UNIT -1

(PART 1)

INTRODUCTION TO PYTHON

SYLLABUS :

Introduction: Introduction to Python, Program Development Cycle, Input, Processing, and Output, Displaying Output with the Print Function, Comments, Variables, Reading Input from the Keyboard, Performing Calculations, Operators. Type conversions, Expressions, More about Data Output.

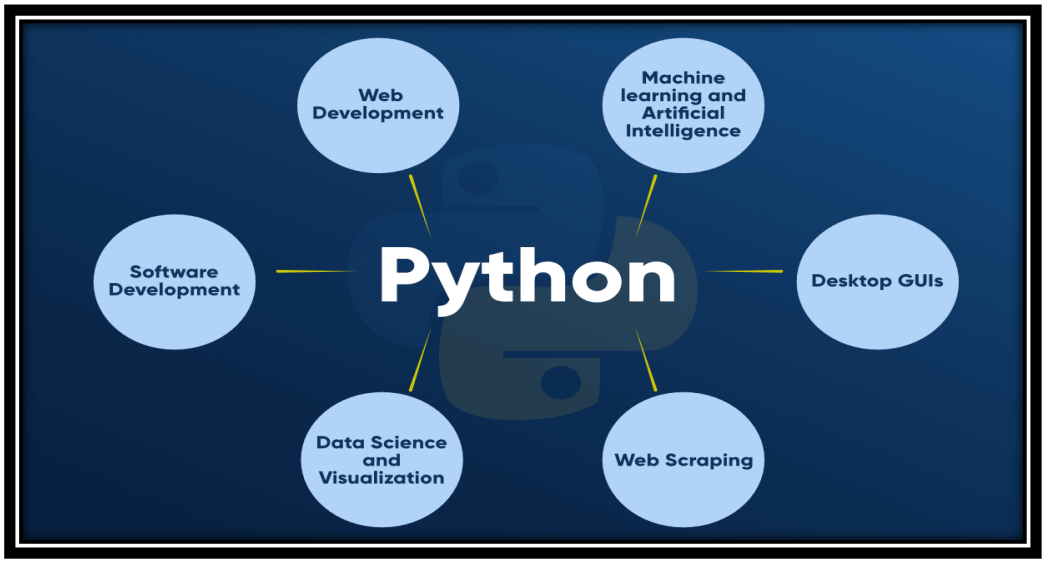
Data Types, and Expression: Strings Assignment, and Comment, Numeric Data Types and Character Sets, Using functions and Modules.

Decision Structures and Boolean Logic: if, if-else, if-elif-else Statements, Nested Decision Structures, Comparing Strings, Logical Operators, Boolean Variables. Repetition Structures: Introduction, while loop, for loop, Calculating a Running Total, Input Validation Loops, Nested Loops.

Introduction to Python

Python was developed by Guido Van Rossum in 1991 at National Research Institute for Mathematics and Computer Science in the Netherlands.

Application Areas of Python:



Features of Python :

- **Simple**

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

- Python is a simple programming language. Reading a program written in python feels almost like reading English.
- Programmers can concentrate on the solution rather than the language.

- **Easy to learn**
  - A python program is clearly defined and easily readable. The structure of the program is very simple.
- **Free and open source**
  - Python is an example of open source software.
  - Anyone can freely distribute it, read the source code, edit it, and even use the code to write new programs
- **High-level Language**
  - When writing programs in python, the programmers don't have to worry about the low-level details like managing memory used by the program.
- **Portable**
  - Python is a portable language. The programs work on any of the operating systems like Linux, Windows, Macintosh, Solaris, Palm OS..etc.
- **Interpreted**
  - Python is processed at runtime by the interpreter. No need to compile the program before executing it.
  - Python converts the source code to intermediate form called byte code, which is translated into the native language of your computer so that it can be executed.
- **Object Oriented**
  - Python supports procedure-oriented programming as well as object-oriented programming.
  - In procedure-oriented language, the program is built around procedures or functions which are nothing but reusable pieces of programs.

- In object oriented languages, the program is built around objects which combine data and functionality.

➢ **Extensible**

- You can write some of your python code in other languages . It can be extended to other languages.

➢ **Extensive Libraries**

- Python has a huge library functions are compatible on UNIX, Windows, Macintosh and allow programmers to perform wide range of applications varying from text processing, maintaining databases, to GUI programming.

Program Development Cycle:

- Computer scientists refer to the process of planning and organizing a program as **software development.**
- One version of software development is known as the **waterfall model.**

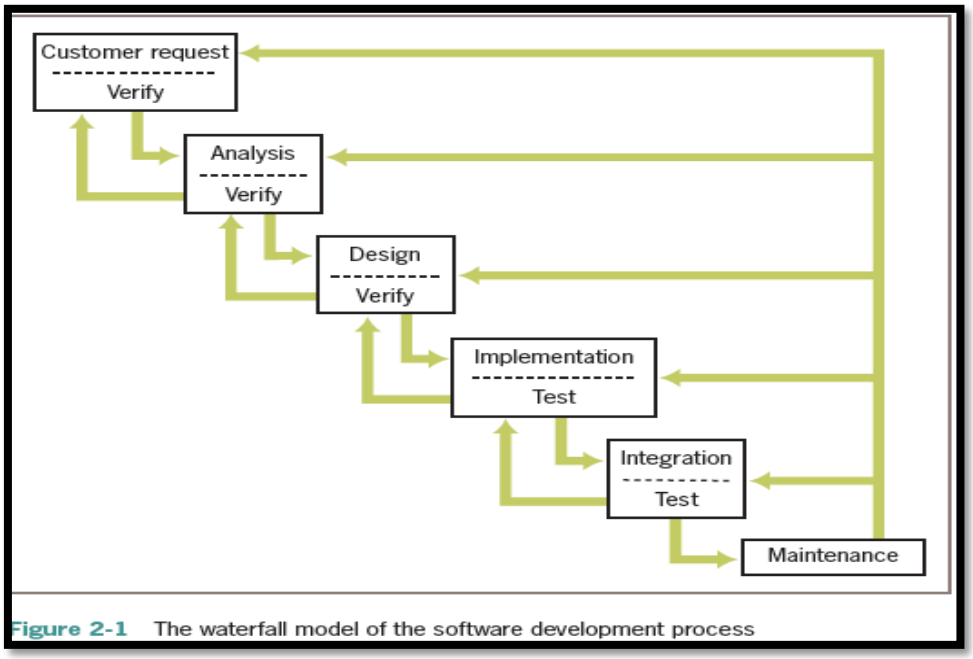


Figure 2-1 The waterfall model of the software development process

- The waterfall model consists of several phases:

1. **Customer request—**

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

In this phase, the programmers receive a broad statement of a problem that is potentially amendable to a computerized solution. This step is also called the user requirements phase.

2. **Analysis—**

   The programmers determine what the program will do. This is sometimes viewed as a process of clarifying the specifications for the problem.

3. **Design—**

   The programmers determine how the program will do its task.

4. **Implementation**—

   The programmers write the program. This step is also called the coding phase.

5. **Integratio**n—

   Large programs have many parts. In the integration phase, these parts are brought together into a smoothly functioning whole, usually not an easy task.

6. **Maintenance**—

   Programs usually have a long life; a life span of 5 to 15 years is common for software.

   During this time, requirements change, errors are detected, and minor or major modifications are made.

- There is more to software development than writing code.
- If you want to reduce the overall cost of software development, write programs that are easy to maintain.
  - This requires thorough analysis, careful design, and a good coding style.
- Keep in mind that mistakes found early are much less expensive to correct than those found late.
- Following figure illustrates some relative costs of repairing mistakes when found in different phases.

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

– These are not just financial costs but also costs in time and effort.

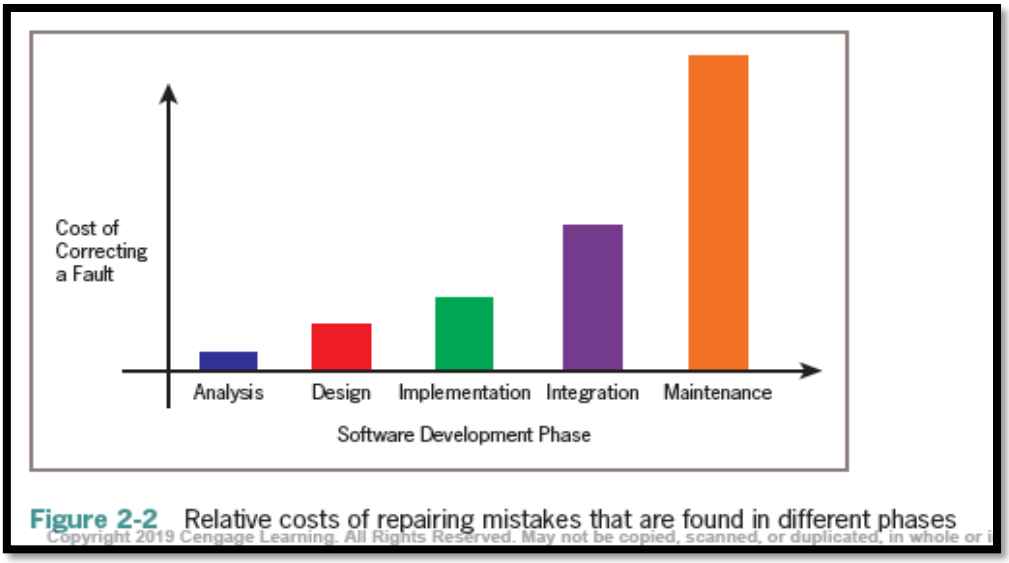


Figure 2-2 Relative costs of repairing mistakes that are found in different phases

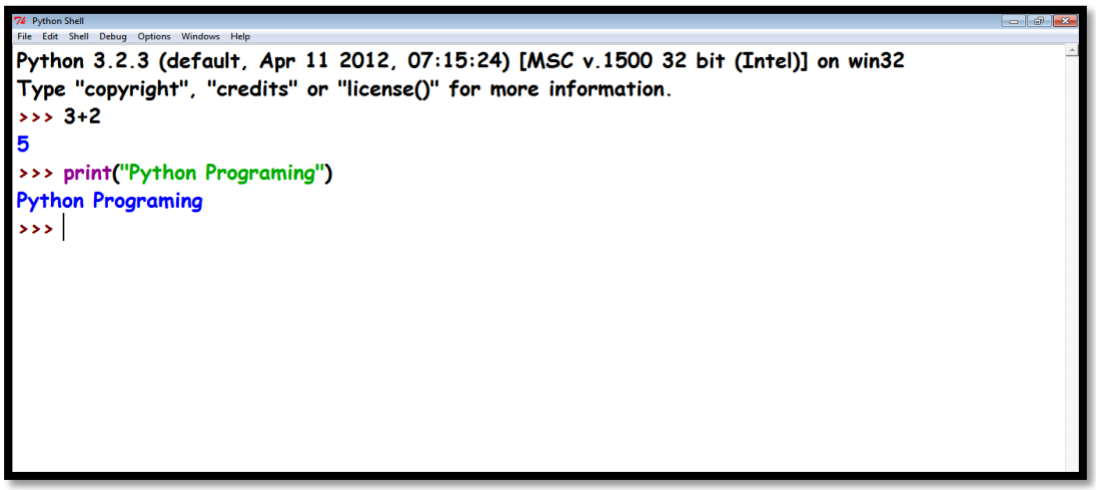
Surya Lakshmi Kantham Vinti
CSE Dept
ACET

Getting Started with Python Programming:

- Python is a high-level, general-purpose programming language for solving problems on modern computer systems.
- The language and many supporting tools are free, and Python programs can run on any operating system.
- You can download Python, its documentation, and related materials from www.python.org.

Running Code in the Interactive Shell :

- You can run simple Python expressions and statements in an interactive programming environment called the **shell.**
- The easiest way to open a Python shell is to launch the **IDLE** (Integrated DeveLopment Environment).
  - This is an integrated program development environment that comes with the Python installation.
  - When you do this, a window named Python Shell opens.
- A shell window contains an opening message followed by the special symbol >>>, called **a shell prompt**.
- When you enter an expression or statement, Python evaluates it and displays its result, if there is one, followed by a new prompt.



Surya Lakshmi Kantham Vinti
CSE Dept
ACET

COLOR CODING OF PYTHON PROGRAM ELEMENTS:

Color	Type of Element	Examples
Black	Inputs in the IDLE shell Numbers Operator symbols Variable, function, and method references Punctuation marks	`67, +, name, y = factorial(x)`
Blue	Outputs in the IDLE shell Function, class, and method names in definitions	`'Ken Lambert'`, `def factorial(n)`
Green	Strings	`"Ken Lambert"`
Orange	Keywords	`def, if, while`
Purple	Built-in function names	`abs, round, int`
Red	Program comments Error messages in the IDLE shell	`# Output the results` `ZeroDivisionError: division by zero`

Table 1-1 Color-coding of Python program elements in IDLE

Input, Processing, and Output :

- The Python shell itself is such a program;
  - its inputs are Python expressions or statements.
  - Its processing evaluates these items.
  - Its outputs are the results displayed in the shell.

Output :

- The programmer can force the output of a value by using the print function.
- It can take any of the following forms:

1) **print(<expression>)**

Here <expression> can be replaced with an expression that need to be evaluated or any string that can be printed.

```
>>> print(3+4)
7
```

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

```
>>> print("Hello CSE!!")

Hello CSE!!
```

2) **print(<expression>,..., <expression>)**

- Note the ellipsis (...) in this syntax example. The ellipsis indicates that you could include multiple expressions after the first one.
- The print function evaluates the expressions and displays their results, separated by single spaces, on one line.

```
>>> print(3+4,5*9,"hai cse")

  7  45   hai cse
```

3) **print(<expression>, end = "")**

- Whether it outputs one or multiple expressions, the print function always ends its output with a **newline.**
- To begin the next output on the same line as the previous one, you can place the expression end = "", which says "end the line with an empty string instead of a newline," at the end of the list of expressions

```
>>> def hai():
        print("hai")
        print("hello")
>>> hai()
hai
hello
>>> def hai():
        print("hai",end="")
        print("hello")
```

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

>>> hai()

Haihello

4) print(<expression>,...,<expression>, sep='<literal>')

sep stands for **separator** and is assigned a single space (' ') by default. It determines the value to join elements with.

>>>print(3+4,"hai",5*7)

7 hai 35

>>> print(3+4,"hai",5*7,sep='-->')

7-->hai-->35

>>>print(3+4,"hai",5*7,sep='\n')

7

hai

35

Input :

- you'll often want your programs to ask the user for input. You can do this by using the **input function**.
- This function causes the program to stop and wait for the user to enter a value from the keyboard.
- The form of an assignment statement with the input function is the following:

<variable identifier> = input(<a string prompt>)

- The input function does the following:

1. Displays a prompt for the input. In this example, the prompt is "Enter your name: ".

2. Receives a string of keystrokes, called characters, entered at the keyboard and returns the string to the shell.

```
>>> name=input("Enter a name")
Enter a nameSurya
>>> name
'Surya'
>>> a=input("Enter a number")
Enter a number9
>>> a
'9'                # 9 is treated as String
>>> b=input("Enter a real number")
Enter a real number9.563
>>> b
'9.563'            #9.563  is also treated as string
```

- The input function always builds a string from the user's keystrokes and returns it to the program.
- After inputting strings that represent numbers, the programmer must convert them from strings to the appropriate numeric types.
- In Python, there are two type conversion functions for this purpose, called
  - **int (for integers) and**
  - **float (for floating point** numbers).

```
a=input("Enter a number")
Enter a number9
>>> a
'9'
a=int(input("Enter a number"))
Enter a number9
>>> a
9
```

```
>>> b=input("Enter a real number")
Enter a real number9.563
>>> b
'9.563'
>>> b=float(input("Enter a real number"))
Enter a real number9.563
>>> b
9.563
```

Editing, Saving, and Running a Script :

- To compose and execute programs in this manner, you perform the following steps:

  1. Select the option **New Window from the File menu of the shell window.**

  2. In the new window, enter Python expressions or statements on separate lines, in the order in which you want Python to execute them.

  3. At any point, you may save the file by selecting File/Save. If you do this, you should use a **. py extension.**

  For example, your first program file might be named myprogram.py.

  4. To run this file of code as a Python script, select **Run Module from the Run menu** or press the F5 key.

Comments :

- A comment is a piece of program text that the computer ignores but that provides useful documentation to programmers.

- At the very least, the author of a program can include his or her name and a brief statement about the program's purpose at the beginning of the program file.
- end-of-line comments can document a program. These comments begin with the # symbol and extend to the end of a line. An end-of-line comment might explain the purpose of a variable or the strategy used by a piece of code.

  >>> RATE = 0.85 # Conversion rate for Canadian to US dollars

It's always better on a programmer's side to:

1) Begin a program with a statement of its purpose and other information that would help orient a programmer called on to modify the program at some future date.

2) Accompany a variable definition with a comment that explains the variable's purpose.

3) Precede major segments of code with brief comments that explain their purpose.

4) Include comments to explain the workings of complex or tricky sections of code

Variables :

- **Variable associates a name with a value, making it easy to** remember and use the value later in a program.
- Variables serve two important purposes in a program.
  - They help the programmer keep track of data that change over time.
  - They also allow the programmer to refer to a complex piece of information with a simple name

How to frame a Variable name(identifier) :

- A variable name must begin with either a **letter or an underscore** (_), and can contain any number of letters, digits, or other underscores
- Python variable names are **case sensitive**; thus, the variable WEIGHT is a different name from the variable weight.

