MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution

DEPARTMENT OF COMPUTER APPLICATION

SUBJECT NAME: PROBLEM SOLVING USING PYTHON

SUBJECT CODE: SE21A

SEMESTER: I

PREPARED BY: PROF.GAYATHRY

UNIT - 1

1. Introduction

Computer Science

Solving problems by use of computation is known as computer science.

Python

Python supports both procedural and object-oriented Programming.

Essence of computational problem solving

To solve a problem computationally, two things are needed: i) a *representation* that captures all the relevant aspects of the problem, ii) an *algorithm* that solves the problem by use of the representation. Let's consider a problem known as **Man, Cabbage, Goat, Wolf problem** (Figure 1.1).

Figure –1.1 Man, Cabbage, Goat, Wolf Problem

A man lives on eastern side of a river wishes to bring a cabbage, a goat, and a wolf to a village on west side of the river to sell. His boat is only big enough to hold himself, and either the cabbage, goat, or wolf. Man cannot leave the goat alone with cabbage because goat will eat cabbage, and he cannot leave the wolf alone with goat because the wolf will eat the goat. To solve this game there is a simple algorithmic approach. Try all possible combinations of items that may be rowed back and forth across the river. Trying all possible solutions to a given problem is referredto as a *brute force approach*. A representation that leaves out details of what is being represented is a form of **abstraction**.

Start state of the problem can be represented as follows.

man	cabbage	goat	wolf
[W,	E,	W,	E]

in which the symbol W indicates that the corresponding object is on the west side of the river—in this case, the man and goat. Second state is

man	cabbage	goat	wolf

[W, W, E, E]

Third state is

man cabbage goat wolf

[W, W, E, W]

and the final state is

man cabbage goat wolf

[W, W, W, W]

A solution to this problem is a sequence of steps that converts the initial state,[E, E, E, E]inwhich all objects are on the east side of the river, to the *goal state* ,[W, W, W, W]in which all objects are on the west side of the river. Each step corresponds to the man rowing a particular object across the river . Python programming language provides an easy means of representing sequences of values. Main task is to develop or find an existing algorithm for computationally solving the problem.

Limits of Computational Problem Solving

Once an algorithm for solving a given problem is developed or found check "Cana solution to the problem be found in a reasonable amount of time?". If not, then the particular algorithm is of limited practical use.

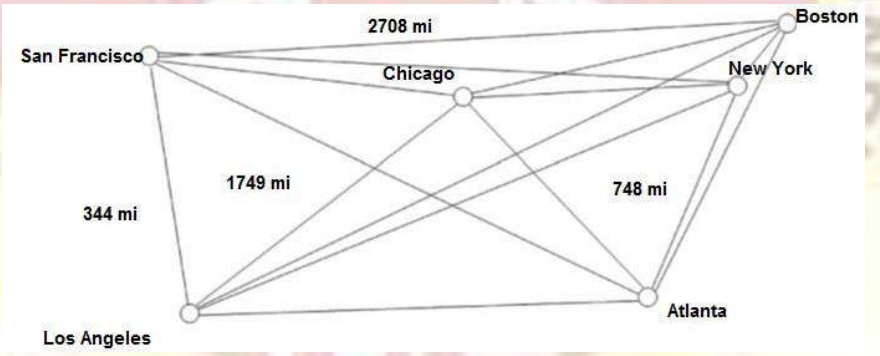


Figure-1.2 Traveling Salesman Problem.

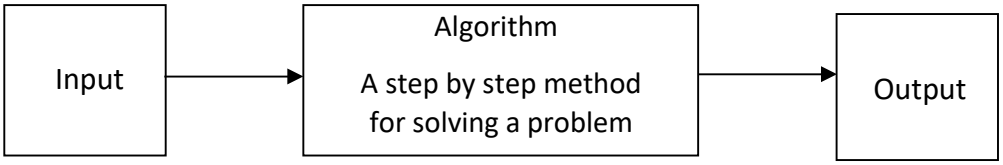
The **Traveling Salesman problem** (Figure 1.2) is a classic computational problem in computer science. The problem is to find the shortest route of travel for a salesman needing to visit a given set of cities. In a brute force approach, the lengths of all possible routes would be calculated and compared to find the shortest one.

Traveling Salesman problem in which a brute-force approach is impractical to use, more efficient problem-solving methods mustbe discovered that find either an exact or an approximate solution to the problem.

Computer Algorithms

What Is an Algorithm?

An algorithm is defined as a step-by-step procedure to solve a problem.



"An **algorithm** can also be defined as a finite number of clearly described, unambiguous "doable" steps. It can be systematically followed to produce a desired result for given input in a finite amount of time ".

Computer algorithms are central to computer science. They provide step-by-step methods of computation that a machine can carry out. Having high-speed machines (computers) that can consistently follow and execute a given set of instructions provides a reliable and effective means of realizing computation. However, *the computation that a given computer performs is only as good as the underlying algorithm used.* Understanding what can be effectively programmed and executed by computers, therefore, relies on the understanding of computer algorithms.

Algorithms and Computers: A Perfect Match

Most algorithms are not as simple or practical to apply manually. Most require the use of computers either because they would require too much time for a person to apply, or involve so much detail as to make human error likely. Because *computers can execute instructions very quickly and reliably without error*, algorithms and computers are a perfect match!

Method for Developing an Algorithm

(1) Define the problem: State the problem to be solved in clear and concise manner.

(2) List the inputs and outputs

(3) Describe the steps needed to convert input to output

(4) Test the algorithm: Choose input data and verify that the algorithm works.

Computer Hardware

Computer hardware comprises the physical part of a computer system. It includes the all-important components of the *central processing unit* (CPU) and *main memory*. It also includes *peripheral components* such as a keyboard, monitor, mouse, and printer.

Digital Computing: It's All about Switches

Computer hardware must be reliable and error free. If the hardware gives incorrect results, then any program run on that hardware is unreliable. The key to developing reliable systems is to keep the design as simple as possible. In digital computing, all information is represented as a series of digits. We are used to representing numbers using base 10 with digits 0–9. Consider if information were represented within a computer system this way, as shown in Figure 1.3

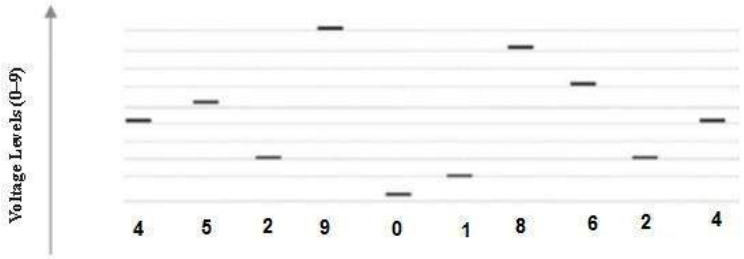


Figure –1.3 Decimal Digitalization.

In current electronic computing, each digit is represented by a different voltage level. The more voltage levels (digits) that the hardware must utilize and distinguish, the more complex the hardware design becomes. This results in greater chance of hardware design errors. It is a fact of information theory, however, that any information can be represented using only *two* symbols. Because of this, *all information within a computer system is represented by the use of only two digits, 0 and 1,* called **binary representation as shown in figure 1.4.**

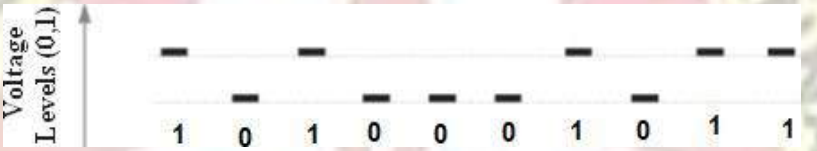


Figure –1.4 Binary Digitalization

In this representation, each digit can be one of only two possible values, similar to a light switch that can be either on or off. Computer hardware, therefore, is based on the use of simple electronic "on/off" switches called **transistors** that switch at very high speed. **Integrated circuits** ("chips"),the building blocks of computer hardware, are comprised of millions or even billions of transistors.

Binary Representation

All information within a computer system is represented using only two digits, 0 and 1, called **binary representation.**

Binary Number System

For representing numbers, any base (radix) can be used. For example, in base 10, there are ten possible digits (0, 1, . . ., 9), in which each column value is a power of ten , as shown in Figure 1.5.

10,000,000	1,000,000	100,000	10,000	1,000	100	10	1
10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0
						9	9 = 99

Figure –1.5 Base 10 Representation.

Other radix systems work in a similar manner. **Base 2** has digits 0 and 1, with place values that are powers of two, as depicted in Figure 1.6.

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	**1**	**1**	**0**	**0**	**0**	**1**	**1**

0 + 64 + 32 + 0 + 0 + 0 + 2 + 1 = **99**

Figure – 1.6 Base 2 Representation

As shown in this figure, converting from base 2 to base 10 is simply a matter of adding up the column values that have a 1.

"The term bit stands for binary digit. Therefore, every bit has the value 0 or 1. A byte is a group of bits operated on as a single unit in a computer system, usually consisting of eight bits".

Although values represented in base 2 are significantly longer than those represented in base 10, binary representation is used in digital computing because of the resulting simplicity of hardware design.

The algorithm for the conversion from base 10 to base 2 is to successively divide a number by two until the remainder becomes 0. The remainder of each division provides the next higher-order(binary) digit, as shown in Figure 1.7

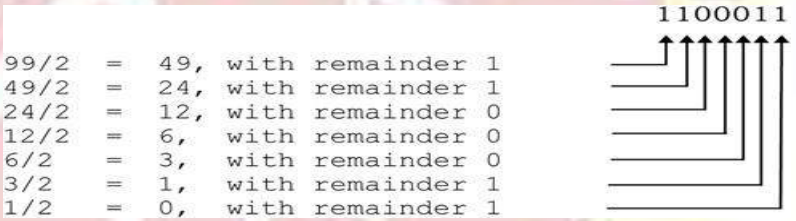


Figure –1.7 Converting from Base10 to Base2

Thus, we get the binary representation of 99 to be 1100011. This is the same as in Figure above, except that we had an extra leading insignificant digit of 0, since we used an eight-bit representation there.

Fundamental Hardware Components

"The **central processing unit (CPU)** is the "brain" of a computer system, containing digital logic circuitry able to interpret and execute instructions". **Main memory** is where currently executing programs reside, which the CPU can directly and very quickly access. Main memory is volatile; that is, the contents are lost when the power is turned off. In contrast, **secondary memory** is nonvolatile, and therefore provides long-term storage of programs and data. This kindof storage, for example, can be magnetic (hard drive), optical (CD or DVD), or nonvolatile flash memory (such as in a USB drive).

Input/output devices include anything that allows for input (such as the mouse and keyboard) or output (such as a monitor or printer). Finally, **buses** transfer data between components within a

computer system, such as between the CPU and main memory. The relationship of these devices is depicted in Figure 1.8.

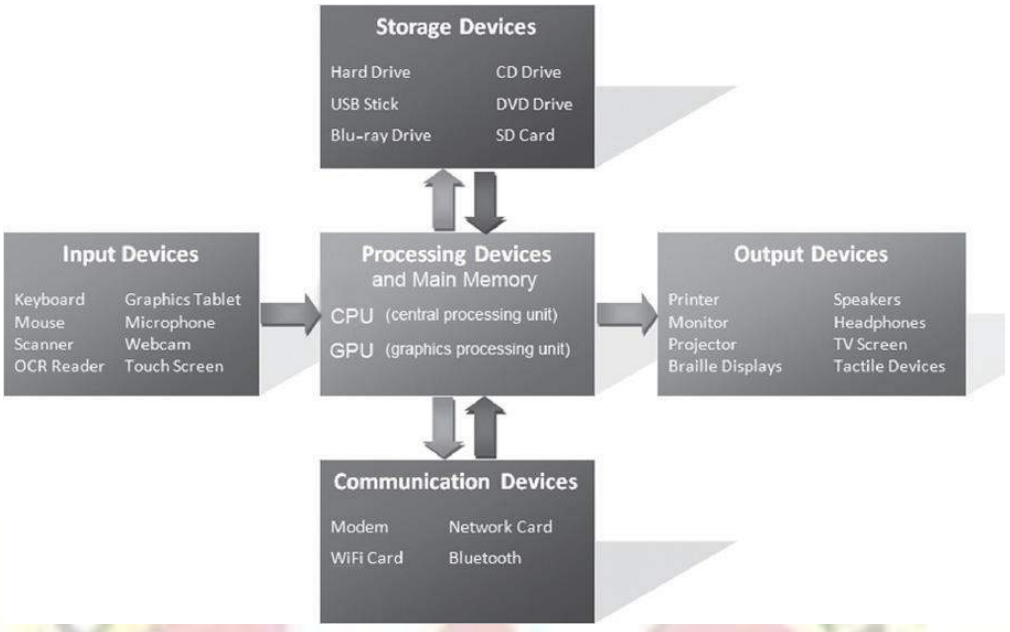


Figure – 1.8 Fundamental Hardware Components

An operating system acts as the "middle man" between the hardware and executing application programs. An **operating system** is software that has the job of managing the hardware resources of a given computer and providing a particular user interface. For example, it controls the allocation of memory for the various programs that may be executing on a computer. Operating systems also provide a particular user interface. Thus, it is the operating system installed on a given computer that determines the "look and feel" of the user interface and how the user interacts with the system, and not the particular model computer as shown in figure 1.9.

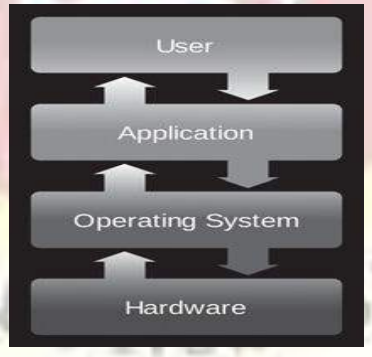


Figure - 1.9 Operating System

Computer Software

The first computer programs ever written were fora mechanical computer designed by Charles Babbage in the mid-1800s. (Babbage's Analytical Engine). "Ada Lovelace "is referred to as "the first computer programmer." who was a talented mathematician.

What Is Computer Software?

"Computer software is a set of program instructions, including related data and documentation, that can be executed by computer". This can be in the form of instructions on paper, or in digital form. While system software is intrinsic to a computer system, **application software** fulfills

users' needs, such as a photo-editing program.

Syntax, Semantics, and Program Translation

What Are Syntax and Semantics?

"**Syntax** of a language is a set of characters and the acceptable arrangements (sequences)of those characters". English, for example, includes the letters of the alphabet, punctuation, and properly spelled words and properly punctuated sentences. The following is a syntactically correct sentence in English,

"Hello there, how are you?"

The following, however, is not syntactically correct,

"Hello there, hao are you?"

In this sentence, the sequence of letters "hao" is not a word in the English language. Now consider the following sentence,

"Colorless green ideas sleep furiously."

This sentence is syntactically correct, but is *semantically* incorrect, and thus has no meaning.

"S**emantics** of a language is the meaning associated with each syntactically correct sequence of characters". In Mandarin, "Hao" is syntactically correct meaning "good." ("Hao" is from a system called pinyin, which uses the Roman alphabet rather than Chinese characters for writing Mandarin.).Thus, every language has its own syntax and semantics, as shown in Figure 1.10.

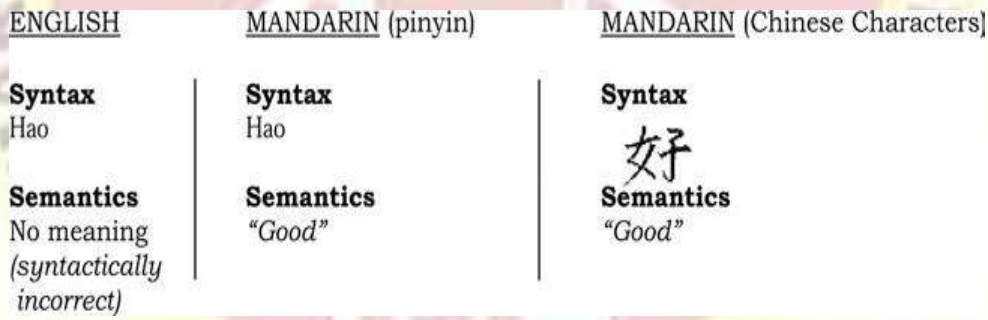


Figure –1.10 Syntax and Semantics of Languages

1.5.3 Program Translation

A central processing unit (CPU) is designed to interpret and execute a specific set of instructions represented in binary form (i.e., 1s and 0s) called **machine code**. Only programs in machine code can be executed by a CPU, depicted in Figure 1.11

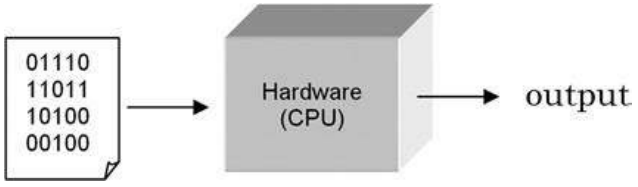


Figure – 1.11 Execution of Machine Code

Writing programs at this "low level" is tedious and error-prone. Therefore, most programs are written in a "high-level" programming language such as Python. Since the instructions of such programs are not in machine code that a CPU can execute, a translator program must be used.

There are two fundamental types of translators. One, called a **compiler** , translates programs directly into machine code to be executed by the CPU, denoted in Figure 1.12.

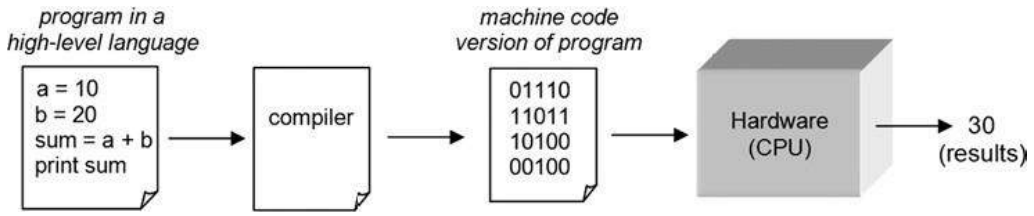


Figure – 1.12 Program Execution by Use of a Compiler

An **interpreter** executes program instructions in place of ("running on top of") the CPUs shown in figure 1.13.

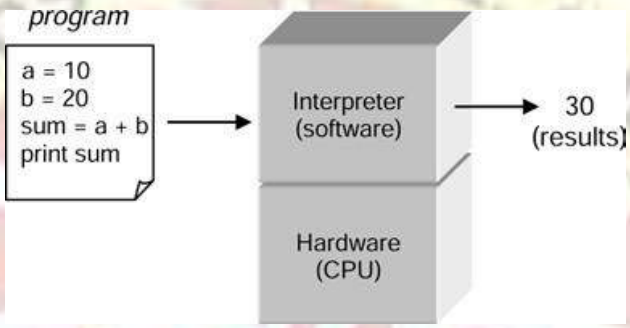


Figure - 1.13 Program Execution by Use of an Interpreter

Thus, an interpreter can immediately execute instructions as they are entered. This is referred to as **interactive mode**. This is a very useful feature for program development. Python is executedby an interpreter. On the other hand, compiled programs generally execute faster than interpreted programs. "A **compiler** is a translator program that translates programs directly into machine code to be executed by the CPU". Any program can be executed by either a compiler or an interpreter, as long there exists the corresponding translator program for the programming language that it is written in.

1.5.4 Program Debugging: Syntax Errors vs. Semantic Errors

Program debugging is the process of finding and correcting errors (**"bugs"**) in a computer program. Programming errors are inevitable during program development. **Syntax errors** are caused by invalid syntax (for example, entering prnt instead of print). Since a translator cannot understand instructions containing syntax errors, translators terminate when encountering such errors indicating where in the program the problem occurred.

In contrast, **semantic errors** (generally called **logic errors)** are errors in program logic. Such errors cannot be automatically detected, since translators cannot understand the intent of a given computation. For eg, if a program computed the average of three numbers as follows,

(num1 + num2 + num3) / 2.0

a translator would have no means of determining that the divisor should be 3 and not 2. *Computers do not understand what a program is meant to do, they only follow the instructions given*. It is up to the programmer to detect such errors. Program debugging is not a trivial task,and constitutes much of the time of program development.

Syntax errors are caused by invalid syntax. **Semantic (logic) errors** are caused by errors in program logic.

PROCESS OF COMPUTATIONAL PROBLEM SOLVING

Computational problem solving does not simply involve the act of computer programming. It is a *process*, with programming being only one of the steps. Before a program is written, a design for the program must be developed. And before a design can be developed, the problem to be solved must be well understood. Once written, the program must be thoroughly tested. These steps are outlined in Figure 1.14

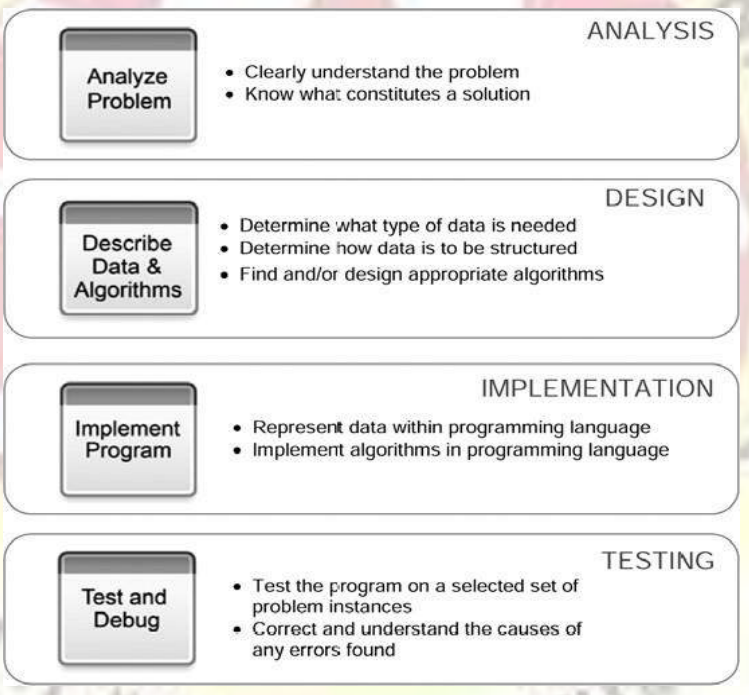


Figure – 1.14 Process of Computational Problem Solving

Problem Analysis

Understanding the Problem

Once a problem is clearly understood, the fundamental computational issues for solving it can be determined. For the Man, Cabbage, Goat, Wolf (MCGW) problem, a brute-force algorithmic approach of trying all possible solutions works very well, since there are a small number of actions that can be taken at each step, and only a relatively small number of steps for reaching a solution. But for a Traveling Salesman problem and the game of chess, the brute- force approach is infeasible. Thus, the computational issue for these problems is to find other, more efficient algorithmic approaches for their solution

Knowing What Constitutes a Solution

Besides clearly understanding a computational problem, one must know what constitutes a solution. For some problems, there is only one solution. For others, there may be a number of solutions. Thus, a program may be stated as finding,

♦ *A* solution

♦ An *approximate* solution

♦ A *best* solution

♦ *All* solutions

Program Design

Describing the Data Needed

For the Man, Cabbage, Goat, Wolf problem, a list can be used to represent the correct location (east and west) of the man, cabbage, goat, and wolf as discussed earlier, reproduced below,

man cabbage goat wolf
[W, E, W, E]

For the Calendar Month problem, the data include the month and year (entered by the user), the number of days in each month, and the names of the days of the week. A useful structuring of the data is given below,

[*month* , *year*]

[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

[‘Sunday’, ‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’, ‘Friday’, ‘Saturday’]

The month and year are grouped in a single list since they are naturally associated. Similarly, the names of the days of the week and the number of days in each month are grouped.

Program Implementation

Design decisions provide *general* details of the data representation and the algorithmic approaches for solving a problem.

Program Testing

Software testing is a crucial part of software development. Testing is done incrementally as a program is being developed, when the programis complete, and when the program needs to be updated. Following are general truisms of software development.

1. Programming errors are pervasive, persistent and inevitable.
2. Software testing is an essential part of software development.
3. Any changes made in correcting a programming error should be fully understood as to why the changes correct the detected error.

Truism 1 reflects the fact that programming errors are inevitable and that we must accept it. As a result of truism 1, truism 2 states the essential role of software testing. Given the inevitability of programming errors, it is important to test a piece of software in a thorough and systematic manner. Finally, truism 3 states the importance of understanding *why* a given change in a

program fixes a specific error.

Python Programming Language

About Python

Guido van Rossum is the creator of the Python programming language, first released in the early 1990s.Python has a simple syntax. Python programs are clear and easy to read. At the same time, Python provides powerful programming features, and is widely used. Companies and organizations that use Python include YouTube, Google, Yahoo, and NASA. Python is well supported and freely available at www.python.org

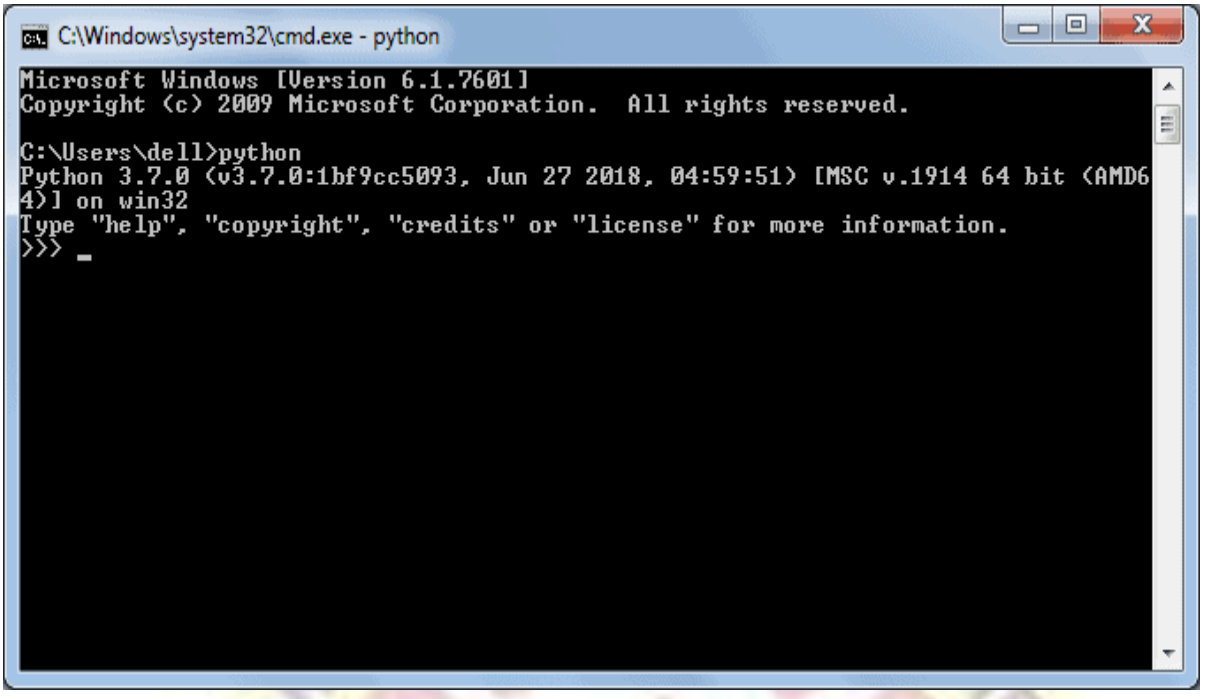
IDLE Python Development Environment

IDLE is an integrated development environment (IDE). An IDE is a bundled set of software tools for program development. This typically includes an editor for creating and modifying programs, a translator for executing programs, and a program debugger . A debugger provides a means of taking control of the execution of a program to aid in finding program errors.

Python is most commonly translated by use of an interpreter. Thus, Python provides the very useful ability to execute in interactive mode. The window that provides this interaction is referred to as the Python shell . Python Shell waits for the input command from the user. As soon as the user enters the command, it executes it and displays the result.

To open the Python Shell on Windows, open the command prompt, write python and press **enter**.

Example use of the Python shell is demonstrated in Figure



1.7.3 The Python Standard Library

The Python Standard Library is a collection of built-in modules , each providing specific functionality. For example, the math module provides additional mathematical functions.

