blank lines, and perhaps the prompt symbol. Type your name arid any other required information, add any discussion you wish, and print the report from the Editor/Debugger window, or save it and import it into the word processor of your choice (change the file name or its extension if you intend to use the script file again).

Debugging Script Files

Debugging a program is the process of finding and removing the “bugs,” or errors, in a program. Such errors usually fall into one of the following categories.

1. Syntax errors such as omitting a parenthesis or comma, or spelling a command name incorrectly. MATLAB usually detects the more obvious errors and displays a message describing the error and its location.
2. Errors due to an incorrect mathematical procedure, called runtime errors. They do not necessarily occur every time the program is executed; their occurrence often depends on the particular input data. A common example is division by zero.

MATLAB error messages usually allow you to find syntax errors. However, runtime errors are more difficult to locate. To locate such an error, try the following:

1. Always test your program with a simple version of the problem, whose answers can be checked by hand calculations.

2. Display any intermediate calculations by removing semicolons at the end of statements.
3. Use the debugging features of the Editor/Debugger, which are covered. However, one advantage of MATLAB is that it requires relatively simple programs to accomplish many types of tasks. Thus you probably will not need to use the Debugger for many of the problems encountered in this text.

Programming Style

Comments may be put anywhere in the script file. However, it is important to note that the first comment line before any executable statement is the line searched by the Lookf o r command, . Therefore, if you intend to use the script file in the future, consider putting key words that describe the script file in this first line (called the HI line).

A suggested structure for a script file is the following.

1. Comments section In this section put comment statements to give:

   a. The name of the program and any key words in the first line.

   b. The date created, and the creators' names in the second line.

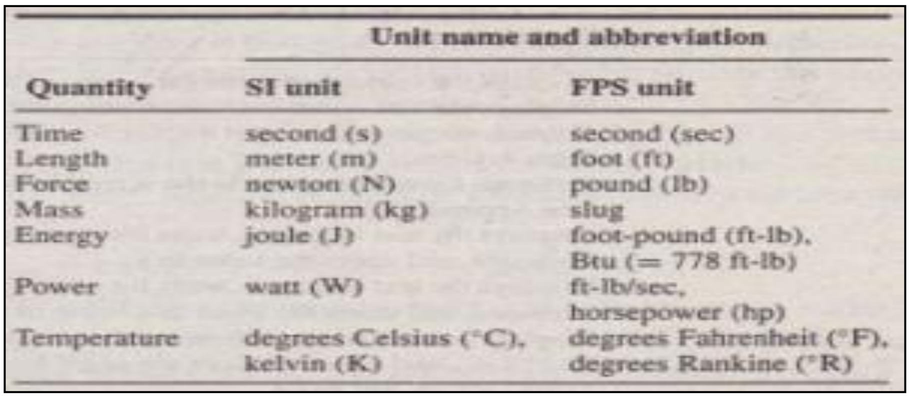
   c. The definitions of the variable names for every input and output variable. Divide this section into at least two subsections, one for input data, and one for output data. A third, optional section may include definitions of variables used in the calculations. Be sure to include the units of measurement for all input and all output variables!

   d. The name of every user-defined function called by the program.
2. Input section In this section put the input data and/or the input functions ·that enable data to be entered. Include comments where appropriate for documentation.
3. Calculation section Put the calculations in this section. Include comments where appropriate for documentation.
4. Output section In this section put the functions necessary to deliver the output in whatever form required. For example, this section might contain functions for displaying the output on the screen. Include comments where appropriate for documentation.
5. The programs in this text usually omit the comments in order to save space. These comments are not necessary here because the text discussion associated with the program provides the required documentation, and because we all know who wrote these programs!

Documenting Units of Measurement

We emphasize again that you must document the units of measurement for all the input and all the output variables. More than one dramatic failure of an engineering system has been traced to the misunderstanding of the units used for the input and output variables of the program used to design the system. Table 1.4-1 lists common units and their abbreviations. The foot-pound-second system (FPS) is also called the U.S. Customary System and the British Engineering System. SI (Systeme Internationale) is the international metric system.

Using Script Files to Store Data



	Unit name and abbreviation	
Quantity	SI unit	FPS unit
Time	second (s)	second (sec)
Length	meter (m)	foot (ft)
Force	newton (N)	pound (lb)
Mass	kilogram (kg)	slug
Energy	joule (J)	foot-pound (ft-lb), Btu (= 778 ft-lb)
Power	watt (W)	ft-lb/sec, horsepower (hp)
Temperature	degrees Celsius (°C), kelvin (K)	degrees Fahrenheit (°F), degrees Rankine (°R)

Table : SI and FPS units

a set of daily temperature measurements at a particular location, which are needed from time to time for calculations. As a short example, consider the following script file, whose name is my data. m.The array temp_F contains temperatures in degrees Fahrenheit.

```
% File mydata.m: Stores temperature data.
% Stores the array temp_F,
% which contains temperatures in degrees Fahrenheit.
temp_F ~ [72, 68, 75, 77, 83, 79)
```

A session to access this data from the Command window, and convert the temperatures
to degrees Celsius, is

```
» mydata
temp _ F =
72 68 75 77 83 79
» temp _ C = 5 * ( temp_F – 32 ) / 9
temp _ C =
22.2222 20.0000 23.8889 25.0000 28.3333 26.1111
```

Thus 68° Fahrenheit corresponds to 20° Celsius

Controlling Input and Output

MATLAB provides several useful commands for obtaining input from the user and for formatting the output (the results obtained by executing the MATLAB commands). Table 1.4-2 summarizes these commands. The methods presented in this section are particularly useful with script files.

You already know how to determine the current value of any variable by t) ping its name and pressing Enter at the command prompt. However. This method. which is useful in the interactive mode, is not useful for script files. The disp function (short for "display") can be used instead. Its syntax is di sp (A) , where A represents a MATL.A B variable name. Thus typing disp (Speed) causes the value of the variable Speed to appear on the screen, but not the variable's name.

The disp function can also display text. You enclose the text within single quotes. For example, the command disp (‘The predicted speed is: ‘) causes the message to appear on the screen. This command can be used with the first form of the disp function in a script file as follows (assuming the value of Speed is 63)

dispt (‘The predict_ed speed is:’)
disp (Speed)

When the file is run, these lines produce the following on the screen:

The predicted speed is:
63

User Input

The input function displays text on the screen, waits for the user to enter something from the keyboard, and then stores the input in the specified variable. For example, the command x = input (‘Please enter the value of x:’) causes the message to appear on the screen. If you type 5 and press. **Enter:** the variable x will have the value 5.

A string variable is composed of text (alphanumeric characters). If you want to store a text input as a string variable, use the other form of the inpu t command. For example, the command Calendar = input (,Enter the day of the week:’ s “) prompts you to enter the day of the week. If you type Wednesday, this text will be Stored in the string variable Calendar. Use the menu function to generate a menu of choices for user input. Its syntax is

k = menu(‘title’, ‘option1’, ‘option2’, …)

The function displays the menu whose title is in the string variable ‘ title and whose choices are string variables’ option1′, ‘option2’, and soon. The returned value of k is I, 2,… depending on whether you click on the button for option1, option2, and so forth. For example, the following script uses a menu to select the data marker for a graph, assuming that the vectors x and y already exist.

k = menu(‘Choose a data marker’, ‘a’, ‘*’,’x’);
type = [‘ 0’ , ‘ *’ , ‘ x ‘):
plot (x,y,x,y,type (k)).

MATLAB offers several options for getting help on MathWorks® products. You can access abbreviated function help text in the Command Window, or search the documentation for in-depth, comprehensive help topics and examples. For information on specific issues not addressed in the documentation, contact technical support.

Functions

doc	Reference page in Help browser
help	Help for functions in Command Window
docsearch	Help browser search
lookfor	Search for keyword in all help entries
demo	Access product examples in Help browser
echodemo	Run example script step-by-step in Command Window

7. MATLAB controls: Relational Logical Variables. Conditional Statements: if – else – else if, switch Loops: for – while – break, continue. User-Defined Functions

In MATLAB environment, every variable is an array or matrix.

You can assign variables in a simple way. For example,

```
x = 3            % defining x and initializing it with a value
```

MATLAB will execute the above statement and return the following result −

```
x = 3
```

It creates a 1-by-1 matrix named *x* and stores the value 3 in its element. Let us check another example,

```
x = sqrt(16)          % defining x and initializing it with an expression
```

MATLAB will execute the above statement and return the following result −

```
x = 4
```

Please note that −

- Once a variable is entered into the system, you can refer to it later.
- Variables must have values before they are used.
- When an expression returns a result that is not assigned to any variable, the system assigns it to a variable named ans, which can be used later.

For example,

```
sqrt(78)
```

MATLAB will execute the above statement and return the following result −

```
ans =  8.8318
```

You can use this variable **ans** −

```
sqrt(78);
9876/ans
```

MATLAB will execute the above statement and return the following result −

```
ans =  1118.2
```

Let's look at another example −

```
x = 7 * 8;
y = x * 7.89
```

MATLAB will execute the above statement and return the following result −

```
y =  441.84
```

Multiple Assignments

You can have multiple assignments on the same line. For example,

```
a = 2; b = 7; c = a * b
```

MATLAB will execute the above statement and return the following result −

```
c = 14
```

I have forgotten the Variables!