- In the case of variable names that consist of more than one word, it's common to begin each word in the variable name (except for the first one) with an uppercase letter. This makes the variable name easier to read.
- For example, the name interestRate is slightly easier to read than the name interestrate
- Programmers use all uppercase letters for the names of variables that contain values that the program never changes. Such variables are known as **symbolic constants.**
- Variables receive their initial values and can be reset to new values with an **assignment statement.**

SYNTAX :

The simplest form of an assignment statement is the following:

<variable name> = <expression>

- The Python interpreter first evaluates the expression on the right side of the assignment symbol and then binds the variable name on the left side to this value.
- When this happens to the variable name for the first time, it is called **defining or initializing the variable.**

```
>>>9rate=78     #Started with number
SyntaxError: invalid syntax
>>> rate=78
>>> rate
78
>>> _rate=47
>>> rate
78
```

```
>>> _rate
47
>>> @si=98.45          #Started with special symbol
SyntaxError: invalid syntax
>>> si=98.45
```

Python Operators :

- Operators are used to perform operations on variables and values.
- Python divides the operators in the following groups:
  - Arithmetic operators
  - Comparison operators
  - Logical operators
  - Identity operators
  - Membership operators
  - Bitwise operators
  - Assignment operators

1. Arithmetic operators:

Some basic arithmetic operators are +,-,*,/,%,** and //

We can apply these operators in numbers as well as variables to perform corresponding operations

Let a= 10, b= 20

Operator	Description	Example	Output
+	It is used to perform addition operation (add the operands)	a+b	30
-	Perform substation operation (subtract the operands)	a-b	-10
*	Perform the multiplication operation	a*b	200
/	It returns quotient	a/b	0.5
%	It returns the reminder	b%a	0
**	Perform power calculation(Exponent)	a**b	10^{20}
//	Floor division	a//b	20

Example:

```
a = float (input (“enter a number”))
b= float (input (“enter b value “))
print (“Addition of “,a,” and “, b,” is “,a+b)
print (“Subtraction of “,a,” and “,b,” is “,a-b)
print(“Product of “,a,” and “,b,” is “,a*b)
```

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

```
print ("Division of ",a," and ",b," is ",a/b)
print("Modulus division of ",b," and ",a," is ",b%a)
print("Floor division of ",a," and ",b," is ",a//b)
print("Exponent of ",a," and ",b," is ",a**b)
```

2) **Comparison Operator: (Relational Operators):**

- It is used to compare the values on its either sides (Left and Right) of the operator and determines the relation between them.
- Let a = 100, b = 200

Operator	Description	Example	Output
==	It returns true if the two values on either side of the operator are exactly same	a==b	FALSE
!=	It returns true if the two values on either side of the operator are not same	a!=b	TRUE
<	It returns true if the value at LHS of the operator is less than the value at the RHS of the operator, otherwise false	a<b	TRUE
>	It returns true if the value at LHS of the operator is greater than the value at the RHS of the operator, otherwise false	a>b	FALSE
<=	It returns true if the value at LHS of the operator is less than or equal to the value at the RHS of the operator, otherwise false	a<=b	TRUE
>=	It returns true if the value at LHS of the operator is greater than or equal to the value at the RHS of the operator, otherwise false	a>=b	FALSE

Example:

```
a = float(input("Enter the value of a: "))
```

```
b = float(input("Enter the value of b: "))

print(a," == ",b,a==b)

print(a," != ",b,a!=b)

print(a," < ",b,a<b)

print(a," > ",b,a>b)

print(a," <= ",b,a<=b)

print(a," >= ",b,a>=b)
```

3) Logical Operators:

- Logical operators are used to evaluate two or more expressions with relational operators.
- Python supports 3 logical Operators
    1. Logical AND (and)
    2. Logical OR (or)
    3. Logical NOT (not)

Logical AND (and): If expression on both sides of the logical operator are true, then the whole expression is true.

Truth Table:

Exp 1	Exp 2	Exp 1 and Exp2
T	T	**T**
T	F	**F**
F	T	**F**
F	F	**F**

Example:

Let a = 10, b = 20, c = 30, d = 40

(a<d) and (b>c)

(True) and (False)

Result = False

Logical OR (or): If one or both the expressions is true then the whole expression is true.

Truth Table:

Exp 1	Exp 2	Exp1 or Exp2
T	T	**T**
T	F	**T**
F	T	**T**
F	F	**F**

Example: Let a = 10, b = 20, c = 30, d = 40

(a<d) or (b>c)

(True) or (False)

Result = True

Logical NOT (not):

- Logical NOT operator takes a single expression and negates the value of expression.
- Logical NOT produces a zero if the expression evaluates to a non-zero and produces 1 if the expression produces zero.

Truth Table:

Expression	!(Expression)
T (1)	F (0)
F (0)	T (1)

Example: Let a = 10, b = 20

not (a>b)

not (False)

Result : True

4) Identity Operators:

These operators compare the memory locations of two objects. Python supports two types of Identity Operators

- **is** operator: It returns true if operands (or) values on both sides of the operator point to the same object and False otherwise.

  Example: If a is b – it returns 1 if id (a) is same as id (b)

- **is not** operator : It returns true if operands (or) values on both sides of the operator do not point to the same object and False otherwise.

  Example: If a is not b – it returns 1 if id (a) is not same as id (b)

5) Membership Operators:

Python supports two types of membership operators

- **in**
- **not in**

The names itself suggests that, test for membership in a sequence such as string, list, tuple.

in – The operator returns true if a variable is found in the specified sequence and false otherwise.

Example: a in nums - it returns True if 'a' is present in the nums

not in – The operator returns true if a variable is not found in the sequence.

Example: a not in nums - it returns 1 if 'a' is not present in the nums

```
>>>name="Surya"
>>> 'u' in name
True
>>> 'b' in name
False
>>> m=[45,67,12,42,55,564,21]
>>> 45 in m
True
>>> 34 in m
False
>>> 45 not in m
False
>>> 34 not in m
True
```

6) Bitwise Operators :

Bitwise Operators perform operations at bit level. These operators include

- Bitwise AND (&)
- Bitwise OR (|)
- Bitwise NOT (~)
- Bitwise XOR (^)
- Bitwise Shift

Bitwise operators expect their operands to be of integers and track them as a sequence of bits.

Bitwise AND (&):

The bit in the first operand is Anded with the corresponding bit in the second operand. If both the bits are 1 then the corresponding bit result is 1 and 0 otherwise**.**

Truth Table:

Bit1	Bit2	Bit1 & Bit2
1	**1**	**1**
1	**0**	**0**
0	**1**	**0**
0	**0**	**0**

Example:

1010101010

&

1111010101

1010000000

Bitwise OR (|):

The bit in the first operand is Ored with the corresponding bit in the second operand. If either of the bits are 1 then the corresponding bit result is 1 and 0 otherwise.

Truth Table:

Bit1	Bit2	Bit1 & Bit2
1	**1**	**1**
1	**0**	**1**

0	**1**	**1**
0	**0**	**0**

Example:

1 0 1 0 1 0 1 0 1 0

|

1 1 1 1 0 1 0 1 0 1

1 1 1 1 1 1 1 1 1 1

Bitwise XOR (^):

The bit in the first operand is XORed with the corresponding bit in the second operand. If one of the bits is 1, then the corresponding bit result is 1 otherwise 0

Truth Table:

Bit1	Bit2	Bit1 & Bit2
1	**1**	**0**
1	**0**	**1**
0	**1**	**1**
0	**0**	**0**

Example:

1 0 1 0 1 0 1 0 1 0

^

1 1 1 1 0 1 0 1 0 1

0 1 0 1 1 1 1 1 1 1

Bitwise NOT (~):

It is a unary operation, which performs logical negation on each bit of the operand. It produces 1's compliment of the given value

Truth Table:

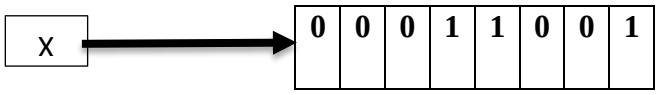
Bit	~(Bit)
0	**1**
1	**0**

Example:

~ (10101010) = 01010101

Shift Operators:

- Python supports two bitwise shift operators
  1. Shift Left (<<)
  2. Shift Right (>>)
- These operators are used to shift bits to the left or right.
- Syntax of shift operation is operand op num
- Let x = 00011010

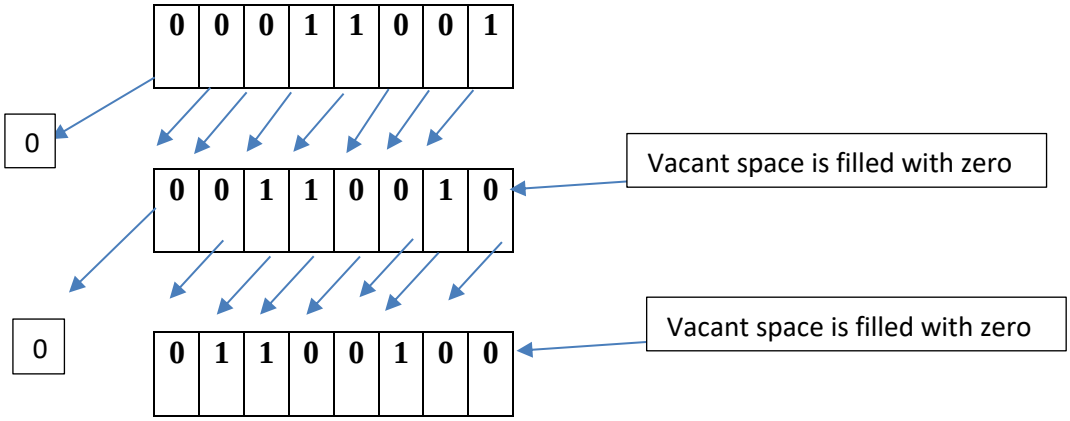


LSB **MSB**

- **Example 1: shift Left**

  x=3

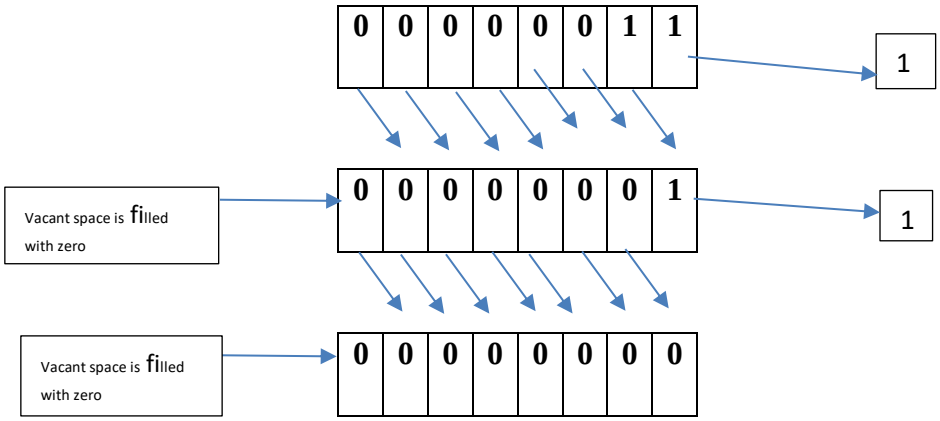
  x<<2



- **Example 1: shift Right**

x=3

x>>2



>>> x=5

>>> y=3

>>> x&y

1

>>> x|y

7

>>> x^y

6

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

```
>>> ~x
-6
>>> x<<1
10
>>> x>>1
2
>>> x>>2
1
```

7) Assignment and In place OR shortcut operators

Assignment Operator: It is used to assign a value to the operand.

Syntax: variablename=value

variablename=Expression

Example 1 : a=10 Example 2: c=a+b

b=20, c=30

Inplace Operator:

In place operators are called as shortcut operators that includes +=,-=, *=, /=, %=, //=, **= allow you to write code like num = num + 10 more concisely as num+ = 3

Operator	Description	Example
=	Assign right side value to the left hand side variable	a = b
+=	Add and assign	a+=b=> a=a+b

-=	Subtract and assign	a-=b=> a=a-b
***=**	Multiply and assign	a*=b=> a=a*b
/=	Divide and assign	a/=b=> a=a/b
%=	Modulus and assign	a%=b=> a=a%b
//=	Floor division and assign	a//=b=> a=a//b
****=**	Exponent and assign	a**=b=> a=a**b

Unary Operators:

- Acts on single operand
- Python supports single minus (-)
- When an operand is preceded by a minus sign, the unary operator negates its value.

Example: b=10

print(-b)

Operator Precedence and Associativity:

The following table is the list from higher precedence to lower precedence.

**	Exponentiation
~, +, -	Unary Operator
*, /, //, %	Multiply, division, floor division, modulus division
+, -	Addition and subtraction

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

>>, <<	Right and left bitwise shift operator
&	Bitwise AND
^, |	Bitwise XOR and regular OR
<=, >=, <, >	Comparison operator
<>, ;, ==, !=	Equality operator
=, %=, +=, -=	Assignment operators
Is, is not	Identity operators
In, not in	Membership operators
NOT, OR, AND	Logical Operators

Expressions in Python:

- An expression is any logical combination of symbols (like variables, constants and operators) that represent a value.
- Every language has some set of rules to define whether the expression is valid/invalid.
- In python, an expression must have at least have one operand can have one or more operators.

Example:a+b*c-5

In above expression

" +, *, - " are operators

" a, b, c " are operators

" 5 " is operators

- Valid Expression:

x = a/b

y = a*b

z = a^b

x = a>b

- Invalid Expressions:a+<y++

Types of Expression:

Python supports different types of expressions that can be classified as follows

- Based on position of operators in an expression
- Based on the datatype of the result obtained on evaluating an expression

- **Based on position of operators in an expression**

These type of expressions include:

Infix expression: In this type of expression the operator is placed in between the operands

Example: a=b-c

Prefix expression: In this type of expression the operator is placed before the operands

Example: a=-bc

Post fix expression: In this type of expression the operator is placed after the operands

Example: a=bc-

- **Based on the data type of the result obtained on evaluating an expression:**

These type of expressions include:

Constant Expression:

This type of expression involves only constants.

Example: 8+9-2

Floating Point Expression:

This type of expression produces floating point results

Example: a = 10, b = 5

a*b/2

Integer Expression:

This type of expression produces integer result after evaluating the expression.

Example: a = 10

b = 5

c = a*b

Relational Expression:

This type of expression returns either true or false

Example: c = a>b

Result: **True**

Logical Expression:

This type of expression combines two or more relational expressions and return a value as true or false

Example: (a>b) and (a!=b)

False and True

Result:True

Bitwise Expressions:

This type of expressions manipulate data at bit level.

Example: x = y&z

Assignment Expression:

In this type expression will assign a value or expression to a variable

Example: c=a+b

c=10

UNIT -1

PART -2

1. Standard Data Types

- The data stored in memory can be of many types.
- Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.
- Python has five standard data types-
    - Numbers
    - String
    - List
    - Tuple
    - Dictionary

1.1 Python Numbers

- Number data types store numeric values. Number objects are created when you assign a value to them.
- Python supports different numerical types −
    - int (signed integers)
    - float (floating point real values)
    - complex (complex numbers)
    - A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are real numbers and j is the imaginary unit.
    - ***bool*** (Boolean values)
        - True False
- A number is also an **immutable type**, meaning that changing or updating its value results in a newly allocated object.

1.1.1 How to Create and Assign Numbers (Number objects)

- Creating numbers is as simple as assigning a value to a variable:

anInt = 1

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

aFloat = 3.14159265358979323846264338327 95

aComplex = 1.23+4.56J

1.1.2 How to Update Numbers :

- update an existing number by (re)assigning a variable to another number.
- Every time you assign another number to a variable, you are creating a new object and assigning it
    - anInt += 1

1.1.3 How to Remove Numbers :

- delete a reference to a number object, just use the **del statement**
    - del anInt
    - del aLong, aFloat, aComplex

i) Standard (Regular or Plain) Integers

- Most machines (32-bit) running Python will provide a range of -2^31 to 2^31 -1, that is -2, 147,483,648 to 2,147,483,647.
- If Python is compiled on a 64-bit system with a 64-bit compiler, then the integers for that system will be 64-bit.
- Examples of Python integers:

0101 84 -237 0x80 (mention x for hexadeximal)

017 -680 -0x92 0o234 (mention o for octal)

ii) Boolean

- Objects of this type have two possible values, True or False.

```
>>>t=4>8
>>>type(t)
<class 'bool'>
```

iii) Floating Point Numbers :

- Floats in Python are values that can be represented in straightforward decimal or scientific notations.