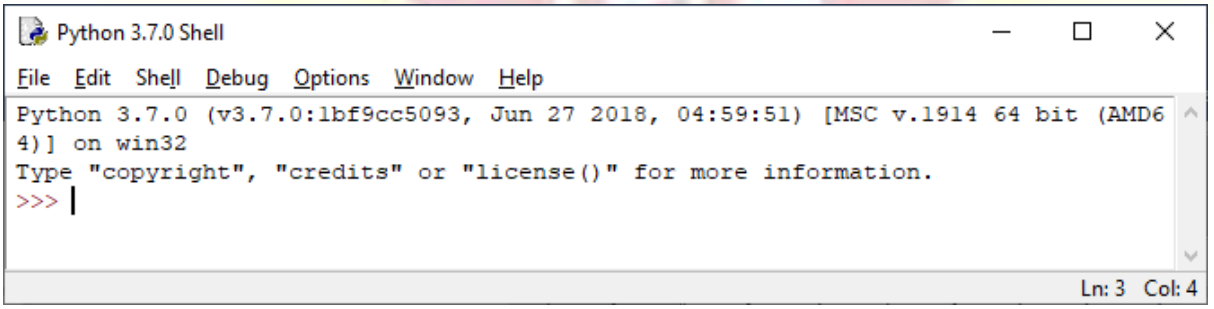
Learning How to Use IDLE

In order to become familiar with writing your own Python programs using IDLE, we will create a simple program that asks the user for their course of study and responds with course name. This program utilizes the following concepts:

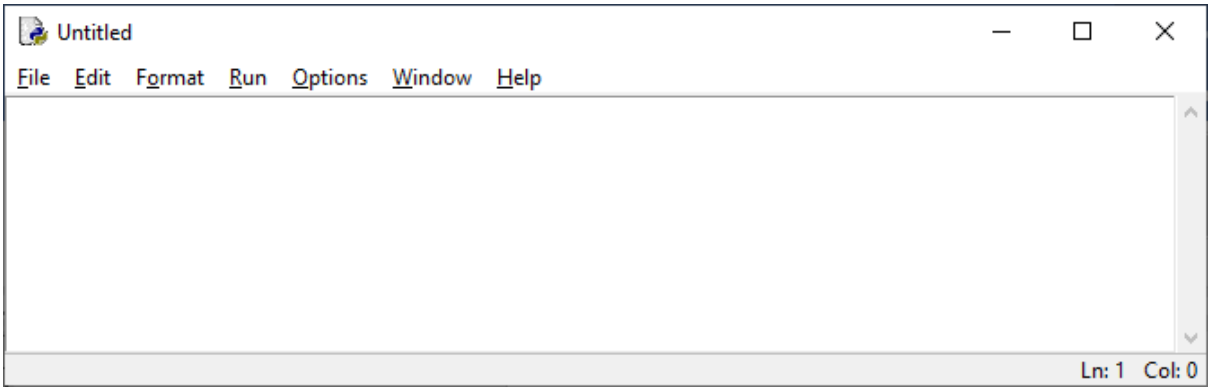
♦ creating and executing Python programs

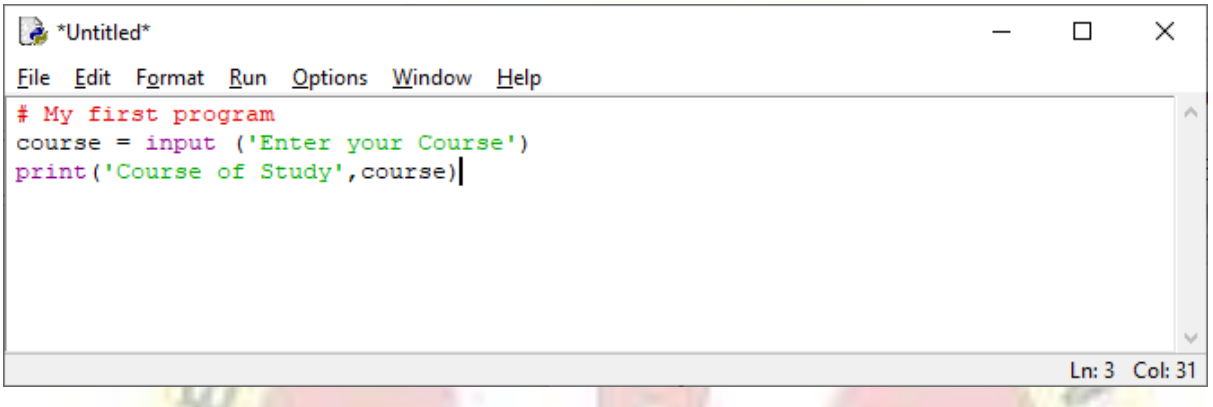
♦ input and print

First, to create a Python program file, select New Window from the File menu in the Python shell as shown in Figure



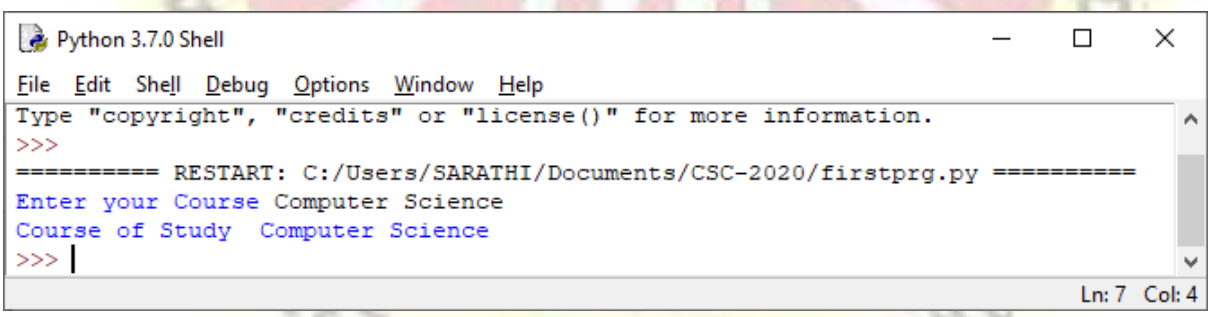
A new, untitled program window appear.





When finished, save the program file by selecting Save As under the File menu, and save in the appropriate folder with the name FirstPrg.py. To run the program, select Run Module from the Run menu (or simply hit function key F5).

Sample Output:



1.8 Literals

A literal is a sequence of one or more characters that stands for itself.

Numeric Literals

A numeric literal is a literal containing only the digits 0–9, an optional sign character (+ or -), and a possible decimal point. If a numeric literal contains a decimal point, then it denotes a floating-point value , or " float " (e.g., 10.24); otherwise, it denotes an integer value (e.g., 10). Commas are never used in numeric literals

Limits of Range in Floating-Point Representation

There is no limit to the size of an integer that can be represented in Python. Floating-point values, however, have both a limited range and a limited precision . Python uses a double-precision standard format (IEEE 754) providing a range of 10^{-308} to 10^{308} with 16 to 17 digits of precision.

It is important to understand the limitations of floating-point representation. For example, the multiplication of two values may result in arithmetic overflow , a condition that occurs when a calculated result is too large in magnitude (size) to be represented, ...

```
>>> 1.5e200 * 2.0e210
>>> inf
```

This results in the special value inf (“infinity”) rather than the arithmetically correct result 3.0e410, indicating that arithmetic overflow has occurred. Similarly, the division of two numbers may result in arithmetic underflow , a condition that occurs when a calculated result is too small in magnitude to be represented,

```
>>> 1.0e2300 / 1.0e100
0.0
```

Arithmetic overflow occurs when a calculated result is too large in magnitude to be represented.
Arithmetic underflow occurs when a calculated result is too small in magnitude to be represented.

1.8.1Built-in format Function

The built-in format function can be used to produce a numeric string of a given floating-point value rounded to a specific number of decimal places.

Because floating-point values may contain an arbitrary number of decimal places, the built-in format function can be used to produce a numeric string version of the value containing a specific number of decimal places,

```
>>>12/5
2.4
>>>5/7
0.7142857142857143
>>>format(12/5, '.2f')
'2.40'
>>> format(5/7, '.2f')
'0.71'
```

In these examples, format specifier ‘.2f' rounds the result to two decimal places of accuracy in the string produced.

, a comma in the format specifier adds comma separators to the result,

```
>>> format(13402.25, ' , .2f')
13,402.24
```

1.8.2 String Literals

A string literal, or string, is a sequence of characters denoted by a pair of matching single or double quotes in Python. String literals, or " strings ," represent a sequence of characters,

'Hello'

'Smith, John'

"csc,mgc 2020"

In Python, string literals may be delimited (surrounded) by a matching pair of either single (') or double (") quotes. Strings must be contained all on one line

```
>>> print('Welcome to Python!')
Welcome to Python!
```

a string may contain zero or more characters, including letters, digits, special characters, and blanks. A string consisting of only a pair of matching quotes (with nothing in between) is called the empty string ,

The Representation of Character Values

There needs to be a way to encode (represent) characters within a computer. Although various encoding schemes have been developed, the Unicode encoding scheme is intended to be a universal encoding scheme. Unicode is actually a collection of different encoding schemes utilizing between 8 and 32 bits for each character. The default encoding in Python uses UTF-8 , an 8-bit encoding compatible with ASCII, an older, still widely used encoding scheme.

UTF-8 encodes characters that have an ordering with sequential numerical values. For example, 'A' is encoded as 01000001 (65), 'B' is encoded as 01000010 (66), and so on. This is true for character digits as well, '0' is encoded as 00110000 (48) and '1' is encoded as 00110001 (49).

Python has means for converting between a character and its encoding. The ord function gives the UTF-8 (ASCII) encoding of a given character. For example, ord('A') is 65. The chr function gives the character for a given encoding value, thus chr(65) is 'A'.

Control Characters

Control characters are special characters that are not displayed on the screen. Control characters do not have a corresponding keyboard character. Therefore, they are represented by a combination of characters called an escape sequence .

An escape sequence begins with an escape character that causes the sequence of characters following it to "escape" their normal meaning. The backslash (\) serves as the escape character in Python. For example, the escape sequence '\n', represents the newline control character , used to begin a new screen line. An example of its use is given below,

print('Hello\nJennifer Smith') which is displayed as follows,

Hello

Jennifer Smith

1.8.3 String Formatting

The format function can be used to control how strings are displayed. The format function has the form,

format(value, format_specifier)

where value is the value to be displayed, and format_specifier can contain a combination of formatting options. For example, to produce the string 'Hello' left-justified in a field width of 20 characters would be done as follows,

format('Hello', ' < 20') → 'Hello ‘

To right-justify the string, the following would be used,

format('Hello', ' > 20') → ' Hello'

Formatted strings are left-justified by default

Finally blanks, by default, are the fill character for formatted strings. However, a specific fill character can be specified as shown below,

>>> print('Hello World', format('.', '. <30’), ‘have a nice day’)

Hello World ..... have a nice day

1.8.4 Implicit and Explicit Line Joining

Sometimes a program line may be too long to fit in the Python-recommended maximum length of 79 characters. There are two ways in Python to do deal with such situations—implicit and explicit line joining.

Implicit Line Joining

There are certain delimiting characters that allow a logical program line to span more than one physical line. This includes matching parentheses, square brackets, curly braces, and triple quotes. For example, the following two program lines are treated as one logical line,

print('Name:', student_name, 'Address:', student_address, 'Number of Credits:', total_credits, 'GPA:', current_gpa)

Matching quotes must be on the same physical line. For example, the following will generate an error,

print('This program will calculate a restaurant tab for a couple with a gift certifi cate, and a restaurant tax of 3%')

Matching parentheses, square brackets, and curly braces can be used to span a logical program line on more than one physical line.

Explicit Line Joining

In addition to implicit line joining, program lines may be explicitly joined by use of the backslash (\) character.

1.9 What is a Variable?

A variable is a name (identifier) that is associated with a value, as for variable num depicted in

Num ->10

A variable can be assigned different values during a program's execution—hence, the name "variable." Wherever a variable appears in a program (except on the left-hand side of an assignment statement), it is the value associated with the variable that is used , and not the variable's name,

num + 1 → 10 +1 → 11

Variables are assigned values by use of the assignment operator , =

num =10

num = num + 1

in Python the same variable can be associated with values of different type during program execution, as indicated below.

var =12 integer

var =12.45 float

var = 'Hello' string

A variable is a name that is associated with a value. The assignment operator, =, is used to assign values to variables. An immutable value is a value that cannot be changed.

Variable Assignment and Keyboard Input

The value can come from the user by use of the input function

>>>name = input('What is your first name?')

What is your first name? John

the variable name is assigned the string 'John'

All input is returned by the input function as a string type. For the input of numeric values, the response must be converted to the appropriate type. Python provides built-in type conversion functions int() and float()

a = int(input('Enter any value'))

gpa=float(input('What is your grade point average?'))

All input is returned by the input function as a string type. Built-in functions int() and float() can be used to convert a string to a numeric type.

1.10 What is an Identifier?

An identifier is a sequence of one or more characters used to provide a name for a given program element. Python is case sensitive , thus, Line is different from line. Identifiers may contain letters and digits, but cannot begin with a digit. The underscore character, _, is also allowed to aid in the readability of long identifi er names. It should not be used as the first character, however, as identifiers beginning with an underscore have special meaning in Python.

Spaces are not allowed as part of an identifier. Thus, any identifier containing a space character would be considered two separate identifiers

In Python, an identifier may contain letters and digits, but cannot begin with a digit. The special underscore character can also be used.

Keywords and Other Predefined Identifiers in Python

A keyword is an identifier that has predefined meaning in a programming language. Therefore, keywords cannot be used as "regular" identifiers

```
>>> and = 10
SyntaxError: invalid syntax
```

1.11 What is an Operator?

An operator is a symbol that represents an operation that may be performed on one or more operands . For example, the + symbol represents the operation of addition. An operand is a value that a given operator is applied to, such as operands 2 and 3 in the expression 2 + 3. A unary operator operates on only one operand, such as the negation operator . A binary operator operates on two operands, as with the addition operator.

Operators that take one operand are called unary operators. Operators that take two operands are called binary operators

1.11.1 Arithmetic Operators

Python provides the arithmetic operators given in below table . The + , - , * (multiplication) and / (division) arithmetic operators perform the usual operations. Note that the - symbol is used both as a unary operator (for negation) and a binary operator (for subtraction).

Arithmetic Operators in Python

Arithmetic Operators	Example	Results
-x Negation	-10	-10
x+ y Addition	10 + 20	30
x – y Subtraction	10 – 20	-10
X * y Multiplication	10 * 20	200
X / y Division	25/10	2.5
X // y truncating div	25 // 10	2
X % y modulus	25 % 10	5
X ** y exponentiation	10 ** 2	100

Python also includes an exponentiation (**) operator. Integer and floating-point values can be used in both the base and the exponent, 2**4 → 16

Python provides two forms of division. "true" division is denoted by a single slash, /. Thus, 25 / 10 evaluates to 2.5. Truncating division is denoted by a double slash, //, providing a truncated result based on

the type of operands applied to. When both operands are integer values, the result is a truncated integer referred to as integer division. When at least one of the operands is a fl oat type, the result is a truncated floating point. Thus, 25 // 10 evaluates to 2, while 25.0 // 10 becomes 2.0

The division operator, /, produces “true division” regardless of its operand types. The truncating division operator, //, produces either an integer or float truncated result based on the type of operands applied to. The modulus operator (%) gives the remainder of the division of its operands.

```
>>> a = 4
>>> b = 3
>>> +a
4
>>> -b
-3
>>> a + b
7
>>> a - b
1
>>> a * b
12
>>> a / b
1.3333333333333333
>>> a % b
1
>>> a ** b
64
```

1.11.2 What Is an Expression?

An expression is a combination of symbols that evaluates to a value. Expressions, most commonly, consist of a combination of operators and operands,

4 + (3 * k)

An expression can also consist of a single literal or variable.

Expressions that evaluate to a numeric type are called arithmetic expressions . A subexpression is any expression that is part of a larger expression. Subexpressions may be denoted by the use of parentheses, as shown above. Thus, for the expression 4 + (3 * 2), the two operands of the addition operator are 4 and (3 * 2), and thus the result it equal to 10. If the expression were instead written as (4 + 3) * 2, then it would evaluate to 14.

1.11.3 Operator Precedence

When operators appear between their operands, is referred to as infix notation . For example, the

expression 4 + 3 is in infix notation since the + operator appears between its two operands, 4 and 3. There are other ways of representing expressions called prefix and postfix notation, in which operators are placed before and after their operands, respectively.

The expression 4 + (3 * 5) is also in infix notation. It contains two operators, + and *. The parentheses denote that (3 * 5) is a subexpression. Therefore, 4 and (3 * 5) are the operands of the addition operator, and thus the overall expression evaluates to 19.

Operator Precedence of Arithmetic Operators in Python

Operator	Associativity
** (Exponentiation)	Right-to-left
- (negation)	Left-to-right
*, / //, %	Left-to-right
+ , -	Left-to-right

Operator precedence is the relative order that operators are applied in the evaluation of expressions, defined by a given operator precedence table.

Operator Associativity

For operators following the associative law, the order of evaluation doesn't matter,

(2 + 3) + 4 → 9

2 + (3 + 4) → 9

In this case, we get the same results regardless of the order that the operators are applied. Division and subtraction, however, do not follow the associative law,

(a) (8 - 4) - 2 → 4 -2 → 2 8 - (4 - 2) → 8 - 2 → 6

(b) (8 / 4) / 2 → 2 / 2 → 1 8 / (4 / 2) → 8 / 2 → 4

Operator associativity is the order that operators are applied when having the same level of precedence, specific to each operator.

1.12 What Is a Data Type?

A data type is a set of values, and a set of operators that may be applied to those values. For example, the integer data type consists of the set of integers, and operators for addition, subtraction, multiplication, and division, among others. Integers, floats, and strings are part of a set of predefined data types in Python called the built-in types .

Finally, there are two approaches to data typing in programming languages. In static typing , a variable is declared as a certain type before it is used, and can only be assigned values of that type. Python, however, uses dynamic typing . In dynamic typing , the data type of a variable depends only on the type of value that the variable is currently holding. Thus, the same variable may be assigned values of different type during the execution of a program.

1.12.1 Mixed-Type Expressions

A mixed-type expression is an expression containing operands of different type. The CPU can only perform operations on values with the same internal representation scheme, and thus only on operands of the same type. Operands of mixed-type expressions therefore must be converted to a common type. Values can be converted in one of two ways—by implicit (automatic) conversion, called coercion , or by explicit type conversion

1.12.2 Coercion vs. Type Conversion

Coercion is the implicit (automatic) conversion of operands to a common type. Coercion is automatically performed on mixed-type expressions only if the operands can be safely converted, that is, if no loss of information will result. The conversion of integer 2 to floating-point 2.0 below is a safe conversion—the conversion of 4.5 to integer 4 is not, since the decimal digit would be lost

3 + 4.5 → 2.0 + 4.5 → 6.5 safe (automatic conversion of int to float)

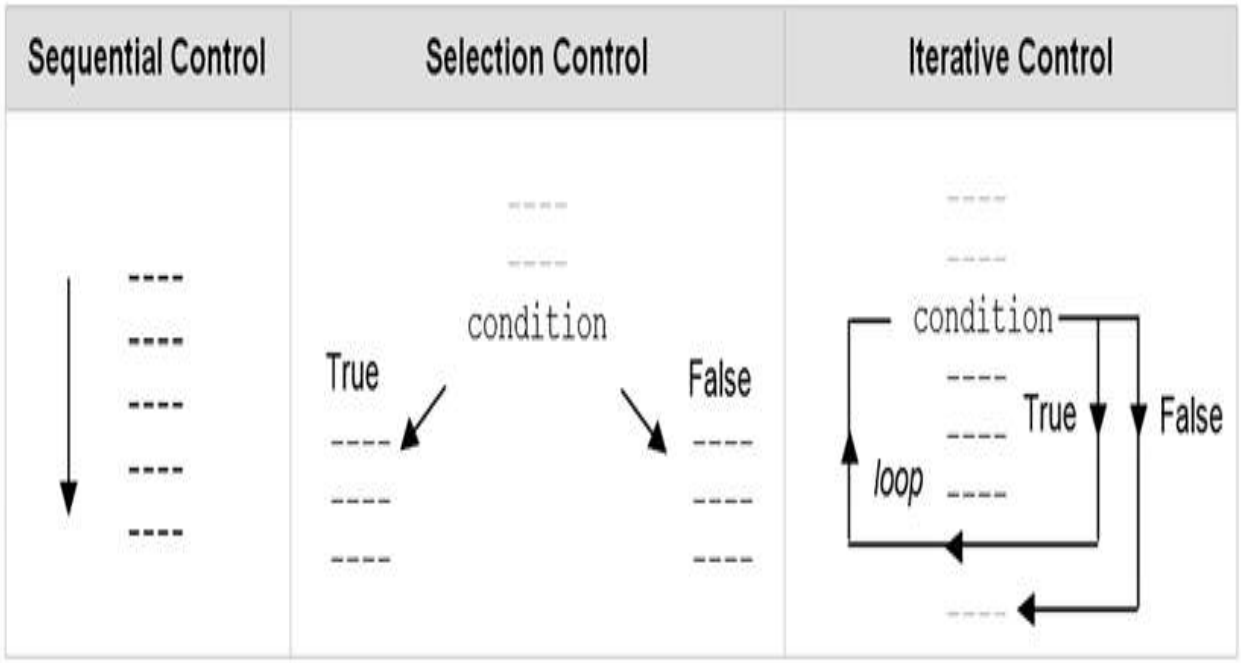
Type conversion is the explicit conversion of operands to a specific type. Type conversion can be applied even if loss of information results. Python provides built-in type conversion functions int() and float(), with the int() function truncating results as given

float(2) + 4.5 → 2.0 + 4.5 → 6.5

UNIT – II

2.1 CONTROL STRUCTURES

Control flow is the order that instructions are executed in a program. A control statement is a statement that determines the control flow of a set of instructions. There are three fundamental forms of control that programming languages provide— sequential control , selection control , and iterative control. Sequential control is an implicit form of control in which instructions are executed in the order that they are written. A program consisting of only sequential control is referred to as a "straight-line program." Selection control is provided by a control statement that selectively executes instructions, while iterative control is provided by an iterative control statement that repeatedly executes instructions. Each is based on a given condition. Collectively a set of instructions and the control statements controlling their execution is called a control structure.



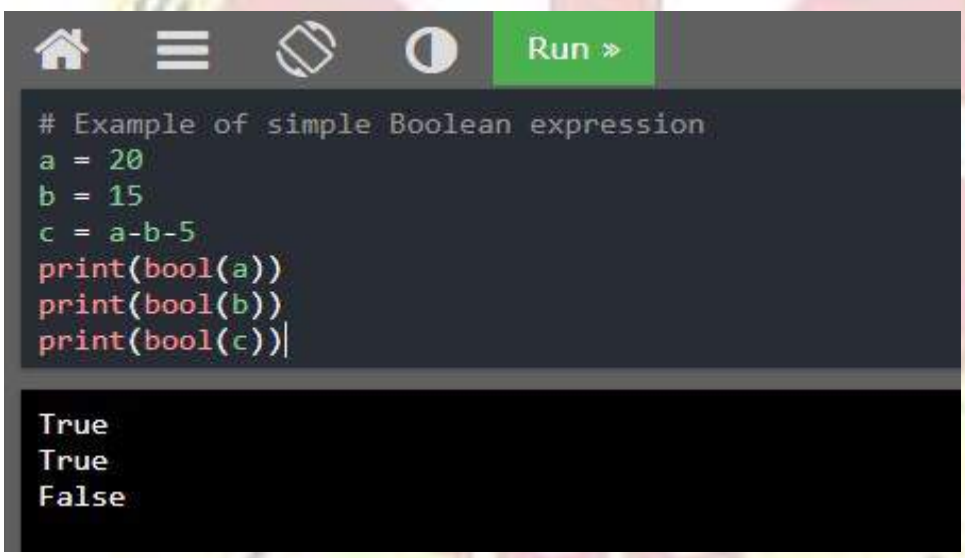
2.2 Boolean Expressions

A Boolean type represents special values ‘True’ and ‘False’. They are represented as 1 and 0, and can be used in numeric expressions as value. Evaluation of any expression in Python, it returns the true or false value. The bool() function is used to evaluate any value and returns true of false.

Example

```
# Example of simple Boolean expression
a = 20
b = 15
c = a-b-5
print(bool(a))
 print(bool(b))
 print(bool(c))
```

output:



2.2.1 Relational Operators

The relational operators in Python perform the usual comparison operations. Relational expressions are a type of Boolean expression, since they evaluate to a Boolean result. These operators not only apply to numeric values, but to any set of values that has an ordering, such as strings.

The Relational operators are = =, ! =, >, <, >=, <=.

Operators	Name	Example	Result
= =	Equal	100 = = 100	True
! =	Not Equal to	100 !=101	False
<	Less than	10 < 20	True

>	Greater than	20 > 10	True
<=	Less than or equal to	10 <=20	True
>=	Greater than or equal to	10>=20	False

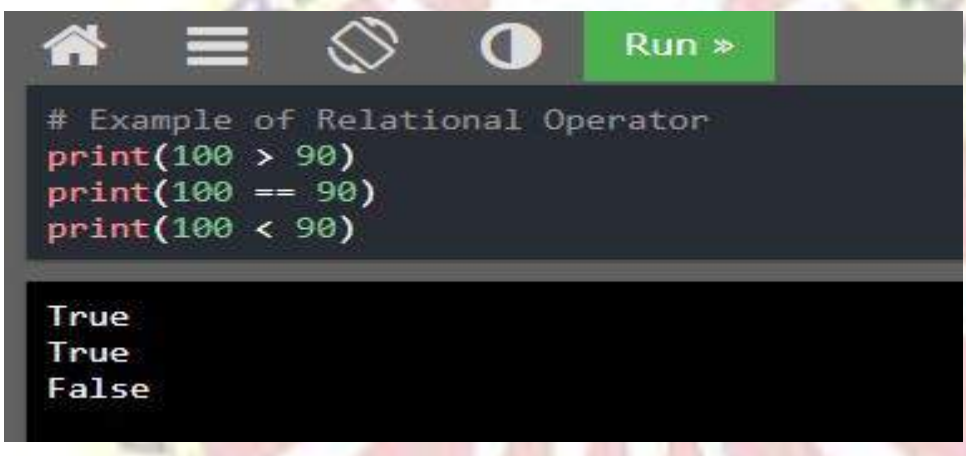
Fig. 3.2

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

Example:

```
# Example of Relational Operator
print(100 > 90)
print(100 == 90)
print(100 < 90)
```

Output:



2.2.2 Membership Operators

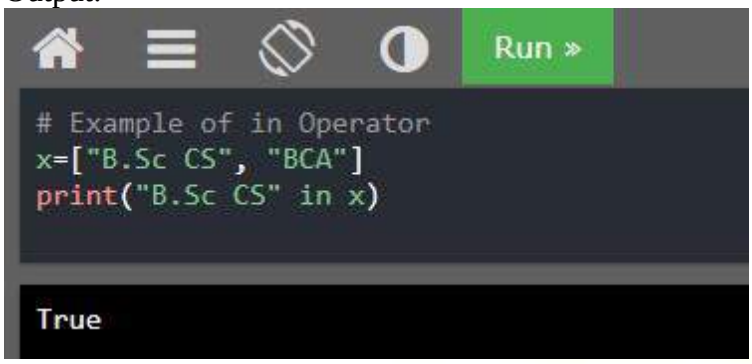
Python provides a convenient pair of membership operators. These operators can be used to easily determine if a particular value occurs within a specified list of values. The in operator is used to determine if a specific value is in a given list, returning True if found, and False otherwise. The not in operator returns the opposite result.

Operators	Description	Example
In	Returns true if a sequence with the specified values is present in the object.	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Example #1

```
# Example of in Operator
x=["B.Sc CS", "BCA"]
print("B.Sc CS" in x)
```

Output:



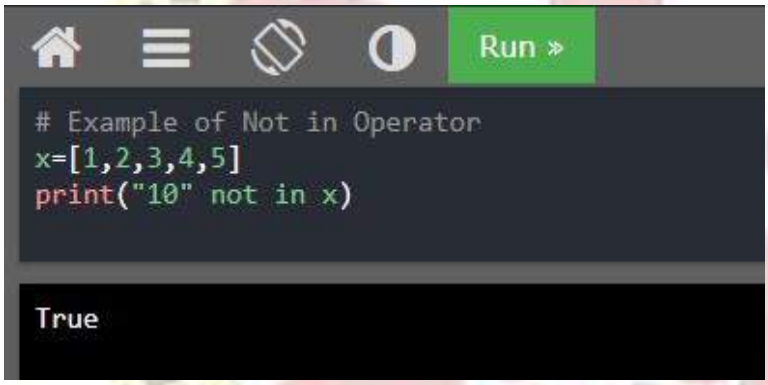
Example #2

Example of Not in Operator

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

print("10" not in x)

Output:



2.2.3 Boolean Operators

Boolean algebra contains a set of **Boolean** (logical) **operators**, denoted by and, or, and not in Python. These logical operators can be used to construct arbitrarily complex Boolean expressions.