The **who** command displays all the variable names you have used.

```
who
```

MATLAB will execute the above statement and return the following result −

```
Your variables are:
a    ans  b    c
```

The **whos** command displays little more about the variables −

- Variables currently in memory
- Type of each variables
- Memory allocated to each variable
- Whether they are complex variables or not

```
whos
```

MATLAB will execute the above statement and return the following result −

```
Attr Name        Size        Bytes      Class
==== ====        ====        ====       =====
   a            1x1          8       double
   ans           1x70        757       cell
   b            1x1          8       double
   c            1x1          8       double

Total is 73 elements using 781 bytes
```

The **clear** command deletes all (or the specified) variable(s) from the memory.

```
clear x     % it will delete x, won't display anything
clear       % it will delete all variables in the workspace
            %  peacefully and unobtrusively
```

Long Assignments

Long assignments can be extended to another line by using an ellipses (...). For example,

```
initial_velocity = 0;
acceleration = 9.8;
time = 20;
final_velocity = initial_velocity + acceleration * time
```

MATLAB will execute the above statement and return the following result −

final_velocity = 196

The format Command

By default, MATLAB displays numbers with four decimal place values. This is known as **short format**.

However, if you want more precision, you need to use the **format** command.

The **format long** command displays 16 digits after decimal.

For example −

```
format long
x = 7 + 10/3 + 5 ^ 1.2
```

MATLAB will execute the above statement and return the following result−

x = 17.2319816406394

Another example,

```
format short
x = 7 + 10/3 + 5 ^ 1.2
```

MATLAB will execute the above statement and return the following result −

x = 17.232

The **format bank** command rounds numbers to two decimal places. For example,

```
format bank
daily_wage = 177.45;
weekly_wage = daily_wage * 6
```

MATLAB will execute the above statement and return the following result −

weekly_wage = 1064.70

MATLAB displays large numbers using exponential notation.

The **format short e** command allows displaying in exponential form with four decimal places plus the exponent.

For example,

```
format short e
4.678 * 4.9
```

MATLAB will execute the above statement and return the following result −

ans = 2.2922e+01

The **format long e** command allows displaying in exponential form with four decimal places plus the exponent. For example,

```
format long e
x = pi
```

MATLAB will execute the above statement and return the following result −

```
x = 3.141592653589793e+00
```

The **format rat** command gives the closest rational expression resulting from a calculation. For example,

```
format rat
4.678 * 4.9
```

MATLAB will execute the above statement and return the following result −

```
ans = 34177/1491
```

Creating Vectors

A vector is a one-dimensional array of numbers. MATLAB allows creating two types of vectors

- Row vectors
- Column vectors

Row vectors are created by enclosing the set of elements in square brackets, using space or comma to delimit the elements.

For example,

```
r = [7 8 9 10 11]
```

MATLAB will execute the above statement and return the following result −

```
r =[ 7    8    9   10   11]
```

Another example,

```
r = [7 8 9 10 11];
t = [2, 3, 4, 5, 6];
res = r + t
```

MATLAB will execute the above statement and return the following result −

```
res =  9         11         13         15         17
```

Column vectors are created by enclosing the set of elements in square brackets, using semicolon(;) to delimit the elements.

```
c = [7;  8;  9;  10; 11]
```

MATLAB will execute the above statement and return the following result −

```
c =
      7
      8
      9
```

```
    10
    11
```

Creating Matrices

A matrix is a two-dimensional array of numbers.

In MATLAB, a matrix is created by entering each row as a sequence of space or comma separated elements, and end of a row is demarcated by a semicolon. For example, let us create a 3-by-3 matrix as −

```
m = [1 2 3; 4 5 6; 7 8 9]
```

MATLAB will execute the above statement and return the following result −

```
m =
     1        2        3
     4        5        6
     7        8        9
```

Loops and Conditional Statements

Control flow and branching using keywords, such as if, for, and while

MATLAB Language Syntax

if, elseif, else	Execute statements if condition is true
for	for loop to repeat specified number of times
parfor	Parallel for loop
switch, case, otherwise	Execute one of several groups of statements
try, catch	Execute statements and catch resulting errors
while	while loop to repeat when condition is true
break	Terminate execution of for or while loop
continue	Pass control to next iteration of for or while loop
end	Terminate block of code or indicate last array index
pause	Stop MATLAB execution temporarily
return	Return control to invoking script or function

Conditional Statements

Conditional statements enable you to select at run time which block of code to execute. The simplest conditional statement is an if statement.

For example:

```
% Generate a random number
```

```
a = randi(100, 1);
% If it is even, divide by 2
if rem(a, 2) == 0
   disp('a is even')
   b = a/2;
end
```

if statements can include alternate choices, using the optional keywords elseif or else.

For example:

```
a = randi(100, 1);

if a < 30
   disp('small')
elseif a < 80
   disp('medium')
else
   disp('large')
end
```

Alternatively, when you want to test for equality against a set of known values, use a switch statement.

For example:

```
[dayNum, dayString] = weekday(date, 'long', 'en_US');

switch dayString
  case 'Monday'
    disp('Start of the work week')
  case 'Tuesday'
    disp('Day 2')
  case 'Wednesday'
    disp('Day 3')
  case 'Thursday'
    disp('Day 4')
  case 'Friday'
    disp('Last day of the work week')
  otherwise
    disp('Weekend!')
    end
```

For both if and switch, MATLAB® executes the code corresponding to the first true condition, and then exits the code block. Each conditional statement requires the end keyword.

In general, when you have many possible discrete, known values, switch statements are easier to read than if statements. However, you cannot test for inequality between switch and case values. For example, you cannot implement this type of condition with a switch:

```
    yourNumber = input('Enter a number: ');

    if yourNumber < 0
       disp('Negative')
    elseif yourNumber > 0
       disp('Positive')
    else
       disp('Zero')
    end
```

Use if, elseif, and else for Conditional Assignment

Create a matrix of 1s.

```
nrows = 4;
ncols = 6;
A = ones(nrows,ncols);
```

Loop through the matrix and assign each element a new value. Assign 2 on the main diagonal, -1 on the adjacent diagonals, and 0 everywhere else.

```
for c = 1:ncols
   for r = 1:nrows
      if r == c
         A(r,c) = 2;
      elseif abs(r-c) == 1
         A(r,c) = -1;
      else
         A(r,c) = 0;
      end

   end
end
A
```

Output:

A = *4×6*

```
     2    -1     0     0     0     0
    -1     2    -1     0     0     0
     0    -1     2    -1     0     0
     0     0    -1     2    -1     0
```

Loops in mat lab document attached

User-Defined Functions

Custom function blocks such as MATLAB Function, MATLAB System, Simulink Function, and Initialize Function

Blocks

C Caller	Integrate C code in Simulink
Fcn	Apply specified expression to input
Function Caller	Call Simulink or exported Stateflow function
Initialize Function	Executes contents on a model initialize event
Interpreted MATLAB Function	Apply MATLAB function or expression to input
Level-2 MATLAB S-Function	Use Level-2 MATLAB S-function in model
MATLAB Function	Include MATLAB code in models that generate embeddable C code
MATLAB System	Include System object in model
Reset Function	Executes contents on a model reset event
S-Function	Include S-function in model
S-Function Builder	Integrate C or C++ code to create S-functions
Simulink Function	Function defined with Simulink blocks
Terminate Function	Execute contents on a model terminate event

8. Arrays, Matrices and Matrix Operations Debugging MATLAB Programs. Working with Data Files, and Graphing Functions: XY Plots – Sub-plots.

Arrays, Matrices and Matrix Operations Debugging MATLAB Programs

All variables of all data types in MATLAB are multidimensional arrays. A vector is a one-dimensional array and a matrix is a two-dimensional array.

We have already discussed vectors and matrices. In this chapter, we will discuss multidimensional arrays. However, before that, let us discuss some special types of arrays.

Special Arrays in MATLAB

In this section, we will discuss some functions that create some special arrays. For all these functions, a single argument creates a square array, double arguments create rectangular array.

The **zeros()** function creates an array of all zeros −

For example −

```
zeros(5)
```

MATLAB will execute the above statement and return the following result −

```
ans =
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
```

The **ones()** function creates an array of all ones −

For example −

```
ones(4,3)
```

MATLAB will execute the above statement and return the following result −

```
ans =
    1    1    1
    1    1    1
    1    1    1
    1    1    1
```

The **eye()** function creates an identity matrix.

For example −

```
eye(4)
```

MATLAB will execute the above statement and return the following result −

```
ans =
    1    0    0    0
    0    1    0    0
    0    0    1    0
    0    0    0    1
```

The **rand()** function creates an array of uniformly distributed random numbers on (0,1) −

For example −

```
rand(3, 5)
```

MATLAB will execute the above statement and return the following result −

```
ans =
  0.8147    0.9134    0.2785    0.9649    0.9572
  0.9058    0.6324    0.5469    0.1576    0.4854
  0.1270    0.0975    0.9575    0.9706    0.8003
```

A Magic Square

A **magic square** is a square that produces the same sum, when its elements are added row-wise, column-wise or diagonally.

The **magic()** function creates a magic square array. It takes a singular argument that gives the size of the square. The argument must be a scalar greater than or equal to 3.

```
magic(4)
```

MATLAB will execute the above statement and return the following result −

```
ans =
  16    2    3   13
   5   11   10    8
   9    7    6   12
   4    14    15     1
```

Multidimensional Arrays

An array having more than two dimensions is called a multidimensional array in MATLAB. Multidimensional arrays in MATLAB are an extension of the normal two-dimensional matrix. Generally to generate a multidimensional array, we first create a two-dimensional array and extend it.