- These 8-byte (64-bit) values conform to the IEEE 754 definition (52M/11E/1S) where 52 bits are allocated to the mantissa, 11 bits to the exponent , and the final bit to the sign.
- Floating point values are denoted by a decimal point (.) in the appropriate place and an optional "e" suffix representing scientific notation.
- Here are some floating point values:

  0.0 -777. 1 -5.555567119 9.6e3 9.384e-23

iv) Complex Numbers

- A complex number is any ordered pair of floating point real numbers (x, y) denoted by x + yj where x is the real part and y is the imaginary part of a complex number.
- Facts about Python's support of complex numbers:
  - Imaginary numbers by themselves are not supported in Python (they are paired with a real part of 0.0 to make a complex number)
  - Complex numbers are made up of real and imaginary parts
  - Syntax for a complex number: *real+imagj*
  - Both real and imaginary components are floating point values
  - Imaginary part is suffixed with letter "J" lowercase (j) or uppercase (J)
- The following are examples of complex numbers:

  64.375+1j 4.23-8.5j 0.23-8.55j

1.2 Strings:

- Strings are sequence of individual characters enclosed in single or double quotes.

  Ex :- >>>str="hai how are u?"

  >>> new= 'this is another example'
- Strings are immutable, meaning that changing an element of a string requires creating a new string.

1.2.1 How to Create and Assign Strings

- Creating strings is as simple as using a scalar value and assign it to a variable:

```
>>> aString = 'Hello World!'        # using single quotes
```

```
>>> another = "Python is cool!"   # double quotes
>>> print(aString)                # print, no quotes!
Hello World!
>>> another                       # no print, quotes!
'Python is cool!'
```

1.2.2 How to Access Values in Strings

- Python does not support a character type; these are treated as strings of length one, thus also considered a substring.
- To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring:

```
>>> aString = 'Hello World!'
>>> aString[0]
'H'
>>> aString[1:5]
'ello'
>>> aString[6:]
'World!'
```

1.2.3 How to Update Strings

- You can "update" an existing string by (re)assigning a variable to another string.

```
>>>aString=‘Hello World’
>>> aString = aString[:6] + 'Python!'
>>> aString
'Hello Python!'
>>> aString = 'different string altogether'
>>> aString
'different string altogether‘
```

1.2.4 How to Remove Characters and Strings

- Strings are immutable, so you cannot remove individual characters from an existing string.
- To clear or remove a string, you assign an empty string or use the del statement, respectively:

  >>> aString = ''

  >>> aString

  ‘’

  >>> del aString

EXAMPLES :

str = 'Hello World!'

>>> print (str) # Prints complete string

Hello World!

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

H

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

llo

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

llo World!

>>> print (str * 2) # Prints string two times

Hello World!Hello World!

>>> print (str + "TEST") # Prints concatenated string

Hello World!TEST

1.3 LISTS :

- Lists provide sequential storage through an index offset and access to single or consecutive elements through slices.
- Lists are flexible container objects that hold an arbitrary number of Python objects

- >>>a=[45,7,9,78]

  >>>a

  [45,7,9,78]

1.3.1 How to Create and Assign Lists

- Creating lists is as simple as assigning a value to a variable.
- Lists are delimited by surrounding square brackets ([ ])

```
>>>a=[8,9,'a',6.8,"hello"]

>>>a

[8, 9, 'a', 6.8, 'hello']

>>>b=[8,9,'a',6.8,[66,"bye"],"hello"]

>>>b

[8, 9, 'a', 6.8, [66, 'bye'], 'hello']
```

1.3.2 How to Access Values in Lists

- Slicing works similar to strings; use the square bracket slice operator ([ ]) along with the index or indices.

```
>>>aList = [123, 'abc', 4.56, ['inner', 'list'], 7-9j]

>>> aList[0]

123

>>> aList[1:4]

['abc', 4.56, ['inner', 'list']]

>>> aList[:3]

[123, 'abc', 4.56]

>>> aList[3][1]

'list'
```

1.3.3 How to Update Lists

- You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method:

```
aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> aList[2]
4.56
>>> aList[2] = 'float replacer'
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>>aList.append(84)
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j),84]
```

1.3.4 How to Remove List Elements and Lists

- To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

```
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>> del  aList[1]
>>> aList
[123, 'float replacer', ['inner', 'list'], (7-9j)]
>>> aList.remove(123)
>>> aList
['float replacer', ['inner', 'list'], (7-9j)]
```

1.4 Tuples :

- Tuples are another container type extremely similar in nature to lists.
- The only visible difference between tuples and lists is that tuples use parentheses () and lists use square brackets[].

- Functionally, there is a more significant difference, and that is the fact that tuples are **immutable**. Because of this, tuples can do something that lists cannot do . . . be a dictionary key.

1.4.1 How to Create and Assign Tuples

- Creating and assigning tuples are practically identical to creating and assigning lists, i.e., enclosing objects in paranthesis() and assigning the tuple to an identifier.

```
>>> a=(34,56,"hai",21.54)

>>> a

(34, 56, 'hai', 21.54)

>>> tuple("hai")

('h', 'a', 'i')

>>> a=(34,(56,"hai"),21.54)

>>> a

(34, (56, 'hai'), 21.54)
```

1.4.2 How to Access Values in Tuples

- Slicing works similarly to lists. Use the square bracket slice operator ([ ]) along with the index or indices.

```
>>> a=(34,56,"hai",21.54)

>>> a[2]

'hai'

>>> a[1:4]

(56, 'hai', 21.54)
```

1.4.3 How to Update Tuples

- Like numbers and strings, tuples are immutable, which means you cannot update them or change values of tuple elements.
- We were able to take portions of an existing tuple to create a new tuple.

```
>>> a

(34, 56, 'hai', 21.54)

>>> b=(a[2],a[1],a[0])
```

>>> b

('hai', 56, 34)

1.4.4 How to Remove Tuple Elements and Tuples

- Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.
- To explicitly remove an entire tuple, just use the **del** statement to reduce an object's reference count

  del aTuple

1.5 Dictionary :

- Dictionaries are the sole mapping type in Python. Mapping objects have a one-to-many correspondence between *hashable values (keys) and the objects they represent (values).*
- *They* can be generally considered as *mutable hash tables.*
- *A dictionary object itself is mutable and is yet* another container type that can store any number of Python objects, including other container types.

1.5.1 How to Create and Assign Dictionaries

- The syntax of a dictionary entry is *key:value* . *Also,* dictionary entries are enclosed in braces ({ })
- Creating dictionaries simply involves assigning a dictionary to a variable, regardless of whether the dictionary has elements or not:

  >>> dict1={}

  >>> dict2={"name":"Surya","last":"vinti","college":"CMREC"}

  >>> dict1

  {}

  >>> dict2

  {'name': 'Surya', 'last': 'vinti', 'college': 'CMREC'}

  >>> dict3={1:23,3:45,2:98}

  >>> dict3

  {1: 23, 3: 45, 2: 98}

1.5.2 How to Access Values in Dictionaries

- To traverse a dictionary (normally by key):

```
>>> dict3={1:23,3:45,2:98}
>>> dict3[1]
23
>>> dict2
{'name': 'Surya', 'last': 'vinti', 'college': 'CMREC'}
>>> dict2["name"]
'Surya'
```

1.5.3 How to Update Dictionaries

- You can update a dictionary by adding a new entry or element (i.e., a key-value pair), modifying an existing entry, or deleting an existing entry

```
>>> dict2= {'name': 'Surya', 'last': 'vinti', 'college': 'CMREC'}
>>> dict2["name"]="Lakshmi"
>>> dict2
{'name': 'Lakshmi', 'last': 'vinti', 'college': 'CMREC'}
>>> dict3={1:23,3:45,2:98}
>>> dict3[3]=45.7
>>> dict3
{1: 23, 3: 45.7, 2: 98}
```

1.5.4 How to Remove Dictionary Elements and Dictionaries

- Either remove individual dictionary elements or clear the entire contents of a dictionary

```
>>> dict3
{1: 23, 3: 45.7, 2: 98}
>>> del dict3[2]
>>> dict3
{1: 23, 3: 45.7}
```

```
>>> del dict3
>>> dict3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'dict3' is not defined
```

2. Character Sets

- In Python, character literals look just like string literals and are of the string type.
- All data and instructions in a program are translated to binary numbers before being run on a real computer.
- To support this translation, the characters in a string each map to an integer value.
- This mapping is defined in character sets, among them the
  - **ASCII set , and the**
  - **Unicode set**

2.1 ASCII:

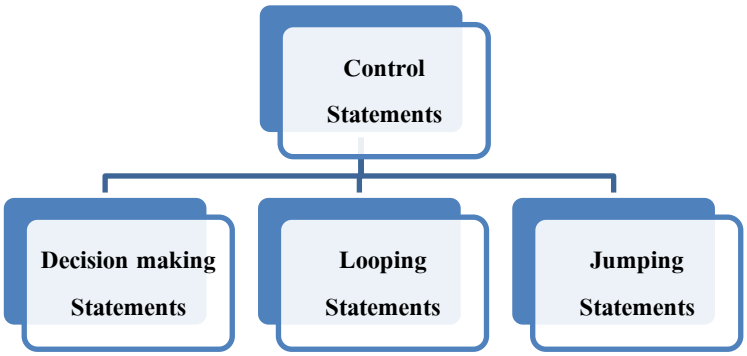
- The term ASCII stands for American Standard Code for Information Interchange , In the 1960s, the original ASCII set encoded each keyboard character and several control characters using the integers from 0 through 127.
- An example of a control character is Control1D, which is the command to terminate a shell window.
- As new function keys and some international characters were added to keyboards, the ASCII set doubled in size to 256 distinct values in the mid-1980s.

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DCI	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	"	#	$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	I	j	k	l	m
11	n	o	P	q	r	S	t	u	v	w
12	X	y	z	{	|	}	~	DEL		

2.2 Unicode :

- When characters and symbols were added from languages other than English, the **Unicode set** was created to support 65,536 values in the early 1990s.
- Unicode supports more than 128,000 values at the present time.

3. CONTROL STATEMENTS



3.1 Decision making Statements :

- Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions.
- Python programming language assumes any non-zero and non-null values as TRUE, and any zero or null values as FALSE value.
- Decision making Statements:
  - if
  - if else
  - if-elif-else
  - nested if

3.1.1 Simple if

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

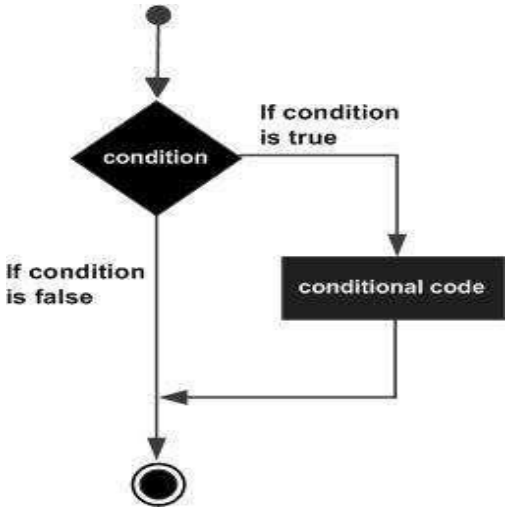
- Syntax

if expression:

statement(s)

statement-x

- If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed.
- In Python, statements in a block are uniformly indented after the : symbol.
- If boolean expression evaluates to FALSE, then the first set of code after the end of block is executed.



>>>if 5>3:

print("hai")

Output:

hai

3.1.2 if-else

- The if-else statement is the most common type of selection statement. It is also called a two-way selection statement, because it directs the computer to make a choice between two alternative courses of action
- Python syntax for the if-else statement:

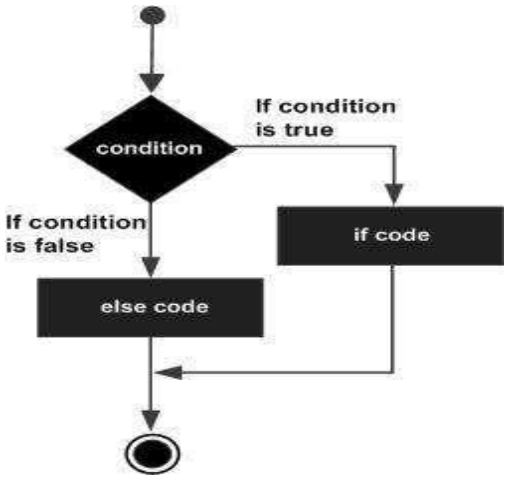
if <condition>:

<sequence of statements–1>

else:

<sequence of statements–2>

- The condition in the if-else statement must be a Boolean expression—that is, an expression that evaluates to either true or false



Code:

```
n=int(input("Enter a
 number"))

if n>0:
   print("Number is +ve")
else:
   print("Number is -ve")
print("Out of if else")
```

Output:

```
>>> ifelse()
Enter a number3
Number is +ve
Out of if else
>>> ifelse()
Enter a number-5
Number is -ve
Out of if else
```

3.1.3. Multi-Way if Statement(if-elif-else)

- The process of testing several conditions and responding accordingly can be described in code by a multi-way selection statement.
- The syntax of the multi-way if statement is the following:

if <condition-1>:

<sequence of statements-1>

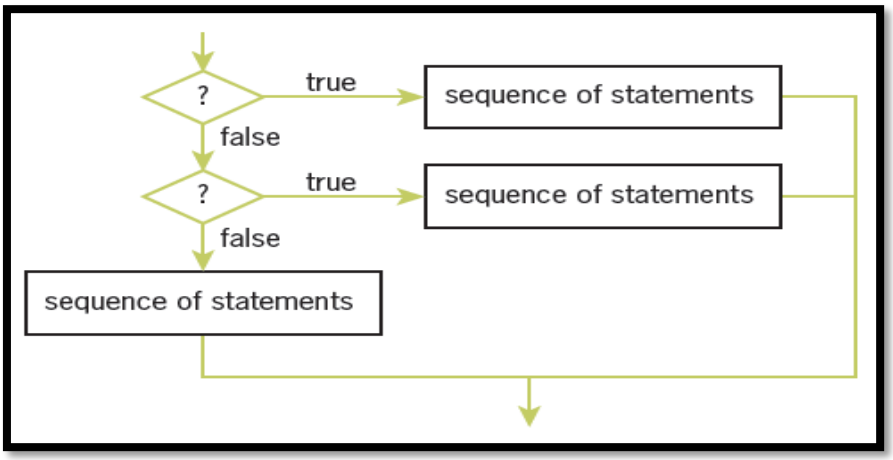
elif <condition-*n*>:

<sequence of statements-*n*>

else:

<default sequence of statements>

- The multi-way if statement considers each condition until one evaluates to True or they all evaluate to False.
- When a condition evaluates to True, the corresponding action is performed and control skips to the end of the entire selection statement.
- If no condition evaluates to True, then the action after the trailing else is performed.



Code:

```
n=int(input("Enter a number"))

  if n>0:
     print("Number is +ve")
  elif n<0:
     print("Number is -ve")
  else:
     print("Number is neither
  +ve nor -ve")
```

Output :

```
>>> ifelif()
Enter a number5
Number is +ve
>>> ifelif()
Enter a number-4
Number is -ve
>>> ifelif()
Enter a number0
Number is neither +ve nor -ve
```

3.1.4 Nested if -else

- Nested if-else statements means an if-else statement inside another if/else statement or both.

- Syntax:

```
if  <expression1>:
        if  <expression2>:
            statement block1
        else:
            statement block2
    else:
        if <expression3>:
            statement block3
        else:
            statement block4
```

Code

```
a=int(input("Enter 1st number:"))
b=int(input("Enter 2nd number:"))
c=int(input("Enter 3rd number:"))
if a>b:
    if a>c:
        print(a,"is biggest")
    else:
        print(c,"is biggest")
else:
    if b>c:
        print(b,"is biggest")
    else:
```

print(c,"is biggest")

Output:

nestedif()

Enter 1st number:5

Enter 2nd number:2

Enter 3rd number:8

8 is biggest

3.2 Looping Statements

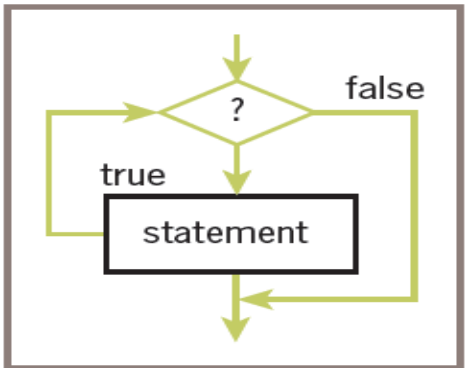
- In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.
- A loop statement allows us to execute a statement or group of statements multiple times.
- Looping Statements supported by Python :
  - for
  - while

3.2.1 while loop

- The while loop is also called an entry-control loop, because its condition is tested at the top of the loop.
- This implies that the statements within the loop can execute zero or more times.
- Syntax:

  while <condition>:

  <sequence of statements>



Surya Lakshmi Kantham Vinti
CSE Dept
ACET

Code :

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

Output:

1 2 3 4 5 6 7 8 9 10

3.2.2 for loop :

- The for statement in Python has the ability to iterate over the items of any sequence, such as a list or a string.
- The basic form of for loop is

```
for <variable> in range(<lowerbound>,<upperbound+1)):
    <statement-1>
    .
    .
    <statement-n>
```

Examples:

```
>>>for i in "Surya":
        print(i,end=",")
S,u,r,y,a,
>>>for i in range(5,10):
        print(i)
```

5

6

7

8

9

Nested Loops :

- Python programming language allows the use of one loop inside another loop. The following section shows a few examples to illustrate the concept.
- Syntax for nested for loop:

```
for iterating_var in sequence:
        for iterating_var in sequence:
            statement(s) –block1
        statement(s)- block2
```

- The syntax for a nested while loop statement in Python programming language is as follows:

```
while expression:
        while expression:
            statement(s)-block1
        statement(s)-block2
```

Examples:

```
>>>for i in range(1,4):
        for j in range(1,4):
            print("*" ,end=' ')
        print("\r")
```

Output:

* * *

* * *

* * *

4 Input Validation : Calculating a Running Total

```
vchoice=["y","Y"]
invchoice=["n","N"]
choice=vchoice+invchoice
num=[]
s=0
ch="y"
while ch in vchoice:
    n=int(input("Enter a number:"))
    num.append(n)
    s=s+n
    ch=input("Do u want to enter another number:[Y/N]")
    while ch not in choice:
        ch=input("Enter a proper choice:")
print("All the elements u entered till now")
for i in num:
    print(i,end=" ")
print("Sum of all the numbers u entered are:",s)
```

OUTPUT:

```
Enter a number:5
Do u want to enter another number:[Y/N]y
Enter a number:6
Do u want to enter another number:[Y/N]Y
Enter a number:2
Do u want to enter another number:[Y/N]u
Enter a proper choice:i
```

Enter a proper choice:o

Enter a proper choice:y

Enter a number:87

Do u want to enter another number:[Y/N]n

All the elements u entered till now

5 6 2 87 Sum of all the numbers u entered are: 100

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

UNIT -2

Control Statement: Definite iteration for Loop, Formatting Text for output, Selection if and if else Statement Conditional Iteration The While Loop

Strings and Text Files: Accessing Character and Substring in Strings, Data Encryption, Strings and Number Systems, String Methods Text Files.