X	y	x and y	x or y	Not x	Not y
True	True	True	True	False	False
True	False	False	True	False	True
False	True	False	True	True	False
False	False	False	False	True	True

Fig. Boolean Logic Truth Table

2.2.4 Operator Precedence and Boolean Expressions

Boolean expressions can contain arithmetic as well as relational and Boolean operators, the precedence of all operators needs to be collectively applied.

Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right
<, >, <=, >=, !=, == (relational operators)	left-to-right
not	left-to-right
and	left-to-right
or	left-to-right

Fig. **Operator Precedence of Arithmetic, Relational, and Boolean Operators**

All arithmetic operators are performed before any relational or Boolean operator. For Example,

25 + 25 < 25 + 50

50 < 75 ⟶ True

2.2.5 Short-Circuit (Lazy) Evaluation

In short-circuit (Lazy) Evaluation, the second operand of Boolean operators AND and OR is not evaluated if the value of the Boolean expression can be determined from the first operand alone.

For example, the expression

if n ! 5 0 and 1/n, tolerance:

would evaluate without error for all values of n when short-circuit evaluation is used. If programming in a language not using short-circuit evaluation, however, a “divide by zero” error would result when n is equal to 0. In such cases, the proper construction would be,

if n ! 5 0:

if 1/n , tolerance:

In the Python programming language, short-circuit evaluation is used.

2.2.6 Logically Equivalent Boolean Expressions

In numerical algebra, there are arithmetically equivalent expressions of different form. For example, x(y + z) and xy + xz are equivalent for any numerical values x, y, and z.

Similarly, there are logically equivalent Boolean expressions of different form.

For Example,

(num!=0)

not(num= =0)

2.3 Selection Control

A selection control statement is a control statement providing selective execution of instructions. A selection control structure is a given set of instructions and the selection control statement(s) controlling their execution.

2.3.1 If Statement

If statement is a selection control statement based on the value of a given Boolean expression. Statements that contain other statements are referred to as a compound statement.

Syntax

```
if condition:
        statements
else:
        statements
```

For example,

```
if mark >=60:
        print ('Grade is A')
else:
        print ('Fail')
```

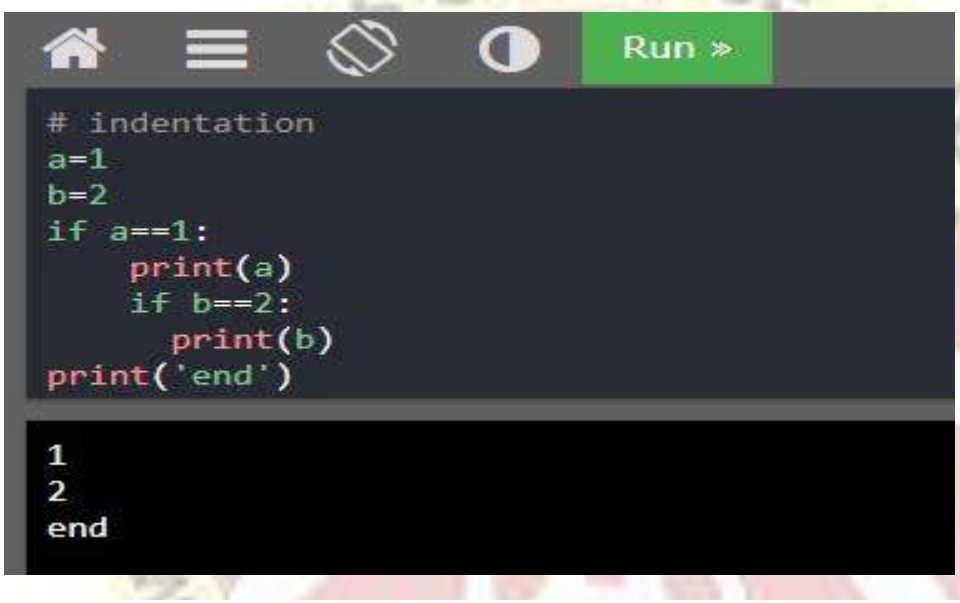
2.3.2 Indentation in Python

One fairly unique aspect of Python is that the amount of indentation of each program line is significant. In most programming languages, indentation has no affect on program logic—it is simply used to align program lines to aid readability. In Python, indentation is used to associate and group statements.

Example,

```
a=1
b=2
if a==1:
    print(a)
    if b==2:
      print(b)
print('end')
```

Output:



In the above code, the first and last line of the statement is related to the same suite because there is no indentation in front of them. So after executing first "if statement", the Python interpreter will go into the next statement. If the condition is not true, it will execute the last line of the statement. By default, Python uses four spaces for indentation, and the programmer can manage it.

2.3.3 Multi-way Selection

In a Multi-way selection more than one if statements can be nested. Thus, if-else statements must be nested to achieve multi-way selection. This is called an if ladder.

Syntax

```
if condition:
        Statement
else:
   if condition:
        Statement
    else:
        if condition:
            Statement
        else:
            Statement
```

Example #1

```
# Example of else if statement
mark=70
if mark >=80:
  print("Grade of  A")
else:
    if mark>=65:
       print("Grade of  B")
    else:
       if mark>= 50:
          print("Grade of  C")
       else:
          print("Grade of D")
```

The "elif" header is used to provides multi-way selection in a single if statement. For example,

```
# Example of elif statement
```

```
mark=70
if mark >=80:
   print(“Grade of A”)
elif mark >=65:
    print(“Grade of  B”)
elif mark >=50:
     print(“Grade of  C”)
else:
      print(“Grade of  D”)
```

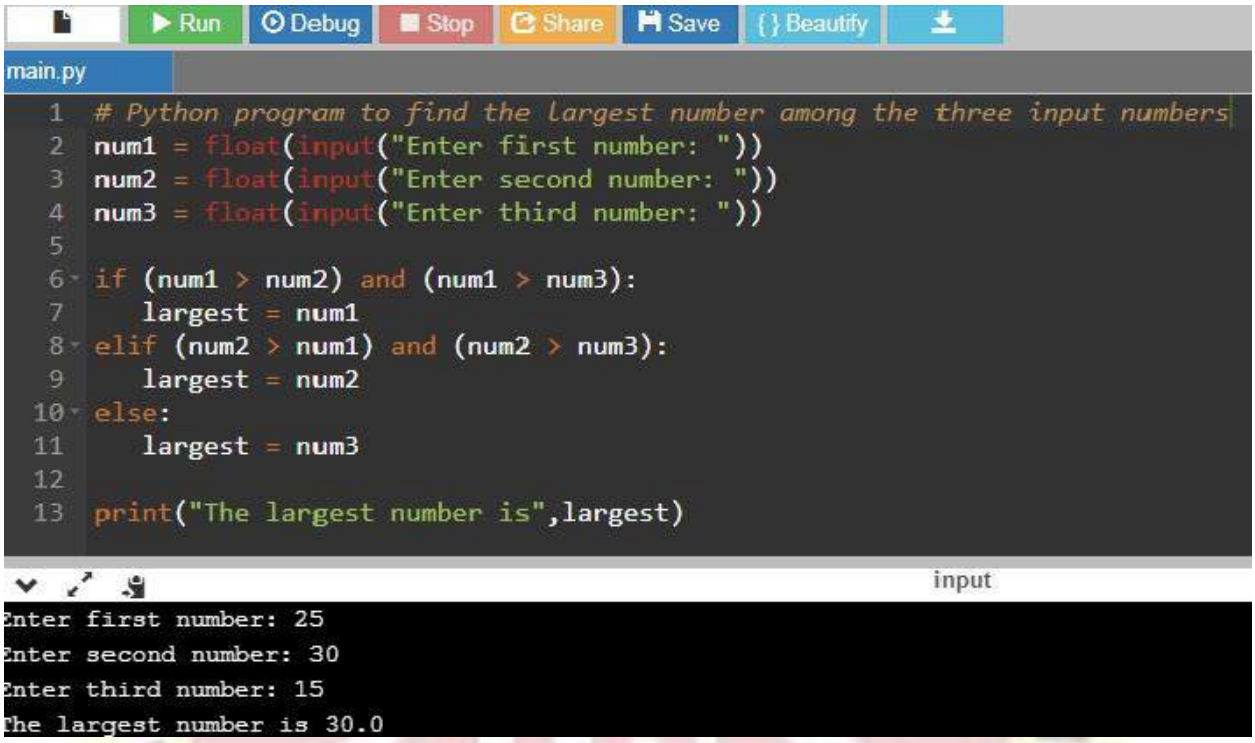
Output:

Example #2

```
# Python program to find the largest number among the three input numbers
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
num3 = float(input("Enter third number: "))
if (num1 > num2) and (num1 > num3):
   largest = num1
elif (num2 > num1) and (num2 > num3):
   largest = num2
```

```
else:
   largest = num3
 print("The largest number is",largest)
```

Output:



2.4 Iterative Control

Iteration is the process of executing a set of statements repeatedly. An iterative control statement is a control statement providing the repeated execution of a set of instructions. An iterative control structure is a set of instructions and the iterative control statement(s) controlling their execution. Because of their repeated execution, iterative control structures are commonly referred to as “loops.” Python has two primitive loop commands.

- while loop
- for loop

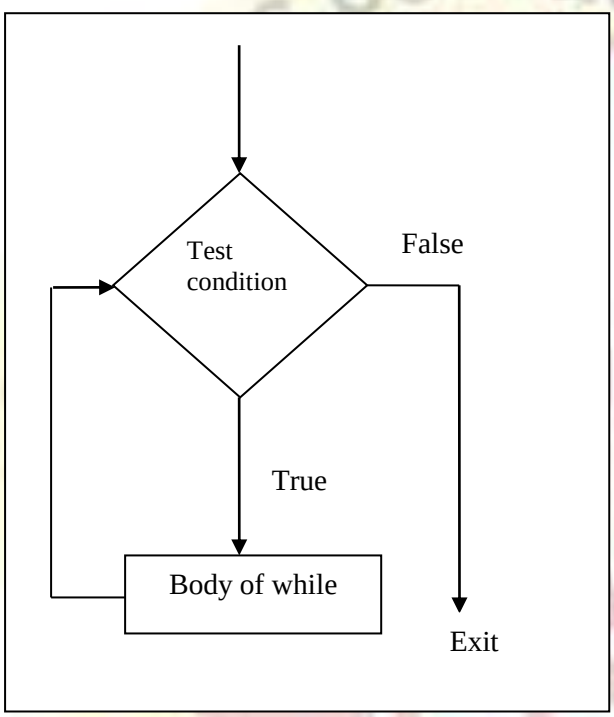
2.4.1 While loop

As long as the condition is true the body of the loop is executed. When the condition is false the body of the loop is exited. A while statement is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression.

Syntax

```
while test_condition:
    Body of the loop
```

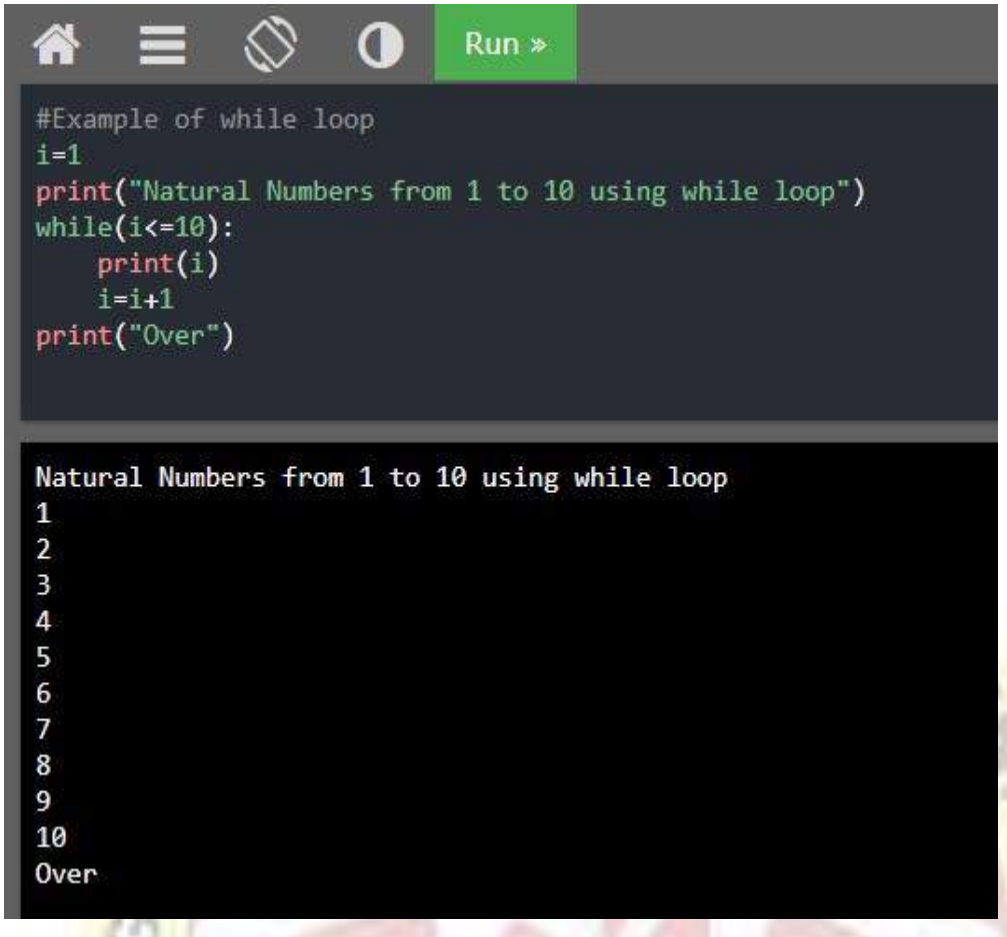
Flowchart of while loop



Example #1

```
#Example of while loop
i=1
print("Natural Numbers from 1 to 10 using while loop")
while(i<=10):
    print(i)
    i=i+1
print("Over")
```

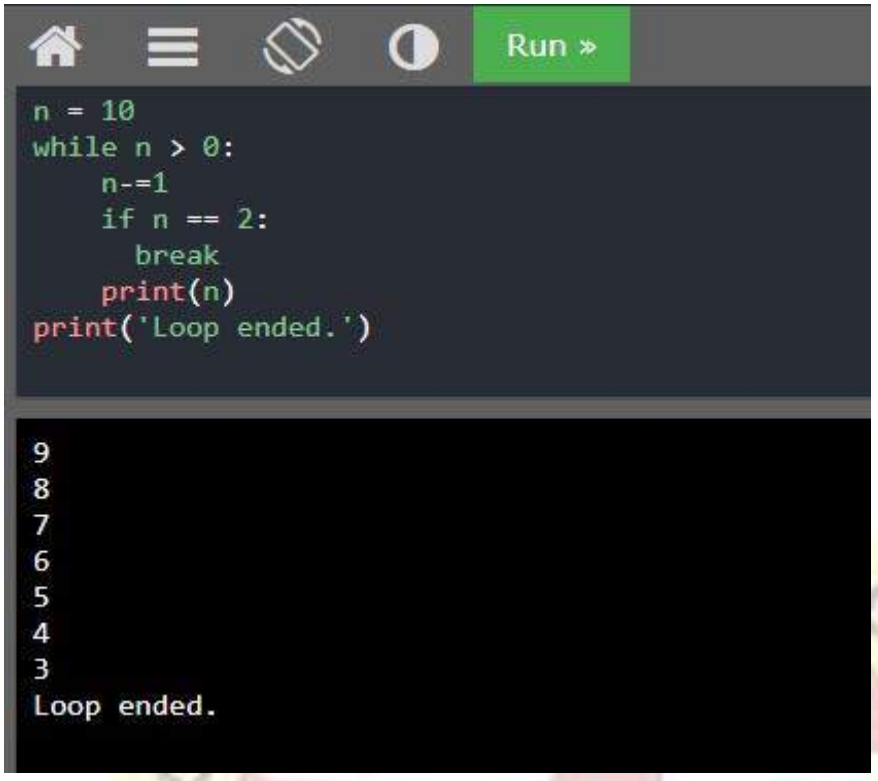
Output



Example #2

```
n = 10
while n > 0:
        n-=1
        if n == 2:
          break
        print(n)
print('Loop ended.')
```

Output



2.4.3 Infinite Loops

An **infinite loop** is an iterative control structure that never terminates. Infinite loops are generally the result of programming errors. For example, if the condition of a while loop can never be false, an infinite loop will result when executed.

Example:

```
sum=0
current=1
n=int(input("Enter the N value\n"))
print(n)
while current<=n:
    sum=sum+current
```

Here current is initialized to 1, it would remain 1 in all iterations, causing the expression current , 5 n to be always be true. Thus, the loop would never terminate.

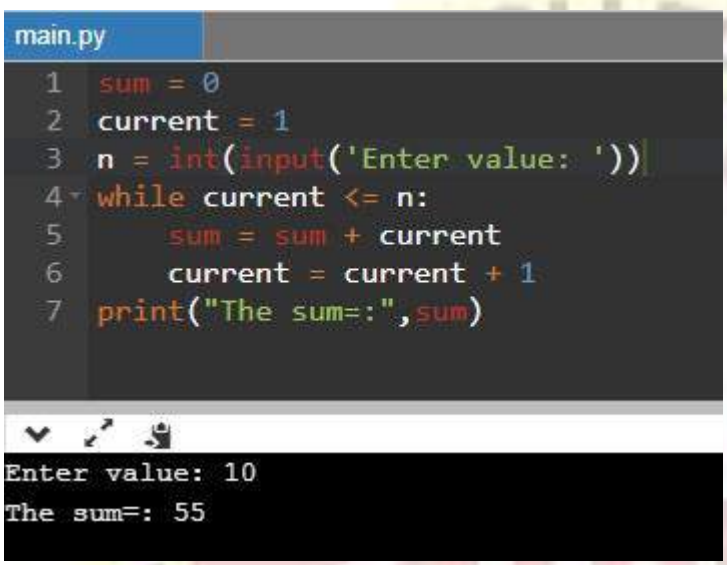
2.4.4. Definite vs. Indefinite Loops

A **definite loop** is a program loop in which the number of times the loop will iterate can be determined before the loop is executed. A definite loop is a loop in which the number of times it is going to execute is known in advance before entering the loop.

Example

```
sum =0
current = 1
n =int( input('Enter value: '))
while current <= n:
        sum = sum + current
        current = current + 1
print(“The sum=:”, sum)
```

Output



An **indefinite loop** is a program loop in which the number of times that the loop will iterate cannot be determined before the loop is executed.

2.4.5 Boolean Flags and Indefinite Loops

A single Boolean variable used as the condition of a given control statement is called a Boolean flag.

2.4 List Structures

2.4.1 List

A List is a linear data structure, thus its elements have a linear ordering. It is a collection which is ordered and changeable. In Python lists are written with square brackets. A single list may contain Data types like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation. List in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index.

For example,

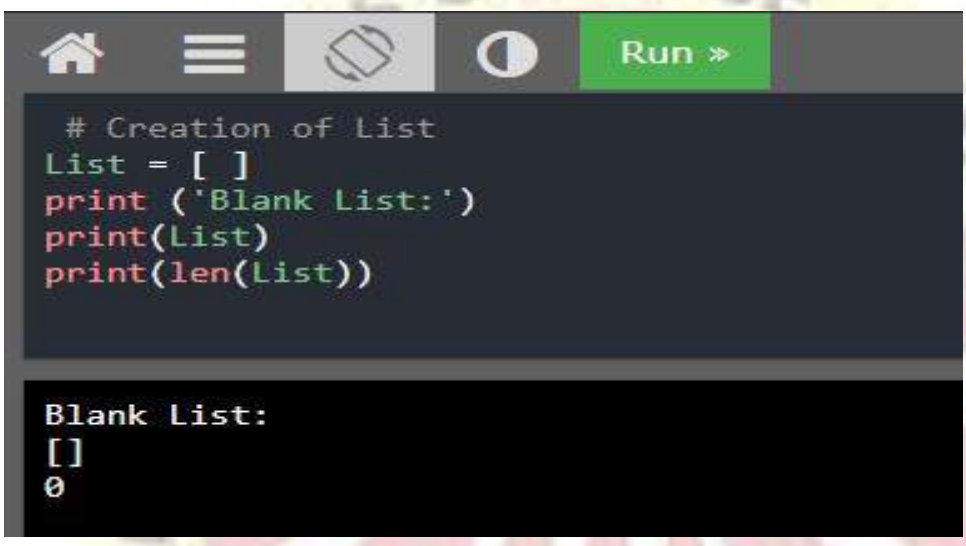
16	Index 0
14	Index 1
12	Index 2
10	Index 3
8	Index 4
6	Index 5
4	Index 6
2	Index 7

Example Program

Example #1

```
# Creation of List
List = [ ]
print ('Blank List:')
print(List)
 print(len(List))
```

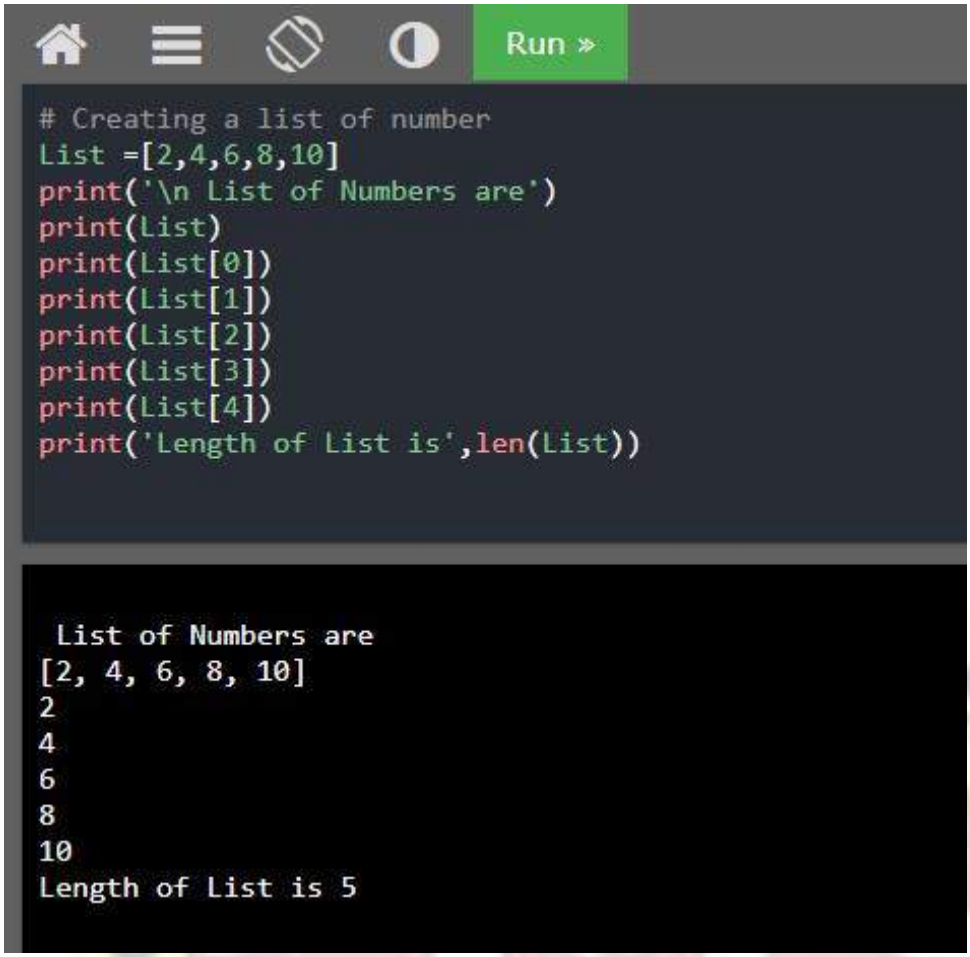
Output:



Example #2

```
# Creating a list of number
List =[2,4,6,8,10]
print('\n List of Numbers are')
print(List)
print(List[0])
print(List[1])
print(List[2])
print(List[3])
print(List[4])
 print('Length of List is',len(List))
```

Output:



Example #3

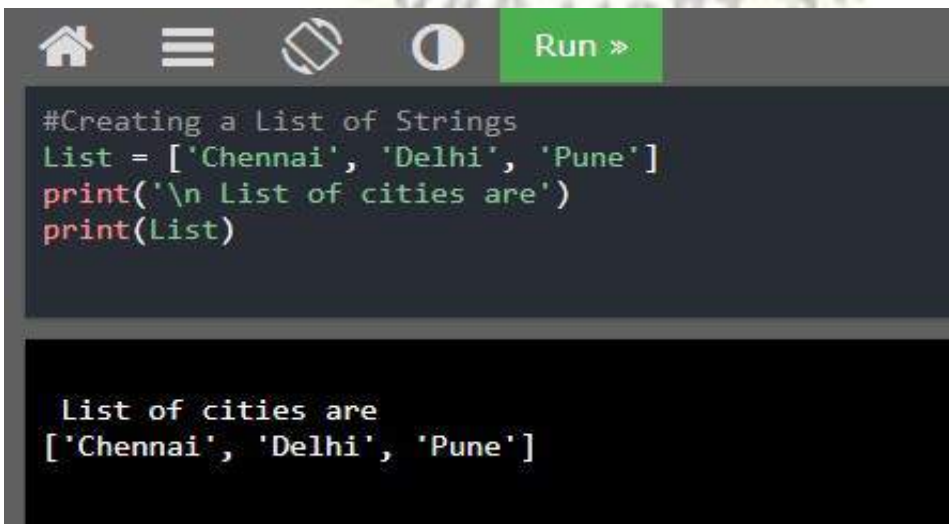
#Creating a List of Strings

List = ['Chennai', 'Delhi', 'Pune']

print('\n List of cities are')

print(List)

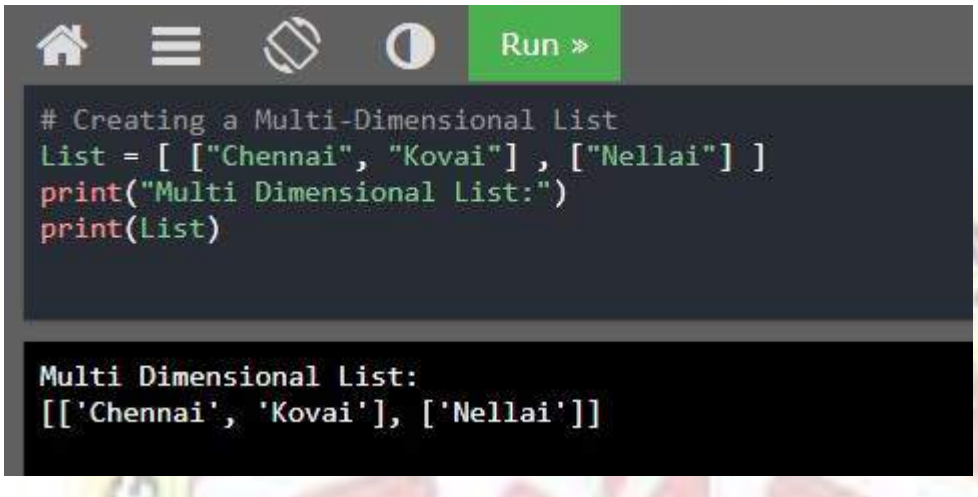
Output:



Example #4

```
# Creating a Multi-Dimensional List
List = [ ["Chennai", "Kovai"] , ["Nellai"] ]
print("Multi Dimensional List:")
print(List)
```

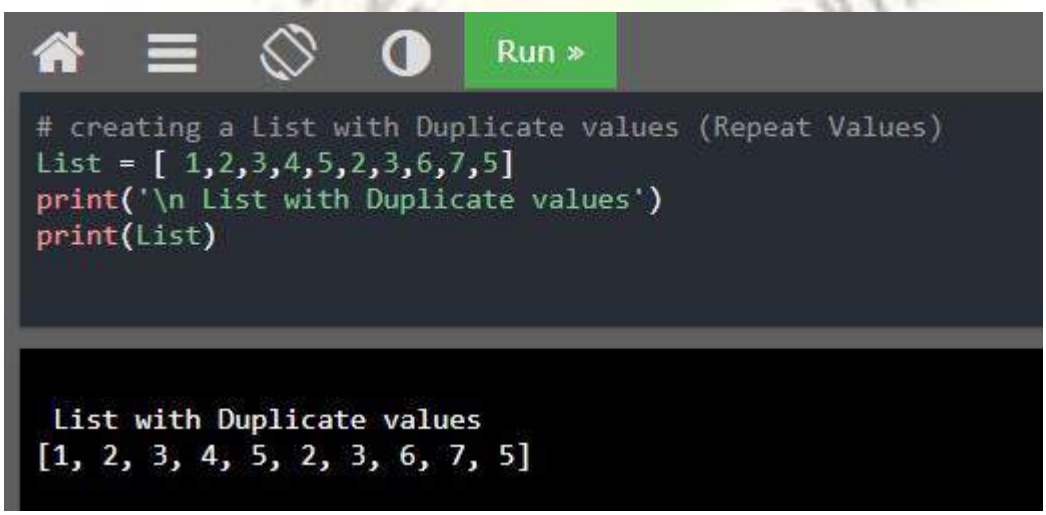
Output:



Example #5

```
# creating a List with Duplicate values (Repeat Value)
List = [ 1,2,3,4,5,2,3,6,7,5]
print('\n List with Duplicate values')
print(List)
```

Output:



Example #6

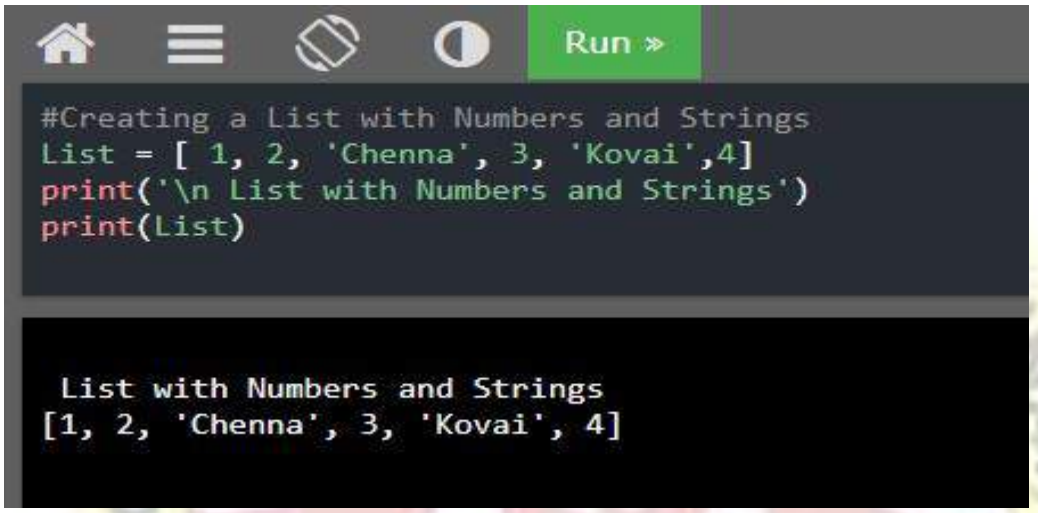
#Creating a List with Numbers and Strings

List = [1, 2, 'Chenna', 3, 'Kovai',4]

print('\n List with Numbers and Strings')

print(List)

Output:



2.4.2 Common List Operations

Common List Operations append, update, insert, delete, retrieve, extend etc.

a) append( )

It is used to add elements to the last position of the element of List.