For example, let's create a two-dimensional array a.

```
a = [7 9 5; 6 1 9; 4 3 2]
```

MATLAB will execute the above statement and return the following result −

```
a =
   7     9     5
   6     1     9
   4     3     2
```

The array *a* is a 3-by-3 array; we can add a third dimension to *a*, by providing the values like −

```
a(:, :, 2)= [ 1 2 3; 4 5 6; 7 8 9]
```

MATLAB will execute the above statement and return the following result −

```
a =
ans(:,:,1) =

   0   0   0
   0   0   0
   0   0   0
ans(:,:,2) =

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

We can also create multidimensional arrays using the ones(), zeros() or the rand() functions.

For example,

```
b = rand(4,3,2)
```

MATLAB will execute the above statement and return the following result −

```
b(:,:,1) =
   0.0344    0.7952    0.6463
```

```
  0.4387    0.1869    0.7094
  0.3816    0.4898    0.7547
  0.7655    0.4456    0.2760


b(:,:,2) =
  0.6797    0.4984    0.2238
  0.6551    0.9597    0.7513
  0.1626    0.3404    0.2551
  0.1190    0.5853    0.5060
```

We can also use the **cat()** function to build multidimensional arrays. It concatenates a list of arrays along a specified dimension −

Syntax for the cat() function is −

```
B = cat(dim, A1, A2...)
```

Where,

- *B* is the new array created
- *A1*, *A2*, ... are the arrays to be concatenated
- *dim* is the dimension along which to concatenate the arrays

Example

Create a script file and type the following code into it −

```
a = [9 8 7; 6 5 4; 3 2 1];
b = [1 2 3; 4 5 6; 7 8 9];
c = cat(3, a, b, [ 2 3 1; 4 7 8; 3 9 0])
```

When you run the file, it displays −

```
c(:,:,1) =
    9    8    7
    6    5    4
    3    2    1
c(:,:,2) =
    1    2    3
    4    5    6
    7    8    9
c(:,:,3) =
    2    3    1
    4    7    8
    3     9     0
```

Array Functions

MATLAB provides the following functions to sort, rotate, permute, reshape, or shift array contents.

Function	Purpose
length	Length of vector or largest array dimension
ndims	Number of array dimensions
numel	Number of array elements
size	Array dimensions
iscolumn	Determines whether input is column vector
isempty	Determines whether array is empty
ismatrix	Determines whether input is matrix
isrow	Determines whether input is row vector
isscalar	Determines whether input is scalar
isvector	Determines whether input is vector
blkdiag	Constructs block diagonal matrix from input arguments
circshift	Shifts array circularly
ctranspose	Complex conjugate transpose
diag	Diagonal matrices and diagonals of matrix
flipdim	Flips array along specified dimension
fliplr	Flips matrix from left to right
flipud	Flips matrix up to down
ipermute	Inverses permute dimensions of N-D array

permute	Rearranges dimensions of N-D array
repmat	Replicates and tile array
reshape	Reshapes array
rot90	Rotates matrix 90 degrees
shiftdim	Shifts dimensions
issorted	Determines whether set elements are in sorted order
sort	Sorts array elements in ascending or descending order
sortrows	Sorts rows in ascending order
squeeze	Removes singleton dimensions
transpose	Transpose
vectorize	Vectorizes expression

Examples

The following examples illustrate some of the functions mentioned above.

Length, Dimension and Number of elements −

Create a script file and type the following code into it −

```
x = [7.1, 3.4, 7.2, 28/4, 3.6, 17, 9.4, 8.9];
length(x)      % length of x vector
y = rand(3, 4, 5, 2);
ndims(y)       % no of dimensions in array y
s = ['Zara', 'Nuha', 'Shamim', 'Riz', 'Shadab'];
numel(s)       % no of elements in s
```

When you run the file, it displays the following result −

```
ans =  8
ans =  4
ans =  23
```

Circular Shifting of the Array Elements −

Create a script file and type the following code into it −

```
a = [1 2 3; 4 5 6; 7 8 9]  % the original array a
```

```
b = circshift(a,1)        %  circular shift first dimension values down by 1.
c = circshift(a,[1 -1])    % circular shift first dimension values % down by 1
                          % and second dimension values to the left % by 1.
```

When you run the file, it displays the following result −

```
a =
  1    2    3
  4    5    6
  7    8    9

b =
  7    8    9
  1    2    3
  4    5    6

c =
  8    9    7
  2    3    1
  5    6    4
```

Sorting Arrays

Create a script file and type the following code into it −

```
v = [ 23 45 12 9 5 0 19 17]  % horizontal vector
sort(v)                   % sorting v
m = [2 6 4; 5 3 9; 2 0 1]    % two dimensional array
sort(m, 1)                 % sorting m along the row
sort(m, 2)                 % sorting m along the column
```

When you run the file, it displays the following result −

```
v =
  23    45    12     9     5     0    19    17
ans =
  0     5     9    12    17    19    23    45
m =
  2    6    4
  5    3    9
  2    0    1
```

```
ans =
  2    0    1
  2    3    4
  5    6    9
ans =
  2    4    6
  3    5    9
  0    1    2
```

Cell Array

Cell arrays are arrays of indexed cells where each cell can store an array of a different dimensions and data types.

The **cell** function is used for creating a cell array. Syntax for the cell function is −

```
C = cell(dim)
C = cell(dim1,...,dimN)
D = cell(obj)
```

Where,

- *C* is the cell array;
- *dim* is a scalar integer or vector of integers that specifies the dimensions of cell array C;
- *dim1, ... , dimN* are scalar integers that specify the dimensions of C;
- *obj* is One of the following −
    - Java array or object
    - .NET array of type System.String or System.Object

Example

Create a script file and type the following code into it −

```
c = cell(2, 5);
c = {'Red', 'Blue', 'Green', 'Yellow', 'White'; 1 2 3 4 5}
```

When you run the file, it displays the following result −

```
c =
{
  [1,1] = Red
  [2,1] =  1
  [1,2] = Blue
  [2,2] =  2
  [1,3] = Green
  [2,3] =  3
  [1,4] = Yellow
  [2,4] =  4
  [1,5] = White
  [2,5] =  5
}
```

Accessing Data in Cell Arrays

There are two ways to refer to the elements of a cell array −

- Enclosing the indices in first bracket (), to refer to sets of cells
- Enclosing the indices in braces {}, to refer to the data within individual cells

When you enclose the indices in first bracket, it refers to the set of cells.

Cell array indices in smooth parentheses refer to sets of cells.

For example −

```
c = {'Red', 'Blue', 'Green', 'Yellow', 'White'; 1 2 3 4 5};
c(1:2,1:2)
```

MATLAB will execute the above statement and return the following result −

```
ans =
{
  [1,1] = Red
  [2,1] =  1
  [1,2] = Blue
  [2,2] =  2
}
```

You can also access the contents of cells by indexing with curly braces.

For example −

```
c = {'Red', 'Blue', 'Green', 'Yellow', 'White'; 1 2 3 4 5};
c{1, 2:4}
```

MATLAB will execute the above statement and return the following result −

```
ans = Blue
ans = Green
ans = Yellow
```

Matrix

A matrix is a two-dimensional array of numbers.

In MATLAB, you create a matrix by entering elements in each row as comma or space delimited numbers and using semicolons to mark the end of each row.

For example, let us create a 4-by-5 matrix *a* −

```
a = [ 1 2 3 4 5; 2 3 4 5 6; 3 4 5 6 7; 4 5 6 7 8]
```

MATLAB will execute the above statement and return the following result −

```
a =
     1    2    3    4    5
     2    3    4    5    6
     3    4    5    6    7
     4    5    6    7    8
```

Referencing the Elements of a Matrix

To reference an element in the mth row and nth column, of a matrix *mx*, we write −

mx(m, n);

For example, to refer to the element in the 2nd row and 5th column, of the matrix *a*, as created in the last section, we type −

```
a = [ 1 2 3 4 5; 2 3 4 5 6; 3 4 5 6 7; 4 5 6 7 8];
a(2,5)
```

MATLAB will execute the above statement and return the following result −

```
ans =  6
```

To reference all the elements in the mth column we type A(:,m).

Let us create a column vector v, from the elements of the 4th row of the matrix a −

```
a = [ 1 2 3 4 5; 2 3 4 5 6; 3 4 5 6 7; 4 5 6 7 8];
v = a(:,4)
```

MATLAB will execute the above statement and return the following result −

```
v =
     4
     5
     6
     7
```

You can also select the elements in the mth through nth columns, for this we write −

```
a(:,m:n)
```

Let us create a smaller matrix taking the elements from the second and third columns −

```
a = [ 1 2 3 4 5; 2 3 4 5 6; 3 4 5 6 7; 4 5 6 7 8];
a(:, 2:3)
```

MATLAB will execute the above statement and return the following result −

```
ans =
     2     3
     3     4
     4     5
     5     6
```

In the same way, you can create a sub-matrix taking a sub-part of a matrix.