CONTROL STATEMENTS :Looping Statements

- In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.
- A loop statement allows us to execute a statement or group of statements multiple times.
- Looping Statements supported by Python :
  - **for**
  - **while**
- There are two types of loops—
  - those that repeat an action a predefined number of times**(definite iteration)**, and
  - those that perform the action until the program determines that it needs to stop **(indefinite iteration)**.

1 Definite iteration for Loop

- **Executing a Statement a Given Number of Times**
  - The form of this type of for loop is

```
for <variable> in range(<an integer expression>):
        <statement-1>
        .
        .
        <statement-n>
```

- The first line of code in a loop is sometimes called the **loop header,**which denotes the number of iterations that the loop performs.
  - The colon (:) ends the loop header.

Surya Lakshmi Kantham Vinti
CSE DEPT
ACET

- The **loop body comprises** the statements in the remaining lines of code, below the header.These statements are executed in sequence on each pass through the loop.
  - The statements in the loop body *must be indented and aligned in the same column.*

```
>>> for i in range(4):
        print(i)
0
1
2
3
```

Count-Controlled Loops:

- Loops that count through a range of numbers are also called **count-controlled loops.**
- **The** value of the count on each pass is often used in computations.
- To count from an explicit lower bound, the programmer can supply a second integer expression in the loop header. When two arguments are supplied to range, the count ranges from the first argument to the second argument minus 1
- The only thing in this version to be careful about is the second argument of range, which should specify an integer greater by 1 than the desired upper bound of the count.
- Here is the form of this version of the for loop:

  for <variable> in range(<lower bound>, <upper bound + 1>):

  <loop body>

```
>>>for i in range(5,10):
            print(i)
5
6
7
8
9
```

Loop Errors: Off-by-One Error:

- The loop fails to perform the expected number of iterations. Because this number is typically off by one, the error is called an **off-by-one error**

Traversing the Contents of a Data Sequence:

The values contained in any sequence can be visited by running a for loop , as follows:

for <variable> in <sequence>:

<do something with variable>

– On each pass through the loop, the variable is bound to or assigned the next value in the sequence, starting with the first one and ending with the last one.

```
>>> name="Surya Lakshmi"
>>> nl=[45,36,644]
>>> nt=(4,22,6,1)
>>> for i in name:              #Traversing string
        print(i,end=",")
S,u,r,y,a, ,L,a,k,s,h,m,i,
>>>for i in nl:                #Traversing list
        print(i,end=",")
45,36,644,
>>>for i in nt:                #Traversing tuple
        print(i,end=",")
4,22,6,1,
```

Specifying the Steps in the Range :

- A variant of Python's range function expects a third argument that allows you to nicely skip some numbers.
- The third argument specifies a step value, or the interval between the numbers used in the range, as shown in the examples that follow:

```
>>> list(range(1, 6, 1)) # Same as using two arguments
[1, 2, 3, 4, 5]
>>> list(range(1, 6, 2)) # Use every other number
```

[1, 3, 5]

>>> list(range(1, 6, 3)) # Use every third number

[1, 4]

Loops That Count Down

- Once in a while, a problem calls for counting in the opposite direction, from the upper bound down to the lower bound.
- a loop displays the count from 10 down to 1 to show how this would be done:

```
>>> for count in range(10, 0, -1):
        print(count, end = " ")
10  9 8 7 6 5 4 3 2 1
```

- When the step argument is a negative number, the range function generates a sequence of numbers from the first argument down to the second argument plus 1.

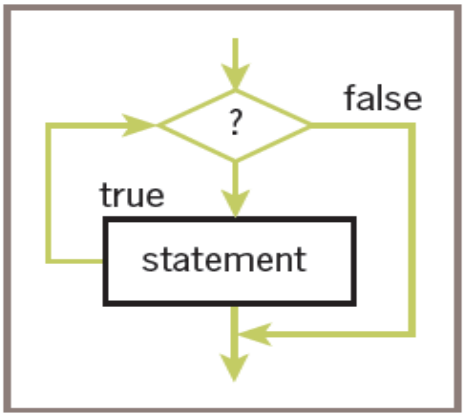
2. Conditional Iteration: The while Loop

- A loop continues to repeat as long as a condition remains true.This is called Conditional iteration
- **The Structure and Behavior of a while Loop :**
- Conditional iteration requires that a condition be tested within the loop to determine whether the loop should continue. Such a condition is called the loop's **continuation condition.**
  - If the continuation condition is false, the loop ends.
  - If the continuation condition is true, the statements within the loop are executed again.
- **Syntax:**

  while <condition>:

  <sequence of statements>

- The while loop is also called an **entry-control loop, because its condition is tested at** the top of the loop.
- This implies that the statements within the loop can execute zero or more times.




```
# Summation with a while loop

theSum = 0

count = 1

while  count <= 10:

theSum += count

count += 1

print(theSum)
```

- The while loop is also called an **entry-control loop, because its condition is tested at** the top of the loop.
  - This implies that the statements within the loop can execute zero or more times.

The while True Loop and the break Statement

- If the loop must run at least once, use a while True loop and delay the examination of the termination condition until it becomes available in the body of the loop.
- Ensure that something occurs in the loop to allow the condition to be checked and a break statement to be reached eventually.

```
while True:

    number = int(input("Enter the numeric grade: "))

    if number >= 0 and number <= 100:

            print(number) # Just echo the valid input

            break
```

else:

print("Error: grade must be between 100 and 0")

OUTPUT

A trial run with just this segment shows the following interaction:

Enter the numeric grade: 101

Error: grade must be between 100 and 0

Enter the numeric grade: –1

Error: grade must be between 100 and 0

Enter the numeric grade: 45

45

Random Numbers

- To simulate randomness in computer applications, programming languages include resources for generating **random numbers.**
- The function **random.randint (from module random)** returns a random number from among the numbers between the two arguments and including those numbers.
- The next session simulates the roll of die 10 times:

>>> import random

>>> for roll in range(10):

print(random.randint(1, 6), end = " ")

2 4 6 4 3 2 3 6 2 2

3. Formatting Text for output

- Many data-processing applications require output that has a **tabular format,** like that used in spreadsheets or tables of numeric data.
- In this format, numbers and other information are aligned in columns that can be either left-justified or right-justified.
  - A column of data is left-justified if its values are vertically aligned beginning with their leftmost characters.

- A column of data is right-justified if its values are vertically aligned beginning with their rightmost characters.

- The total number of data characters and additional spaces for a given datum in a formatted string is called its **field width.**

3.1 FORMATTING STRING:

- The simplest form of this operation is the following:

  <format string> % <datum>

- The following example shows how to right-justify and left- justify the string "four" within a field width of 6:

```
>>> "%6s" % "four"          # Right justify

' four'

>>> "%-6s" % "four“         # Left justify

'four '
```

- When the field width is positive, the datum is right-justified; when the field width is negative, you get left-justification.
- If the field width is less than or equal to the datum's print length in characters, no justification is added.
- The % operator works with this information to build and return a formatted string.

3.2 FORMATTING INTEGERS:

- To format integers, you use the letter d instead of s.
- To format a sequence of data values, you construct a format string that includes a format code for each datum and place the data values in a tuple following the % operator.
- The form of the second version of this operation follows:

  <format string> % (<datum–1>, ..., <datum–*n*>)

WITHOUT FORMATTING (integers):

```
>>> for exponent in range(7, 11):
        print(exponent, 10 ** exponent)
```

Surya Lakshmi Kantham Vinti
CSE DEPT
ACET

7 10000000

8 100000000

9 1000000000

10 10000000000

WITH FORMATTING (integers):

- The first column is left-justified in a field width of 3, and the second column is right-justified in a field width of 12.

```
>>> for exponent in range(7, 11):
        print("%-3d%12d" % (exponent, 10 ** exponent))
7          10000000
8         100000000
9        1000000000
10  10000000000
```

3.3 FORMATTING FLOAT:

- The format information for a data value of type float has the form

  %<field width>.<precision>f

  where .<precision> is optional.

- Example of the use of a format string, which says to use a field width of 6 and a precision of 3 to format the float value 3.14:

```
>>> "%6.3f" % 3.14
' 3.140'
```

- Note that Python adds a digit of precision to the string and pads it with a space to the left to achieve the field width of 6. This width includes the place occupied by the decimal point.

4. Strings

- E-mail, text messaging, Web pages, and word processing all rely on and manipulate data consisting of strings of characters.

- A string is a sequence of characters enclosed in single or double quotation marks.

- The following session with the Python shell shows some example strings:

>>> 'Hello there!'

'Hello there!'

>>> "Hello there!"

'Hello there!'

>>> ''

''

>>> """"

''

4.1 Accessing Character and Substring in Strings

The Structure of Strings:

- A string is a **data structure** i.e., it is a compound unit that consists of several other pieces of data.
- A string is a sequence of zero or more characters.
- The string is an **immutable data structure**. This means that its internal data elements, the characters, can be accessed, but cannot be replaced, inserted, or removed.
- A string's length is the number of characters it contains. Python's **len** function returns this value when it is passed a string, as shown in the following session:

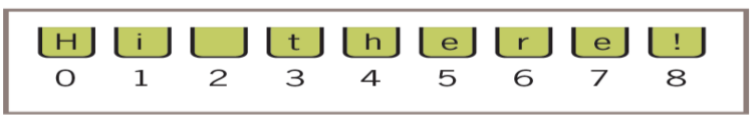
  >>> **len("Hi there!")**

  9

  >>> len("")

  0

- The positions of a string's characters are numbered from 0, on the left, to the length of the string minus 1, on the right.
- The following figure illustrates the sequence of characters and their positions in the string "Hi there!".
- Note that the ninth and last character, '!', is at position 8.



The Subscript Operator:

- The **subscript operator []** inspects one character at a given position without visiting all the characters in a given string / sequence.
- The simplest form of the subscript operation is the following:

  <a string>[<integer or index>]

- The first part of this operation is the string you want to inspect.
- The integer in brackets is the index that indicates the position of a particular character in that string.

```
>>> name = "Alan Turing"
>>> name[0]                 # Examine the first character
'A'
>>> name[3]                 # Examine the fourth character
'n'
>>> name[len(name)]      # Oops! An index error!
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> name[len(name) - 1] # Examine the last character
'g'
>>> name[-l]                # Shorthand for the last character
'g'
>>> name[-2]                # Shorthand for next to last character
'n'
```

- The next code segment uses a count-controlled loop to display the characters and their positions:

```
>>> data = "Hi there!"
>>> for index in range(len(data)):
                print(index, data[index])
```

Surya Lakshmi Kantham Vinti
CSE DEPT
ACET

0 H

1 i

2

3 t

4 h

5 e

6 r

7 e

8 !

4.2 Slicing for Substrings :

- You can use Python's subscript operator to obtain a substring through a process called **slicing.**
- To extract a substring, the programmer places a colon (:) in the subscript. An integer value can appear on either side of the colon
- When two integer positions are included in the slice, the range of characters in the substring extends from the first position up to but not including the second position.
- When the integer is omitted on either side of the colon, all of the characters extending to the end or the beginning are included in the substring

```
>>> name="Surya Lakshmi"
>>> name[:]
'Surya Lakshmi'
>>> name[0:5]
'Surya'
>>> name[0:]
'Surya Lakshmi'
>>> name[4:7]
'a L'
name[-4:-1]
```

'shm'

4.3 Testing for a Substring with the in Operator:

- When used with strings, the left operand of in is a target substring, and the right operand is the string to be searched.
- The operator in returns True if the target string is somewhere in the search string, or False otherwise.

```
name="Surya Lakshmi"
>>> "z" in name
False
>>> "u" in name
True
>>> "Lak" in name
True
```

5. Strings and Number Systems

- The system used to represent numbers are called number systems. Various number systems are:
  - decimal number system(base ten number system)
  - binary number system (base two number system)
  - octal number system (base eight number system)
  - hexadecimal number system (base 16 number system)
- To identify the system being used, you attach the base as a subscript to the number.
- For example, the following numbers represent the quantity 41510 in the binary, octal, decimal, and hexadecimal systems:
  - 415 in binary notation 1100111112
  - 415 in octal notation 6378
  - 415 in decimal notation 41510
  - 415 in hexadecimal notation $19F16$

The Positional System for Representing Numbers

- All of the number systems we have examined use **positional notation**—that is, the value of each digit in a number is determined by the digit's position in the number.
- In other words, each digit has a **positional value**. The positional value of a digit is determined by raising the base of the system to the power specified by the position (***base position)***.
- To determine the quantity represented by a number in any system from base 2 through base 10, you multiply each digit (as a decimal number) by its positional value and add the results.
- The following example shows how this is done for a three-digit number in base 10:

$$
\begin{aligned}
&415_{10} = \\
&4 * 10^2 + 1 * 10^1 + 5 * 10^0 = \\
&4 * 100 + 1 * 10 + 5 * 1 = \\
&400 + 10 + 5 = 415
\end{aligned}
$$

Converting Binary to Decimal

- Each digit or bit in a binary number has a positional value that is a power of 2.
- We occasionally refer to a binary number as a string of bits or a **bit string.**
- Determine the integer quantity that a string of bits represents in the usual manner: Multiply the value of each bit (0 or 1) by its positional value and add the results

$$
\begin{aligned}
&1100111_2 = \\
&1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = \\
&1 * 64 + 1 * 32 + 0 * 16 + 0 * 8 + 1 * 4 + 1 * 2 + 1 * 1 = \\
&64 + 32 + 4 + 2 + 1 = 103
\end{aligned}
$$

- **Python script to convert binary number to decimal number**

```
bitString = input("Enter a string of bits: ")
decimal = 0
exponent = len(bitString) - 1
for digit in bitString:
```

Surya Lakshmi Kantham Vinti
CSE DEPT
ACET

```
        decimal = decimal + int(digit) * 2 ** exponent
        exponent = exponent - 1
    print("The integer value is", decimal)
```

OUTPUT :

Enter a string of bits: 1111

The integer value is 15

Enter a string of bits: 101

The integer value is 5

Converting Decimal to Binary:

- This algorithm repeatedly divides the given decimal number by 2.
- After each division, the remainder (either a 0 or a 1) is placed at the beginning of a string of bits.
- The quotient becomes the next dividend in the process. The string of bits is initially empty, and the process continues while the decimal number is greater than 0.

Python script to convert decimal number to binary number

```
decimal = int(input("Enter a decimal integer: "))
if (decimal == 0):
        print(0)
else:
        print("Quotient Remainder Binary")
        bitString = ""
        while decimal > 0:
                remainder = decimal % 2
                decimal = decimal // 2
                bitString = str(remainder) + bitString
                print("%5d%8d%12s" % (decimal, remainder,bitString))
        print("The binary representation is", bitString)
```

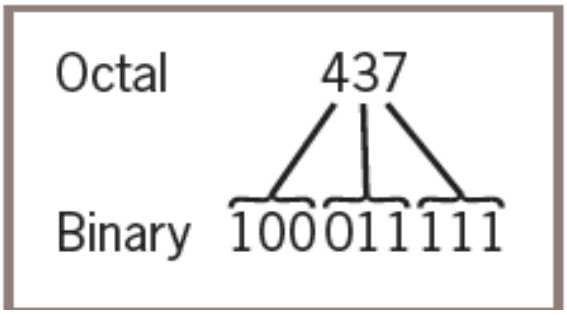
Enter a decimal integer: 34

Quotient	Remainder	Binary
17	0	0
8	1	10
4	0	010
2	0	0010
1	0	00010
0	1	100010

The binary representation is 100010

Octal Numbers

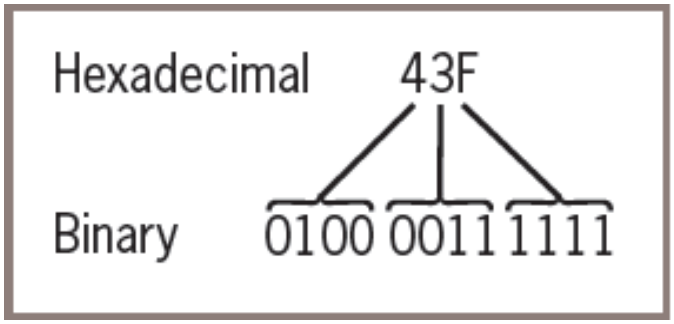
- The **octal system** uses a base of eight and the digits 0 . . . 7.
- Conversions of octal to decimal and decimal to octal use algorithms similar to those discussed thus far (using powers of 8 and multiplying or dividing by 8, instead of 2).
- To convert binary to octal, you begin at the right and factor the bits into groups of three bits each. You then convert each group of three bits to the octal digit they represent.