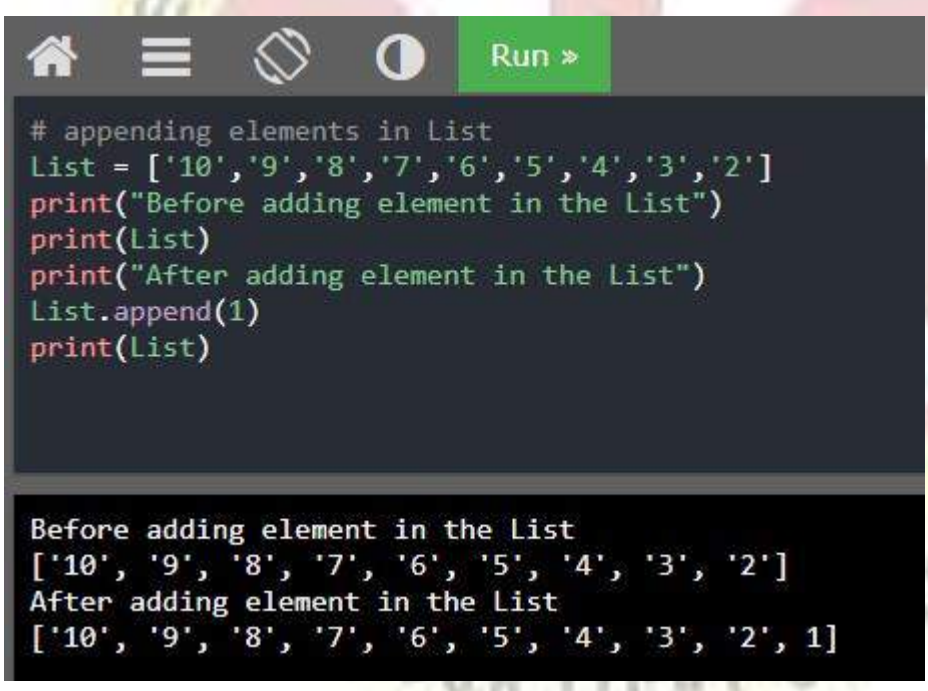
Syntax

```
List.append(element)
```

Example

```
# appending elements in List
List = ['10','9','8','7','6','5','4','3','2']
print("Before adding element in the List")
print(List)
print("After adding element in the List")
List.append(1)
print(List)
```

Output



b) insert( )

It is used to insert an element in the List at specified location.

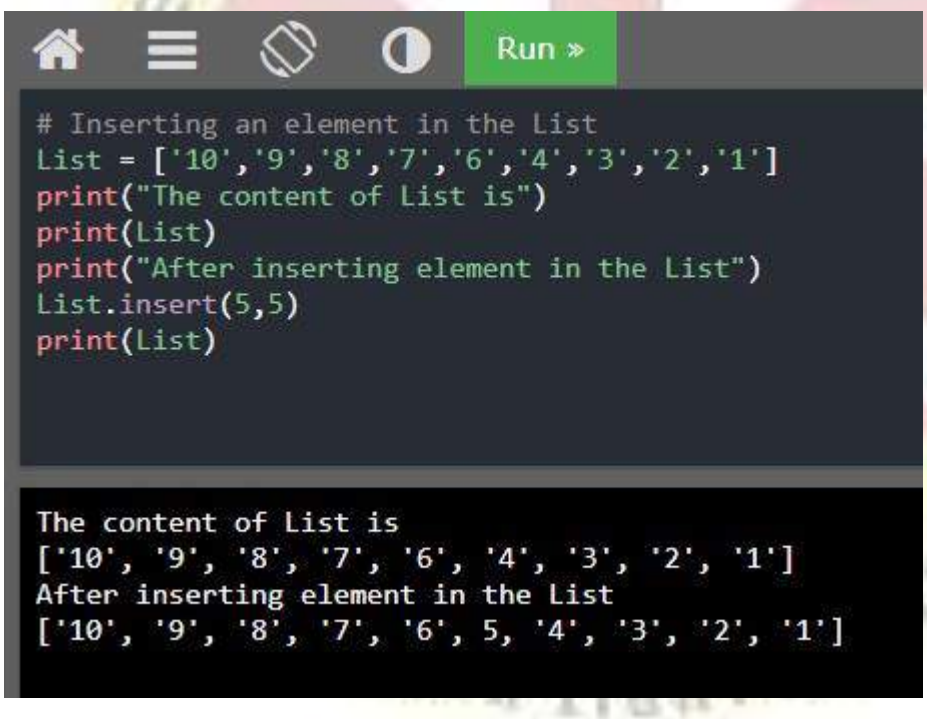
Syntax

```
List.append(element)
```

Example

```
# Inserting an element in the List
List = ['10','9','8','7','6','4','3','2','1']
print("The content of List is")
print(List)
print("After inserting element in the List")
List.insert(5,5)
print(List)
```

Output



c) extend( )

The extend( ) method is used to add multiple elements at the end of the list.

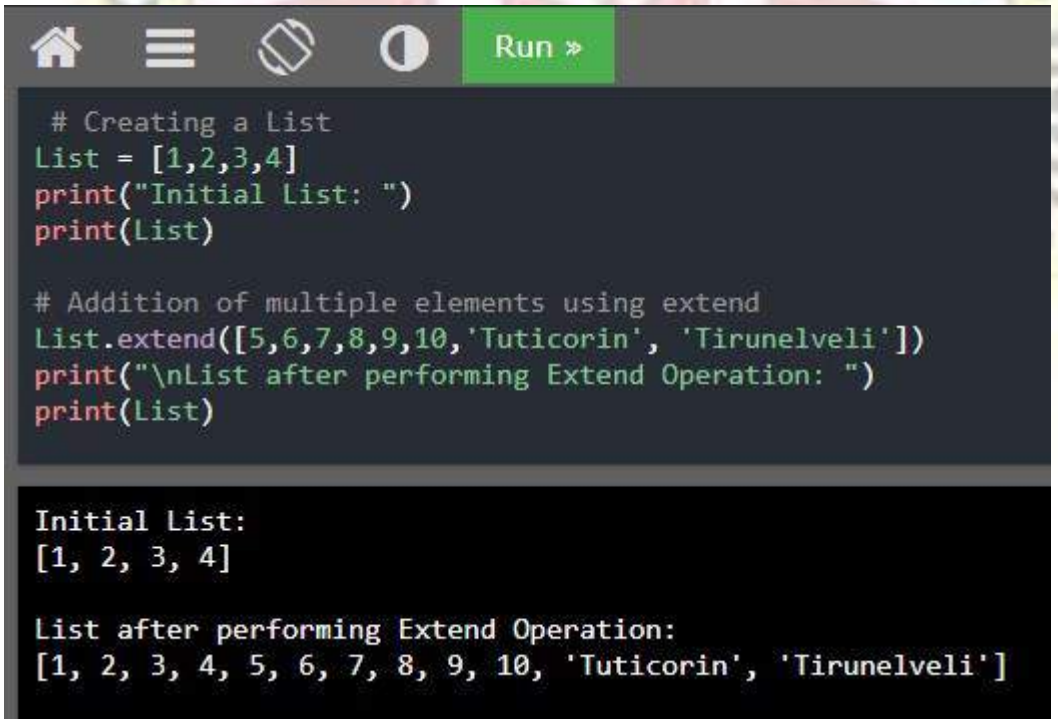
Syntax

```
List1.extend(List2)
```

Example #1

```
# Creating a List
List = [1,2,3,4]
print("Initial List: ")
print(List)
# Addition of multiple elements using extend
List.extend([5,6,7,8,9,10,'Tuticorin', 'Tirunelveli'])
print("\nList after performing Extend Operation: ")
print(List)
```

Output



Example #2

```
# Fruits list
fruits= ['Banana', 'Watermelon', 'Tomato']
# another list of Fruits
fruits1 = ['Apple', 'Jack fruit', 'Pineapple']
fruits.extend(fruits1)
# Extended List
  print('Fruits List: ', fruits)
```

Output:

d) reverse( )

The reverse( ) function is used to reverse all elements in the List. It does not take any value.

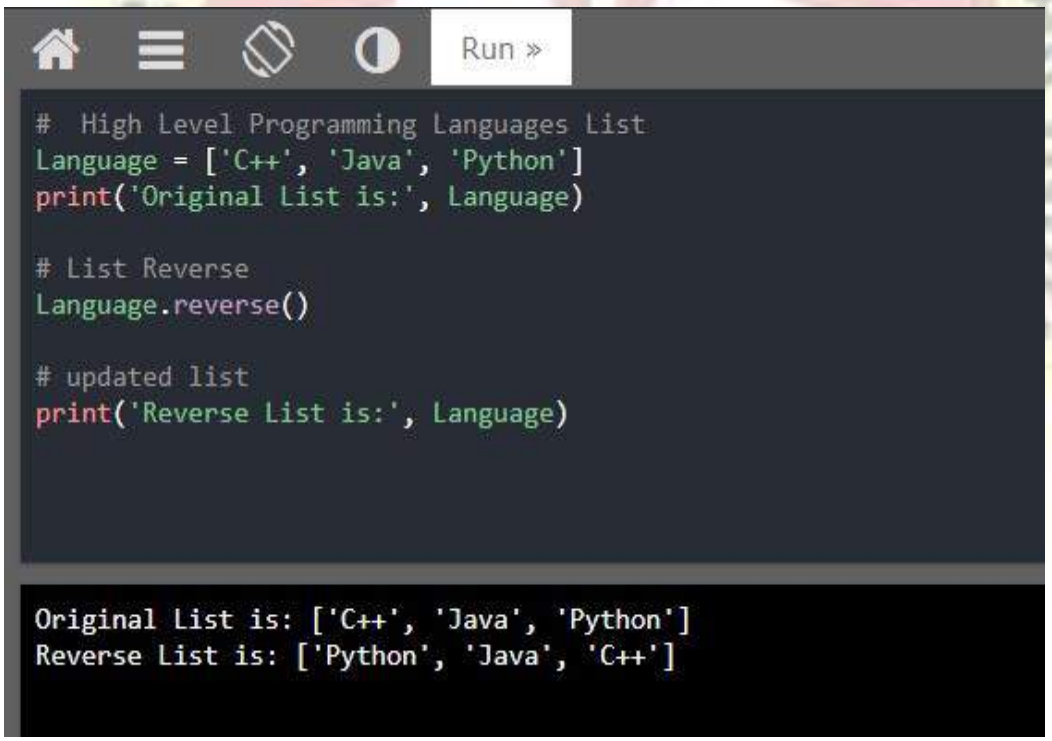
Syntax

```
List.reverse( )
```

Example

```
#  High Level Programming Languages List
Language = ['C++', 'Java', 'Python']
print('Original List is:', Language)
# List Reverse
Language.reverse()
print('Reverse List is:', Language)
```

Output:



e) remove

The remove( ) method is used to removes the specified item.

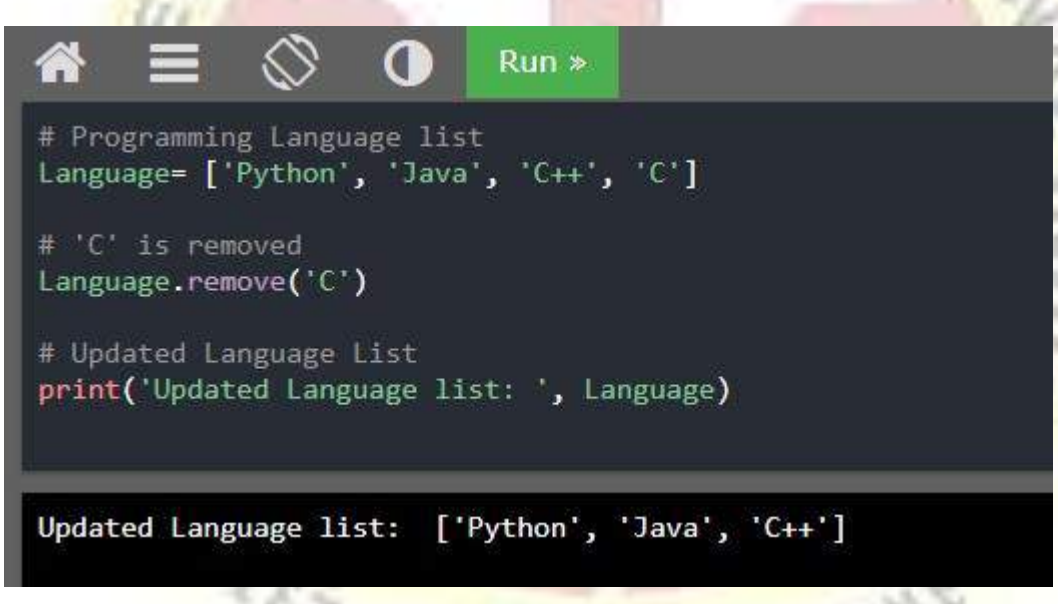
Syntax

```
List.remove(element)
```

Example #1

```
# Programming Language list
Language= ['Python', 'Java', 'C++', 'C']
# 'C' is removed
Language.remove('C')
# Updated Language List
print('Updated animals list: ', Language)
```

Output:



Example #2

```
# Remove duplicate elements
Name_List= ['Rex', 'Nixon', 'Felix', 'Franklin', 'Nixon']
# 'John' is removed
Name_List.remove('Nixon')
# Updated Name list
print('Updated Name list: ', Name_List)
```

Output:

f) pop ( )

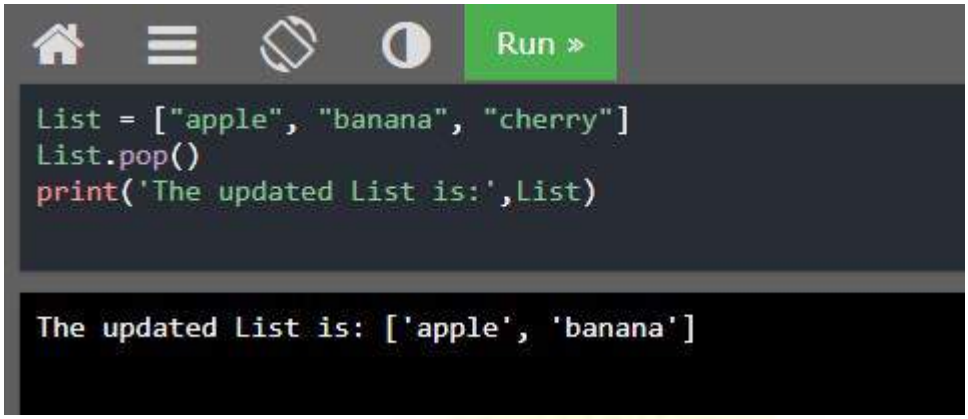
A pop( ) method returns the item present at the given index. It takes a single argument. The pointed item is removed from the list.

```
List.pop(index)
```

Example #1

```
List = ["apple", "banana", "cherry"]

List.pop()

 print('The updated List is:',List)
```

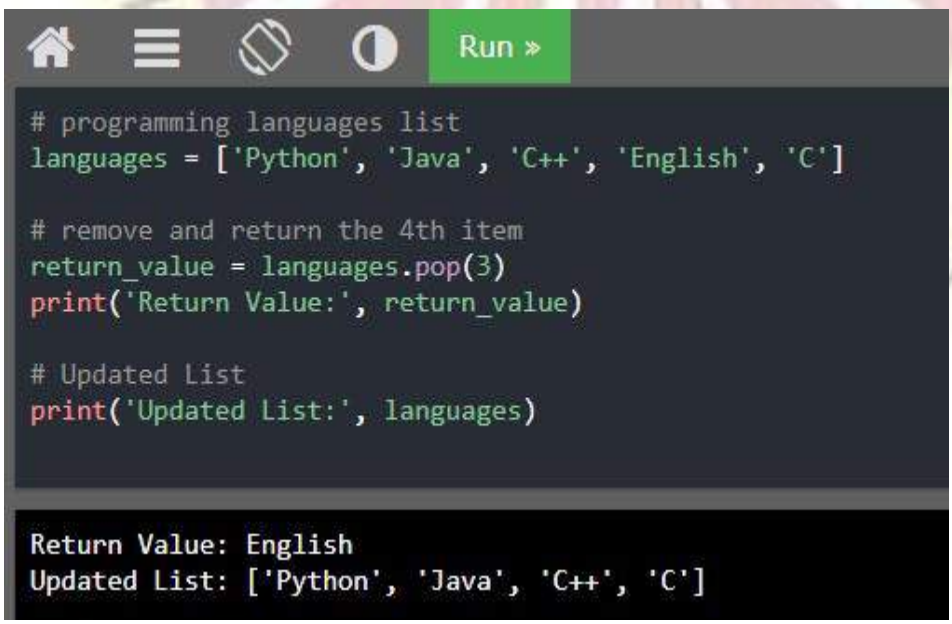
Output:



Example #2

```
# programming languages list
languages = ['Python', 'Java', 'C++', 'English', 'C']
# remove and return the 4th item
return_value = languages.pop(3)
print('Return Value:', return_value)
# Updated List
 print('Updated List:', languages)
```

Output:



g) sort( )

The sort( ) method is used to sort the elements in the list. It is sort the value in ascending order by default. It has two optional parameters.

- Reverse – if true, the elements are sorted in descending.
- Key – function that serves as a key for the sort comparison.

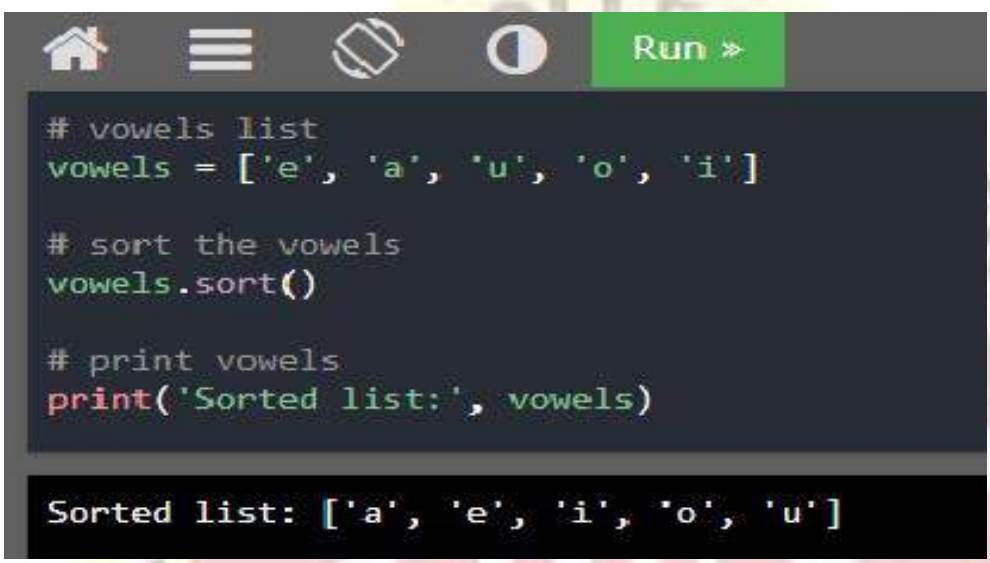
Syntax

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

Example #1

```
# vowels list
vowels = ['e', 'a', 'u', 'o', 'i']
# sort the vowels
vowels.sort()
# print vowels
print('Sorted list:', vowels)
```

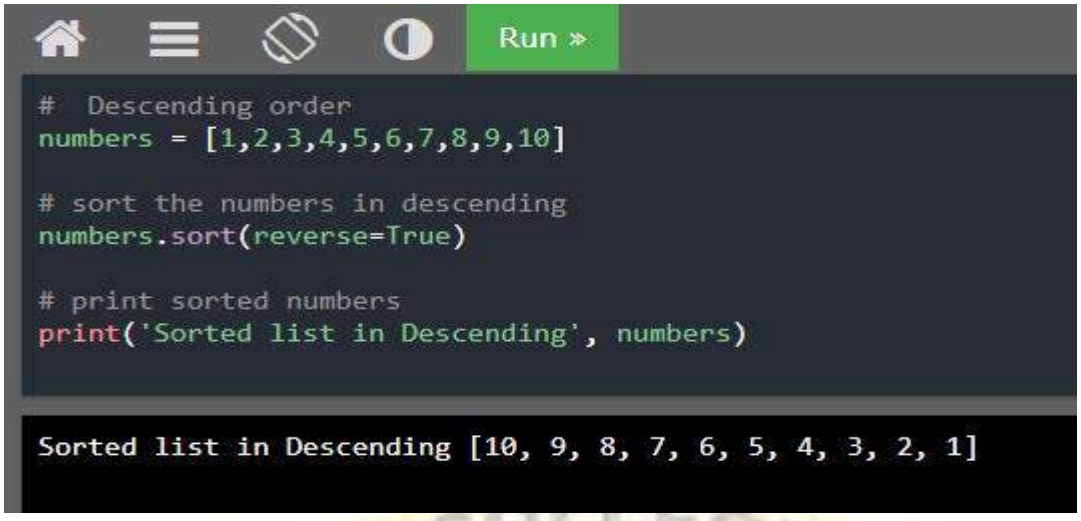
Output:



Example #2

```
#  Descending order
numbers = [1,2,3,4,5,6,7,8,9,10]
# sort the numbers in descending
numbers.sort(reverse=True)
# print sorted numbers
print('Sorted list in Descending', numbers)
```

Output:

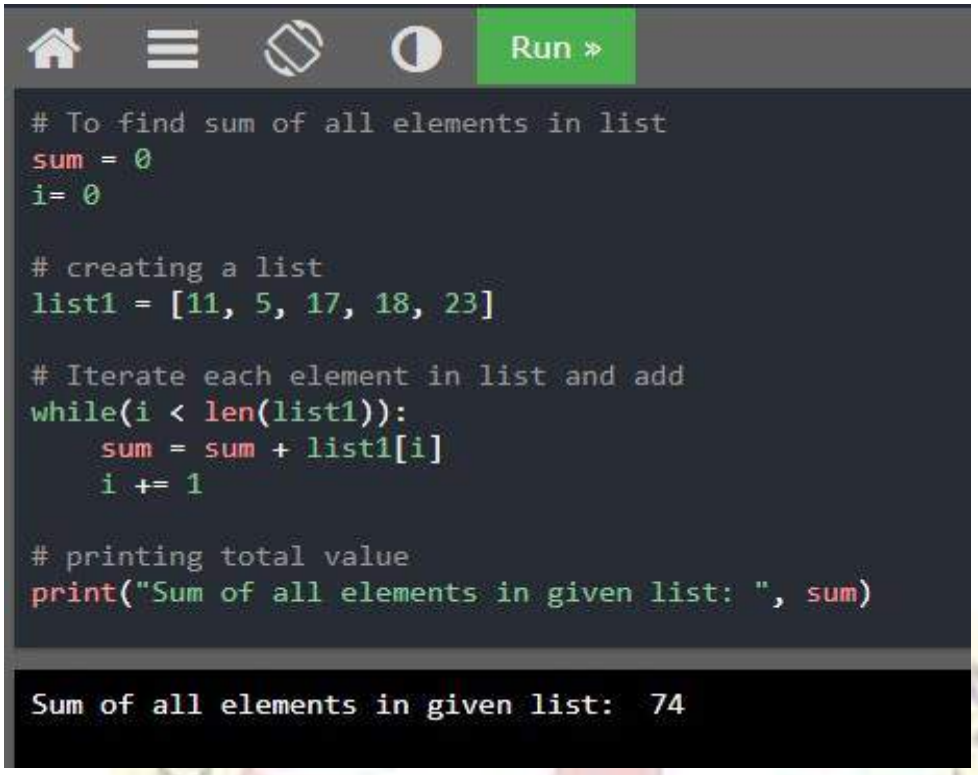


2.4.3 List Traversal

A **list traversal** is a means of accessing, one-by-one, the elements of a list. For example, to add up all the elements in a list of integers, each element can be accessed one-by-one, starting with the first, and ending with the last element. Similarly, the list could be traversed starting with the last element and ending with the first. For Example,

```
# To find sum of all elements in list
sum = 0
i= 0
# creating a list
list1 = [11, 5, 17, 18, 23]
# Iterate each element in list and add
while(i < len(list1)):
    sum = sum + list1[i]
    i += 1
# printing total value
print("Sum of all elements in given list: ", sum)
```

Output:



2.5 Lists (Sequences) in Python

A **list** in Python is a mutable, linear data structure of variable length, allowing mixed-type elements. Mutable means that the contents of the list may be altered. Lists in Python use zero based indexing. Thus, all lists have index values 0 ... n-1, where n is the number of elements in the list.