```
a = [ 1 2 3 4 5; 2 3 4 5 6; 3 4 5 6 7; 4 5 6 7 8];
a(:, 2:3)
```

MATLAB will execute the above statement and return the following result −

```
ans =
     2     3
     3     4
     4     5
     5     6
```

In the same way, you can create a sub-matrix taking a sub-part of a matrix.

For example, let us create a sub-matrix *sa* taking the inner subpart of a −

```
3     4     5
4     5     6
```

To do this, write −

```
a = [ 1 2 3 4 5; 2 3 4 5 6; 3 4 5 6 7; 4 5 6 7 8];
sa = a(2:3,2:4)
```

MATLAB will execute the above statement and return the following result −

```
sa =
     3     4     5
     4     5     6
```

Deleting a Row or a Column in a Matrix

You can delete an entire row or column of a matrix by assigning an empty set of square braces [] to that row or column. Basically, [] denotes an empty array.

For example, let us delete the fourth row of a −

```
a = [ 1 2 3 4 5; 2 3 4 5 6; 3 4 5 6 7; 4 5 6 7 8];
a( 4 , : ) = []
```

MATLAB will execute the above statement and return the following result −

```
a =
    1    2    3    4    5
    2    3    4    5    6
    3    4    5    6    7
```

Next, let us delete the fifth column of a −

```
a = [ 1 2 3 4 5; 2 3 4 5 6; 3 4 5 6 7; 4 5 6 7 8];
a(: , 5)=[]
```

MATLAB will execute the above statement and return the following result −

```
a =
    1    2    3    4
    2    3    4    5
    3    4    5    6
    4    5    6    7
```

Example

In this example, let us create a 3-by-3 matrix m, then we will copy the second and third rows of this matrix twice to create a 4-by-3 matrix.

Create a script file with the following code −

```
a = [ 1 2 3 ; 4 5 6; 7 8 9];
new_mat = a([2,3,2,3],:)
```

When you run the file, it displays the following result −

```
new_mat =
    4    5    6
    7    8    9
    4    5    6
    7    8    9
```

Matrix Operations

In this section, let us discuss the following basic and commonly used matrix operations −

- Addition and Subtraction of Matrices
- Division of Matrices
- Scalar Operations of Matrices
- Transpose of a Matrix
- Concatenating Matrices
- Matrix Multiplication

- Determinant of a Matrix
- Inverse of a Matrix

Working with data files

Tools that Import Multiple File Formats

You can import data into MATLAB from a disk file or the system clipboard interactively.

To import data from a file, do one of the following:

- On the **Home** tab, in the **Variable** section, select **Import Data** .
- Double-click a file name in the Current Folder browser.
- Call uiimport.

To import data from the clipboard, do one of the following:

- On the Workspace browser title bar, click , and then select **Paste**.
- Call uiimport.

To import without invoking a graphical user interface, the easiest option is to use the importdata function.

For a complete list of the formats you can import interactively or with importdata, see Supported File Formats for Import and Export.

Importing Specific File Formats

MATLAB includes functions tailored to import specific file formats. Consider using format-specific functions instead of importing data interactively when you want to import only a portion of a file. Many of the format-specific functions provide options for selecting ranges or portions of data. Some format-specific functions allow you to request multiple optional outputs. This option is not available when you import interactively.

For a complete list of the format-specific functions, see Supported File Formats for Import and Export.

For binary data files, consider Overview of Memory-Mapping. Memory-mapping enables you to access file data using standard MATLAB indexing operations.

Alternatively, MATLAB toolboxes perform specialized import operations. For example, use Database Toolbox™ software for importing data from relational databases. Refer to the documentation on specific toolboxes to see the available import features.

Importing Data with Low-Level I/O

If the Import Wizard, importdata, and format-specific functions cannot read your data, use *low-level I/O functions* such as fscanf or fread. Low-level functions allow the most control over reading from a file, but require detailed knowledge of the structure of your data. For more information, see:

- Import Text Data Files with Low-Level I/O
- Import Binary Data with Low-Level I/O

Importing data in MATLAB means loading data from an external file. The **importdata** function allows loading various data files of different formats. It has the following five forms −

Sr.No.	Function & Description
1	**A = importdata(filename)** Loads data into array A from the file denoted by *filename*.
2	**A = importdata('-pastespecial')** Loads data from the system clipboard rather than from a file.
3	**A = importdata(___, delimiterIn)** Interprets *delimiterIn* as the column separator in ASCII file, filename, or the clipboard data. You can use *delimiterIn* with any of the input arguments in the above syntaxes.
4	**A = importdata(___, delimiterIn, headerlinesIn)** Loads data from ASCII file, filename, or the clipboard, reading numeric data starting from line *headerlinesIn+1*.
5	**[A, delimiterOut, headerlinesOut] = importdata(___)** Returns the detected delimiter character for the input ASCII file in *delimiterOut* and the detected number of header lines in *headerlinesOut*, using any of the input arguments in the previous syntaxes.

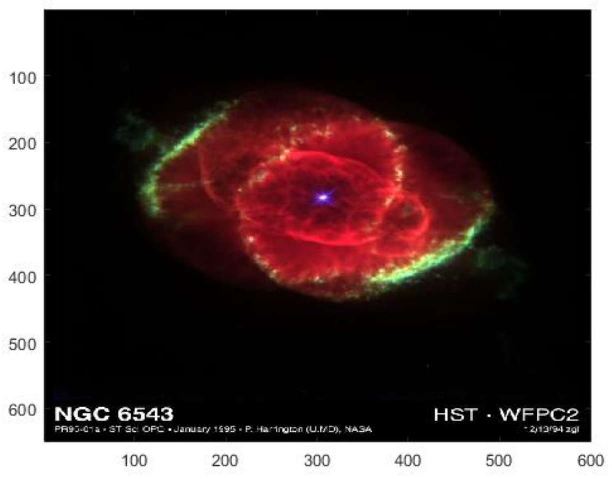
By default, Octave does not have support for *importdata()* function, so you will have to search and install this package to make following examples work with your Octave installation.

Example 1

Let us load and display an image file. Create a script file and type the following code in it −

```
filename = 'random.jpg';
A = importdata(filename);
image(A);
```

When you run the file, MATLAB displays the image file. However, you must store it in the current directory.



Example 2

In this example, we import a text file and specify Delimiter and Column Header. Let us create a space-delimited ASCII file with column headers, named *weeklydata.txt*.

Our text file weeklydata.txt looks like this −

```
SunDay  MonDay  TuesDay  WednesDay  ThursDay  FriDay  SaturDay
95.01   76.21   61.54    40.57      55.79     70.28   81.53
73.11   45.65   79.19    93.55      75.29     69.87   74.68
60.68   41.85   92.18    91.69      81.32     90.38   74.51
48.60   82.14   73.82    41.03      0.99      67.22   93.18
89.13   44.47   57.63    89.36      13.89     19.88   46.60
```

Create a script file and type the following code in it −

```
filename = 'weeklydata.txt';
delimiterIn = ' ';
headerlinesIn = 1;
A = importdata(filename,delimiterIn,headerlinesIn);

% View data
for k = [1:7]
   disp(A.colheaders{1, k})
   disp(A.data(:, k))
   disp(' ')
end
```

When you run the file, it displays the following result −

```
SunDay
```

```
   95.0100
   73.1100
   60.6800
   48.6000
   89.1300

MonDay
   76.2100
   45.6500
   41.8500
   82.1400
   44.4700

TuesDay
   61.5400
   79.1900
   92.1800
   73.8200
   57.6300

WednesDay
   40.5700
   93.5500
   91.6900
   41.0300
   89.3600

ThursDay
   55.7900
   75.2900
   81.3200
   0.9900
   13.8900
```

```
FriDay
   70.2800
   69.8700
   90.3800
   67.2200
   19.8800

SaturDay
   81.5300
   74.6800
   74.5100
   93.1800
   46.6000
```

Example 3

In this example, let us import data from clipboard.

Copy the following lines to the clipboard −

Mathematics is simple

Create a script file and type the following code −

```
A = importdata('-pastespecial')
```

When you run the file, it displays the following result −

```
A =
   'Mathematics is simple'
```

Low-Level File I/O

The *importdata* function is a high-level function. The low-level file I/O functions in MATLAB allow the most control over reading or writing data to a file. However, these functions need more detailed information about your file to work efficiently.

MATLAB provides the following functions for read and write operations at the byte or character level −

Function	Description
fclose	Close one or all open files
feof	Test for end-of-file

ferror	Information about file I/O errors
fgetl	Read line from file, removing newline characters
fgets	Read line from file, keeping newline characters
fopen	Open file, or obtain information about open files
fprintf	Write data to text file
fread	Read data from binary file
frewind	Move file position indicator to beginning of open file
fscanf	Read data from text file
fseek	Move to specified position in file
ftell	Position in open file
fwrite	Write data to binary file

Import Text Data Files with Low-Level I/O

MATLAB provides the following functions for low-level import of text data files −

- The **fscanf** function reads formatted data in a text or ASCII file.
- The **fgetl** and **fgets** functions read one line of a file at a time, where a newline character separates each line.
- The **fread** function reads a stream of data at the byte or bit level.

Example

We have a text data file 'myfile.txt' saved in our working directory. The file stores rainfall data for three months; June, July and August for the year 2012.

The data in myfile.txt contains repeated sets of time, month and rainfall measurements at five places. The header data stores the number of months M; so we have M sets of measurements.

The file looks like this −

Rainfall Data

Months: June, July, August

M = 3

12:00:00

June-2012