Hexadecimal Numbers:

- The **hexadecimal** or base-16 system (called "hex" for short), which uses 16 different digits, provides a more concise notation than octal for larger numbers.
- Base 16 uses the digits 0 . . . 9 for the corresponding integer quantities and the letters A . . . F for the integer quantities 10 . . . 15.
- The conversion between numbers in the two systems works as follows.
- Each digit in the hexadecimal number is equivalent to four digits in the binary number.
- Thus, to convert from hexadecimal to binary, you replace each hexadecimal digit with the corresponding 4-bit binary number.

- To convert from binary to hexadecimal, you factor the bits into groups of four and look up the corresponding hex digits.



5 String Methods :

String Method	What it Does
`s.center(width)`	Returns a copy of `s` centered within the given number of columns.
`s.count(sub [, start [, end]])`	Returns the number of non-overlapping occurrences of substring `sub` in `s`. Optional arguments `start` and `end` are interpreted as in slice notation.
`s.endswith(sub)`	Returns `True` if `s` ends with `sub` or `False` otherwise.
`s.find(sub [, start [, end]])`	Returns the lowest index in `s` where substring `sub` is found. Optional arguments `start` and `end` are interpreted as in slice notation.
`s.isalpha()`	Returns `True` if `s` contains only letters or `False` otherwise.
`s.isdigit()`	Returns `True` if `s` contains only digits or `False` otherwise.
`s.join(sequence)`	Returns a string that is the concatenation of the strings in the sequence. The separator between elements is `s`.
`s.lower()`	Returns a copy of `s` converted to lowercase.
`s.replace(old, new [, count])`	Returns a copy of `s` with all occurrences of substring `old` replaced by `new`. If the optional argument `count` is given, only the first `count` occurrences are replaced.
`s.split([sep])`	Returns a list of the words in `s`, using `sep` as the delimiter string. If `sep` is not specified, any whitespace string is a separator.
`s. startswith(sub)`	Returns `True` if `s` starts with `sub` or `False` otherwise.
`s.strip([aString])`	Returns a copy of `s` with leading and trailing whitespace (tabs, spaces, newlines) removed. If `aString` is given, remove characters in `aString` instead.
`s.upper()`	Returns a copy of `s` converted to uppercase.

Examples :

```
>>> s = "Hi there!"

>>> len(s)
```

```
9
>>> s.center(11)
' Hi there! '
>>> s.count('e')
2
>>> s.endswith("there!")
True
>>> s.startswith("Hi")
True
>>> s.find("the")
3
>>> s.isalpha()
False
>>> 'abc'.isalpha()
True
>>> "326".isdigit()
True
>>> words = s.split()
>>> words
['Hi', 'there!']
>>> " ".join(words)
'Hithere!'
>>> " ". join(words)
'Hi there!'
>>> s.lower()
'hi there!'
>>> s.upper()
```

'HI THERE!'

>>> s.replace('i', 'o')

'Ho there!'

>>> " Hi there! ".strip()

'Hi there!'

6) Data Encryption

- Data travelling on the Internet is vulnerable to spies and potential thieves.
- It is easy to observe data crossing a network, particularly now that more and more communications involve wireless transmissions.
  - For example, a person can sit in a car in the parking lot outside any major hotel and pick up transmissions between almost any two computers if that person runs the right **sniffing software**
  - **Data encryption** is a security method where information is encoded and can only be accessed or decrypted by a user with the correct encryption key.
- The sender encrypts a message by translating it to a secret code, called a **cipher text**.
- At the other end, the receiver decrypts the cipher text back to its original **plaintext** form.
- Both parties to this transaction must have at their disposal one or more keys that allow them to encrypt and decrypt messages.

6.1 Caesar cipher:

- This encryption strategy replaces each character in the plaintext with the character that occurs a given distance away in the sequence.
- For positive distances, the method wraps around to the beginning of the sequence to locate the replacement characters for those characters near its end.
- For example, if the distance value of a Caesar cipher equals three characters, the string "invaders" would be encrypted as “lqydghuv“
- To decrypt this cipher text back to plaintext, you apply a method that uses the same distance value but looks to the left of each character for its replacement.
- This decryption method wraps around to the end of the sequence to find a replacement character for one near its beginning

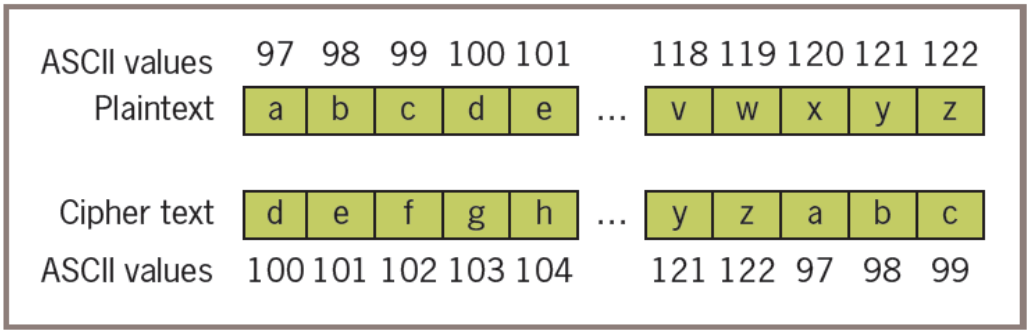


Figure 4-2 A Caesar cipher with distance +3 for the lowercase alphabet

Python Script to encrypt plain text to cipher text using Caesar Cipher

```
plainText = input("Enter a one-word, lowercase message: ")
distance = int(input("Enter the distance value: "))
code = ""
for ch in plainText:
    ordvalue = ord(ch)
    cipherValue = ordvalue + distance
    if cipherValue > ord('z'):
        cipherValue = ord('a') + distance - (ord('z') - ordvalue + 1)
    code += chr(cipherValue)
print(code)
```

OUTPUT :

```
>>>Enter a one-word, lowercase message: python
Enter the distance value: 3
sbwkrq
>>>Enter a one-word, lowercase message: xyz
Enter the distance value: 3
abc
```

Python Script to encrypt cipher text to plain text using Caesar Cipher

```
code = input("Enter the coded text: ")
distance = int(input("Enter the distance value: "))
plainText = ""
for ch in code:
    ordvalue = ord(ch)
    cipherValue = ordvalue - distance
    if cipherValue < ord('a'):
        cipherValue = ord('z') - (distance + (ord('a') - ordvalue - 1))
    plainText += chr(cipherValue)
print(plainText)
```

OUTPUT :

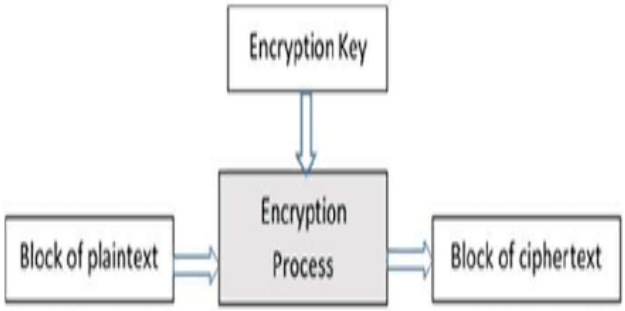
```
>>>Enter the coded text: sbwkrq
Enter the distance value: 3
python
>>>Enter the coded text: abc
Enter the distance value: 3
xyz
```

- The main shortcoming of this encryption strategy is that the plaintext is encrypted one character at a time, and each encrypted character depends on that single character and a fixed distance value.
- In a sense, the structure of the original text is preserved in the cipher text, so it might not be hard to discover a key by visual inspection.

6.2 Block cipher:

- A block cipher uses plaintext characters to compute two or more encrypted characters.
- This is accomplished by using a mathematical structure known as an **invertible matrix to determine the values of** the encrypted characters.

- The matrix provides the key in this method. The receiver uses the same matrix to decrypt the cipher text.
- The fact that information used to determine each character comes from a block of data makes it more difficult to determine the key



UNIT- 3

LISTS , DICTIONARIES, FUNCTIONS AND MODULES

List and Dictionaries: Lists, Defining Simple Functions, Dictionaries.
Design with Function: Functions as Abstraction Mechanisms, Problem Solving with Top Down Design, Design with Recursive Functions, Case Study Gathering Information from a File System, Managing a Program's Namespace, Higher Order Function.
Modules: Modules, Standard Modules, Packages

Lists :

- A list is a sequence of data values called items or elements. An item can be of any type.
- Here are some real-world examples of lists:
  - A shopping list for the grocery store
  - A to-do list
  - A roster for an athletic team
  - A guest list for a wedding
  - A recipe, which is a list of instructions
  - A text document, which is a list of lines
  - The names in a phone book
- Each of the items in a list is ordered by position.
- Like a character in a string, each item in a list has a unique index that specifies its position.
- The index of the first item is 0, and the index of the last item is the length of the list minus 1

List Literals and Basic Operators :

- In Python, a list literal is written as a sequence of data values separated by commas.
- The entire sequence is enclosed in square brackets ([and]).
- Here are some example list literals:
  - [1951, 1969, 1984] # A list of integers
  - ["apples", "oranges", "cherries"] # A list of strings
  - [] # An empty list
- You can also use other lists as elements in a list, thereby creating a list of lists. Here is one example of such a list:
  - [[5, 9], [541, 78]]
- The Python interpreter evaluates a list literal, and each of the elements are also evaluated if required

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

```
>>> import math
>>> x = 2
>>> [x, math.sqrt(x)]
[2, 1.4142135623730951]
>>> [x + 1]
[3]
```

- You can also build lists of integers using the range and list functions

```
>>> second = list(range(1, 5))
>>> second
[1, 2, 3, 4]
```

- The list function can build a list from any iterable sequence of elements, such as a string:

```
>>> third = list("Hi there!")
>>> third
['H', 'i', ' ' , 't', 'h', 'e', 'r', 'e', '!']
```

List Methods :

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

- **append() and extend()** :
  - The method append expects just the new element as an argument and adds the new element to the end of the list.
  - The method extend performs a similar operation, but adds the elements of its list argument to the end of the list.

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.append(3)
>>> example
[1, 2, 3]
>>> example.extend([11, 12, 13])
>>> example
[1, 2, 3, 11, 12, 13]
>>> example + [14, 15]
[1, 2, 3, 11, 12, 13, 14, 15]
>>> example
[1, 2, 3, 11, 12, 13]
```

- **pop() :**
  - The method pop is used to remove an element at a given position. If the position is not specified, pop removes and returns the last element.
  - In that case, the elements that followed the removed element are shifted one position to the left

```
>>> example
[1, 2, 10, 11, 12, 13]
>>> example.pop()              # Remove the last element
13
>>> example
[1, 2, 10, 11, 12]
>>> example.pop(0)             # Remove the first element
1
>>> example
[2, 10, 11, 12]
```

- **Searching a List**
  - first use the **in** operator to test for presence and then the **index** method if this test returns True.
  - The next code segment shows how this is done for an example list and target element:

```
aList = [34, 45, 67]
target = 45
if target in aList:
        print(aList.index(target))
else:
        print(-l)
```

- **Sorting a List** :
  - When the elements can be related by comparing them for less than and greater than as well as equality, they can be sorted.
  - The list method **sort** mutates a list by arranging its elements in ascending order

```
>>> example = [4, 2, 10, 8]
>>> example
[4, 2, 10, 8]
>>> example.sort()
>>> example
[2, 4, 8, 10]
```

NOTE:

- **Mutator Methods and the Value None :**
  - Mutable objects (such as lists) have some methods devoted entirely to modifying the internal state of the object. Such methods are called **mutators. Examples** are the list methods **insert, append, extend, pop**, and **sort**

Dictionaries:

- A dictionary organizes information by **association, not position.**
- **For example, when you** use an english dictionary to look up the definition of "mammal," you don't start at page 1; instead, you turn directly to the words beginning with "M."
  - Phone books, address books, encyclopedias, and other reference sources also organize information by association.
- In computer science, data structures organized by association are also called **tables or association lists.**
- **In** Python, a **dictionary associates a set of keys with values.**

Dictionary Literals:

- A Python dictionary is written as a sequence of key/value pairs separated by commas.
- These pairs are sometimes called **entries. The entire sequence of entries is enclosed in curly braces** ({ and }).
- A colon (:) separates a key and its value. Here are some example dictionaries:
  - phonebook= {"Savannah":"476-3321", "Nathaniel":"351-7743"}
  - Info={"Name":"Molly", "Age":18}
- You can even create an empty dictionary—that is, a dictionary that contains no entries.
  - {}

Adding Keys and Replacing Values :

- You add a new key/value pair to a dictionary by using the subscript operator []. The form of this operation is the following:
  - » <a dictionary>[<a key>] = <a value>
- The next code segment creates an empty dictionary and adds two new entries:

```
>>> info = {}
>>> info["name"] = "Sandy"
>>> info["occupation"] = "hacker"
>>> info
{'name':'Sandy', 'occupation':'hacker'}
```

- The subscript is also used to replace a value at an existing key, as follows:

```
>>> info["occupation"] = "manager"
>>> info
{'name':'Sandy', 'occupation':'manager'}
```

- The same operation is used for two different purposes: insertion of a new entry and modification of an existing entry.

Accessing Values :

- You can also use the subscript to obtain the value associated with a key. However, if the key is not present in the dictionary, Python raises an exception.

```
>>>info["name"]
'Sandy'
>>> info["job"]
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
```

```
info["job"]
KeyError: 'job'
```

Removing Keys :

- To delete an entry from a dictionary, one removes its key using the method pop.
- This method expects a key and an optional default value as arguments.
- If the key is in the dictionary, it is removed, and its associated value is returned. Otherwise, the default value is returned

Traversing a Dictionary :

- When a for loop is used with a dictionary, the loop's variable is bound to each key in an unspecified order. The next code segment prints all of the keys and their values in our info dictionary:

```
>>>info ={"name":"Surya","phone":9876543211}
>>>for key in info:
        print(key, info[key])
phone 9876543211
name Surya
```

- The entries are represented as tuples within the list. A tuple of variables can then access the key and value of each entry in this list within a for loop:

```
>>>for (key, value) in info.items():
        print(key, value)
```

Gives same output as the previous one

- On each pass through the loop, the variables key and value within the tuple are assigned the key and value of the current entry in the list. The use of a structure containing variables to access data within another structure is called **pattern matching**.

Dictionary Operations :

Dictionary Operation	What It Does
`len(d)`	Returns the number of entries in **d**.
`d[key]`	Used for inserting a new key, replacing a value, or obtaining a value at an existing key.
`d.get(key [, default])`	Returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
`d.pop(key [, default])`	Removes the key and returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
`list(d.keys())`	Returns a list of the keys.
`list(d.values())`	Returns a list of the values.
`list(d.items())`	Returns a list of tuples containing the keys and values for each entry.
`d.clear()`	Removes all the keys.
`for key in d:`	**key** is bound to each key in **d** in an unspecified order.

```
d={1:1,2:2**3,3:3**3,4:4**3,5:5**3,6:6**3}
>>> d
{1: 1, 2: 8, 3: 27, 4: 64, 5: 125, 6: 216}
>>> len(d)
6
>>> d[5]
125
>>> d.get(4)
64
>>>d.pop(4)
64
>>> d
{1: 1, 2: 8, 3: 27, 5: 125, 6: 216}
>>> d.keys()
dict_keys([1, 2, 3, 5, 6])
>>> d.values()
dict_values([1, 8, 27, 125, 216])
>>> d.items()
dict_items([(1, 1), (2, 8), (3, 27), (5, 125), (6, 216)])
```

```
>>> d.clear()
>>> d
{}
```

Conversion of hexadecimal to Binary:

- The algorithm visits each digit in the hexadecimal number, selects the corresponding four bits that represent that digit in binary, and adds these bits to a result string.
- If you maintain the set of associations between hexadecimal digits and binary digits in a dictionary, then you can just look up each hexadecimal digit's binary equivalent with a subscript operation. Such a dictionary is sometimes called a lookup table. Here is the definition of the lookup table required for hex-to-binary conversions:
- hexToBinaryTable = {'0':'0000', '1':'0001', '2':'0010', '3':'0011', '4':'0100', '5':'0101', '6':'0110', '7':'0111', '8':'1000', '9':'1001', 'A':'1010', 'B':'1011', 'C':'1100', 'D':'1101', 'E':'1110', 'F':'1111'}

```
def   convert(number, table):
binary = ""
for digit in number:
        binary = binary + table[digit]
return binary

>>> convert("35A", hexToBinaryTable)
'001101011010'
```

FUNCTIONS:

1. Design with Function

- A function packages an algorithm in a chunk of code that you can call by name
- A function can be called from anywhere in a program's code, including code within other functions
- A function can receive data from its caller via **arguments**
- When a function is called, any expressions supplied as arguments are first evaluated.
- Their values are copied to temporary storage locations named by the parameters in the function's definition
- A function may have one or more **return** statements, whose purpose is to terminate the execution of the function and return control to its caller. A return statement may be followed by an expression.

1.1 Functions as Abstraction Mechanisms

- Human brain can wrap itself around just a few things at once , People cope with complexity by developing a mechanism to simplify or hide it. This mechanism is called an **abstraction.**
- An abstraction hides detail and thus allows a person to view many things as just one thing
- **"doing my laundry" :** This expression is simple, but it refers to a complex process that involves
  - fetching dirty clothes from the hamper,
  - separating them into whites and colors,
  - loading them into the washer,
  - Transferring them to the dryer, and
  - folding them and
  - putting them into the dresser
- Without abstractions, most of our everyday activities would be impossible to discuss, plan, or carry out. Likewise, effective designers must invent useful abstractions to control complexity.

1.2 Functions Eliminate Redundancy

- The first way that functions serve as abstraction mechanisms is by eliminating redundant, or repetitious, code.
- To explore the concept of redundancy, let's look at a function named summation, which

returns the sum of the numbers within a given range of numbers.