For Example,

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

The elements in List can be summed as follows,

Sum = List[0] + List[1] + List[2] + List[3] + List[4] or sum = sum(List)

creating a list

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

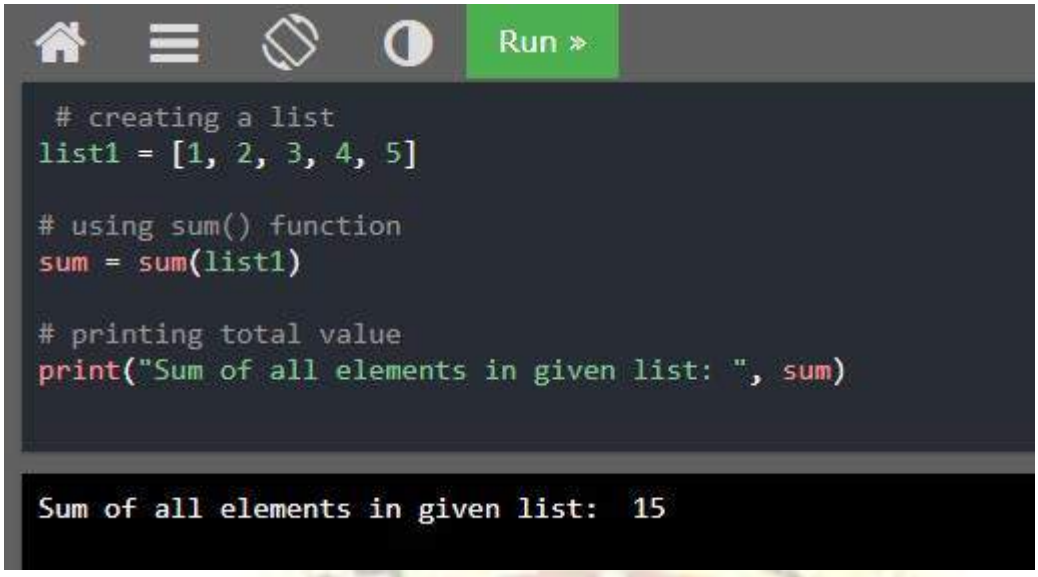
using sum() function

sum = sum(list1)

printing total value

print("Sum of all elements in given list: ", total)

Output:



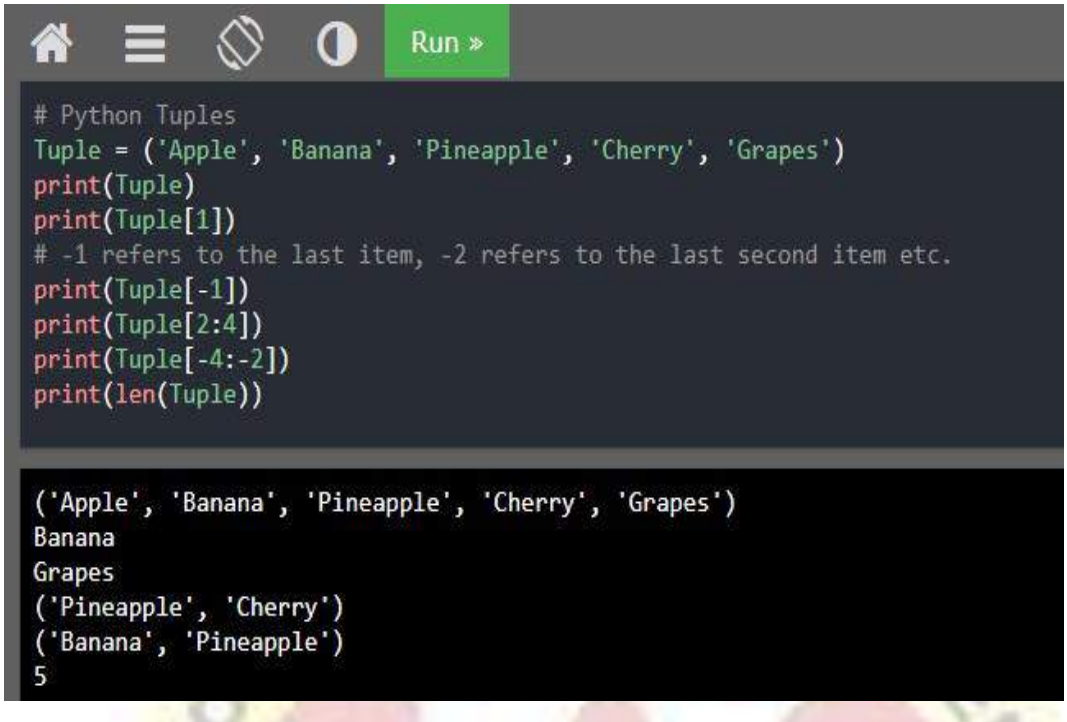
2.5.1 Tuples

A **tuple** is an immutable linear data structure. Once a tuple is defined, it cannot be altered. Tuples are denoted by parenthesis instead of square brackets. For example, Tuple = (1, 2, 3, 4, 5)

Example #1

```
# Python Tuples
Tuple = ('Apple', 'Banana', 'Pineapple', 'Cherry', 'Grapes')
print(Tuple)
print(Tuple[1])
# -1 refers to the last item, -2 refers to the last second item etc.
print(Tuple[-1])
print(Tuple[2:4])
print(Tuple[-4:-2])
print(len(Tuple))
```

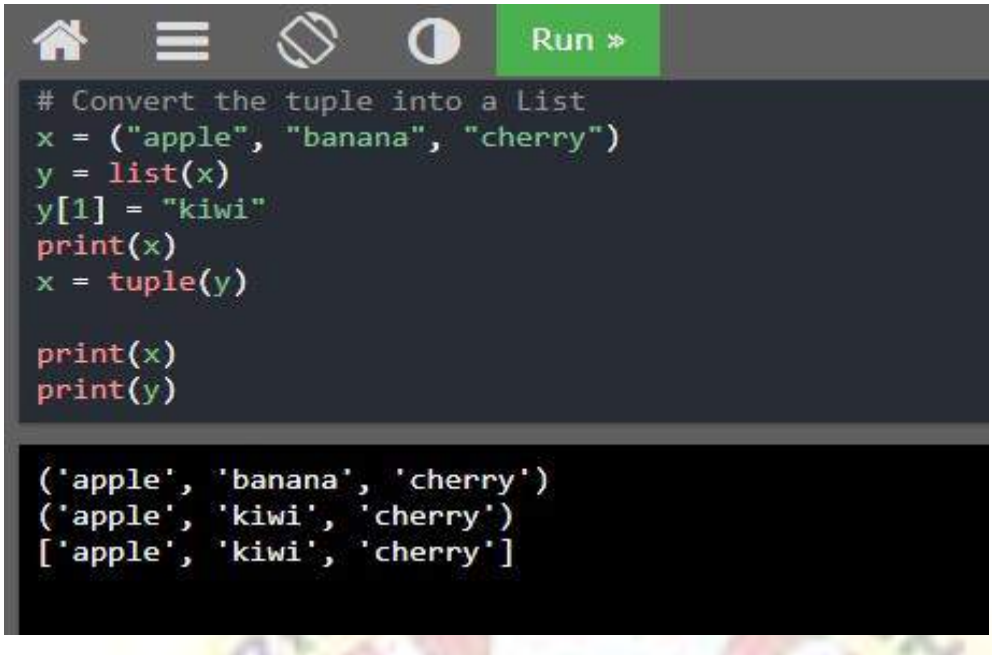
Output:



Example #2

```
# Convert the tuple into a List
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
print(x)
x = tuple(y)
print(x)
print(y)
```

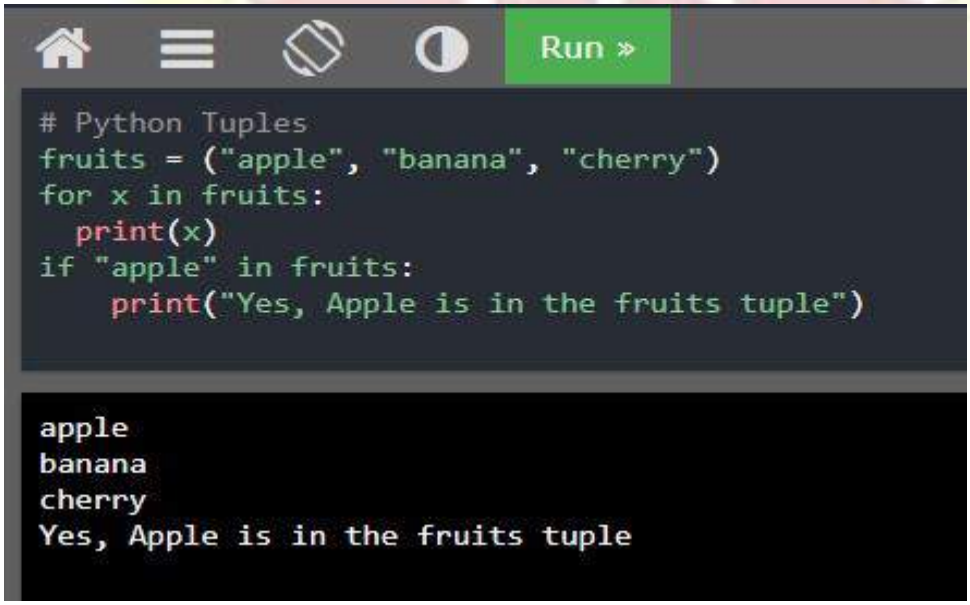
Output:



Example #3

```
# Python Tuples
fruits = ("apple", "banana", "cherry")
for x in fruits:
  print(x)
if "apple" in fruits:
    print("Yes, Apple is in the fruits tuple")
```

Output:

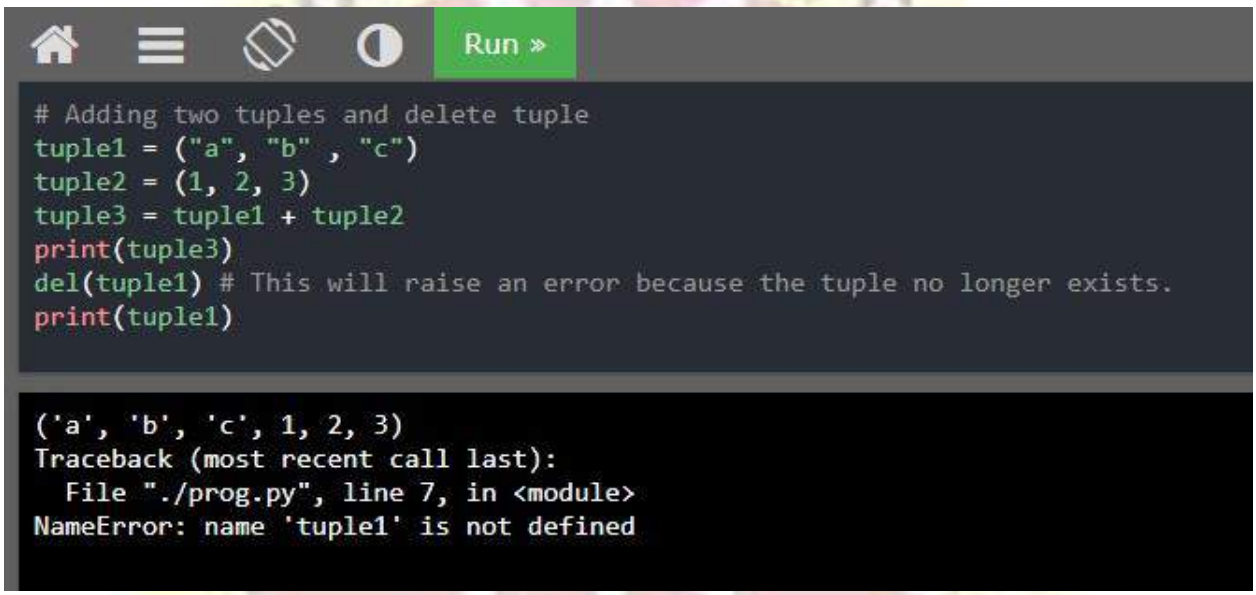


The + operator is used to add two or more tuples. The del keyword is used to delete the tuple completely.

Example #4

```
# Adding two tuples and delete tuple
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
del(tuple1) # This will raise an error because the tuple no longer exists.
print(tuple1)
```

Output:



2.5.2 Difference between List and Tuple

S.NO	List	Tuple
1	List is mutable.	Tuple is immutable.
2	List iteration is slower and is time consuming.	Tuple iteration is faster.
3	List is useful for insertion and deletion operations.	Tuple is useful for read only operations like accessing elements.
4	List consumes more memory.	Tuple consumes less memory.
5	List provides many in-built methods.	Tuples have less in-built methods.
6	List operations are more error prone.	Tuple operations are safe.

2.5.3 Sequences

A **sequence** in Python is a linearly ordered set of elements accessed by an index number. Lists, tuples, and strings are all sequences. Strings, like tuples, are immutable *;*

therefore, they cannot be altered.

For any sequence S,

- len(S) gives its length.
- S [k] retrieves the element at index k.

- The slice operation, s[index1:index2], returns a subsequence of a sequence, starting with

  the first index location up to *but not including* the second.
- The s[index :] form of the slice operation returns a string containing all the list elements starting from the given index location to the end of the sequence.
- The count method returns how many instances of a given value occur within a sequence, and the find method returns the index location of the first occurrence of a specific item,

Operation		**String** S='Hello' W='!'	**Tuple** S=(1,2,3,4) W=(5,6)	**List** S=[1,2,3,4] W=[5,6]
Length	len(S) LEN(W)	5 1	4 2	4 2
Select	S[0]	'h'	1	1
Slice	S[1:4] S[1:]	'ell' 'ello'	(2,3,4) (2,3,4)	[2,3,4] [2,3,4]
Count	S.count('e') S.count(5)	1 error	0 1	0 1
Index	S.index('e') S.index(3)	1 __	-- 2	-- 2
Membership	'h' in S	True	False	False
Concatenation	S + W	'hello!'	(1,2,3,4,5,6)	[1,2,3,4,5,6]
Minimum value	min(S)	'e'	1	1
Maximum value	max (S)	'o'	4	4
Sum	Sum(S)	Error	10	10

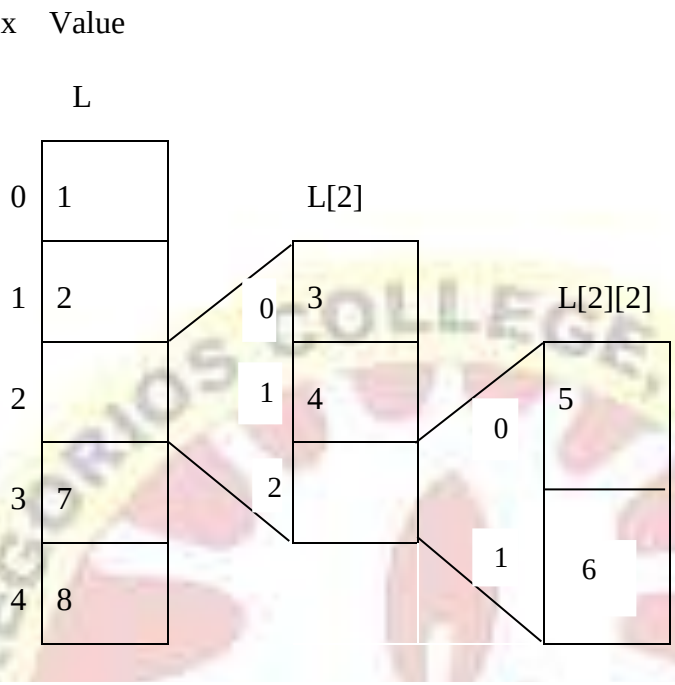
Fig. Sequence Operations in Python

2.5.4 Nested Lists

Lists and tuples can contain elements of any type, including other sequences. Thus, lists and tuples can be nested to create arbitrarily complex data structures.

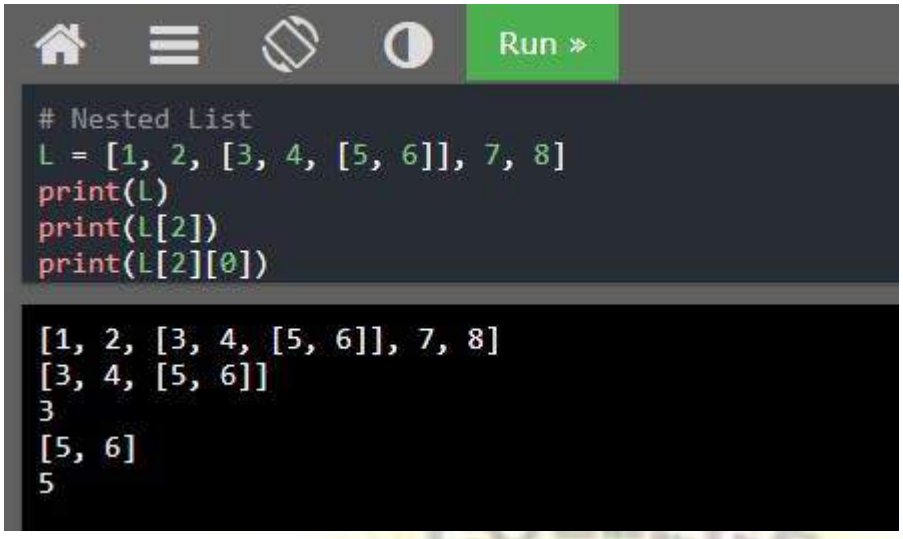
A list can contain any sort object, even another list (sub list), which in turn can contain sub lists themselves, and so on. This is known as nested list.

For Example,




```
# Nested List
L = [1, 2, [3, 4, [5, 6]], 7, 8]
print(L)
print(L[2])
print(L[2][0])
print(L[2][2])
print(L[2][2][0])
```

Output:



2.6 Iterating over lists in Python

Python's for statement provides a convenient means of iterating over lists (and other sequences). In this section, we look at both for loops and while loops for list iteration.

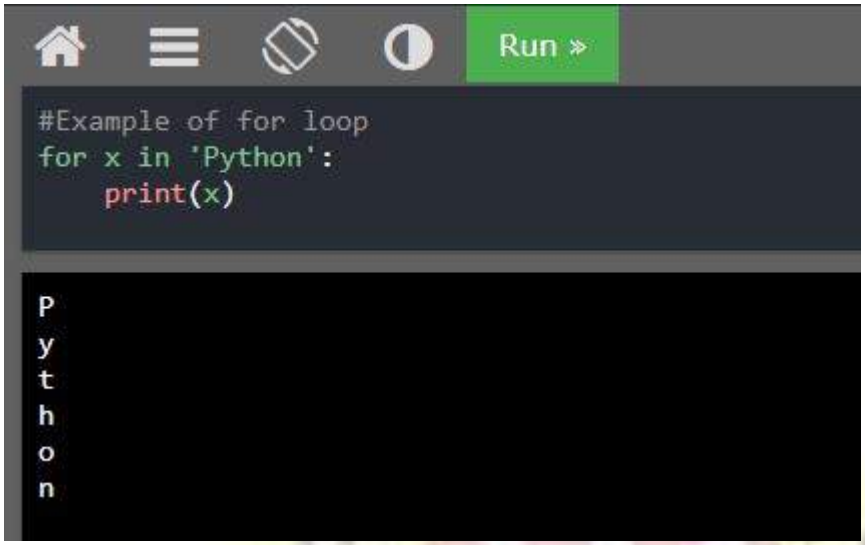
2.6.1 For loops

A **for statement** is an iterative control statement that iterates once for each element in a specified sequence of elements. A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set or a string).

Example #1

```
#Example of for loop
for x in 'Python':
print(x)
```

Output:



Example #2

```
Names = ["Infel", "Rufel",  "Jufel"]
for x in Names:
        print (x)
```

Output:

The break statement is used to stop the loop before it has executed. For example,

Example #3

```
#Using break statement
cities = ["Chennai", "Delhi", "Pune"]
for x in cities:
   print(x)
   if x == "Delhi":
        break
```

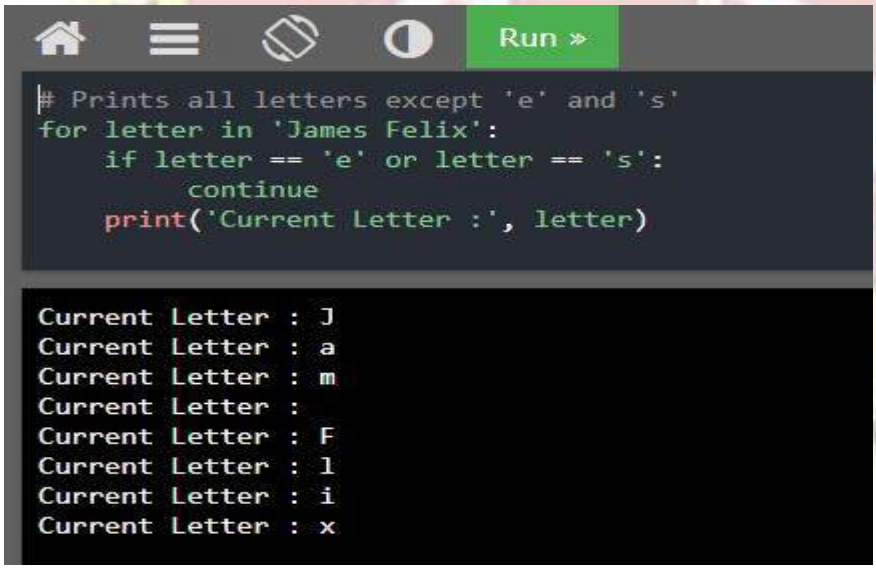
Output:

The continue statement returns the control to the beginning of the loop.

Example #4

```
# Prints all letters except 'e' and 's'
for letter in 'James Felix':
   if letter == 'e' or letter == 's':
      continue
   print('Current Letter :', letter)
```

Output:



2.6.2 The Built-in range Function

The range() function is used to generate a sequence of numbers. range() is commonly used in for looping hence, knowledge of same is key aspect when dealing with any kind of Python code. Most common use of range() function in Python is to iterate sequence type (List, string etc..) with for and while loop. It takes mainly three arguments. The range( )

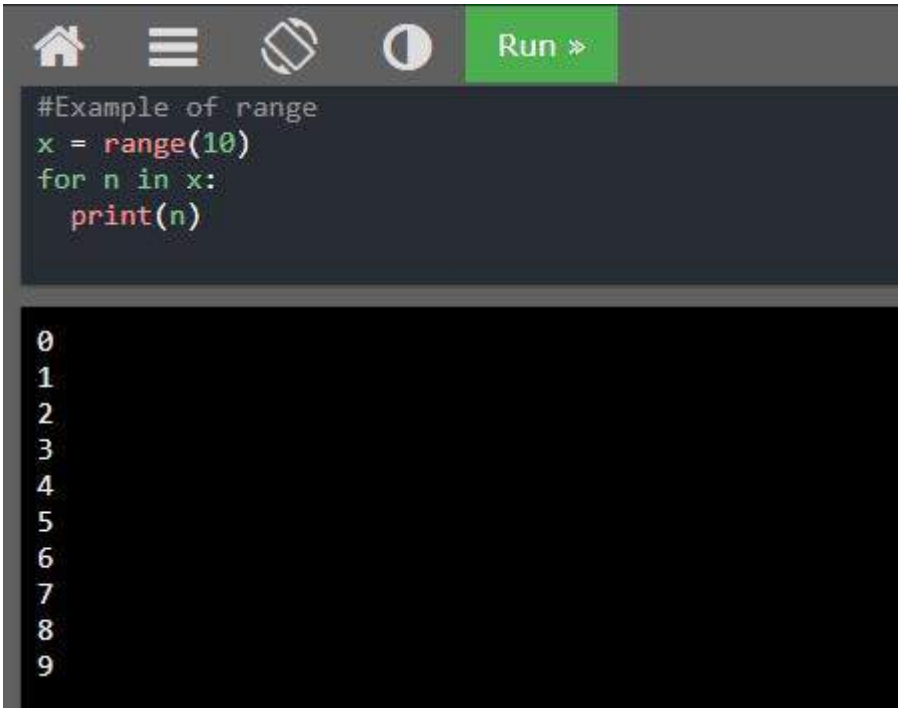
function returns a sequence of numbers, starting from by 0 default, and increments by 1, and ends at a specified number.

- **start**: integer starting from which the sequence of integers is to be returned
- **stop:** integer before which the sequence of integers is to be returned. The range of integers ends at stop – 1.
- **step:** integer value which determines the increment between each integer in the sequence.

Example #1

```
#Example of range
x = range(10)
for n in x:
  print(n)
```

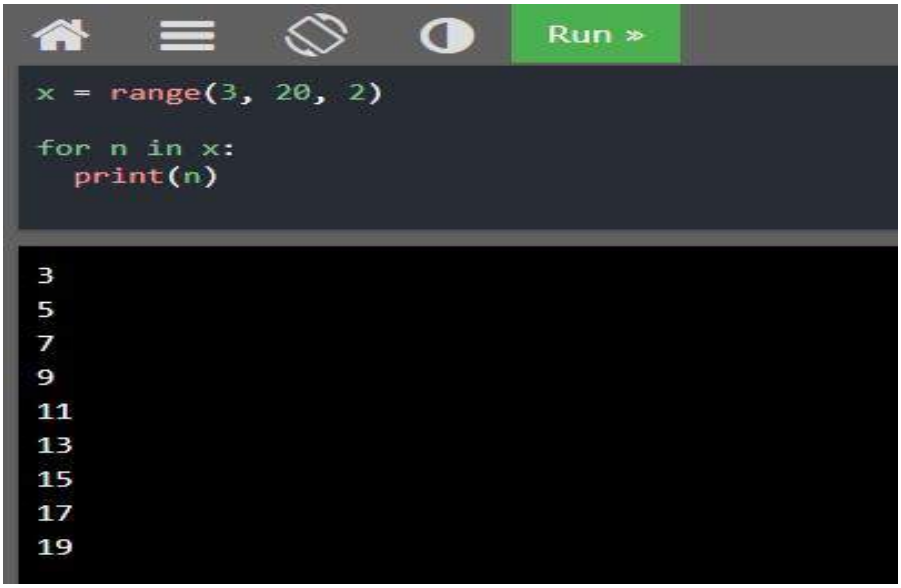
Output:



Example #2

```
x = range(3, 20, 2)
for n in x:
  print(n)
```

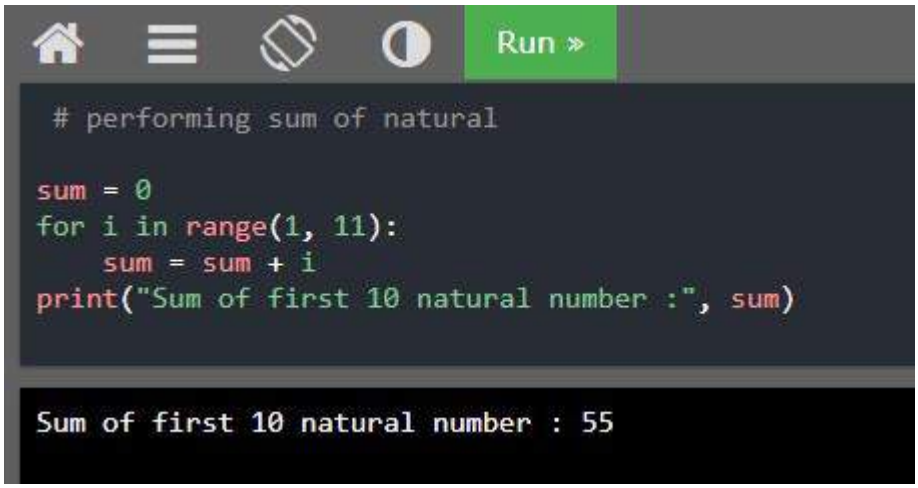
Output:



Example #3

```
# performing sum of natural

sum = 0

for i in range(1, 11):

    sum = sum + i

print("Sum of first 10 natural number :", sum)
```

Output:



Points to remember

- A control statement is a statement that determines the control flow of a set of instructions.
- The **Boolean data type** contains two Boolean values, denoted as **True** and **False**
- A **selection control statement** is a control statement providing selective execution of instructions.
- If statements can be nested in Python, resulting in multi-way selection.
- An **iterative control statement** is a control statement providing the repeated execution of a set of instructions.
- A **while statement** is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression (condition).
- An **infinite loop** is an iterative control structure that never terminates.
- A **definite loop** is a program loop in which the number of times the loop will iterate can be determined before the loop is executed.
- An **indefinite loop** is a program loop in which the number of times that the loop will iterate cannot be determined before the loop is executed.
- A single Boolean variable used as the condition of a given control statement is called a Boolean flag.
- A **list** is a linear data structure, meaning that its elements have a linear ordering.
- Operations commonly performed on lists include retrieve, update, insert, and append.
- A list traversal is a means of accessing, one-by-one, the elements of a list.
- A **list** in Python is a mutable, linear data structure of variable length, allowing mixed-type elements.
- A **tuple** is an immutable linear data structure. Once a tuple is defined, it cannot be altered.