```
17.21  28.52  39.78  16.55 23.67
19.15  0.35   17.57  NaN   12.01
17.92  28.49  17.40  17.06 11.09
9.59   9.33   NaN    0.31  0.23
10.46  13.17  NaN    14.89 19.33
20.97  19.50  17.65  14.45 14.00
18.23  10.34  17.95  16.46 19.34
09:10:02
July-2012
12.76  16.94  14.38  11.86 16.89
20.46  23.17  NaN    24.89 19.33
30.97  49.50  47.65  24.45 34.00
18.23  30.34  27.95  16.46 19.34
30.46  33.17  NaN    34.89  29.33
30.97  49.50  47.65  24.45 34.00
28.67  30.34  27.95  36.46 29.34
15:03:40
August-2012
17.09  16.55  19.59  17.25 19.22
17.54  11.45  13.48  22.55 24.01
NaN    21.19  25.85  25.05 27.21
26.79  24.98  12.23  16.99 18.67
17.54  11.45  13.48  22.55 24.01
NaN    21.19  25.85  25.05 27.21
26.79  24.98  12.23  16.99 18.67
```

We will import data from this file and display this data. Take the following steps −

- Open the file with **fopen** function and get the file identifier.
- Describe the data in the file with **format specifiers**, such as '**%s**' for a string, '**%d**' for an integer, or '**%f**' for a floating-point number.
- To skip literal characters in the file, include them in the format description. To skip a data field, use an asterisk ('*') in the specifier.

  For example, to read the headers and return the single value for M, we write −

```
M = fscanf(fid, '%*s %*s\n%*s %*s %*s %*s\nM=%d\n\n', 1);
```

- By default, **fscanf** reads data according to our format description until it does not find any match for the data, or it reaches the end of the file. Here we will use for loop for reading 3 sets of data and each time, it will read 7 rows and 5 columns.
- We will create a structure named *mydata* in the workspace to store data read from the file. This structure has three fields - *time*, *month*, and *raindata* array.

Create a script file and type the following code in it −

```
filename = '/data/myfile.txt';
rows = 7;
cols = 5;

% open the file
fid = fopen(filename);

% read the file headers, find M (number of months)
M = fscanf(fid, '%*s %*s\n%*s %*s %*s %*s\nM=%d\n\n', 1);

% read each set of measurements
for n = 1:M
   mydata(n).time = fscanf(fid, '%s', 1);
   mydata(n).month = fscanf(fid, '%s', 1);

   % fscanf fills the array in column order,
   % so transpose the results
   mydata(n).raindata  = ...
      fscanf(fid, '%f', [rows, cols]);
end
for n = 1:M
   disp(mydata(n).time), disp(mydata(n).month)
   disp(mydata(n).raindata)
end

% close the file
fclose(fid);
```

When you run the file, it displays the following result −

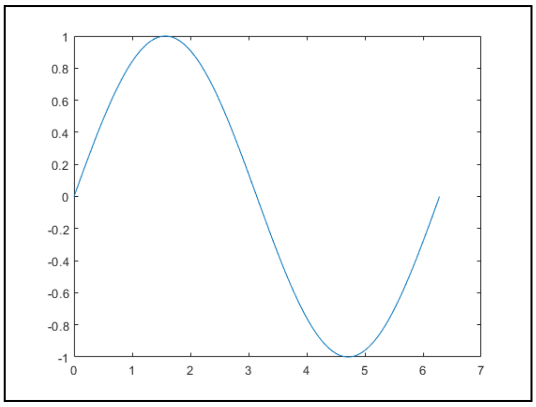
```
12:00:00
June-2012
   17.2100   17.5700   11.0900   13.1700   14.4500
   28.5200       NaN    9.5900       NaN   14.0000
   39.7800   12.0100    9.3300   14.8900   18.2300
   16.5500   17.9200       NaN   19.3300   10.3400
   23.6700   28.4900    0.3100   20.9700   17.9500
   19.1500   17.4000    0.2300   19.5000   16.4600
   0.3500   17.0600   10.4600   17.6500   19.3400

09:10:02
July-2012
   12.7600       NaN   34.0000   33.1700   24.4500
   16.9400   24.8900   18.2300       NaN   34.0000
   14.3800   19.3300   30.3400   34.8900   28.6700
   11.8600   30.9700   27.9500   29.3300   30.3400
   16.8900   49.5000   16.4600   30.9700   27.9500
   20.4600   47.6500   19.3400   49.5000   36.4600
   23.1700   24.4500   30.4600   47.6500   29.3400

15:03:40
August-2012
   17.0900   13.4800   27.2100   11.4500   25.0500
   16.5500   22.5500   26.7900   13.4800   27.2100
   19.5900   24.0100   24.9800   22.5500   26.7900
   17.2500       NaN   12.2300   24.0100   24.9800
   19.2200   21.1900   16.9900       NaN   12.2300
   17.5400   25.8500   18.6700   21.1900   16.9900
   11.4500   25.0500   17.5400   25.8500   18.6700
```

Line Plots

To create two-dimensional line plots, use the plot function. For example, plot the value of the sine function from 0 to 2π:

```
x = 0:pi/100:2*pi;
y = sin(x);
```

plot(x,y)

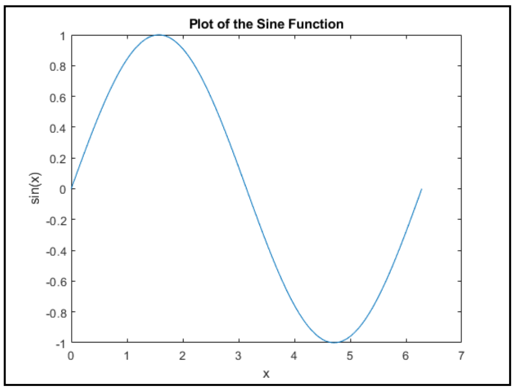


You can label the axes and add a title.

xlabel('x')

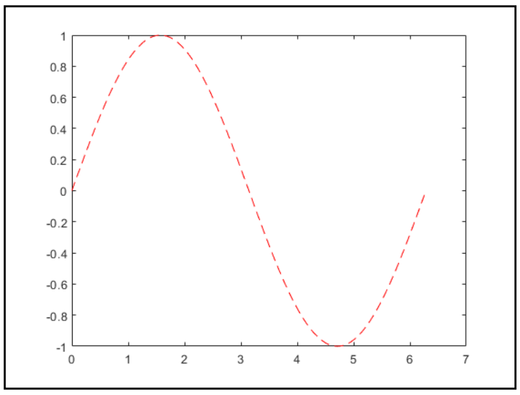
ylabel('sin(x)')

title('Plot of the Sine Function')



By adding a third input argument to the plot function, you can plot the same variables using a red dashed line.

plot(x,y,'r--')

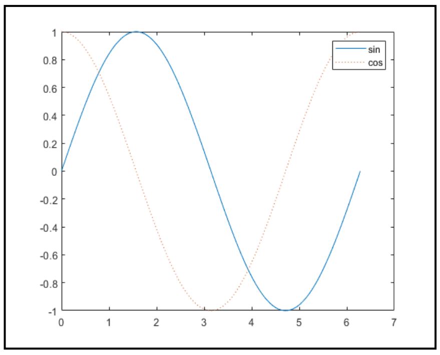


'r--' is a *line specification*. Each specification can include characters for the line color, style, and marker. A marker is a symbol that appears at each plotted data point, such as a +, o, or *. For example, 'g:*' requests a dotted green line with * markers.

Notice that the titles and labels that you defined for the first plot are no longer in the current *figure* window. By default, MATLAB® clears the figure each time you call a plotting function, resetting the axes and other elements to prepare the new plot.

To add plots to an existing figure, use hold on. Until you use hold off or close the window, all plots appear in the current figure window.

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)

hold on

y2 = cos(x);
plot(x,y2,':')
legend('sin','cos')
hold off
```



3-D Plots

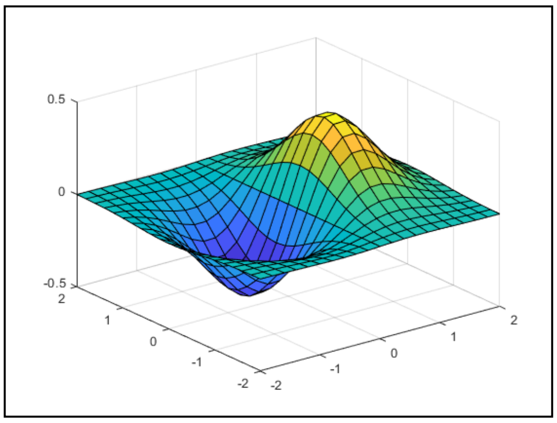
Three-dimensional plots typically display a surface defined by a function in two variables, $z = f(x,y)$.