```
def summation(lower, upper):
    result = 0
    while lower <= upper:
        result += lower
        lower += 1
    return result
```

```
>>> summation(1,4) # The summation of the numbers 1..4
10
>>> summation(50,100) # The summation of the numbers 50..100
3825
```

- Code redundancy is bad for several reasons. For one thing, it requires the programmer to laboriously enter or copy the same code over and over, and to get it correct every time.
- Then, if the programmer decides to improve the algorithm by adding a new feature or making it more efficient, he or she must revise each instance of the redundant code throughout the entire program leading to many maintainance problems
- By relying on a single function definition, instead of multiple instances of redundant code, the programmers free themselves to write only a single algorithm in just one place—say, in a library module.
- Any other module or program can then import the function for its use. Once imported, the function can be called as many times as necessary.
- When the programmer needs to debug, repair, or improve the function, she needs to edit and test only the single function definition. There is no need to edit the parts of the program that call the function

1.3 Functions Hide Complexity

- Functions serve as abstraction mechanisms is by hiding complicated details.
- A function call expresses the idea of a process to the programmer, without forcing him or her to wade through the complex code that realizes that idea
  - In summation function, although the idea of summing a range of numbers is simple, the code for computing a summation is not.

- There are three variables to manipulate, as well as count-controlled loop logic to construct.

1.4 Functions Support General Methods with Systematic Variations

- An algorithm is a **general method for solving a class of problems. The individual problems** that make up a class of problems are known as **problem instances.**
- The problem instances for the summation algorithm are the pairs of numbers that specify the lower and upper bounds of the range of numbers to be summed.
- The summation function contains both the code for the summation algorithm and the means of supplying problem instances to this algorithm. The problem instances are the data sent as arguments to the function.

1.5 Functions Support the Division of Labor

- In a computer program, functions can enforce a division of labor.
- Ideally, each function performs a single coherent task, such as computing a summation or formatting a table of data for output.
- Each function is responsible for using certain data, computing certain results, and returning these to the parts of the program that requested them.
- Each of the tasks required by a system can be assigned to a function, including the tasks of managing or coordinating the use of other functions.

2 . Problem Solving with Top-Down Design

- The top down strategy starts with a global view of the entire problem and breaks the problem into smaller, more manageable sub problems—a process known as **problem decomposition.**
- **As each subproblem is isolated, its solution is** assigned to a function. Problem decomposition may continue down to lower levels, because a subproblem might in turn contain two or more lower-level problems to solve.
- As functions are developed to solve each subproblem, the solution to the overall problem is gradually filled out in detail. This process is also called **stepwise refinement.**

2.1 The Design of the Text-Analysis Program

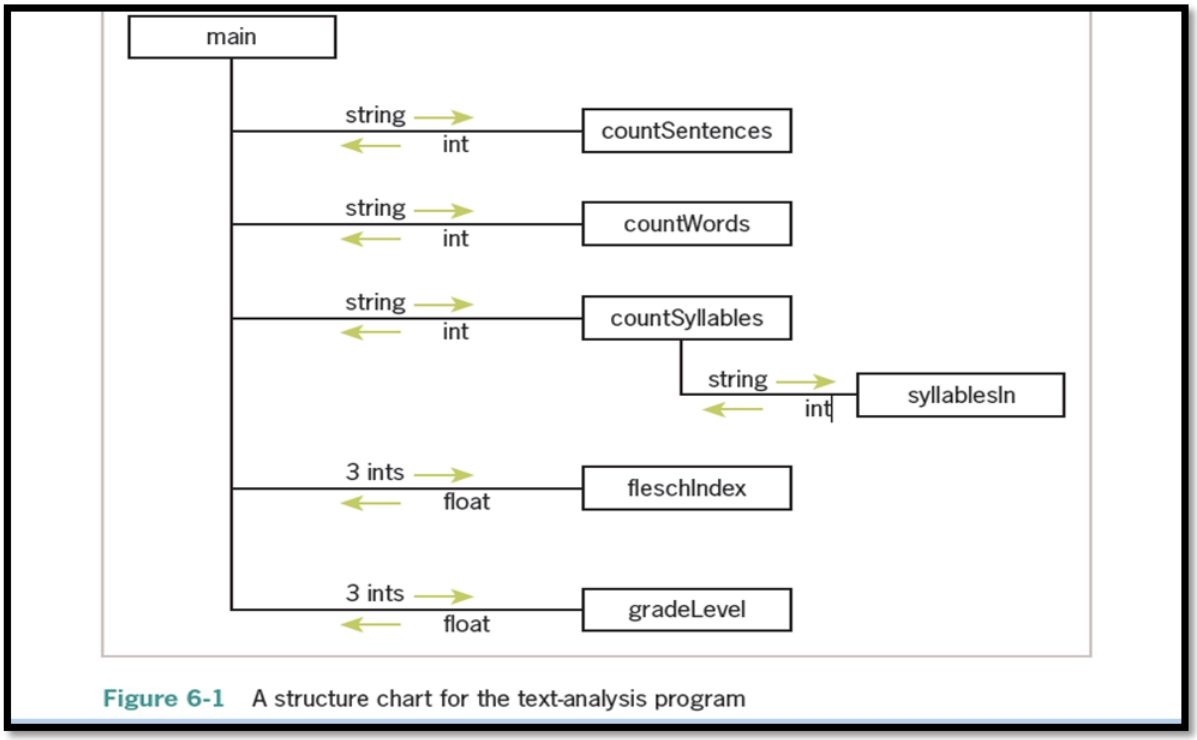


Figure 6-1 A structure chart for the text-analysis program

- The program requires simple input and output components, so these can be expressed as statements within a main function.
- The processing of the input is complex enough to decompose into smaller subprocesses, such as obtaining the counts of the sentences, words, and syllables and calculating the readability scores.
- We develop a new function for each of these computational tasks. The relationships among the functions in this design are expressed in the structure chart

Structure chart

- A **structure chart is a** diagram that shows the relationships among a program's functions and the passage of data between them.
- Each box in the structure chart is labeled with a function name. The main function at the top is where the design begins, and decomposition leads us to the lower-level functions on which main depends.
- The lines connecting the boxes are labeled with data type names, and arrows indicate the flow of data between them. For example, the function countSentences takes a string as an argument and returns the number of sentences in that string.
- Note that all functions except one are just one level below main

3 . Design with Recursive Functions

- In some cases of top down design , you can decompose a complex problem into smaller problems of the same form.
  - In these cases, the subproblems can all be solved by using the same function. This design strategy is called **recursive design, and the resulting** functions are called **recursive functions.**

Defining a Recursive Function :

- A recursive function is a function that calls itself.
- To prevent a function from repeating itself indefinitely, it must contain at least one selection statement. This statement examines a condition called a **base case** to determine whether to stop or to continue with another **recursive step.**

#Python **recursive** function for summation

```
def  summation(lower, upper):
    """Returns the sum of the numbers from lower through upper."""
    if   lower > upper:
        return 0
    else:
        return lower + summation (lower + 1, upper)
```

The recursive call of summation adds up the numbers from lower + 1 through upper .The function then adds lower to this result and returns it.

Using Recursive Definitions to Construct Recursive Functions

- A recursive definition consists of equations that state what a value is for one or more base cases and one or more recursive cases.
- For example, the Fibonacci sequence is a series of values with a recursive definition. The first and second numbers in the Fibonacci sequence are 1. Thereafter, each number in the sequence is the sum of its two predecessors, as follows:

  1 1 2 3 5 8 13 . . .
- More formally, a recursive definition of the *nth* ***Fibonacci number is the following:***

  Fib(n) = 1, when n = 1 or n = 2

  Fib(n) = Fib(n - 1) + Fib(n - 2), for all n > 2
- Given this definition, you can construct a recursive function that computes and returns

the *nth Fibonacci number. Here it is:*

```
def  fib(n):
        """Returns the nth Fibonacci number."""
        if n < 3:
                return 1
        else:
                return fib(n - 1) + fib(n - 2)
```

Infinite Recursion:

- Infinite recursion arises when the programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process.
- In fact, the Python virtual machine eventually runs out of memory resources to manage the process, so it halts execution with a message indicating a **stack overflow error.**
- **The next session defines a** function that leads to this result:

```
def runForever(n):
        if n > 0:
                runForever(n)
        else:
                runForever(n - 1)
>>> runForever(1)
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
runForever(1)
File "<pyshell#5>", line 3, in runForever
runForever(n)
File "<pyshell#5>", line 3, in runForever
runForever(n)
File "<pyshell#5>", line 3, in runForever
runForever(n)
[Previous line repeated 989 more times]
File "<pyshell#5>", line 2, in runForever
if n > 0:
```

RecursionError: maximum recursion depth exceeded in comparison

The PVM keeps calling runForever(1) until there is no memory left to support another recursive call. Unlike an infinite loop, an infinite recursion eventually halts execution with an error message.

The Costs and Benefits of Recursion :

- The run-time system on a real computer, such as the PVM(Python Virtual Machine), must devote some overhead to recursive function calls.
- At program startup, the PVM reserves an area of memory named a **call stack. For each call of a function,** recursive or otherwise, the PVM must allocate on the call stack a small chunk of memory called a **stack frame.**
- In this type of storage, the system places the values of the arguments and the return address for each function call. Space for the function call's return value is also reserved in its stack frame.
- When a call returns or completes its execution, the return address is used to locate the next instruction in the caller's code, and the memory for the stack frame is deallocated.
- When, because of a design error, the recursion is infinite, the stack frames are added until the PVM runs out of memory, which halts the program with an error message.

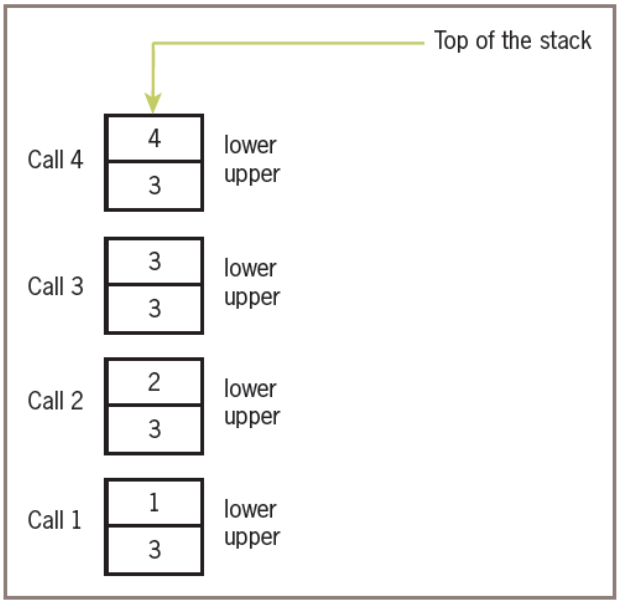


Figure 6-4 The stack frames for `displayRange(1, 3)`

4. Case Study Gathering Information from a File System

- Modern file systems come with a graphical browser, allowing the user to navigate to files or folders by selecting icons of folders, opening these by double-clicking, and selecting commands from a drop-down menu. Information on a folder or a file, such as the size and contents, is also easily obtained in several ways.
- Users of terminal-based user interfaces must rely on entering the appropriate commands at the terminal prompt to perform these functions.
- In this case study, we develop a simple terminal-based file system navigator that provides some information about the system.
- In the process, we will have an opportunity to exercise some skills in top-down design and recursive design.

Request:

Write a program that allows the user to obtain information about the file system.

Analysis:

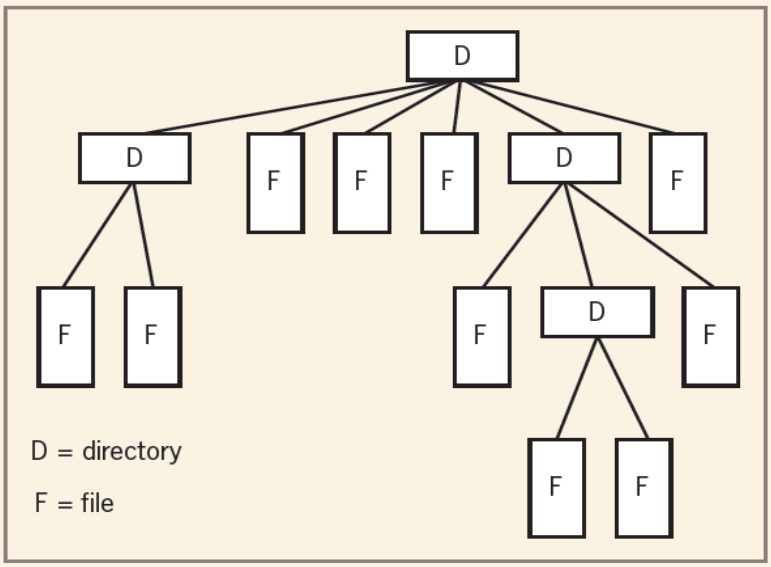


Figure 6-5 The structure of a file system

Command	What It Does
List the current working directory	Prints the names of the files and directories in the current working directory (CWD).
Move up	If the CWD is not the root, move to the parent directory and make it the CWD.
Move down	Prompts the user for a directory name. If the name is not in the CWD, print an error message; otherwise, move to this directory and make it the CWD.
Number of files in the directory	Prints the number of files in the CWD and all of its subdirectories.
Size of the directory in bytes	Prints the total number of bytes used by the files in the CWD and all of its subdirectories.
Search for a filename	Prompts the user for a search string. Prints a list of all the filenames (with their paths) that contain the search string, or "String not found."
Quit the program	Prints a signoff message and exits the program.

Table 6-1 The commands in the **filesys** program

```
import os, os.path
QUIT = '7'
COMMANDS = ('1', '2', '3', '4', '5', '6', '7')

MENU = """1 List the current directory
2 Move up
3 Move down
4 Number of files in the directory
5 Size of the directory in bytes
6 Search for a filename
7 Quit the program"""
def main():
    while True:
```

```
        print(os.getcwd())
        print(MENU)
        command = acceptCommand() #takes choice
        runCommand(command)
        if command == QUIT:
            print("Have a nice day!")
            break
def acceptCommand():
    """Inputs and returns a legitimate command number."""
    command = input("Enter a number: ")
    if command in COMMANDS:
        return command
    else:
        print("Error: command not recognized")
        return acceptCommand()
def runCommand(command):
    """Selects and runs a command."""
    if command == '1':
        listCurrentDir(os.getcwd())
    elif command == '2':
        moveUp()
    elif command == '3':
        moveDown(os.getcwd())
    elif command == '4':
        print("The total number of files is", \
        countFiles(os.getcwd()))
    elif command == '5':
        print("The total number of bytes is", \
        countBytes(os.getcwd()))
    elif command == '6':
        target = input("Enter the search string: ")
        fileList = findFiles(target, os.getcwd())
        if not fileList:
```

```
            print("String not found")
        else:
            for f in fileList:
                print(f)

def listCurrentDir(dirName):
    """Prints a list of the cwd's contents."""
    lyst = os.listdir(dirName)
    for element in lyst:
        print(element)

def moveUp():
    """Moves up to the parent directory."""
    os.chdir("..")

def moveDown(currentDir):
    """Moves down to the named subdirectory if it exists."""
    newDir = input("Enter the directory name: ")
    if os.path.exists(currentDir + os.sep + newDir) and os.path.isdir(newDir):
        os.chdir(newDir)
    else:
        print("ERROR: no such name")

def countFiles(path):
    """Returns the number of files in the cwd and all its subdirectories."""
    count = 0
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            count += 1
        else:
            os.chdir(element)
            count += countFiles(os.getcwd())
```