Two mark Questions

1. Define “Straight Line Program”
2. What is the use of bool( ) function?
3. What are the relational operators?
4. Mention the Membership operators.
5. How the infinite loop will execute?
6. What do you mean by List?
7. Deine “tuple” in Python.
8. Give the syntax of while loop.
9. What is the use o range( ) function?
10. What is the use of for loop?

Five mark Questions

1. Explain about control structures.
2. Write short notes on Boolean operators.
3. Illustrate the flow chart and syntax of if-elif- else statements.
4. Write a program to calculate the simple interest using python.
5. How a tuple is iterated? Explain with an example.

Ten Mark Questions

1. Develop a program to find the largest among three numbers.
2. Discuss on various operations of List in Python.
3. Explain the basic Tuple Operations with examples

UNIT – III

3.1 Functions

A function is a set of statements that take inputs, do some specific computation and produces output. The idea is to put some commonly or repeatedly done task together and make a function, so that instead of writing the same code again and again for different inputs, we can Call the function.

Python provides built-in functions like print(), etc. but we can also create your own functions. These functions are called user-defined functions. A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

• Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.

• Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place

. • Dividing a long program into functions allows you to debug the parts one at a time and

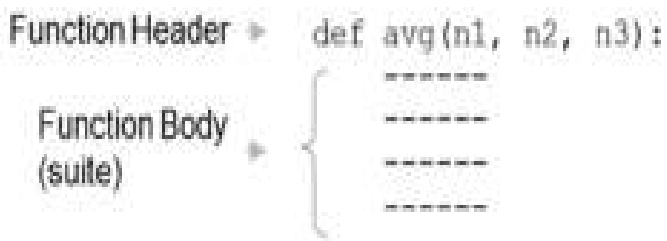
then assemble them into a working whole.

• Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

3.1.1Creating a Function

In Python a function is defined using the def keyword:

The elements of a function definition are given in Figure 5-1



The first line of a function definition is the function header. A function header starts with the keyword def, followed by an identifier (avg), which is the function's name. The function name is followed by a comma-separated (possibly empty) list of identifiers (n1, n2, n3) called formal parameters , or simply "parameters." Following the parameter list is a colon (:). Following the function header is the body of the function.

3.2 Non-Value-Returning Functions

A non-value-returning function is called not for a returned value, but for its side effects . A side effect is an action other than returning a function value, such as displaying output on the screen.

Example

```
def my_function():
  print("function example")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
```

```
  print("functionexample")
```

my_function()

3.3 Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):
  print(fname+ "CSC")

my_function("Deepa")
my_function("Divya")
my_function("Ramesh")
```

```
# A simple Python function to check whether x is even or odd
x=int(input(“enter any number\n”))
def evenOdd( x ):
    if (x % 2 == 0):
        print "even"
    else:
        print "odd"
  evenOdd(x)
```

3.3.1Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example

```
def my_function(fname,lname):
```

```
  print(fname+ "" +lname)

my_function("MGC", "Chennai")
```

3.3.2Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Example

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Rajesh", "Ramesh", "Vignesh")
```

3.3.3Keyword Arguments

You can also send arguments with the *key = value* syntax.

This way the order of the arguments does not matter.

Example

```
def my_function(child3,child2,child1):
  print("The youngest child is " + child3)

my_function(child1 = "Rajesh", child2 = "Vishnu", child3 = "Vijay")
```

3.3.4Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

Example

If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Venkatesh", lname = "Kumar")
```

Arbitrary Kword Arguments are often shortened to ***kwargs* in Python documentations.

3.3.5 Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

Example

```
def my_function(country= "Norway"):
  print("Iamfrom" + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

3.4 Passing a List as an Argument

Example

```
def my_function(food):
  for x in food:
    print(x)

fruits=["apple", "banana", "cherry"]

my_function(fruits)
```

3.5 Calling Value-Returning Functions

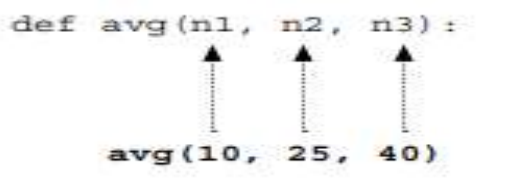
To let a function return a value, use the return statement:

Example

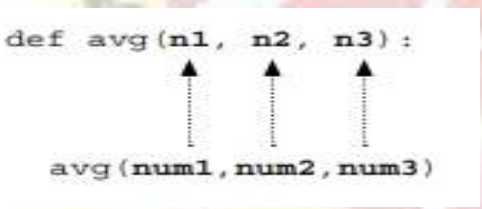
```
def my_function(x):
  return 5 *x


print(my_function(3))
print(my_function(5))
print(my_function(9))
```

3.6 Mutable vs. Immutable Arguments

We know that when a function is called, the current values of the arguments passed become the initial values of their corresponding formal parameters,

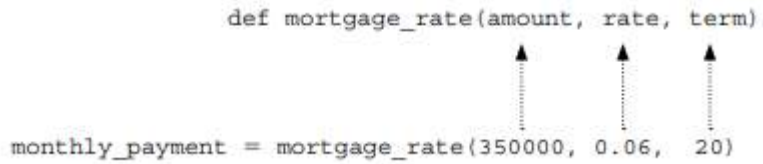


In this case, literal values are passed as the arguments to function avg. When variables are passed as actual arguments, however, as shown below,



3.7 Keyword Arguments in Python

A positional argument is an argument that is assigned to a particular parameter based on its position in the argument list, as illustrated below



This function computes and returns the monthly mortgage payment for a given loan amount (amount), interest rate (rate), and number of years of the loan (term)

Python provides the option of calling any function by the use of keyword arguments. A keyword argument is an argument that is specified by parameter name

```
def mortgage_rate(amount, rate, term)
monthly_payment = mortgage_rate(rate=0.06, term=20, amount=350000)
```

3.8Variable Scope

3.8.1Local Scope and Local Variables

A local variable is a variable that is only accessible from within a given function. Such variables are said to have local scope . In Python, any variable assigned a value in a function becomes a local variable of the function.

```
def func1():
    n = 10
    print('n in func1 = ', n)

def func2():
    n = 20
    print('n in func2 before call to func1 = ', n)
    func1()
    print('n in func2 after call to func1 = ', n)
```

```
>>> func2()
n in func2 before call to func1 =  20
n in func1 =  10
n in func2 after call to func1 =  20
```

The period of time that a variable exists is called its lifetime . Local variables are automatically created (allocated memory) when a function is called, and destroyed (deallocated) when the function terminates. Thus, the lifetime of a local variable is equal to the duration of its function's execution.

3.8.2 Global Variables and Global Scope

A global variable is a variable that is defined outside of any function definition. Such variables are said to have global scope .

```
max = 100  # global variable

def func1(count):
  if count < max:
    .

def func2(count):
  for i in range(1, max):
    .
```

global variable max accessed

global variable max accessed

Variable max is defined outside func1 and func2 and therefore "global" to each. As a result, it is directly accessible by both functions.

UNIT -IV

4.1 What Is an Object?

An object contains a set of attributes, stored in a set of instance variables , and a set of functions called methods that provide its behavior. For example, when sorting a list in procedural programming, there are two distinct entities—a sort function and a list to pass it, as depicted in Figure 6-2.

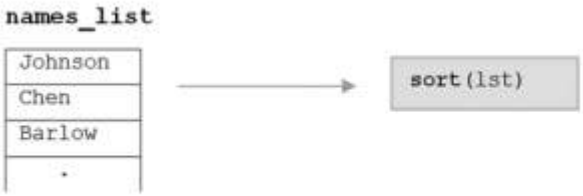


FIGURE 6-2 Procedural Programming Approach

In object-oriented programming, the sort routine would be part of the object containing the list, depicted in Figure 6-3.

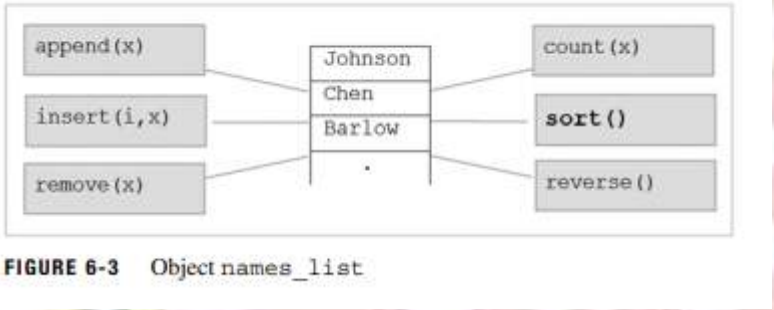


FIGURE 6-3 Object names_list

Here, names_list is an object instance of the Python built-in list type. All list objects contain the same set of methods. Thus, names_list is sorted by simply calling that object's sort method, names_list.sort()

The period is referred to as the dot operator , used to select a member of a given object.

4.1.1Object References

References in Python

In Python, objects are represented as a reference to an object in memory, as shown in Figure 6-5.

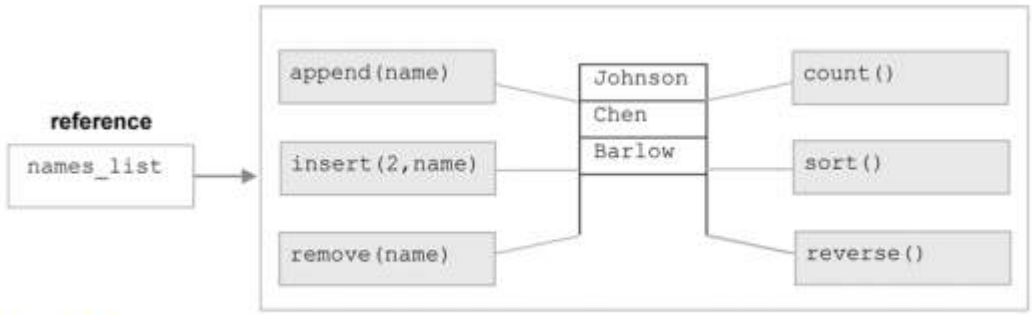


FIGURE 6-5 Object Reference

A reference is a value that references, or "points to," the location of another entity. Thus, when a new object in Python is created, two entities are stored—the object, and a variable holding a reference to the object. All access to the object is through the reference value. This is depicted in Figure 6-6.

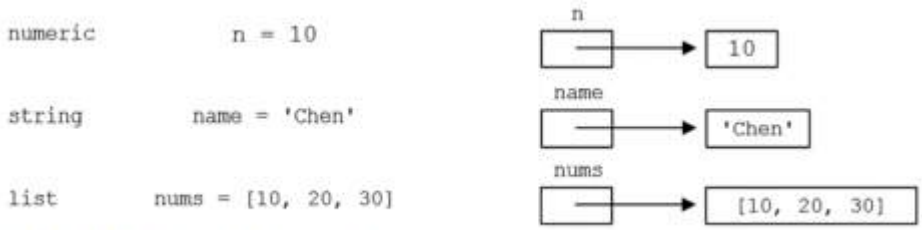


FIGURE 6-6 Object References to Python Values

The value that a reference points to is called the dereferenced value . This is the value that the variable represents, as shown in Figure 6-7.

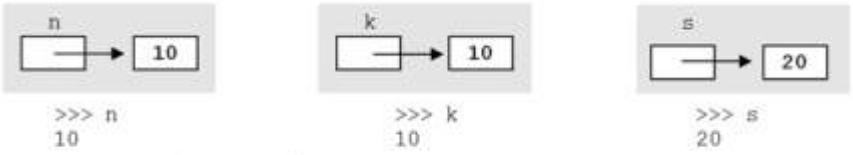


FIGURE 6-7 Variables' Dereferenced Values

We can get the reference value of a variable (that is, the location in which the corresponding object is stored) by use of built-in function id

```
>>> id(n)        >>> id(k)              >>> id(s)
505498136        505498136             505498296
```

4.1.2The Assignment of References

With our current understanding of references, consider what happens when variable n is assigned to variable k, depicted in Figure 6-8.

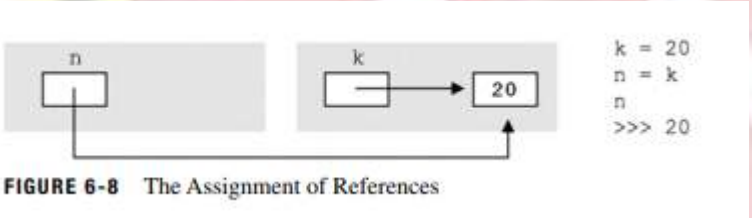


FIGURE 6-8 The Assignment of References

When variable n is assigned to k, it is the reference value of k that is assigned, not the dereferenced value 20, as shown in Figure 6-8. This can be determined by use of the built-in id function, as demonstrated below

```
>>> id(k)                    >>> id(k) == id(n)
505498136                    True

>>> id(n)                    >>> n is k
505498136                    True
```

4.1.3Memory Deallocation and Garbage Collection

Next we consider what happens when in addition to variable k being reassigned, variable n is reassigned as well. The result is depicted in Figure 6-10.

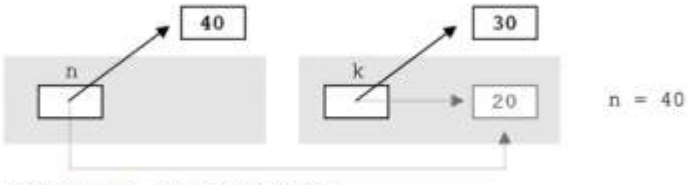


FIGURE 6-10 Inaccessible Values

After n is assigned to 40, the memory location storing integer value 20 is no longer referenced— thus, it can be deallocated . To deallocate a memory location means to change its status from “currently in use” to “available for reuse.” In Python, memory deallocation is automatically performed by a process called garbage collection . Garbage collection is a method of automatically determining which locations in memory are no longer in use and deallocating them. The garbage collection process is ongoing during the execution of a Python program.

4.1.4List Assignment and Copying

We know that when a variable is assigned to another variable referencing a list, each variable ends up referring to the same instance of the list in memory, depicted in Figure 6-11.

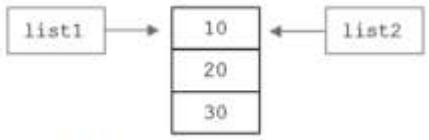


FIGURE 6-11 List Assignment

Thus, any changes to the elements of list1 results in changes to list2,

We also learned that a copy of a list can be made as follows,

list() is referred to as a list constructor . The result of the copying is depicted in Figure 6-12.

FIGURE 6-12 Copying of Lists by Use of the List Constructor

A copy of the list structure has been made. Therefore, changes to the list elements of list1 will not result in changes in list2.

The situation is different if a list contains sublists, however. The resulting list structure after the assignment is depicted in Figure 6-13.

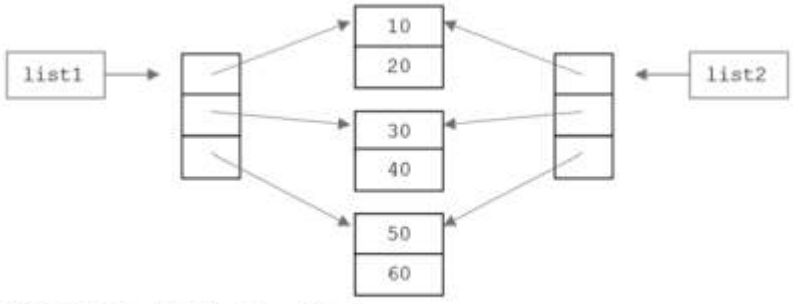


FIGURE 6-13 Shallow Copy List Structures

We see that although copies were made of the top-level list structures, the elements within each list were not copied. This is referred to as a shallow copy

A deep copy operation of a list (structure) makes a copy of the complete structure, including

sublists. The result of this form of copying is given in Figure 6-16.

6.1

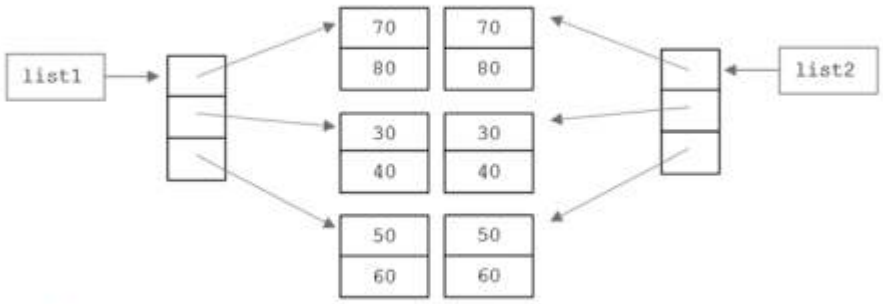


FIGURE 6-16 Deep Copy List Structures

4.2 Turtle Graphics

Turtle graphics refers to a means of controlling a graphical entity (a "turtle") in a graphics window with x,y coordinates. A turtle can be told to draw lines as it travels, therefore having the ability to create various graphical designs.

4.2.1Creating a Turtle Graphics Window

The first step in the use of turtle graphics is the creation of a turtle graphics window (a turtle screen). Figure 6-17 shows how to create a turtle screen of a certain size with an appropriate title bar.

```
import turtle

# set window size
turtle.setup(800, 600)

# get reference to turtle window
window = turtle.Screen()

# set window title bar
window.title('My First Turtle Graphics Program')
```

FIGURE 6-17 Creating a Turtle Graphics Window

Assuming that the import turtle form of import is used, each of the turtle graphics methods must be called in the form turtle. methodname. The first method called, setup, creates a graphics window of the specified size (in pixels). In this case, a window of size 800 pixels wide by 600 pixels high is created. The center point of the window is at coordinate (0,0). The top-left, top-right, bottom-left, and bottom-right coordinates for a window of size (800, 600) are as shown in Figure 6-18. A turtle graphics window in Python is also an object. Therefore, to set the title of this window, we need the reference to this object. This is done by call to method Screen

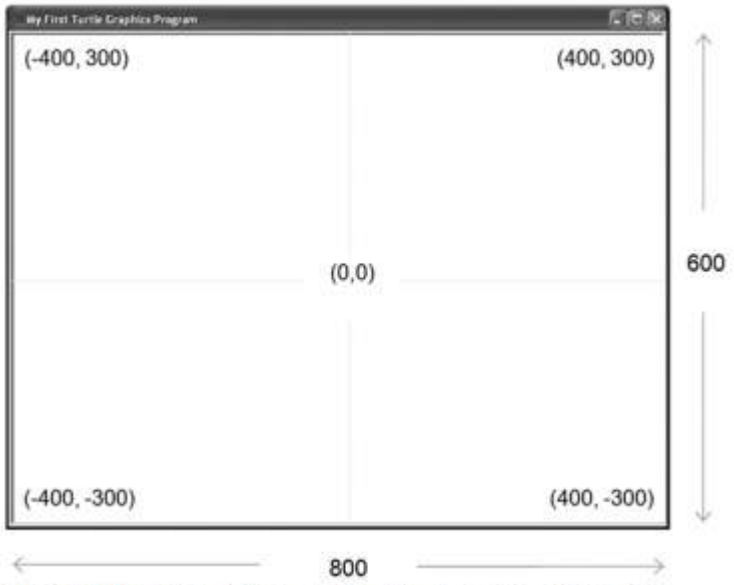


FIGURE 6-18 Python Turtle Graphics Window (of size 800 × 600)

The background color of the turtle window can be changed from the default white background color. This is done using method bgcolor,

```
window = turtle.Screen()
window.bgcolor('blue')
```

4.2.2The "Default" Turtle

A "turtle" is an entity in a turtle graphics window that can be controlled in various ways. Like the graphics window, turtles are objects. A "default" turtle is created when the setup method is called. The reference to this turtle object can be obtained by,

```
the_turtle = turtle.getturtle()
```

A call to getturtle returns the reference to the default turtle and causes it to appear on the screen. The initial position of all turtles is the center of the screen at coordinate (0,0), as shown in Figure 6-19.

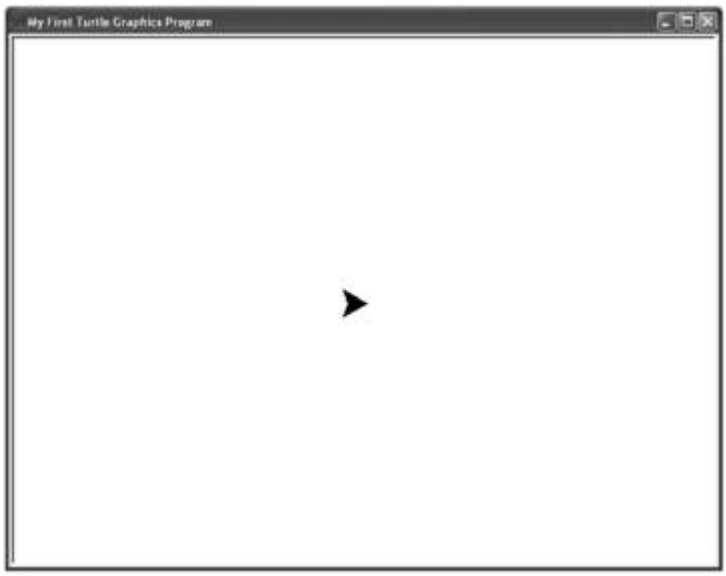


FIGURE 6-19 The Default Turtle

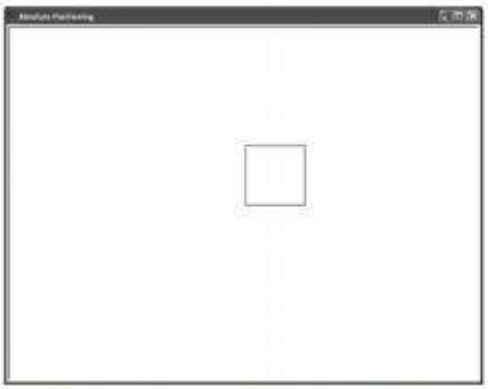
The default turtle shape is an arrowhead

4.2.3Fundamental Turtle Attributes and Behavior

Turtle objects have three fundamental attributes: position, heading (orientation), and pen attributes

Absolute Positioning

Method position returns a turtle's current position. For newly created turtles, this returns the tuple (0, 0). A turtle's position can be changed using absolute positioning by moving the turtle to a specific x,y coordinate location by use of method setposition. An example of this is given in Figure 6-20.

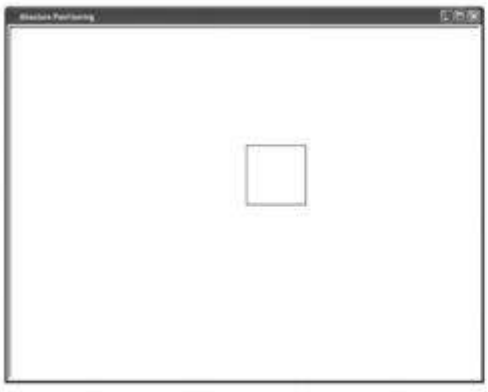


```
# set window title
window = turtle.Screen()
window.title('Absolute Positioning')

# get default turtle and hide
the_turtle = turtle.getturtle()
the_turtle.hideturtle()

# create square (absolute positioning)
the_turtle.setposition(100, 0)
the_turtle.setposition(100, 100)
the_turtle.setposition(0, 100)
the_turtle.setposition(0, 0)

# exit on close window
turtle.exitonclick()
```

FIGURE 6-20 Absolute Positioning of Turtle

Turtle Heading and Relative Positioning

A turtle's position can also be changed through relative positioning . In this case, the location that a turtle moves to is determined by its second fundamental attribute, its heading. A newly created turtle's heading is to the right, at 0 degrees. A turtle with heading 90 degrees moves up; with a heading 180 degrees moves left; and with a heading 270 degrees moves down. A turtle's heading can be changed by turning the turtle a given number of degrees left, left(90), or right, right(90). The forward method moves a turtle in the direction that it is currently heading. An example of relative positioning is given in Figure 6-21.



```
# set window title
window = turtle.Screen()
window.title('Relative Positioning')

# get default turtle and hide
the_turtle = turtle.getturtle()
the_turtle.hideturtle()

# create box (relative positioning)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)
the_turtle.left(90)
the_turtle.forward(100)

# exit on close window
turtle.exitonclick()
```

FIGURE 6-21 Relative Positioning of Turtle

Pen Attributes

The pen attribute of a turtle object is related to its drawing capabilities. The most fundamental of these attributes is whether the pen is currently “up” or “down,” controlled by methods penup() and pendown(). When the pen attribute value is “up,” the turtle can be moved to another location without lines being drawn

The pen size of a turtle determines the width of the lines drawn when the pen attribute is “down.” The pensize method is used to control this: the_turtle.pensize(5). The width is given in pixels, and is limited only by the size of the turtle screen. Example pen sizes are depicted in Figure 6-23.

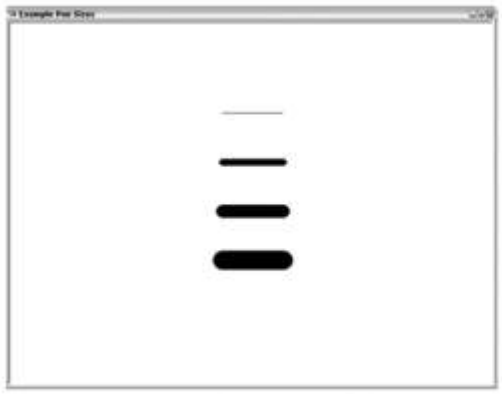


FIGURE 6-23 Example Turtle Pen Sizes

The pen color can also be selected by use of the pencolor method: the_turtle. pencolor('blue'). The name of any common color can be used, for example 'white', 'red', 'blue', 'green', 'yellow', 'gray', and 'black'. Colors can also be specified in RGB (red/green/blue) component values. These values can be specified in the range 0–255 if the color mode attribute of the turtle window is set as given below,

4.2.4Additional Turtle Attributes

Turtle Visibility

a turtle’s visibility can be controlled by use of methods hideturtle() and showturtle()

Turtle Size

The size of a turtle shape can be controlled with methods resizemode and turtlesize as shown in Figure 6-24.

```
# set to allow user to change turtle size
the_turtle.resizemode('user')

# set a new turtle size
the_turtle.turtlesize(3, 3)
```

FIGURE 6-24 Changing the Size of a Turtle

The first instruction sets the resize attribute of a turtle to ‘user’. This allows the user (programmer) to change the size of the turtle by use of method turtlesize.

There are two other values that method resizemode may be set to. An argument value of 'auto' causes the size of the turtle to change with changes in the pen size, whereas a value of 'noresize' causes the turtle shape to remain the same size.

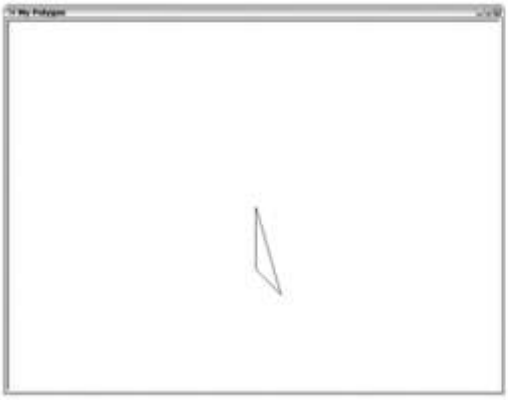
Turtle Shape

There are a number of ways that a turtle's shape (and fill color) may be defi ned to something other than the default shape (the arrowhead) and fill color (black). First, a turtle may be assigned one of the following provided shapes: 'arrow', 'turtle', 'circle', 'square', 'triangle', and 'classic' (the default arrowhead shape), as shown in Figure 6-25.