To evaluate z, first create a set of (x,y) points over the domain of the function using meshgrid.

```
[X,Y] = meshgrid(-2:.2:2);
Z = X .* exp(-X.^2 - Y.^2);
```

Then, create a surface plot.

```
surf(X,Y,Z)
```



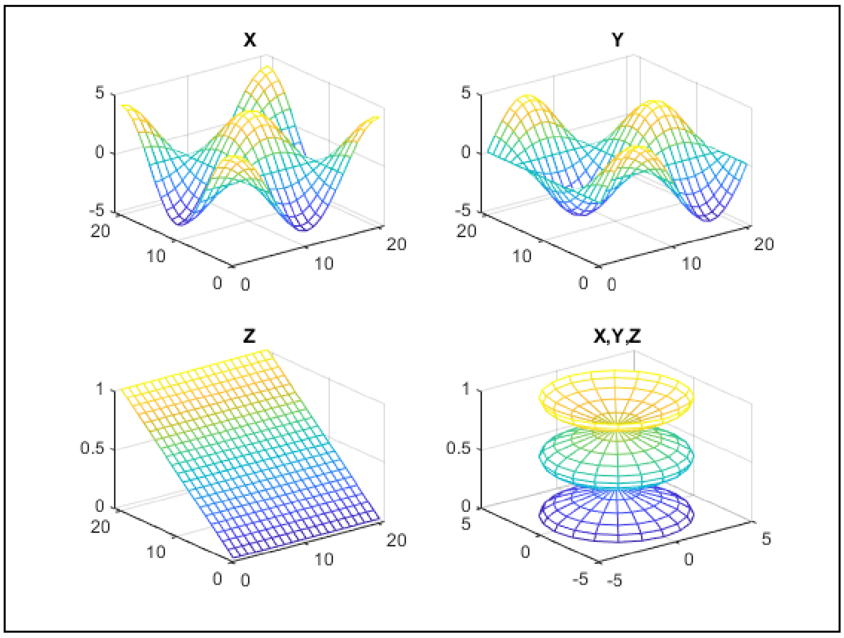
Both the surf function and its companion mesh display surfaces in three dimensions. surf displays both the connecting lines and the faces of the surface in color. mesh produces wireframe surfaces that color only the lines connecting the defining points.

Subplots

You can display multiple plots in different subregions of the same window using the subplot function.

The first two inputs to subplot indicate the number of plots in each row and column. The third input specifies which plot is active. For example, create four plots in a 2-by-2 grid within a figure window.

```
t = 0:pi/10:2*pi;
[X,Y,Z] = cylinder(4*cos(t));
subplot(2,2,1); mesh(X); title('X');
subplot(2,2,2); mesh(Y); title('Y');
subplot(2,2,3); mesh(Z); title('Z');
subplot(2,2,4); mesh(X,Y,Z); title('X,Y,Z');
```



ADDITIONAL PROGRAMS

PROGRAM 9 : Python Program to detect if two strings are anagrams.

Problem Description

Aim: The program takes two strings and checks if the two strings are anagrams.

Problem Solution

1. Take two strings from the user and store them in separate variables.
2. Then use sorted() to sort both the strings into lists.
3. Compare the sorted lists and check if they are equal.
4. Print the final result.
5. Exit.

Program/Source Code

Here is source code of the Python Program to detect if two strings are anagrams. The program output is also shown below.

```
s1=raw_input("Enter first string:")
s2=raw_input("Enter second string:")
if(sorted(s1)==sorted(s2)):
     print("The strings are anagrams.")
else:
     print("The strings aren't anagrams.")
```

Program Explanation

1. User must enter both the strings and store them in separate variables.
2. The characters of both the strings are sorted into separate lists.
3. They are then checked whether they are equal or not using an if statement.
4. If they are equal, they are anagrams as the characters are simply jumbled in anagrams.
5. If they aren't equal, the strings aren't anagrams.
6. The final result is printed.

Runtime Test Cases

```
 Case 1:
Enter first string:anagram
Enter second string:nagaram
The strings are anagrams.

Case 2:
Enter first string:hello
Enter second string:world
The strings aren't anagrams.
```

PROGRAM 10.Python Program to calculate the number of digits and letters in a string.

Problem Description

Aim: The program takes a string and calculates the number of digits and letters in a string.

Problem Solution

1. Take a string from the user and store it in a variable.
2. Initialize the two count variables to 0.
3. Use a for loop to traverse through the characters in the string and increment the first count variable each time a digit is encountered and increment the second count variable each time a character is encountered.
4. Print the total count of both the variables.
5. Exit.

Program/Source Code

Here is source code of the Python Program to calculate the number of digits and letters in a string. The program output is also shown below.

```
string=raw_input("Enter string:")
count1=0
count2=0
for i in string:
    if(i.isdigit()):
        count1=count1+1
    count2=count2+1
print("The number of digits is:")
print(count1)
print("The number of characters is:")
print(count2)
```

Program Explanation

1. User must enter a string and store it in a variable.
2. Both the count variables are initialized to zero.
3. The for loop is used to traverse through the characters in the string.
4. The first count variable is incremented each time a digit is encountered and the second count

variable is incremented each time a character is encountered.

5. The total count of digits and characters in the string are printed.

Runtime Test Cases

```
Case 1:
Enter string:Hello123
The number of digits is:
3
The number of characters is:
8

Case 2:
Enter string:Abc12
The number of digits is:
2
The number of characters is:
5
```

PROGRAM 11.Python Program to check common letters in the two input strings.

Problem Description

Aim: The program takes two strings and checks common letters in both the strings.

Problem Solution

1. Enter two input strings and store it in separate variables.
2. Convert both of the strings into sets and find the common letters between both the sets.
3. Store the common letters in a list.
4. Use a for loop to print the letters of the list.
5. Exit.

Program/Source Code

Here is source code of the Python Program to check common letters in the two input strings. The program output is also shown below.

```
s1=raw_input("Enter first string:")
s2=raw_input("Enter second string:")
a=list(set(s1)&set(s2))
print("The common letters are:")
for i in a:
    print(i)
```

Program Explanation

1. User must enter two input strings and store it in separate variables.
2. Both of the strings are converted into sets and the common letters between both the sets are found using the ‘&’ operator.
3. These common letters are stored in a list.
4. A for loop is used to print the letters of the list.

Runtime Test Cases

```
Case 1:
Enter first string:Hello
Enter second string:How are you
The common letters are:
H
e
o

Case 2:
Enter first string:Test string
Enter second string:checking
The common letters are:
i
e
g
n
```

PROGRAM 12.Python Program to find the factorial of a number using recursion.

Problem Description

Aim: The program takes a number and determines the factorial of the number using recursion.

Problem Solution

1. Take a number from the user and store it in a variable.
2. Pass the number as an argument to a recursive factorial function.
3. Define the base condition as the number to be lesser than or equal to 1 and return 1 if it is.
4. Otherwise call the function recursively with the number minus 1 multiplied by the number itself.
5. Then return the result and print the factorial of the number.
6. Exit.

Program/Source Code

Here is source code of the Python Program to find the factorial of a number using recursion. The program output is also shown below.

```
def factorial(n):
    if(n <= 1):
        return 1
    else:
        return(n*factorial(n-1))
n = int(input("Enter number:"))
print("Factorial:")
print(factorial(n))
```

Program Explanation

1. User must enter a number and store it in a variable.
2. The number is passed as an argument to a recursive factorial function.
3. The base condition is that the number has to be lesser than or equal to 1 and return 1 if it is.
4. Otherwise the function is called recursively with the number minus 1 multiplied by the number itself.
5. The result is returned and the factorial of the number is printed.

Runtime Test Cases

```
Case 1:
Enter number:5
Factorial:
120

Case 2:
Enter number:9
Factorial:
362880
```

PROGRAM 13. Python Program to find the area of a rectangle using classes.

Problem Description

Aim: The program takes the length and breadth from the user and finds the area of the rectangle using classes.

Problem Solution

1. Take the value of length and breadth from the user.
2. Create a class and using a constructor initialise values of that class.
3. Create a method called as area and return the area of the class.
4. Create an object for the class.
5. Using the object, call the method area() with the parameters as the length and breadth taken from the user.
6. Print the area.
7. Exit

Program/Source Code

Here is the source code of the Python Program to take the length and breadth from the user and find the area of the rectangle. The program output is also shown below.

```
class rectangle():
    def __init__(self,breadth,length):
        self.breadth=breadth
        self.length=length
    def area(self):
        return self.breadth*self.length
a=int(input("Enter length of rectangle: "))
b=int(input("Enter breadth of rectangle: "))
obj=rectangle(a,b)
print("Area of rectangle:",obj.area())
 print()
```

Program Explanation

1. User must enter the value of length and breadth.
2. A class called rectangle is created and the __init__() method is used to initialise values of that class.
3. A method called as area, returns self.length*self.breadth which is the area of the class.
4. An object for the class is created.
5. Using the object, the method area() is called with the parameters as the length and breadth taken from the user.
6. The area is printed.

Runtime Test Cases

```
 Case 1:
Enter length of rectangle: 4
Enter breadth of rectangle: 5
Area of rectangle: 20

Case 2:
Enter length of rectangle: 15
Enter breadth of rectangle: 13
Area of rectangle: 195
```

PROGRAM 14. Python Program to find the area of a rectangle using classes.

Problem Description

Aim: The program takes the length and breadth from the user and finds the area of the rectangle using classes.

Problem Solution

1. Create a class and using a constructor to initialize values of that class.
2. Create methods for adding, substracting, multiplying and dividing two numbers and returning the respective results.
3. Take the two numbers as inputs and create an object for the class passing the two numbers as parameters to the class.
4. Using the object, call the respective function depending on the choice taken from the user.
5. Print the final result.
6. Exit

Program/Source Code

Here is the source code of the Python Program to take the length and breadth from the user and find the area of the rectangle. The program output is also shown below.

```
class cal():
    def __init__(self,a,b):
        self.a=a
        self.b=b
    def add(self):
        return self.a+self.b
    def mul(self):
        return self.a*self.b
    def div(self):
        return self.a/self.b
    def sub(self):
        return self.a-self.b
a=int(input("Enter first number: "))
b=int(input("Enter second number: "))
```

```
obj=cal(a,b)
choice=1
while choice!=0:
   print("0. Exit")
   print("1. Add")
   print("2. Subtraction")
   print("3. Multiplication")
   print("4. Division")
   choice=int(input("Enter choice: "))
   if choice==1:
      print("Result: ",obj.add())
   elif choice==2:
      print("Result: ",obj.sub())
   elif choice==3:
      print("Result: ",obj.mul())
   elif choice==4:
      print("Result: ",round(obj.div(),2))
   elif choice==0:
      print("Exiting!")
   else:
      print("Invalid choice!!")


print()
```

Program Explanation

1. A class called cal is created and the __init__() method is used to initialize values of that class.
2. Methods for adding, substracting, multiplying, dividing two numbers and returning their respective results is defined.
3. The menu is printed and the choice is taken from the user.
4. An object for the class is created with the two numbers taken from the user passed as parameters.
5. Using the object, the respective method is called according to the choice taken from the user.
6. When the choice is 0, the loop is exited.
7. The final result is printed.