```
            os.chdir("..")
    return count
def countBytes(path):
    """Returns the number of bytes in the cwd and all its subdirectories."""
    count = 0
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            count += os.path.getsize(element)
        else:
            os.chdir(element)
            count += countBytes(os.getcwd())
            os.chdir("..")
    return count
def findFiles(target, path):
    """Returns a list of the filenames that contain the target string in the cwd and all its
    subdirectories."""
    files = []
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            if target in element:
                files.append(path + os.sep + element)
            else:
                os.chdir(element)
                files.extend(findFiles(target, os.getcwd()))
                os.chdir("..")
    return files

if __name__ == "__main__":
    main()
```

5. Managing a Program's Namespace

Namespaces in Python

- A namespace is a collection of currently defined symbolic names along with information about the object that each name references.
- You can think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves.
  - Each key-value pair maps a name to its corresponding object
- In a Python program, there are four types of namespaces:
  - Built-In
  - Global
  - Enclosing
  - Local

i) The Built-In Namespace

- The **built-in namespace** contains the names of all of Python's built-in objects. These are available at all times when Python is running.
- You can list the objects in the built-in namespace with the following command:

```
>>> dir(__builtins__)
```

The Python interpreter creates the built-in namespace when it starts up. This namespace remains in existence until the interpreter terminates.

ii) The Global Namespace

- The **global namespace** contains any names defined at the level of the main program.
- Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.
- The interpreter also creates a global namespace for any **module** that your program loads with the import statement.

iii) The Local and Enclosing Namespaces

The interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates

```
def f():
    print('Start f()')
```

```
def g():
    print('Start g()')
    print('End g()')
    return
g()
print('End f()')
return
```

Output :

```
>>> f()
Start f()
Start g()
End g()
End f()
```

- When the main program calls f(), Python creates a new namespace for f(). Similarly, when f() calls g(), g() gets its own separate namespace.
- The namespace created for g() is the **local namespace**, and the namespace created for f() is the **enclosing namespace**.
- Each of these namespaces remains in existence until its respective function terminates.

Scope:

- In Python, a name's scope is the area of program text in which the name refers to a given value
- In general, the meanings of temporary variables are restricted to the body of the functions in which they are introduced, and they are invisible elsewhere in a module.
- The restricted visibility of temporary variables befits their role as temporary working storage for a function.
- Although a Python function can reference a module variable for its value, it cannot under normal circumstances assign a new value to a module variable.
- When such an attempt is made, the PVM creates a new, temporary variable of the same name within the function.
- The following script shows how this works:

```
x = 5
def f():
```

```
    x = 10   # Attempt to reset x
f()                  # Does the top-level x change?
print(x)             # No, this displays 5
```

- When the function f is called, it does not assign 10 to the module variable x; instead, it assigns 10 to a temporary variable x.
- In fact, once the temporary variable is introduced, the module variable is no longer visible within function f. In any case, the module variable's value remains unchanged by the call

Lifetime:

- A variable's lifetime is the period of time during program execution when the variable has memory storage associated with it.
- When a variable comes into existence, storage is allocated for it; when it goes out of existence, storage is reclaimed by the PVM.
- The concept of lifetime explains the existence of two variables called x in our last example session.
  - The module variable x comes into existence before the temporary variable x and survives the call of function f.
  - During the call of f, storage exists for both variables, so their values remain distinct.
- **Using Keywords for Default and Optional Arguments:**
- The programmer can also specify **optional arguments with default values in any function** definition.
- Here is the syntax:

  def <function name>(<required arguments>, <key-1> = <val-1>, ... <key-*n*> = <*val-n*>)
- The required arguments are listed first in the function header. These are the ones that are "essential" for the use of the function by any caller.
- Following the required arguments are one or more **default arguments or keyword arguments.** These are assignments of values to the argument names. When the function is called without these arguments, their default values are automatically assigned to them. When the function is called with these arguments, the default values are overridden by the caller's values.
- When using functions that have default arguments, you must provide the required

arguments and place them in the same positions as they are in the function definition's header.

- The default arguments that follow can be supplied in two ways:

1. **By position**. In this case, the values are supplied in the order in which the arguments occur in the function header. Defaults are used for any arguments that are omitted.

2. **By keyword**. In this case, one or more values can be supplied in any order, using the syntax **<key> = <value>** in the function call.

- Here is an example of a function with one required argument and two default arguments and a session that shows these options:

```
>>> def example(required, option1 = 2, option2 = 3):
                    print(required, option1, option2)
>>> example(1)                  # Use all the defaults
1 2 3
>>> example(1, 10)    # Override the first default
1 10 3
>>> example(1, 10, 20)          # Override all the defaults
1 10 20
>>> example(1, option2 = 20) # Override the second default
1 2 20
>>> example(1, option2 = 20, option1 = 10) # In any order
1    10 20
```

6. Anonomyous Function or Lambda function:

- An anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.

lambda arguments: expression

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned.

EXAMPLE 1:

```
>>> d = lambda x: x * 2
>>> print(d(5))
10
```

- We use lambda functions when we require a nameless function for a short period of time.
- In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments).
- Lambda functions are used along with built-in functions like filter(), map() etc.

EXAMPLE 2 :Lambda with filter():

- The filter() function in Python takes in a function and a list as arguments.
- The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True

```
>>> my_list = [1, 5, 4, 6, 8, 11, 3, 12]
>>> new_list = list(filter(lambda x: (x%2 == 0) , my_list))
>>> print(new_list)
[4, 6, 8, 12]
```

EXAMPLE 3: Lambda with map():

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

```
>>> my_list = [1, 5, 4, 6, 8, 11, 3, 12]
>>> new_list=list(map(lambda x:x**2 , my_list))
>>> new_list
[1, 25, 16, 36, 64, 121, 9, 144]
```

7. Higher Order Functions

- A function is called **Higher Order Function** if it contains other functions as a parameter or returns a function as an output i.e, the functions that operate with another function are known as Higher order Functions
- The 3 mostly used higher order functions are:
  - map()
  - filter()
  - reduce()

map() :

- **map()** function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)
- **Syntax :**

  map(fun, iter)
- **Parameters :**
  - **fun :** It is a function to which map passes each element of give iterable.
  - **iter :** It is a iterable which is to be mapped.

 # Python program to demonstrate working of map.

```
# Return double of n
def addition(n):
    return n + n
# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

Output:

[2, 4, 6, 8]

- **filter()**
  - The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.
  - **Syntax:**

    filter(function, sequence)
  - **Parameters:**
    - function: function that tests if each element of a sequence true or not.
    - sequence: sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.
  - **Returns:** returns an iterator that is already filtered.

```
# function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
```

```
    if (variable in letters):
        return True
    else:
        return False
# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']
# using filter function
filtered = filter(fun, sequence)
print('The filtered letters are:')
for s in filtered:
    print(s)
```

OUTPUT :

The filtered letters are: e e

- **reduce() :**
  - The Python functools module includes a reduce function that expects a function of two arguments and a list of values. The reduce function returns the result of applying the function as just described.

  - The following example shows reduce used twice—once to produce a sum and once to produce a product:

```
>>> from functools import reduce
>>> def   add(x, y):
                return  x + y
>>> def   multiply(x, y):
                return  x * y
>>> data = [1, 2, 3, 4]
>>> reduce(add, data)
10
>>> reduce(multiply, data)
24
```

8. Modules in Python:

- Modules refer to a file containing Python statements and definitions.
- A file containing Python code, for example: example.py, is called a module, and its module name would be example
- Modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

User defined module :

Let us create a module. Type the following and save it as example.py.

```
# Python Module example

def add(a, b):

  """This program adds two  numbers and return the result"""

    result = a + b

     return result
```

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

```
>>> import example
```

This does not import the names of the functions defined in example directly in the current symbol table. It only imports the module name example there.

Using the module name we can access the function using the dot . operator. For example:

```
>>> example.add(4,5.5)

9.5
```

- Modules are imported by using import statement

Syntax:

i) import module_name

Example:

```
>>>import math

>>>print(math.sqrt(25))

5.0
```

ii) from….import statement:

A module may contain definition of many functions and variables.

When you import a module, you can use any variable or any function defined in that module but if we want to use only selective variables and functions then we will use the "from……import statement"

Syntax:

from module_name import function_name/variable_name

e.g.1

```
>>>from time import asctime
print(asctime())
```

Thu Aug 26 15:08:52 2021

e.g.2

```
>>>from math import pi
>>>print("pi= ", pi)
```

To import more than one item from the module, we use a comma separated list like below

```
from math import sqrt, pow
print(sqrt(25), pow(10,2))
```

iii) "as keyword":

To avoid the confusion in function names we use as keyword to give a alias name

e.g.

```
>>>from math import sqrt as square_root
>>>print(square_root(25))
```

Creating a module: num.py

```
def square(x):
    return(x*x)
def cube(x):
    return(x*x*x)
def power(x, y):
```

return(x**y)

Example program:

```
import  num
print(“Square of 10”,num.square(10))
print(“Cube of 10”,num.cube(10))
print(“Power of 10, 2 is “,num.power(10,5))
```

9. Packages in Python:

Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.

A package can contain one or more relevant modules. Physically, a package is actually a folder containing one or more module files

Creating a Package:

Let's create a package named mypackage, using the following steps:

- Create a new folder named C:\MyApp.
- Inside MyApp, create a subfolder with the name 'mypackage'.
- Create an empty __init__.py file in the mypackage folder.
- Using a Python-aware editor like IDLE, create modules greet.py and functions.py with the following code:

greet.py

```
def  SayHello(name):
    print("Hello ", name)
```

functions.py

```
def sum(x,y):
        return x+y
def average(x,y):
```

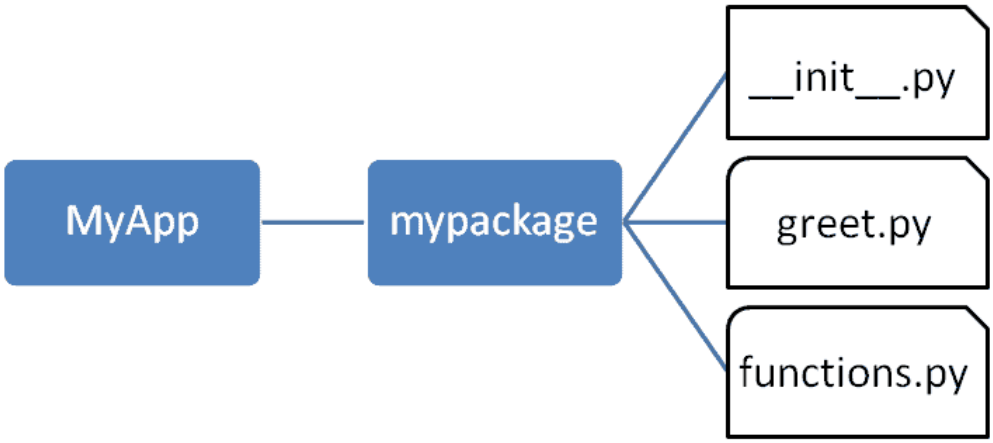
```
    return (x+y)/2

def power(x,y):

    return x**y
```

__init__.py :

The package folder contains a special file called __init__.py, which stores the package's content. It serves two purposes:

- The Python interpreter recognizes a folder as the package if it contains __init__.py file.
- __init__.py exposes specified resources from its modules to be imported.

An empty __init__.py file makes all functions from the above modules available when this package is imported. Note that __init__.py is essential for the folder to be recognized by Python as a package.



- Import the functions module from the mypackage package and call its power() function.

```
>>> from mypackage import functions

>>> functions.power(3,2)

9
```

- It is also possible to import specific functions from a module in the package.

```
>>> from mypackage.functions import sum

>>> sum(10,20)
```

```
30
>>> average(10,12)
Traceback (most recent call last):
File "<pyshell#13>", line 1, in <module>
NameError: name 'average' is not defined
```

UNIT-4

Syllabus:

File Operations: Understanding read functions, read (), readline () and readlines (), Understanding write functions, write () and writelines (), Manipulating file pointer using seek, Programming using file operations, Reading config files in python, Writing log files in python.

Object Oriented Programming: Concept of class, object and instances, Constructor, class attributes and destructors, Real time use of class in live projects, Inheritance, overlapping and overloading operators, Adding and retrieving dynamic attributes of classes, Programming using Oops support.

Design with Classes: Objects and Classes, Data modelling Examples, Case Study An ATM, Structuring Classes with Inheritance and Polymorphism.

Files in Python:

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files. Python provides us with an important feature for reading data from the file and writing data into a file. Mostly, in programming languages, all the values or data are stored in some variables which are volatile in nature. Because data will be stored into those variables during run-time only and will be lost once the program execution is completed. Hence it is better to save these data permanently using files. Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a file object.

Opening and Closing Files

The open () Method

Before you can read or write a file, you have to open it using Python's built-in open () function. This function creates a file object, which would be utilized to call other support methods associated with it.

Syntax: file object = open (filename, access mode)

Here are parameter details –

file_name − The file_name argument is a string value that contains the name of the file that you want to access.

access_mode − The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).

Here is a list of the different modes of opening a file –

Sno	Modes & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**rb** Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	**rb+** Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	**w** Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

6	**wb** Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
7	**w+** Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	**wb+** Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	**a** Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	**ab** Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11	**a+** Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12	**ab+** Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The file Object Attributes:

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all attributes related to file object −

Sno	Attribute & Description
1	**file.closed** Returns true if file is closed, false otherwise.
2	**file.mode** Returns access mode with which file was opened.
3	**file.name** Returns name of the file.

Example:

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file

#File object attributes

print('Name of the file: ', f.name)

print('Closed or not : ', f.closed)

print('Opening mode : ', f.mode)

f.close()
```

The close () Method

The close () method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done. It is a good practice to use the close () method to close a file.

Syntax: fileObject.close()

Example:

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file
#File object attributes
print('Name of the file: ', f.name)
print('Closed or not : ', f.closed)
print('Opening mode : ', f.mode)
f.close()
```

Reading and Writing Files

The file object provides a set of access methods. Now, we will see how to use read (), readline (), readlines () and write (), writelines () methods to read and write files.

Understanding write () and writelines ()

The write () Method

- The write () method writes any string (binary data and text data) to an open file.
- The write () method does not add a newline character ('\n') to the end of the string

Syntax: fileObject.write(string)

Here, passed parameter is the content to be written into the opened file.

Example:

```
f=open('sample.txt','w') #creates a new file sample.txt give write permissions on file
#writing content into file sample.txt using write method
f.write( "Python is a great language.")
f.close()
```

The writelines () method:

Python file method **writelines ()** writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value .

Syntax: fileObject.writelines(sequence)

Parameters

Sequence – This is the Sequence of the strings.

Return Value-This method does not return any value.

Example

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file

#writing content into file using write method

f.writelines (['python is easy\n','python is portable\n','python is  comfortable']

)

 f.close()
```

Understanding read (), readline () and readlines ():

The read () Method

The read () method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax: fileObject.read([count])

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

Example

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file
#writing content into file using write method

f.writelines(['python is easy\n','python is portable\n','python is comfortable'])


 f.close()

f=open('sample.txt','r')

#reading first 20 bytes from the file using read() method

print(f.read(20))
```

The readline () Method

Python file method **readline()**reads one entire line from the file. A trailing newline character is kept in the string. If the *size* argument is present and non-negative, it is a maximum byte count including the trailing newline and an incomplete line may be returned.

An empty string is returned only when EOF is encountered immediately.

Syntax: fileObject.readline(size)

Parameters

- **size** − This is the number of bytes to be read from the file.

Return Value

- This method returns the line read from the file.

Example

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file

#writing content into file using write method

f.writelines(['python is easy\n','python is portable\n','python is comfortable'])

f.close()
```

```
f=open('sample.txt','r')

#reading first line of the file using readline() method
print(f.readline())
```

The readlines () Method

Python file method **readlines()** reads until EOF using readline() and returns a list containing the lines. If the optional *sizehint* argument is present, instead of reading up to EOF, whole lines totalling approximately *sizehint* bytes (possibly after rounding up to an internal buffer size) are read.

An empty string is returned only when EOF is encountered immediately.

Syntax: fileObject.readlines(sizehint)

Parameters

- **sizehint** − This is the number of bytes to be read from the file.

Return Value

This method returns a list containing the lines.

Example

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file

#writing content into file using write method

f.writelines(['python is easy\n','python is portable\n','python is comfortable'])

 f.close()

f=open('sample.txt','r')

#reading all the line of the file using readlines() method

print(f.readlines())
```

Manipulating file pointer using seek():

tell (): The tell () method tells you the current position within the file

Syntax: file_object.tell()

Example:

```
# Open a file

fo = open("sample.txt", "r+")

str = fo.read(10)

print("Read String is : ", str)

# Check current position

position = fo.tell()

print("Current file position : ", position)

fo.close()
```

seek (): The seek (offset, from_what) method changes the current file position.

Syntax: f.seek(offset, from_what) #where f is file pointer

Parameters:

Offset: Number of postions to move forward

from_what: It defines point of reference.

Returns: Does not return any value

The reference point is selected by the **from_what** argument. It accepts three values:

0: sets the reference point at the beginning of the file

1: sets the reference point at the current file position

2: sets the reference point at the end of the file

By default from_what argument is set to 0.

Note: Reference point at current position / end of file cannot be set in text mode except when offset is equal to 0.