FIGURE 6-25 Available Turtle Shapes

The shape and fill colors are set by use of the shape and fillcolor methods,

New shapes may be created and registered with (added to) the turtle screen's shape dictionary . One way of creating a new is shape by providing a set of coordinates denoting a polygon, as shown in Figure 6-26.



```
turtle.setup(800, 600)
window = turtle.Screen()
window.title('My Polygon')
the_turtle = turtle.getturtle()

turtle.register_shape('mypolygon',
((0, 0), (100, 0), (140, 40)))

the_turtle.shape('mypolygon')
the_turtle.fillcolor('white')
```

FIGURE 6-26 Creating a New Polygon Turtle Shape

Turtle Speed

A turtle's speed can be set to a range of speed values from 0 to 10, with a "normal" speed being around 6. To set the speed of the turtle, the speed method is used, the_turtle.speed(6). The following speed values can be set using a descriptive rather than a numeric value,

4.3Modules

A module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

```
# A simple module, calc.py

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

```
def subtract(x, y):
    return (x-y)
```

4.3.1The import statement

We can use any Python source file as a module by executing an import statement in some other Python source file.

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches for importing a module. For example, to import the module calc.py, we need to put the following command at the top of the script :

```
# importing  module calc.py
import calc

print(add(10, 2))
```

4.3.2The from import Statement

Python's from statement lets you import specific attributes from a module. The from .. import .. has the following syntax :

```
# importing sqrt() and factorial from the
# module math
from math import sqrt, factorial

# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16))
print(factorial(6))
```

4.3.3The from import * Statement

The * symbol used with the from import the statement is used to import all the names from a module to a current namespace.

Syntax:

```
from module_name import *
```

4.3.4The dir() function

The dir() built-in function returns a sorted list of strings containing the names defined by a module. The list contains the names of all the modules, variables, and functions that are defined in a module.

4.4Files

Python has several functions for creating, reading, updating, and deleting files.

4.4.1File Handling

The key function for working with files in Python is the open() function.

The open() function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

4.4.2Open a File on the Server

Assume we have the following file, located in the same folder as Python:

demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in open() function.

The open() function returns a file object, which has a read() method for reading the content of the file:

Example

```
f = open("demofile.txt", "r")
print(f.read())
```

If the file is located in a different location, you will have to specify the file path, like this:

Example

Open a file on a different location:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

4.4.3Read Only Parts of the File

By default the read() method returns the whole text, but you can also specify how many characters you want to return:

Example

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines

You can return one line by using the readline() method:

Example

Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

By calling readline() two times, you can read the two first lines:

Example

Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

By looping through the lines of the file, you can read the whole file, line by line:

Example

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

4.4.4Close Files

It is a good practice to always close the file when you are done with it.

Example

Close the file when you are finish with it:

```
f                                                    = open("demofile.txt", "r")
print(f.readline())
f.close()
```

4.5String Processing

The operations performed on strings is called string processing .

Sequences Operations Applicable to Strings			
Length	`len(str)`	Membership	`'h' in s`
Select	`s[index_val]`	Concatenation	`s + w`
Slice	`s[start:end]`	Minimum Value	`min(s)`
Count	`s.count(char)`	Maximum Value	`max(s)`
Index	`s.index(char)`	Comparison	`s == w`

FIGURE 8-4 Sequence Operations on Strings

We give examples of each of these operations for s = 'Hello Goodbye!'.

```
>>> len(s)                  s.count('o')             >>> s + '!!'
14                          3                        'Hello Goodbye!!!'

>>> s[6]                    >>> s.index('b')         >>> min(s)
'G'                         10                       ' '

>>> s[6:10]                 >>> 'a' in s             >>> max(s)
'Good'                      False                    'y'
```

4.5.1String Methods

Checking the Contents of a String			
`str.isalpha()`	Returns True if `str` contains only letters.	`s = 'Hello'`	`s.isalpha()` → `True`
		`s = 'Hello!'`	`s.isalpha()` → `False`
`str.isdigit()`	Returns True if `str` contains only digits.	`s = '124'`	`s.isdigit()` → `True`
		`s = '124A'`	`s.isdigit()` → `False`
`str.islower()` `str.isupper()`	Returns True if `str` contains only lower (upper) case letters.	`s = 'hello'`	`s.islower()` → `True`
		`s = 'Hello'`	`s.isupper()` → `False`
`str.lower()` `str.upper()`	Return lower (upper) case version of `str`.	`s = 'Hello!'`	`s.lower()` → `'hello!'`
		`s = 'hello!'`	`s.upper()` → `'HELLO!'`
Searching the Contents of a String			
`str.find(w)`	Returns the index of the first occurrence of `w` in `str`. Returns -1 if not found.	`s = 'Hello!'`	`s.find('l')` → `2`
		`s = 'Goodbye'`	`s.find('l')` → `-1`

Replacing the Contents of a String			
`str.replace(w,t)`	Returns a copy of *str* with all occurrences of `w` replaced with t.	`s = 'Hello!'`	`s.replace('H', 'J')` → `'Jello'`
		`s = 'Hello'`	`s.replace('ll', 'r')` → `'Hero'`
Removing the Contents of a String			
`str.strip(w)`	Returns a copy of *str* with all leading and trailing characters that appear in w removed.	`s = ' Hello! '` `s = 'Hello\n'`	`s.strip(' !')` → `'Hello'` `s.strip('\n')` → `'Hello'`
Splitting a String			
`str.split(w)`	Returns a list containing all strings in *str* delimited by w.	`s = 'Lu, Chao'`	`s.split(',')` → `['Lu', 'Chao']`

4.6Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>>
```

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions. Standard exception names are built-in identifiers (not reserved keywords).

4.6.1 Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the

user to interrupt the program ; note that a user-generated interruption is signalled by raising the KeyboardInterrupt exception.

```
>>>
```

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

The try statement works as follows.

- First, the *try clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

The try block lets you test a block of code for errors.

The except block lets you handle the error.

The finally block lets you execute code, regardless of the result of the try- and except blocks.

4.6.2Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the try statement:

Example

The try block will generate an exception, because x is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error:

Example

This statement will raise an error, because x is not defined:

```
print(x)
```

4.6.3Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a NameError and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Else

You can use the else keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the try block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

Example

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

This can be useful to close objects and clean up resources:

Example

Try to open and write to a file that is not writable:

```
try:
  f = open("demofile.txt")
  f.write("example prg")
except:
  print("Something went wrong when writing to the file")
finally:
  f.close()
```

The program can continue, without leaving the file object open.

4.6.4Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the raise keyword.

Example

Raise an error and stop the program if x is lower than 0:

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

The raise keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

Example

Raise a TypeError if x is not an integer:

```
x = "hello"
```

```
if not type(x)                                                    is int:
  raise TypeError("Only integers are allowed")
```

UNIT – V

5. DICTIONARIES AND SETS

5.1 Dictionary

A dictionary is a mutable, associative data structure of variable length denoted by the use of curly braces. Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds key : value pair. Key value is provided in the dictionary to make it more optimized. Dictionary keys are case sensitive, same name but different cases of Key will be treated distinctly. The dictionary operations are,

S.No	Operation	Description
1	Dict( )	Creates a new, empty dictionary.
2	Dict(S)	Creates a new dictionary with key values and their associated values from sequence S.
3	Len(d)	Length (number of key / value pairs) of dictionary d.
4	d[key]=value	Sets the associated value for key to value, used to either add a new key / value pair, or replace the value of an existing key / value pair.
5	Del d[key]	Remove key and associated value from dictionary d.
6	Key in d	True if key value exists in dictionary d, otherwise returns false.

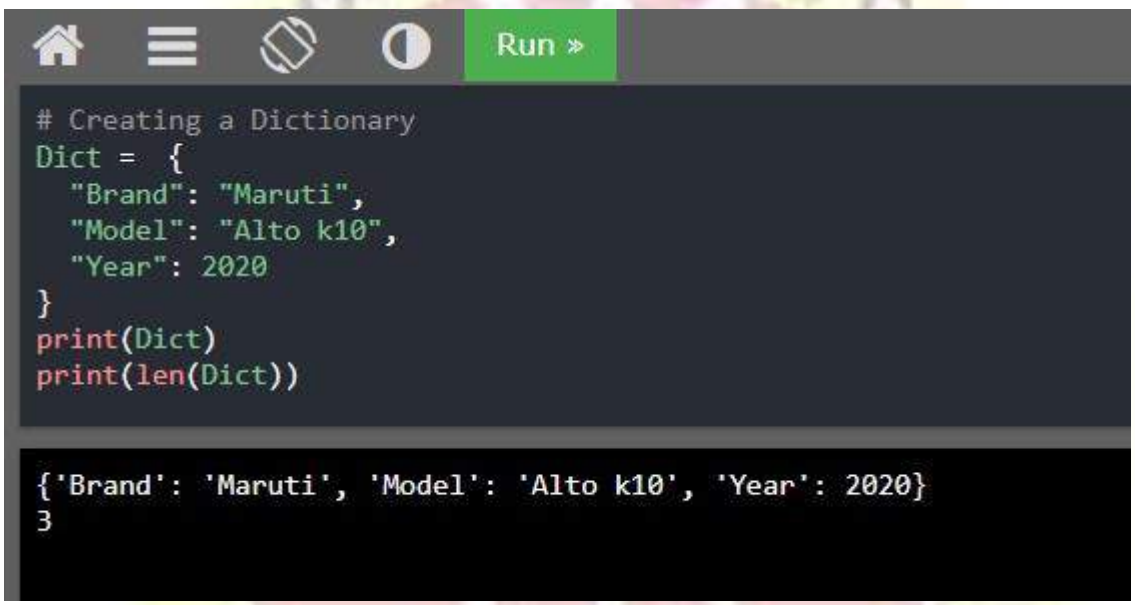
Dictionary can be created by placing sequence of elements within curly **{}** braces, separated by 'comma'. Dictionary holds a pair of values, one being the Key and the other corresponding pair element being its Key: value. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be *immutable.*

a) Create a Dictionary

Example #1

```
# Creating a Dictionary
Dict =  {
  "Brand": "Maruti",
  "Model": "Alto k10",
  "Year": 2020
}
print(Dict)
print(len(Dict))
```

Output:

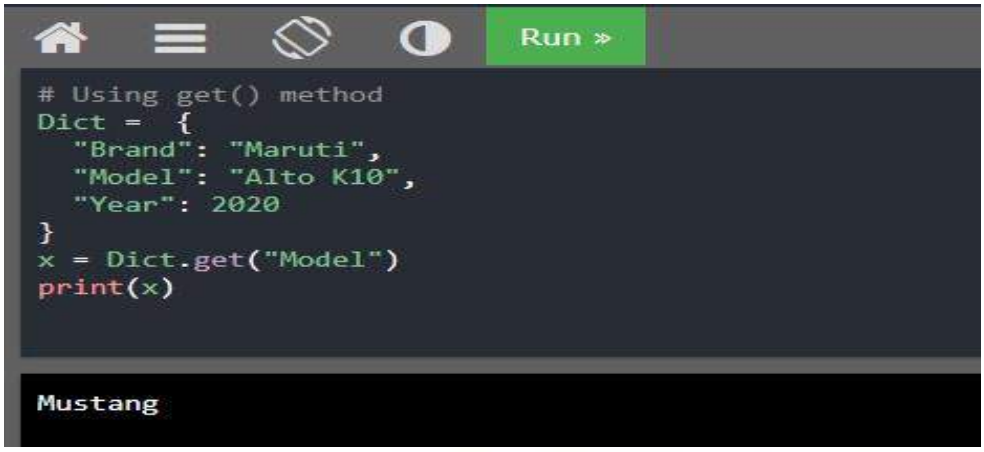


Example #2

The get( ) method is used to get the value form the dictionary.

```
# Using get() method
Dict =  {
  "Brand": "Maruti",
  "Model": "Alto K10",
  "Year": 2020
}
x = Dict.get("Model")
print(x)
```

Output:



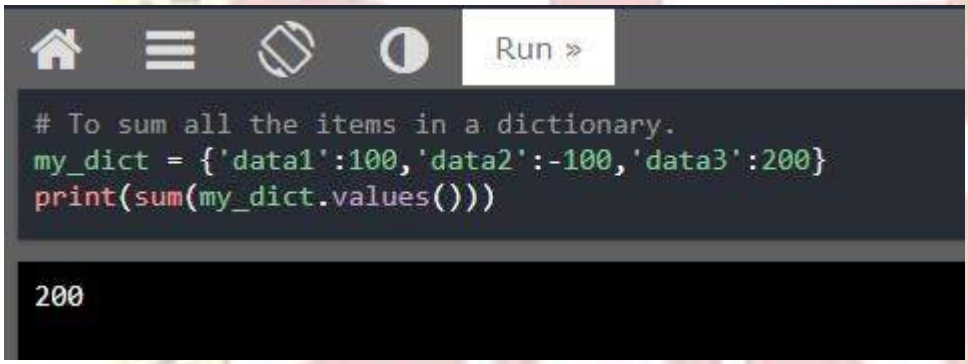
Example #3

To sum all the items in a dictionary.

my_dict = {'data1':100,'data2':-100,'data3':200}

print(sum(my_dict.values()))

Output:



b) Changing the Dictionary Value

We can change the value of a specific item by referring to its key name.

Example

Changing the Dictionary Value

Dict ={

"Brand": "Maruti",

"Model": "Alto k10",

"Year": 2020

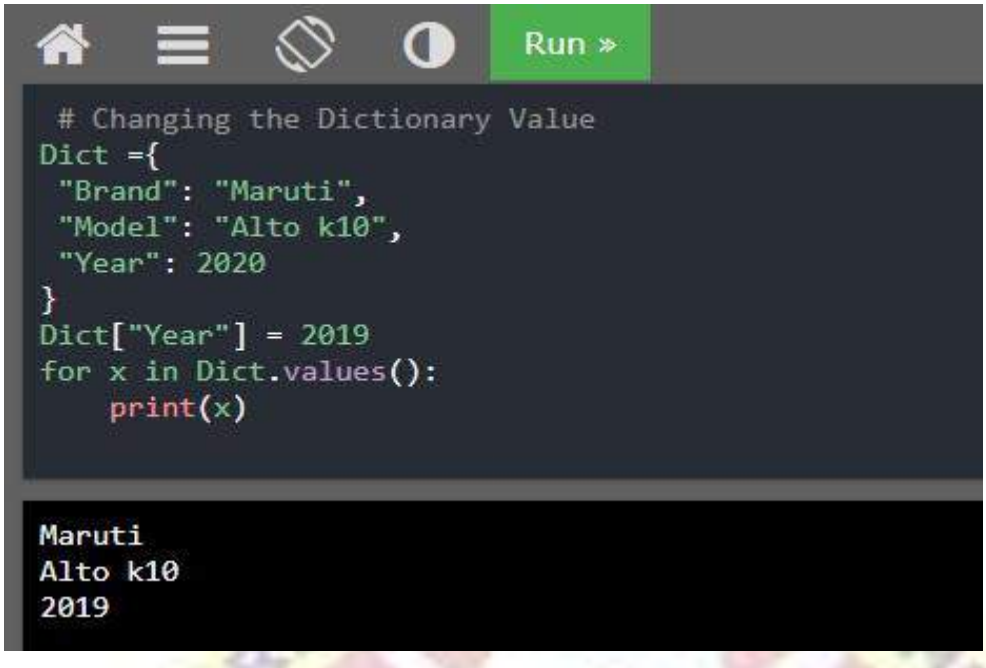
}

Dict["Year"] = 2019

for x in Dict.values():

print(x)

Output:



c) Add and Remove an Element

Adding an item to the dictionary is done by using a new index key and assigning a value to it.

The pop( ) method removes the item with the specified key name.

Example

```
# Adding and Removing an element in Dictionary
Dict =  {
  "Brand": "Maruti",
  "Model": "Alto k10",
  "Year": 2020
}
Dict["Color"] = "Red"
Dict.pop("Model")
print(Dict)
```

Output:

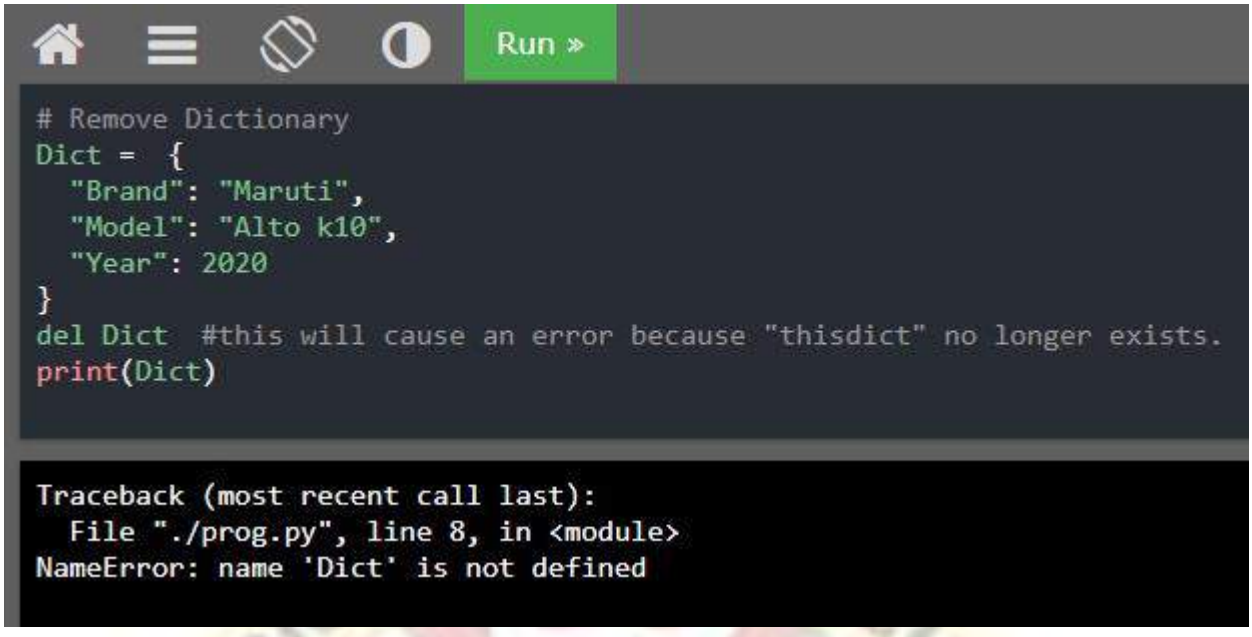
d) Remove Dictionary

The del( ) keyword also delete the dictionary completely.

Example

```
# Remove Dictionary
Dict =  {
  "Brand": "Maruti",
  "Model": "Alto k10",
  "Year": 2020
}
del Dict  #this will cause an error because "thisdict" no longer exists.
print(Dict)
```

Output:



5.2 Set data type

A set is a mutable data structure with non duplicate, unordered values, providing the usual set operations. A Set is an unordered collection data type that is iterable, mutable and has no duplicate elements. Python's set class represents the mathematical notion of a set. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table.

5.2.1 Set Operators

We assume, Set A = {1,2,3} Set B = {3,4,5,6}

Set operator			
Membership	1 in A	True	True if 1 is a member of set
Add	A.add(4)	{1,2,3,4}	Adds new member to set
Remove	A.remove(2)	{1,3}	Removes member form set
Union	A | B	{1,2,3,4,5,6}	Set of elements in either set A or set B
intersection	A & B	{3}	Set of elements in both set A and set B
difference	A-B	{1,2}	Set of elements in set A, but not set B
symmetric difference	A ^ B	{1,2,4,5,6}	Set of elements in set A or Set B, but not both
size	len(A)	3	Number of elements in set.

a) Create a set

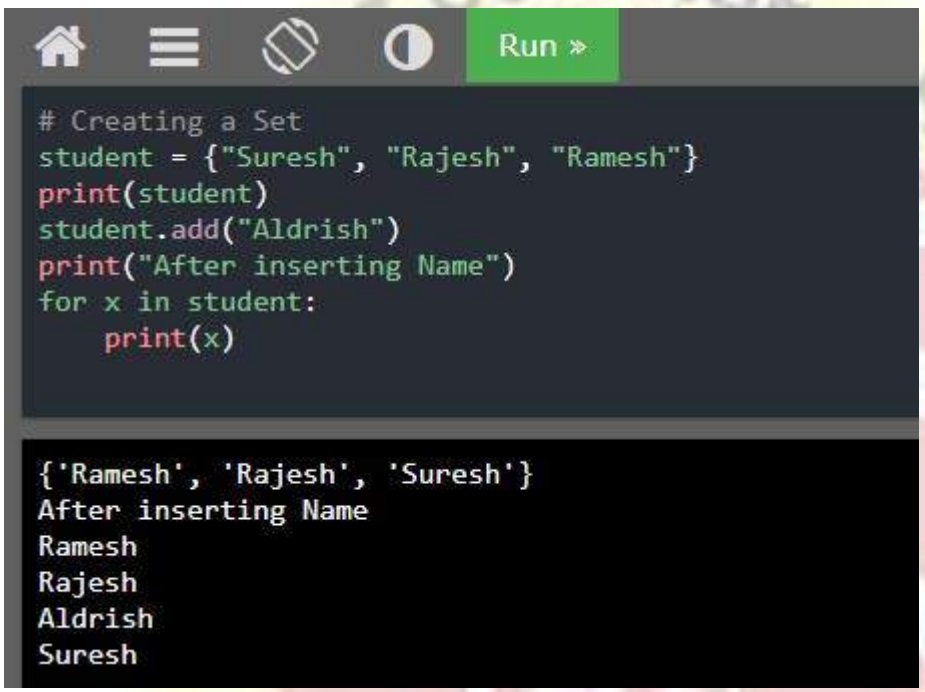
Python sets are written with curly brackets. Insertion in set is done through set.add() function,

where an appropriate record value is created to store in the hash table.

Example

```
# Creating a Set
student = {"Suresh", "Rajesh", "Ramesh"}
print(student)
student.add("Aldrish")
print("After inserting Name")
for x in student:
        print(x)
```

Output:



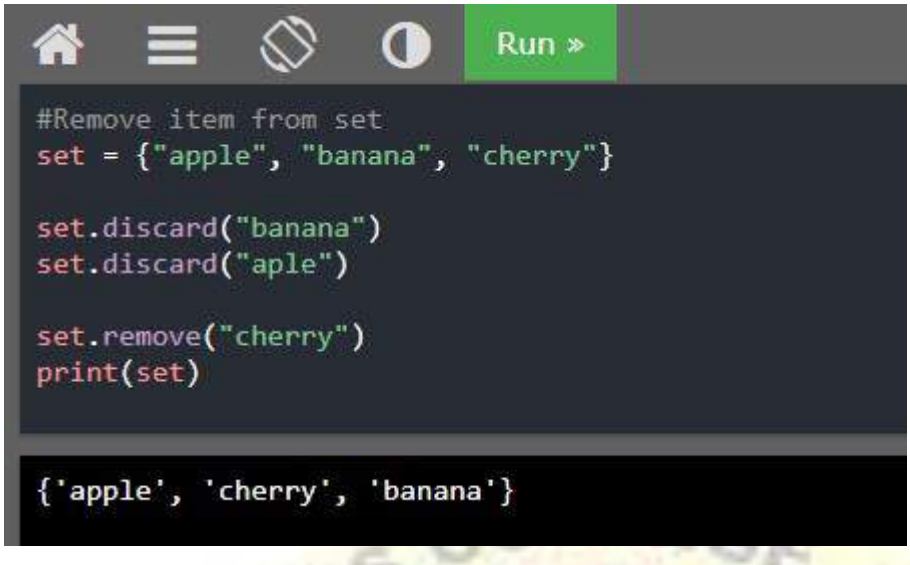
b) **Remove an element**

The remove() and discard() method is used to remove the item from the set. If the item to remove does not exist, discard() will not raise an error. If the item to remove does not exist, remove( ) will raise an error.

Example

```
#Remove item from set
set = {"apple", "banana", "cherry"}
set.discard("banana")
set.discard("aple")
set.remove("cherry")
print(set)
```

Output:



c) Frozen Set

A frozenset is a immutable set type.

Example

#Example of Frozen Set

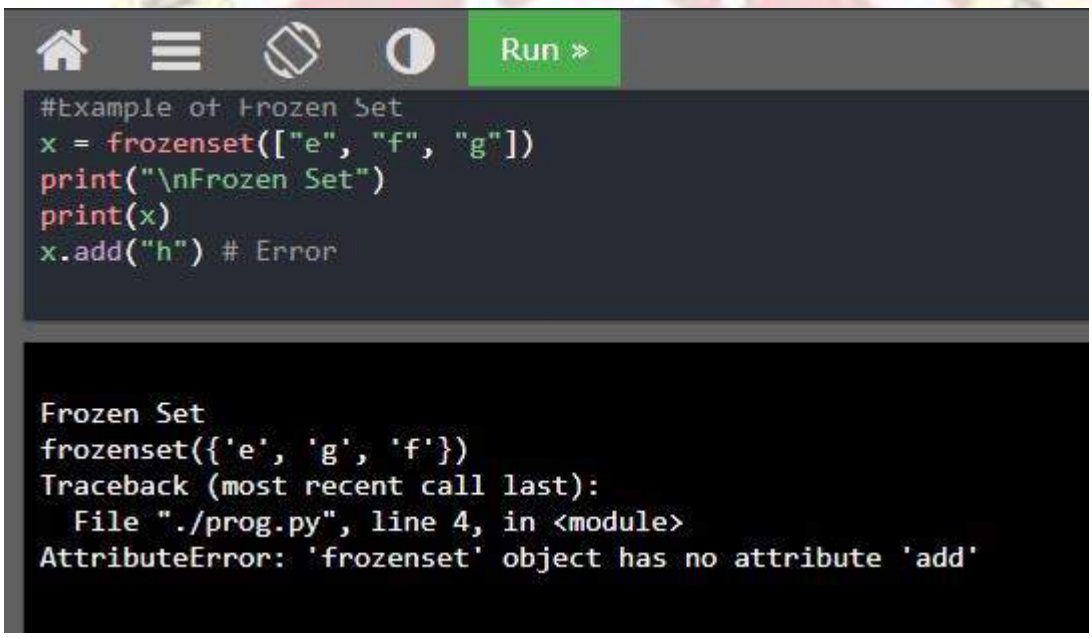
x = frozenset(["e", "f", "g"])

print("\nFrozen Set")

print(x)

x.add("h") # Error

Output:



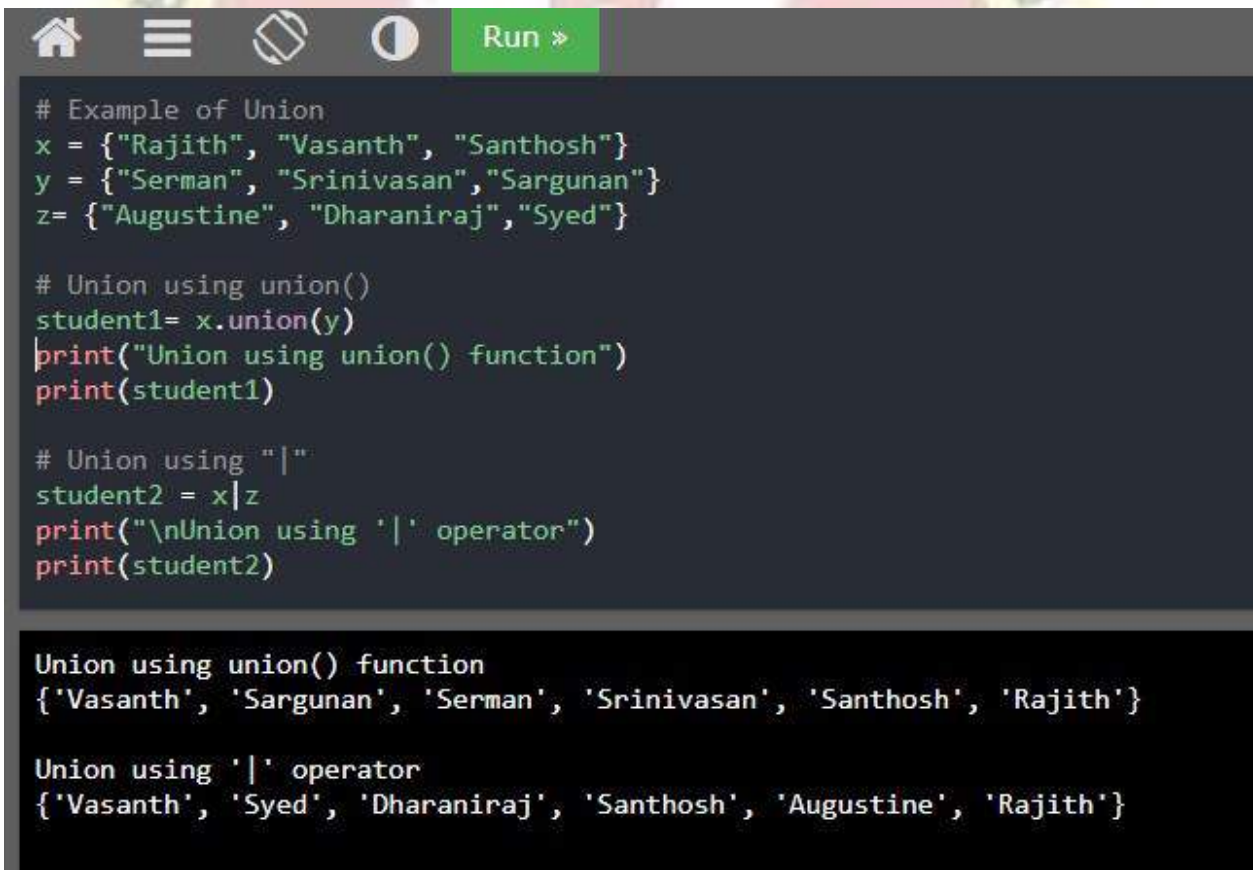
d) Union

Two sets can be merged using union() function or | operator.