Runtime Test Cases

```
Case 1:
Enter first number: 2
Enter second number: 4
0. Exit
1. Add
2. Subtraction
3. Multiplication
4. Division
Enter choice: 1
Result:  6
0. Exit
1. Add
2. Subtraction
3. Multiplication
4. Division
Enter choice: 3
Result:  8
0. Exit
1. Add
2. Subtraction
3. Multiplication
4. Division
Enter choice: 0
Exiting!

Case 2:
Enter first number: 150
Enter second number: 50
0. Exit
1. Add
2. Subtraction
3. Multiplication
```

```
4. Division
Enter choice: 2
Result:  100
0. Exit
1. Add
2. Subtraction
3. Multiplication
4. Division
Enter choice: 4
Result:  3.0
0. Exit
1. Add
2. Subtraction
3. Multiplication
4. Division
Enter choice: 0
Exiting!
```

PROGRAM 15: Extracting the individual element(s) of a matrix

Aim: retrieve single element from the matrix

```
A = [3 5; 2  4]; c =
A(2,2)+A(1,2)
```

Output:

```
c = 9
```

PROGRAM 16: Create a prompt to request user input

Aim: reading input from the user

```
num1 = input('Enter your age'); if (num1 > 17)

fprintf('You are eligible to vote') else fprintf('You are not eligible to vote')
end
```

PROGRAM 17: MAT LAB program to develop An application - determine whether a given year is a leap year (try to change the given value of nyear and observe the outcome)

Aim: Find weather given year is leap year or not

```
nyear = 1975;

if (mod(nyear, 400) == 0) fprintf('%6u is a leap year', nyear)

elseif (mod(nyear,4) == 0) & (mod(nyear,100) ~= 0) fprintf('%6u is a
leap year', nyear)

else

fprintf('%6u is not a leap year', nyear) end
```

Output:

```
1975 is not a leap year
```

VIVA VOCE

PYTHON VIVA QUESTIONS

1.What are the applications of Python?

Python is used in various software domains some application areas are given below.

- Web and Internet Development
- Games
- Scientific and computational applications
- Language development
- Image processing and graphic design applications
- Enterprise and business applications development
- Operating systems
- GUI based desktop applications

2.What are the advantages of Python?

- Interpreted
- Free and open source
- Extensible
- Object-oriented
- Built-in data structure
- Readability
- High-Level Language
- Cross-platform

  Interpreted: Python is an interpreted language. It does not require prior compilation of code and executes instructions directly.
- Free and open source: It is an open source project which is publicly available to reuse. It can be downloaded free of cost.
- Portable: Python programs can run on cross platforms without affecting its performance.

3. What do you mean by Python literals?

Literals can be defined as a data which is given in a variable or constant. Python supports the following literals:

String Literals

String literals are formed by enclosing text in the single or double quotes. For example, string literals are string values.

E.g.:"Aman", '12345'.

4. Explain Python Functions?

A function is a section of the program or a block of code that is written once and can be executed whenever required in the program. A function is a block of self-contained statements which has a valid name, parameters list, and body. Functions make programming more functional and modular to perform modular tasks. Python provides several built-in functions to complete tasks and also allows a user to create new functions as well.

There are two types of functions:

- Built-In Functions: copy(), len(), count() are the some built-in functions.
- User-defined Functions: Functions which are defined by a user known as user-defined functions.

5.What is Python's parameter passing mechanism?

There are two parameters passing mechanism in Python:

- Pass by references
- Pass by value

6.Is python a case sensitive language?

Yes! Python is a case sensitive programming language.

7.What are the supported data types in Python?

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

8.What is the output of print str if str = 'Hello World!'?

It will print complete string. Output would be Hello World!.

9.What is the output of print str[0] if str = 'Hello World!'?

It will print first character of the string. Output would be H.

10.What is the output of print str[2:5] if str = 'Hello World!'?

It will print characters starting from 3rd to 5th. Output would be llo.

11.What is the output of print str[2:] if str = 'Hello World!'?

It will print characters starting from 3rd character. Output would be llo World!.

12.What is the output of print str * 2 if str = 'Hello World!'?

It will print string two times. Output would be Hello World!Hello World!.

13.What is the output of print str + "TEST" if str = 'Hello World!'?

It will print concatenated string. Output would be Hello World!TEST.

14.What is the output of print list if list = ['abcd', 786 , 2.23, 'john', 70.2]?

It will print complete list. Output would be ['abcd', 786, 2.23, 'john', 70.200000000000003].

15.What is the output of print list[0] if list = ['abcd', 786 , 2.23, 'john', 70.2]?

It will print first element of the list. Output would be abcd.

16.What is the output of print list[1:3] if list = ['abcd', 786 , 2.23, 'john', 70.2]?

It will print elements starting from 2nd till 3rd. Output would be [786, 2.23].

17.What is the output of print list[2:] if list = ['abcd', 786 , 2.23, 'john', 70.2]?

It will print elements starting from 3rd element. Output would be [2.23, 'john', 70.200000000000003].

18.What is the output of print tinylist * 2 if tinylist = [123, 'john']?

It will print list two times. Output would be [123, 'john', 123, 'john'].

19.What is the output of print list1 + list2, if list1 = ['abcd', 786 , 2.23, 'john', 70.2] and ist2 = [123, 'john']?

It will print concatenated lists. Output would be ['abcd', 786, 2.23, 'john', 70.2, 123, 'john']

20.What are tuples in Python?

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

21.What is the difference between tuples and lists in Python?

The main differences between lists and tuples are − Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists.

22.What is the output of print tuple if tuple = ('abcd', 786 , 2.23, 'john', 70.2)?

It will print complete tuple. Output would be ('abcd', 786, 2.23, 'john', 70.200000000000003).

23.What is the output of print tuple[0] if tuple = ('abcd', 786 , 2.23, 'john', 70.2)?

It will print first element of the tuple. Output would be abcd.

24.What is the output of print tuple[1:3] if tuple = ('abcd', 786 , 2.23, 'john', 70.2)?

It will print elements starting from 2nd till 3rd. Output would be (786, 2.23).

25.What is the output of print tuple[2:] if tuple = ('abcd', 786 , 2.23, 'john', 70.2)?

It will print elements starting from 3rd element. Output would be (2.23, 'john', 70.200000000000003).

26.What is the output of print tinytuple * 2 if tinytuple = (123, 'john')?

It will print tuple two times. Output would be (123, 'john', 123, 'john').

27.What is the output of print tuple + tinytuple if tuple = ('abcd', 786 , 2.23, 'john', 70.2) and tinytuple = (123, 'john')?

It will print concatenated tuples. Output would be ('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john').

28.What are Python's dictionaries?

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

29.How will you create a dictionary in python?

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
```

30.How will you get all the keys from the dictionary?

Using dictionary.keys() function, we can get all the keys from the dictionary object.

```
print dict.keys()   # Prints all the keys
```

31.How will you get all the values from the dictionary?

Using dictionary.values() function, we can get all the values from the dictionary object.