Example

```
# Example of Union
x = {"Rajith", "Vasanth", "Santhosh"}
y = {"Serman", "Srinivasan","Sargunan"}
z= {"Augustine", "Dharaniraj","Syed"}
 # Union using union()
student1= x.union(y)
print("Union using union() function")
print(student1)
 # Union using "|"
student2 = x|z
print("\nUnion using '|' operator")
print(student2)
```

Output:



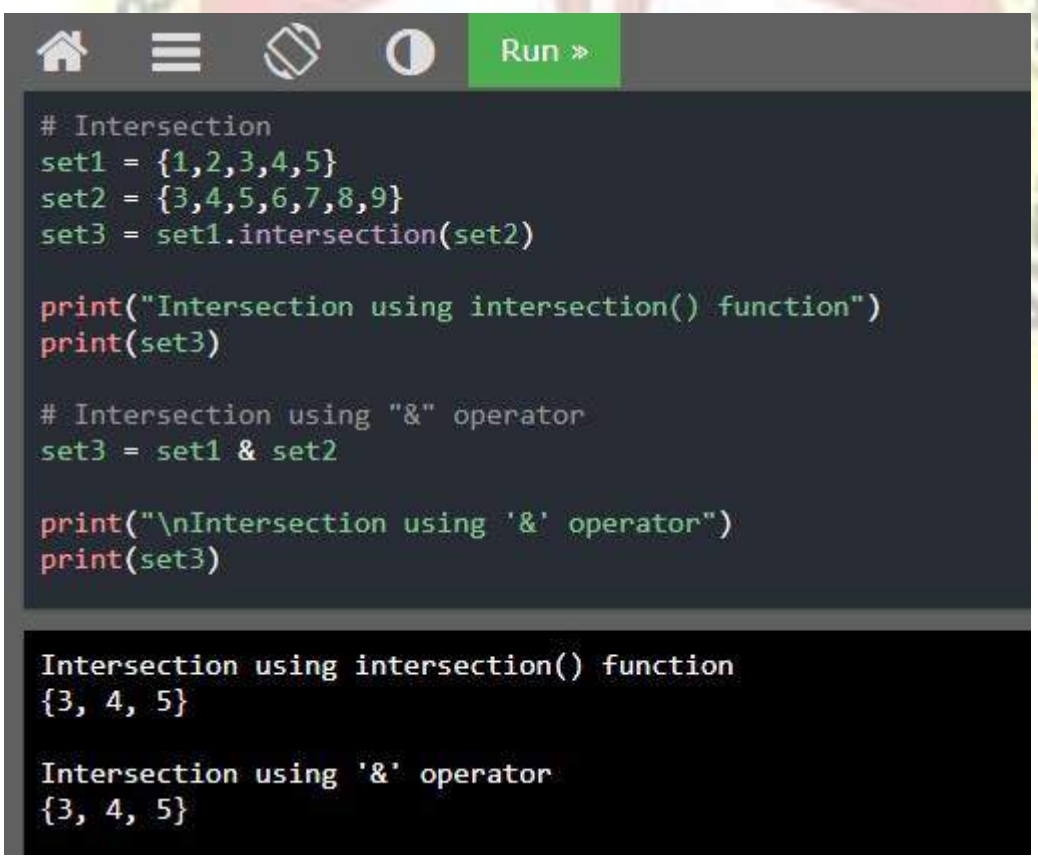
e) Intersection

This can be done through intersection() or & operator. Common Elements are selected.

Example

```
# Intersection
set1 = {1,2,3,4,5}
set2 = {3,4,5,6,7,8,9}
set3 = set1.intersection(set2)
print("Intersection using intersection() function")
print(set3)
# Intersection using "&" operator
set3 = set1 & set2
print("\nIntersection using '&' operator")
print(set3)
```

Output:



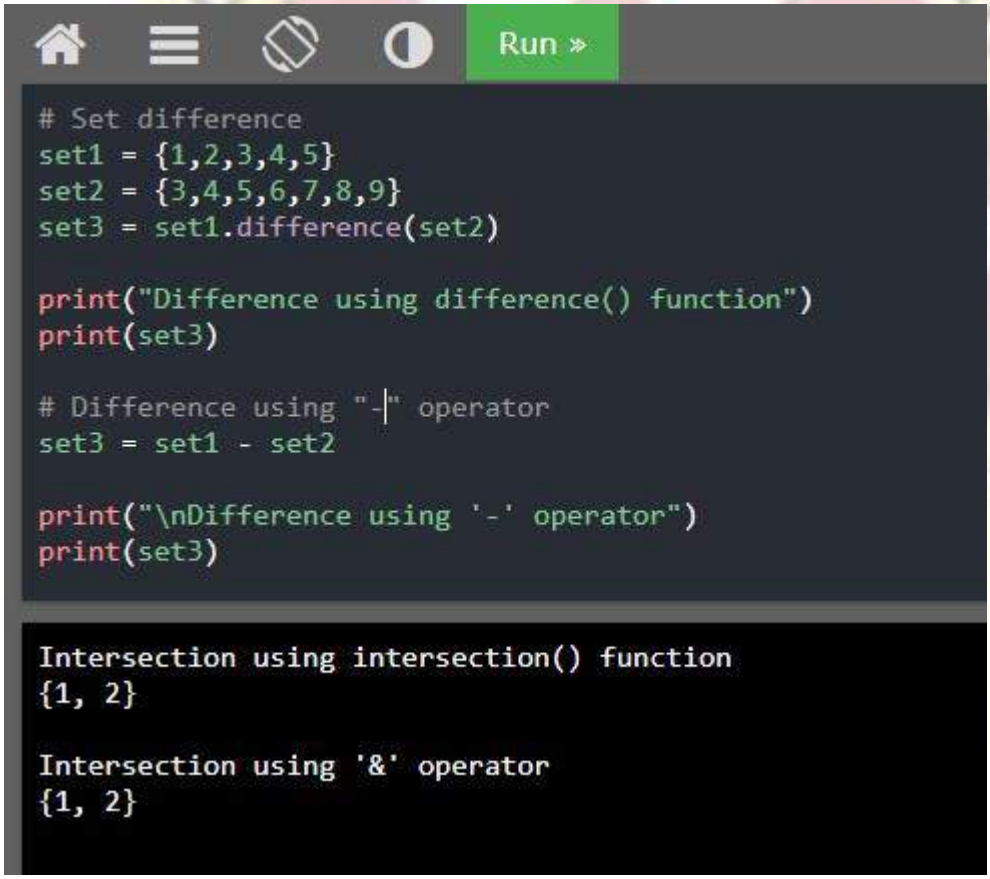
f) Difference

To find difference in between sets. Similar to find difference in linked list. This is done through difference() or – operator.

Example

```
# Set difference
set1 = {1,2,3,4,5}
set2 = {3,4,5,6,7,8,9}
set3 = set1.difference(set2)

print("Difference using difference() function")
print(set3)
# Difference using "-" operator
set3 = set1 - set2
print("\nDifference using '-' operator")
print(set3)
```

Output:



5.3 Object Oriented Programming using Python

Python is an object oriented programming language. An Object Oriented Programming Language is a high-level programming language based on the object-oriented model. OOP provides a clear modular structure for programs. It is easy to maintain and modify existing code as new objects can be created with small differences to existing ones.

An object is an instance of a class. It has states and behaviours. For example, a dog is an object. States of dog is colour, name and breed as well as behaviours of dog is wagging the tail, barking and eating. A class is a template / blueprint that define the variables (states) and methods (behaviours). A class specifies the set of instance variables and methods that are "bundled together" for defining a type of object.

Example

In the above program, Test is a class and obj is an object.

Three fundamental features supporting the design of object-oriented programs are referred to as Encapsulation, Inheritance and polymorphism. Message passing occurs when a method of one object calls a method of another.

5.3.1 Encapsulation

Encapsulation is a means of bundling together instance variables and methods to form a given type (class). Selected members of a class can be made inaccessible ("hidden") from its clients, referred to as information hiding. Information hiding is a form of abstraction. This is an important capability that object-oriented programming languages provide.

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variable**.** A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

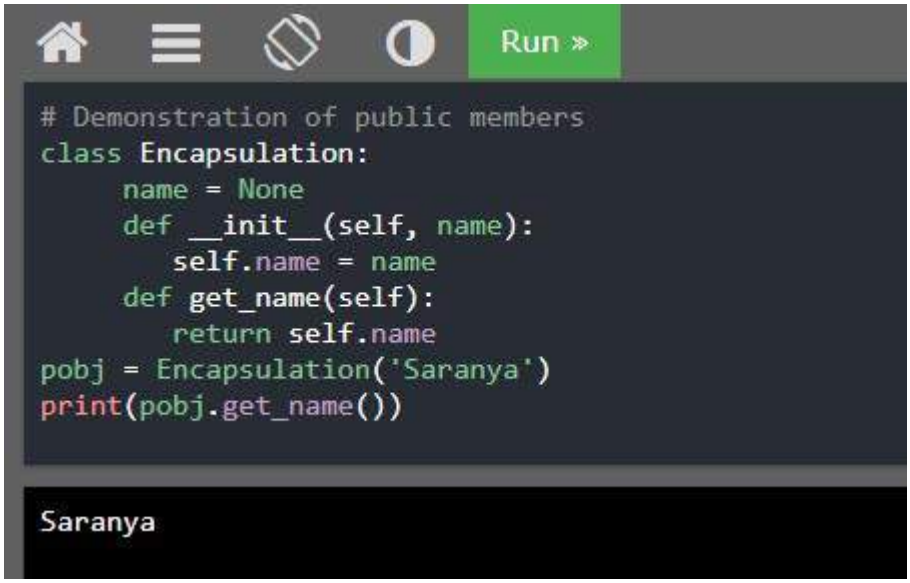
a) Public members

Public members (generally methods declared in a class) are accessible from outside the class. The object of the same class is required to invoke a public method. This arrangement of private instance variables and public methods ensures the principle of data encapsulation.

Example

```
# Demonstration of public members
class Encapsulation:
    name = None
    def __init__(self, name):
        self.name = name
    def get_name(self):
      return self.name
pobj = Encapsulation('Saranya')
print(pobj.get_name())
```

Output:



b) Protected members

Protected members are those members of the class which cannot be accessed outside the class but can be accessed from within the class and its subclasses. To accomplish this in Python, just follow the convention by prefixing the name of the member by a single underscore "_".

The __init__ method is a constructor and runs as soon as an object of a class is instantiated.

Example

```
class Encapsulation:
    _name = None
    def __init__(self, name):
        self._name = name
    def get_name(self):
        return self._name
pobj = Encapsulation('Saranya')
print(pobj.get_name())
```

c) Private members

Private members are similar to protected members; the difference is that the class members declared private should neither be accessed outside the class nor by any base class. In Python, there is no existence of Private instance variables that cannot be accessed except inside a class. However, to define a private member prefix the member name with double underscores "__".

Example

```
# Demonstration of private members
class Encapsulation:
   __name = None
    def __init__(self, name):
        self.__name = name
    def get_name(self):
        return self.__name
pobj = Encapsulation('Saranya')
print(pobj.get_name())
```

Output:

5.3.2 Inheritance

Inheritance, in object-oriented programming, is the ability of a class to inherit members of another class as part of its own definition. The inheriting class is called a subclass (also "derived class" or "child class"), and the class inherited from is called the super class (also "base class" or "parent class").

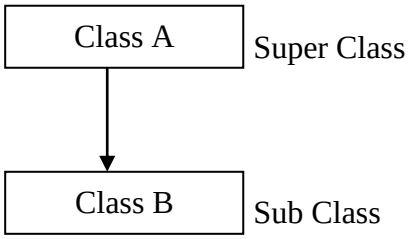
The benefits of Inheritance are,

- It represents real-world relationships well.
- It provides reusability of a code.

Types of Inheritance

i) Single Inheritance

A sub class inherits from only one super class; it is called as single inheritance.



Here, class A is called super (Parent or Base) class and B is called sub (Child or Derived) class. Class B inherits all variables and methods from class A.

Example

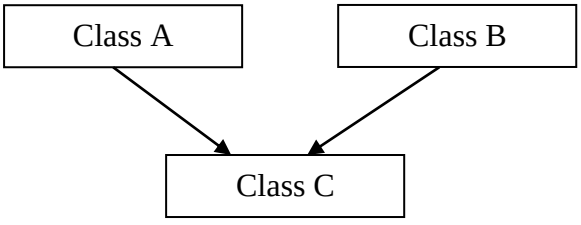
```
# Demonstration of Single Inheritance

class Animal:
```

```
    def speak(self):
        print("Animals are not Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("Dog is barking")
d = Dog()
d.bark()
d.speak()
```

Output:

ii) Multiple Inheritance

When a sub class inherits from multiple super (parent) classes, it is called as multiple inheritances.



In the above diagram, Class A and Class B are called Super class and class C is called as sub class. Class C inherits properties from Class A and Class B.

Example

```
# Demonstration of Multiple Inheritances
class Calculation1:
        def Summation(self,a,b):
          return a+b;
class Calculation2:
        def Multiplication(self,a,b):
           return a*b;
class Derived(Calculation1,Calculation2):
          def Divide(self,a,b):
             return a/b;
d = Derived()
print(d.Summation(20,10))
print(d.Multiplication(20,10))
print(d.Divide(20,10))
```

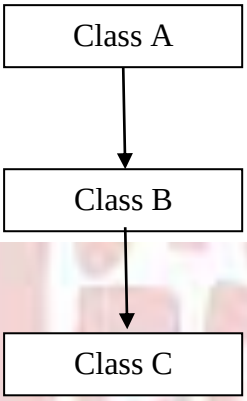
Output:

```
# Demonstration of Multiple Inheritances
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(20,10))
print(d.Multiplication(20,10))
print(d.Divide(20,10))
```

```
30
200
2.0
```

iii) Multilevel Inheritance

Features of the base class and the sub class are inherited into the new derived class.



Here, Class B inherits the properties of Class A and Class C inherits the properties of Class A and Class B.

Example

```
# Demonstration of Multi Level Inheritance
class Animal:
    def speak(self):
        print("Animals are not Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("Dog is barking")
#The child class Dogchild inherits another child class Dog
```

```
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

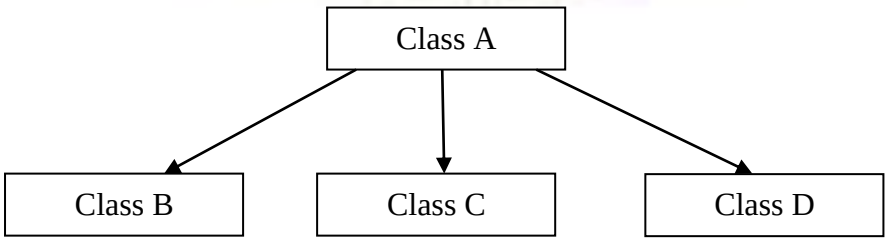
Output:

```
# Demonstration of Multi Level Inheritance
class Animal:
    def speak(self):
        print("Animals are not Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("Dog is barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

```
Dog is barking
Animals are not Speaking
Eating bread...
```

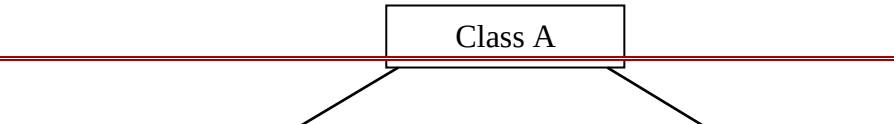
iv) Hierarchical Inheritance

More than one sub classes are created from a single class is called Hierarchical Inheritance.



v) Hybrid Inheritance

The combination of any other two inheritances is called Hybrid inheritance.



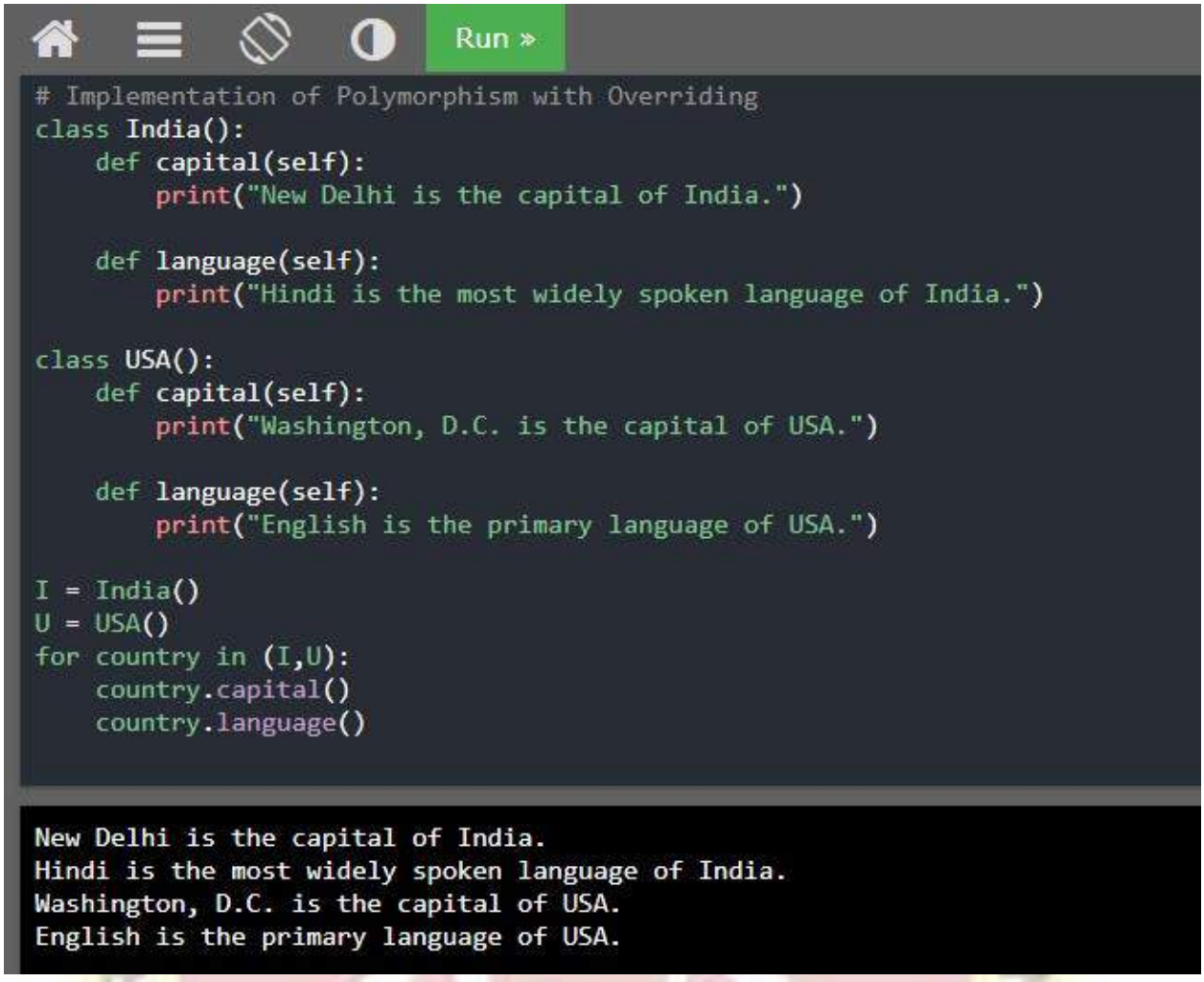
5.3.3 Polymorphism

Polymorphism is a powerful feature of object-oriented programming languages. It allows for the implementation of elegant software that is well designed and easily modified. The word polymorphism derives from Greek meaning something that takes “many forms”.

Example

```
# Implementation of Polymorphism with Overriding
class India():
    def capital(self):
        print("New Delhi is the capital of India.")
    def language(self):
        print("Hindi is the most widely spoken language of India.")
class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")
    def language(self):
        print("English is the primary language of USA.")
I = India()
U = USA()
for country in (I,U):
    country.capital()
    country.language()
```

Output:



```
# Implementation of Polymorphism with Overriding
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

I = India()
U = USA()
for country in (I,U):
    country.capital()
    country.language()
```

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
```

In the above program, two classes are having two methods come in the same name. But, it prints different statements. This concept is called method overriding.

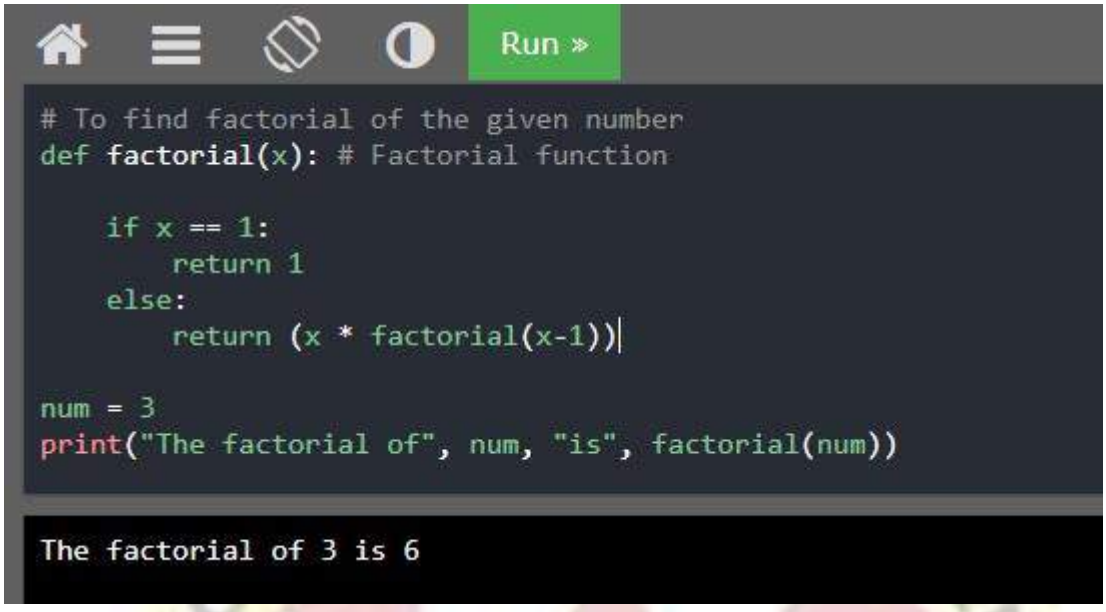
5.4 Recursion

A recursive function is a function that conditionally calls itself. This means that the function will continue to call itself and repeat its behaviour until some condition is met to return a result. Usually, it is returning the return value of this function call.

Example #1

```
# To find factorial of the given number
def factorial(x): # Factorial function

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

Output:

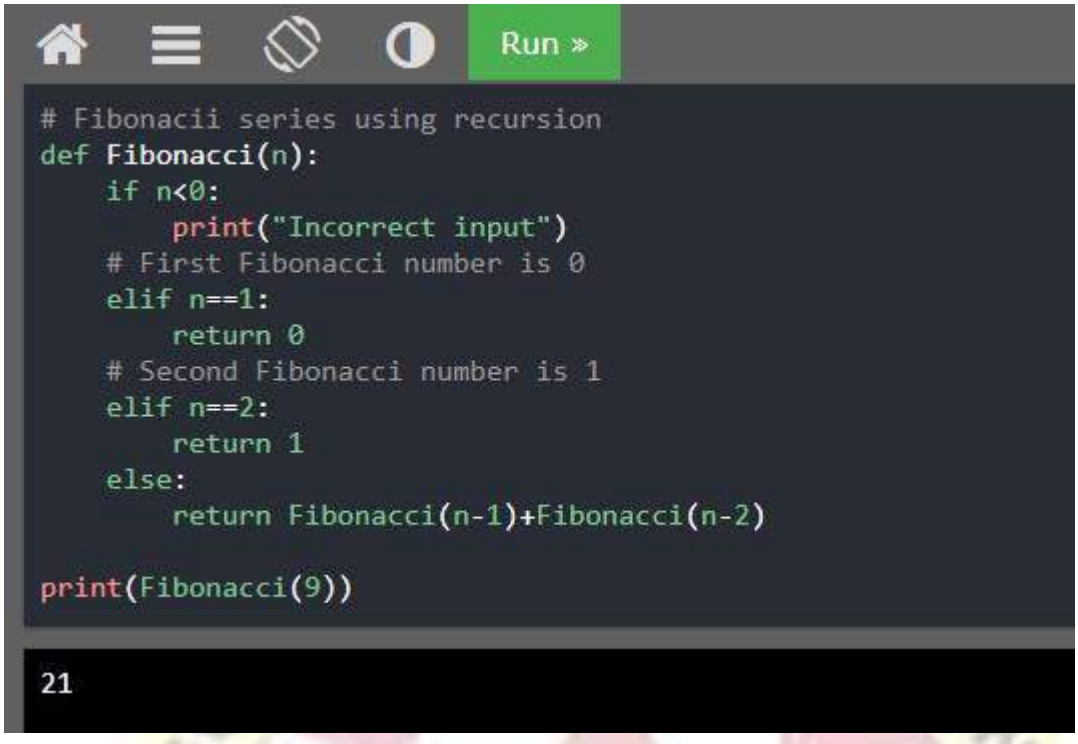


In the above example, factorial() is a recursive function as it calls itself.

Example #2

```
# Fibonacii series using recursion
def Fibonacci(n):
    if n<0:
        print("Incorrect input")
    # First Fibonacci number is 0
    elif n==1:
        return 0
    # Second Fibonacci number is 1
    elif n==2:
        return 1
    else:
        return Fibonacci(n-1)+Fibonacci(n-2)
print(Fibonacci(9))
```

Output:



Example #3

```
# Recursive Python function to solve tower of hanoi
def TowerOfHanoi(n , from_rod, to_rod, aux_rod):
    if n == 1:
        print ("Move disk 1 from rod",from_rod,"to rod",to_rod)
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print ("Move disk",n,"from rod",from_rod,"to rod",to_rod)
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)
n = 4
TowerOfHanoi(n, 'A', 'C', 'B')
```

Output:

```
# Recursive Python function to solve tower of hanoi
def TowerOfHanoi(n , from_rod, to_rod, aux_rod):
    if n == 1:
        print ("Move disk 1 from rod",from_rod,"to rod",to_rod)
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print ("Move disk",n,"from rod",from_rod,"to rod",to_rod)
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)
n = 4
TowerOfHanoi(n, 'A', 'C', 'B')
```

```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

Points to remember

- A dictionary is a mutable, associative data structure of variable length denoted by the use of curly braces.
- The del( ) keyword also delete the dictionary completely.
- A set is a mutable data structure with non duplicate, unordered values, providing the usual set operations.
- A frozenset is a immutable set type.
- An Object Oriented Programming Language is a high-level programming language based on the object-oriented model.
- A class specifies the set of instance variables and methods that are “bundled together” for defining a type of object.
- An object is an instance of a class. It has states and behaviours.
- Encapsulation is a means of bundling together instance variables and methods to form a given type (class).

- The __init__ method is a constructor and runs as soon as an object of a class is instantiated.
- A private member prefix the member name with double underscores "__" and protected member prefix comes with single underscore.
- Inheritance, in object-oriented programming, is the ability of a class to inherit members of another class as part of its own definition.
- A sub class inherits from only one super class; it is called as single inheritance.
- A recursive function is a function that conditionally calls itself.

Two mark Questions

1. What is the use of Dictionary?
2. What is the use of Set data type?
3. What is Encapsulation?
4. What is the use of Object Oriented Program?
5. What is a class?
6. Define "Object".
7. What is Polymorphism?
8. What do you mean by List?
9. Define "Recursion".
10. What is the different set and Frozen Set.

Five mark Questions

11. Explain the properties of Dictionary keys with examples
12. Explain about the Set operators.
13. Investigate on mutability and immutability in python.
14. Explain recursive function. How do recursive functions work?
15. Explain about various operations in dictionaries.

Ten Mark Questions

16.Explain in detail about
   (i) Creating a dictionary
   (ii) Accessing values in a dictionary
   (iii) Updating dictionary
   (i) Deleting elements from dictionary
17.Discuss on various operations of Set.
18. Explain about the concepts of Object Oriented Programming with examples.

*****