```
print dict.values()   # Prints all the values
```

32.How will you convert a string to an int in python?

int(x [,base]) - Converts x to an integer. base specifies the base if x is a string.

33.How will you convert a string to a long in python?

long(x [,base]) - Converts x to a long integer. base specifies the base if x is a string.

34.How will you convert a string to a float in python?

float(x) − Converts x to a floating-point number.

35.How will you convert a object to a string in python?

str(x) − Converts object x to a string representation.

36.How will you convert a object to a regular expression in python?

repr(x) − Converts object x to an expression string.

37.How will you convert a String to an object in python?

eval(str) − Evaluates a string and returns an object.

38.How will you convert a string to a tuple in python?

tuple(s) − Converts s to a tuple.

39.How will you convert a string to a list in python?

list(s) − Converts s to a list.

40.How will you convert a string to a set in python?

set(s) − Converts s to a set.

41.How will you create a dictionary using tuples in python?

dict(d) − Creates a dictionary. d must be a sequence of (key,value) tuples.

42.How will you convert a string to a frozen set in python?

frozenset(s) − Converts s to a frozen set.

43.How will you convert an integer to a character in python?

chr(x) − Converts an integer to a character.

44.How will you convert an integer to an unicode character in python?

unichr(x) − Converts an integer to a Unicode character.

45.How will you convert a single character to its integer value in python?

ord(x) − Converts a single character to its integer value.

46.How will you convert an integer to hexadecimal string in python?

hex(x) − Converts an integer to a hexadecimal string.

47.How will you convert an integer to octal string in python?

oct(x) − Converts an integer to an octal string.

48.What is the purpose of ** operator?

** Exponent − Performs exponential (power) calculation on operators. a**b = 10 to the power 20 if a = 10 and b = 20.

49.What is the purpose of // operator?

// Floor Division − The division of operands where the result is the quotient in which the digits after the decimal point are removed.

50.What is the purpose of is operator?

is − Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. x is y, here is results in 1 if id(x) equals id(y).

51.What is the purpose of not in operator?

not in − Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. x not in y, here not in results in a 1 if x is not a member of sequence y.

52.What is the purpose break statement in python?

break statement − Terminates the loop statement and transfers execution to the statement immediately following the loop.

53.What is the purpose continue statement in python?

continue statement − Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

54.What is the purpose pass statement in python?

pass statement − The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

55.How can you pick a random item from a list or tuple?

choice(seq) − Returns a random item from a list, tuple, or string.

56.How can you pick a random item from a range?

randrange ([start,] stop [,step]) − returns a randomly selected element from range(start, stop, step).

57.How can you get a random number in python?

random() − returns a random float r, such that 0 is less than or equal to r and r is less than 1.

58.How will you set the starting value in generating random numbers?

seed([x]) − Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.

59.How will you randomizes the items of a list in place?

shuffle(lst) − Randomizes the items of a list in place. Returns None.

60.How will you capitalizes first letter of string?

capitalize() − Capitalizes first letter of string.

61.How will you check in a string that all characters are alphanumeric?

isalnum() − Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

62.How will you check in a string that all characters are digits?

isdigit() − Returns true if string contains only digits and false otherwise.

63.How will you check in a string that all characters are in lowercase?

islower() − Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.

64.How will you check in a string that all characters are numerics?

isnumeric() − Returns true if a unicode string contains only numeric characters and false otherwise.

65.How will you check in a string that all characters are whitespaces?

isspace() − Returns true if string contains only whitespace characters and false otherwise.

66.How will you check in a string that it is properly titlecased?

istitle() − Returns true if string is properly "titlecased" and false otherwise.

67.How will you check in a string that all characters are in uppercase?

isupper() − Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.

68.How will you merge elements in a sequence?

join(seq) − Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.

69.How will you get the length of the string?

len(string) − Returns the length of the string.

70.How will you get a space-padded string with the original string left-justified to a total of width columns?

ljust(width[, fillchar]) − Returns a space-padded string with the original string left-justified to a total of width columns.

71.How will you convert a string to all lowercase?

lower() − Converts all uppercase letters in string to lowercase.

72.How will you remove all leading whitespace in string?

lstrip() − Removes all leading whitespace in string.

73.How will you get the max alphabetical character from the string?

max(str) − Returns the max alphabetical character from the string str.

74.How will you get the min alphabetical character from the string?

min(str) − Returns the min alphabetical character from the string str.

75.How will you replaces all occurrences of old substring in string with new string?

replace(old, new [, max]) − Replaces all occurrences of old in string with new or at most max occurrences if max given.

76.How will you remove all leading and trailing whitespace in string?

strip([chars]) − Performs both lstrip() and rstrip() on string.

77.How will you change case for all letters in string?

swapcase() − Inverts case for all letters in string.

78.How will you get titlecased version of string?

title() − Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.

79.How will you convert a string to all uppercase?

upper() − Converts all lowercase letters in string to uppercase.

80.How will you check in a string that all characters are decimal?

isdecimal() − Returns true if a unicode string contains only decimal characters and false otherwise.

81.What is the difference between del() and remove() methods of list?

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

82.What is the output of len([1, 2, 3])?

3.

83.What is the output of [1, 2, 3] + [4, 5, 6]?

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

84.What is the output of ['Hi!'] * 4?

['Hi!', 'Hi!', 'Hi!', 'Hi!']

85.What is the output of 3 in [1, 2, 3]?

True

86.What is the output of for x in [1, 2, 3]: print x?

```
1
2
3
```

87.What is the output of L[2] if L = [1,2,3]?

3, Offsets start at zero.

88.What is the output of L[-2] if L = [1,2,3]?

1, Negative: count from the right.

89.What is the output of L[1:] if L = [1,2,3]?

2, 3, Slicing fetches sections.

90.How will you compare two lists?

cmp(list1, list2) − Compares elements of both lists.

91.How will you get the length of a list?

len(list) − Gives the total length of the list.

92.How will you get the max valued item of a list?

max(list) − Returns item from the list with max value.

93.How will you get the min valued item of a list?

min(list) − Returns item from the list with min value.

94.How will you get the index of an object in a list?

list.index(obj) − Returns the lowest index in list that obj appears.

95.How will you insert an object at given index in a list?

list.insert(index, obj) − Inserts object obj into list at offset index.

96.How will you remove last object from a list?

list.pop(obj=list[-1]) − Removes and returns last object or obj from list.

97.How will you remove an object from a list?

list.remove(obj) − Removes object obj from list.

98.How will you reverse a list?

list.reverse() − Reverses objects of list in place.

99.How will you sort a list?

list.sort([func]) − Sorts objects of list, use compare func if given.

100.What is lambda function in python?

'lambda' is a keyword in python which creates an anonymous function. Lambda does not contain block of statements. It does not contain return statements.

101.What we call a function which is incomplete version of a function?

Stub.

When a function is defined then the system stores parameters and local variables in an area of memory. What this memory is known as?

Stack.

102.A canvas can have a foreground color? (Yes/No)

Yes.

103.Is Python platform independent?

No

There are some modules and functions in python that can only run on certain platforms.

104.Do you think Python has a complier?

Yes

Yes it has a complier which works automatically so we don't notice the compiler of python.

105.What are the applications of Python?

Django (Web framework of Python).

2. Micro Frame work such as Flask and Bottle.

3. Plone and Django CMS for advanced content Management.

106.What is the basic difference between Python version 2 and Python version 3?

Table below explains the difference between Python version 2 and Python version 3.

S.No	Section	Python Version2	Python Version3
1.	Print Function	Print command can be used without parentheses.	Python 3 needs parentheses to print any string. It will raise error without parentheses.
2.	Unicode	ASCII str() types and separate Unicode() but there is no byte type code in Python 2.	Unicode (utf-8) and it has two byte classes − • Byte • Bytearray S.
3.	Exceptions	Python 2 accepts both	Python 3 raises a SyntaxError in

		new and old notations of syntax.	turn when we don't enclose the exception argument in parentheses.
4.	Comparing Unorderable	It does not raise any error.	It raises 'TypeError' as warning if we try to compare unorderable types.

107.Which programming Language is an implementation of Python programming language designed to run on Java Platform?

Jython

(Jython is successor of Jpython.)

108.Is there any double data type in Python?

No

109.Is String in Python are immutable? (Yes/No)

Yes.

110.Can True = False be possible in Python?

No.

111.Which module of python is used to apply the methods related to OS.?

OS.

112.When does a new block begin in python?

A block begins when the line is intended by 4 spaces.

113.Write a function in python which detects whether the given two strings are anagrams or not.

```
def check(a,b):
  if(len(a)!=len(b)):
    return False
  else:
    if(sorted(list(a)) == sorted(list(b))):
      return True
  else:
    return False
```

114.Name the python Library used for Machine learning.

Scikit-learn python Library used for Machine learning

115.What does pass operation do?

Pass indicates that nothing is to be done i.e. it signifies a no operation.

116.Name the tools which python uses to find bugs (if any).

Pylint and pychecker.

117.Write a function to give the sum of all the numbers in list?

Sample list − (100, 200, 300, 400, 0, 500)

Expected output − 1500

Program for sum of all the numbers in list is −

```
def sum(numbers):
  total = 0
  for num in numbers:
    total+=num
  print("Sum of the numbers: ", total)
sum((100, 200, 300, 400, 0, 500))
```

We define a function 'sum' with numbers as parameter. The in for loop we store the sum of all the values of list.

118.Write a program in Python to reverse a string without using inbuilt function reverse string?

Program to reverse a string in given below −

```
def string_reverse(str1):


rev_str = ' '
index = len(str1) #defining index as length of string.
while(index>0):
  rev_str = rev_str + str1[index-1]
  index = index-1
  return(rev_str)


print(string_reverse('1tniop'))
```

First we declare a variable to store the reverse string. Then using while loop and indexing of string (index is calculated by string length) we reverse the string. While loop starts when index is greater than zero. Index is reduced to value 1 each time. When index reaches zero we obtain the reverse of string.

119.Write a program to test whether the number is in the defined range or not?

Program is −

```
def test_range(num):
  if num in range(0, 101):
    print("%s is in range"%str(num))
  else:
    print("%s is not in range"%str(num))
```

Output −

test_range(101)

101 is not in the range

To test any number in a particular range we make use of the method ‘if..in’ and else condition.

120.Write a program to calculate number of upper case letters and number of lower case letters?

Test on String: "Tutorials POINT"

Program is −

```
def string_test(s):

a = { "Lower_Case":0 , "Upper_Case":0} #intiail count of lower and upper
for ch in s: #for loop
  if(ch.islower()): #if-elif-else condition
    a["Lower_Case"] = a["Lower_Case"] + 1
  elif(ch.isupper()):
    a["Upper_Case"] = a ["Upper_Case"] + 1
  else:
    pass

print("String in testing is: ",s) #printing the statements.
print("Number of Lower Case characters in String: ",a["Lower_Case"])
print("Number of Upper Case characters in String: ",a["Upper_Case"])
```

Output −

```
string_test("Tutorials POINT")
String in testing is: Tutorials POINT
Number of Lower Case characters in String: 8
Number of Upper Case characters in String: 6
```

We make use of the methods .islower() and .isupper(). We initialise the count for lower and upper. Using if and else condition we calculate total number of lower and upper case characters