Example:

```
# Open a file
```

```
fo = open("sample.txt", "r+")

str = fo.read(10)
print("Read String is : ", str)

# Check current position

position = fo.tell()

print("Current file position : ", position)

# Reposition pointer at the beginning once again

position = fo.seek(0, 0);

str = fo.read(10)

print("Again read String is : ", str)

# Close opend file

fo.close()
```

File processing operations:

Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

i) os.rename(): The rename() method takes two arguments, the current filename and the new filename.(to rename file)

Syntax: os.rename(current_file_name, new_file_name)

Example:

```
import os

os.rename('sample.txt','same.txt')
```

ii) os.mkdir(): The mkdir() method takes one argument as directory name, that you want to

create.(This method is used to create directory)

Syntax: os.mkdir(directory name)

Example:

```
import os

os.mkdir('python') # Creates python named directory
```

iii) os.rmdir(): The rmdir() method takes one argument as directory name, that you want to remove.(This method is used to remove directory)

Syntax: os.rmdir(directory name)

Example:

```
import os

os.rmdir('python') # removes python named directory
```

iv) os.chdir(): The chdir() method takes one argument as directory name which we want to change.(This method is used to change directory)

Syntax: os.chdir(newdir)

Example:

```
import os

os.chdir('D:\>') # change directory to D drive
```

os.remove(): The remove() method takes one argument, the filename that you want to remove.(This method is used to remove file)

Syntax: os.remove(filename)

Example:

```
import os

os.remove('python,txt') # removes python.txt named file
```

os.getcwd(): The getcwd() method takes zero arguments,it gives current working director.

Syntax: os.getcwd()

Example:

```
import os
os.getcwd( ) # it gives current working directory
```

WRITING AND READING CONFIG FILES IN PYTHON

Config files help creating the initial settings for any project, they help avoiding the hardcoded data. For example, imagine if you migrate your server to a new host and suddenly your application stops working, now you have to go through your code and search/replace IP address of host at all the places. Config file comes to the rescue in such situation. You define the IP address key in config file and use it throughout your code. Later when you want to change any attribute, just change it in the config file. So this is the use of config file.

Creating and writing config file in Python

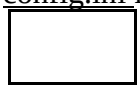
In Python we have configparser module which can help us with creation of config files (.ini format).

Program:

```
from configparser import ConfigParser

#Get the configparser object

config_object = ConfigParser()

#Assume we need 2 sections in the config file, let's call them USERINFO and SERVERCONFIG

config_object["USERINFO"] = {

 "admin": "Chankey Pathak",

 "loginid": "chankeypathak",

 "password": "tutswiki"

}

config_object["SERVERCONFIG"] = {
```

```
 "host": "tutswiki.com",

 "port": "8080",

 "ipaddr": "8.8.8.8"
}

#Write the above sections to config.ini file

with open('config.ini', 'w') as conf:

 config_object.write(conf)
```

Now if you check the working directory, you will notice config.ini file has been created, below



is its content.

```
[USERINFO]

admin = Chankey Pathak

password = tutswiki

loginid = chankeypathak


[SERVERCONFIG]

host = tutswiki.com

ipaddr = 8.8.8.8

port = 8080
```

Reading a key from config file:

So we have created a config file, now in your code you have to read the configuration data so that you can use it by "keyname" to avoid hardcoded data, let's see how to do that

Program:

```
from configparser import ConfigParser
```

```
#Read config.ini file

config_object = ConfigParser()

config_object.read("config.ini")

#Get the password
userinfo = config_object["USERINFO"]

print("Password  is{}".format(userinfo["password"]))
```

output:

Password is tutswiki

UNIT-4
PART-2

Object Oriented Programming: Concept of class, object and instances, Constructor, class attributes and destructors, Real time use of class in live projects, Inheritance, overlapping and overloading operators, Adding and retrieving dynamic attributes of classes, Programming using Oops support

Design with Classes: Objects and Classes, Data modelling Examples, Case Study An ATM, Structuring Classes with Inheritance and Polymorphism

Introduction

We have two programming techniques namely

1. Procedural-oriented programming technique
2. Object-oriented programming technique

Till now we have using the Procedural-oriented programming technique, in which our program is written using functions and block of statements which manipulate data. However a better style of programming is Object-oriented programming technique in which data and functions are combined to form a class. Object Oriented programming (OOP) is a programming paradigm that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects. There are many object-oriented programming languages including JavaScript, C++, Java, and Python.

Classes and objects are the main aspects of object oriented programming.

Overview of OOP Terminology

- **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** − A variable that is defined inside a method and belongs only to the

current instance of a class.

- **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** − The creation of an instance of a class.
- **Method** − A special kind of function that is defined in a class definition.
- **Object** − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** − The assignment of more than one function to a particular operator.

Benefits of OOP

- OOP models complex things as reproducible, simple structures
- Reusable, OOP objects can be used across programs
- Allows for class-specific behavior through polymorphism
- Easier to debug, classes often contain all applicable information to them
- Secure, protects information through encapsulation

Classes:

1. Class is a basic building block in python
2. Class is a blue print or template of a object
3. A class creates a new data type
4. And object is instance(variable) of the class
5. In python everything is an object or instance of some class

   Example :

   All integer variables that we define in our program are instances of class int. >>>

   a=10

   >>> type(a)

   <class 'int'>
6. The python standard library based on the concept of classes and objects

Defining a class:

Python has a very simple syntax of defining a class.

Syntax :

```
Class class-name:
    Statement1
    Statement2
    Statement3
    -
    -
    -
    Statement
```

From the syntax, Class definition starts with the keyword class followed by class-name and a colon(:). The statements inside a class are any of these following

1. Sequential instructions
2. Variable definitions
3. Decision control statements
4. Loop statements
5. Function definitions

Note : the class members are accessed through class object

Note : class methods have access to all data contained in the instance of the object

Creating objects: (creating an object of a class is known as class instantiation)

• Once a class is defined, the next job is to create a object of that class. • The object can then access class variables and class methods using dot operator

Syntax of object creation:

Object-name=class-name()

• Syntax for accessing class members through the class object is
Object-name.class-member-name

Example :

```
class ABC:
    a=10
obj=ABC()
print(obj.a)
```

self variable and class methods:

• Self refers to the object itself (Self is a pointer to the class instance)

• Whenever we define a member function in a class always use a self as a first argument and give rest of the arguments

• Even if it doesn't take any parameter or argument you must pass self to a member function

• We do not give a value for this parameter, when call the method, python will provide it.

• The self in python is equivalent to the this pointer in c++

Example 1 :

```
class Person:
        pc=0            # Class varibles
        def  setFullName(self,fName,lName):
                self.fName=fName # instance variables
                self.lName=lName      # instance variables


        def  printFullName(self):
                print(self.fName," ",self.lName)
                print("Person number : ",self.pc) #access Classvariable
PName=Person()                  #Object PName created
PName.setFullName("vamsi","kurama")
PName.pc=7                      #Attribute pc of PName modified
PName.printFullName()
P=Person()                      #Object P created
P.setFullName("Surya","Vinti")
P.pc=23                         #Attribute pc of P modified
P.printFullName()
```

Output:

```
>>>
vamsi   kurama
Person number :  7
Surya   Vinti
Person number :  23
```

Constructor method:

A constructor is a special type of method (function) that is called when it instantiates an object of a class. The constructors are normally used to initialize (assign values) to the instance variables.

Creating a constructor: (The name of the constructor is always the _ _init_ _().)

The constructor is always written as a function called __init__(). It must always take as its first argument a reference to the instance being constructed.

While creating an object, a constructor can accept arguments if necessary. When you create a class without a constructor, Python automatically creates a default constructor that doesn't do anything.

Every class must have a constructor, even if it simply relies on the default

constructor. Example:

```
class Person:
        pc=0            # Class varibles
        def __init__(self):
                print("Constructor initialised ")
                self.fName="XXXX"
                self.lName="YYYY"
        def setFullName(self,fName,lName):
                self.fName=fName # instance variables
                self.lName=lName      # instance variables

        def printFullName(self):
                print(self.fName," ",self.lName)
                print("Person number : ",self.pc) #access Classvariable

PName=Person()
PName.printFullName()
PName.setFullName("vamsi","kurama")
PName.pc=7
print("After setting Name:")
PName.printFullName()
```

Output:

>>>

Constructor initialised

XXXX YYYY

Person number : 0

After setting Name:

vamsi kurama

Person number : 7

Destructor:

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically. The _ _ del _ _ ( ) method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Syntax of destructor declaration:

```
def __del__(self):
 # body of destructor
```

Note: A reference to objects is also deleted when the object goes out of reference or when the program ends.

Example 1: Here is the simple example of destructor. By using del keyword we deleted the all references of object 'obj', therefore destructor invoked automatically.

```
# Python program to illustrate destructor
class Employee:

        # Initializing
        def  __init__(self):
                print('Employee created.')

        # Deleting (Calling destructor)
        def  __del__(self):
                print('Destructor called, Employee deleted.')

obj = Employee()
del obj
```

Output:

Employee created

Destructor called, Employee deleted

Inheritance:

One of the major advantages of Object Oriented Programming is reusability. Inheritance is one of the mechanisms to achieve the reusability. Inheritance is used to implement is-a relationship.

Definition: A technique of creating a new class from an existing class is called inheritance. The old or existing class is called base class or super class and a new class is called sub class or derived class or child class.

The derived class inherits all the variable and methods of the base class and adds their own variables and methods. In this process of inheritance base class remains unchanged.

Syntax to inherit a class:

Class MySubClass(object):

Pass(Body-of-the-derived-class)

Example :

```
class Pet:
        def __init__(self,name,age):
                self.name=name
                self.age=age
class Dog(Pet):
        def sound(self):
                print("I am {} and My age is {} and I sounds
                Like".format(self.name,self.age)) print("Bow Bow..")
class Cat(Pet):
        def sound(self):
                print("I am {} and My age is {} and I sounds
                Like".format(self.name,self.age)) print("Meow Meow..")
class Parrot(Pet):
        def sound(self):
                print("Hello I am {} and My age is {} ".format(self.name,self.age))
p1=Dog("Dozer",4)
p2=Cat("Edward",3)
p3=Parrot("Jango",6)
p1.sound()
p2.sound()
p3.sound()
```

Example 2:

```
class Person:
        def __init__(self,name,age):
                self.name=name
                self.age=age
        def display(self):
                print("name=",self.name)
```

```
                print("age=",self.age)
class Teacher(Person):
        def __init__(self,name,age,exp,r_area):
                Person.__init__(self,name,age)
                self.exp=exp
                self.r_area=r_area
        def displayData(self):
                Person.display(self)
                print("Experience=",self.exp)
                print("Research area=",self.r_area)
class Student(Person):
        def __init__(self,name,age,course,marks):
                Person.__init__(self,name,age)
                self.course=course
                self.marks=marks
        def displayData(self):
                Person.display(self)
                print("course=",self.course)
                print("marks=",self.marks)
print("********TEACHER***********")
t=Teacher("jai",55,13,"cloud computing")
t.displayData()
print("********STUDENT***********")
s=Student("hari",21,"B.Tech",99)
s.displayData()
```

Types of inheritance:

Python supports the following types of inheritan:

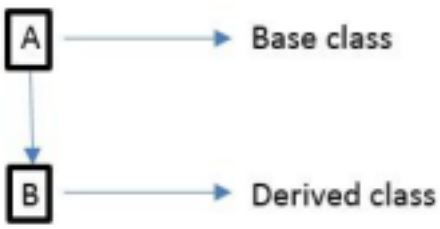
i) Single inheritance
ii) Multiple Inheritance
iii) Multi-level Inheritance
iv) Multi path Inheritance

Single Inheritance:

When a derived class inherits features form only one base class, it is called Single inheritance.

Syntax:

```
class Baseclass:
        <body of base class>
class Derivedclass(Baseclass):
        <body of the derived class>
```



Example:

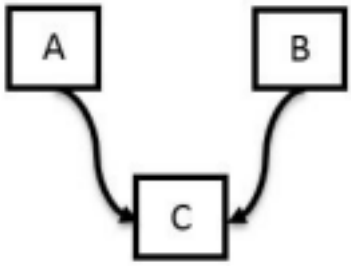
```
class A:
        i=10
class B(A):
        j=20
```

```
obj=B()
print("member of class A is",obj.i)
print("member of class B is",obj.j)
```

Multiple Inheritance:

When derived class inherits features from more than one base class then it is called Multiple Inheritance.

Syntax:




```
class Baseclass1:
        <body of base class1>
class Baseclass2:
        <body of base class2>
class Derivedclass(Baseclass1,Baseclass2):
        <body of the derived class>
```

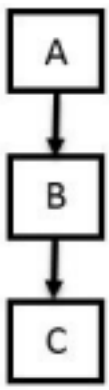
e.g.

```
class A:
        i=10
class B:
        j=20
class C(A,B):
        k=30
obj=C()
print("member of class A is",obj.i)
print("member of class B is",obj.j)
print("member of class C is",obj.k)
```

Multi-Level Inheritance:

When derived class inherits features from other derived classes then it is called Multi-level inheritance.

Syntax:




```
class Baseclass:
        <body of base class>
class Derivedclass1(Baseclass):
        <body of derived class 1>
class Derivedclass2(Derivedclass1):
        <body of the derived class2>
```

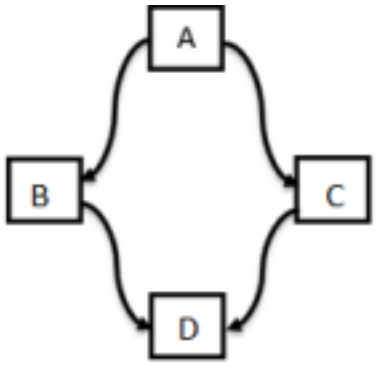
e.g.

```
class A:
        i=10
class B(A):
        j=20
class C(B):
        k=30
obj=C()
print("member of class A is",obj.i)
print("member of class B is",obj.j)
print("member of class C is",obj.k)
```

Multi Path Inheritance:

Syntax:

```
class Baseclass:
        <body of the base class>
class Derived1(Baseclass):
        <body of the derived1>
class Derived2(Baseclass):
        <body of the derived2>
class Derived3 (Derived1,Derived2) :
        <body of derived3>
```



e.g.

```
class A:
        i=10
class B(A):
        j=20
class C(A):
        k=30
class D(B,C):
        ijk=40
obj=D()
print("member of class A is",obj.i)
print("member of class B is",obj.j)
print("member of class C is",obj.k)
print("member of class Cis",obj.ijk)
```

Polymorphism:

The word polymorphism means having many forms. In python we can find the same operator or function taking multiple forms. That helps in re using a lot of code and decreases code complexity.

Polymorphism in operators

- The + operator can take two inputs and give us the result depending on what the inputs are.
- In the below examples we can see how the integer inputs yield an integer and if one of the input is float then the result becomes a float. Also for strings, they simply get concatenated.

Example:

```
a = 23
b = 11
c = 9.5
s1 = "Hello"
s2 = "There!"
print(a + b)
print(type(a + b))
print(b + c)
print(type (b + c))
print(s1 + s2)
print(type(s1 + s2))
```

Polymorphism in built-in functions

We can also see that different python functions can take inputs of different types and then process them differently. When we supply a string value to len() it counts every letter in it. But if we give tuple or a dictionary as an input, it processes them differently.

Example:

```
str = 'Hi There !'
tup = ('Mon','Tue','wed','Thu','Fri')
lst = ['Jan','Feb','Mar','Apr']
dict = {'1D':'Line','2D':'Triangle','3D':'Sphere'}
print(len(str))
print(len(tup))
print(len(lst))
print(len(dict))
```

Polymorphism in inheritance:

Method Overriding:

It is nothing but same method name in parent and child class with different functionalities. In inheritance only we can achieve method overriding. If super and sub classes have the same method name and if we call the overridden method then the method of corresponding class (by using which object we are calling the method) will be executed.

e.g.

```
class A:
        i=10
        def display(self):
                print("I am class A and I have data",self.i)
class B(A):
        j=20
        def display(self):
                print("I am class B and I have data",self.j)
obj=B()
obj.display()
```

OUTPUT :

I am class B and I have data 20

Note: In above program the method of class B will execute. If we want to execute method of class A by using Class B object we use super() concept.

Super():

In method overriding , If we want to access super class member by using sub class object we use super()

e.g

```
class A:
        i=10
        def display(self):
                print("I am class A and I hava data",self.i)
class B(A):
        j=20
```

```
    def display(self):
        super().display()
        print("I am class B and I hava data",self.j)
obj=B()
obj.display()
```

OUTPUT:

I am class A and I have data 10

I am class B and I have data 20

Note: In above example both the functions (display () in class A and display () in class B) will execute

Note: Name mangling is the encoding of function and variable names into unique names so that linkers can separate common names in the language.

overloading operators

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example, operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading.

```
# Python program to show use of + and * operator for different purposes.

print(1 + 2)
# concatenate two strings
print("Learn"+"For")
# Product two numbers
print(3 * 4)
# Repeat the String
print("Learn"*4)
```

Output:

3

LearnFor

12

LearnLearnLearnLearn

Example 2:

Changing the behavior of operator is as simple as changing the behavior of method or function. You define methods in your class and operators work according to that behavior defined in methods. When we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.

```
class A:
        def  __init__(self, a):
                self.a = a
        def  __add__(self, o):                  # adding two objects
                return  self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("sai")
ob4 = A("kumar")
ob5=A([2,5,6,2])
ob6=A([34.6,12])
print(ob1 + ob2)
print(ob3 + ob4)
print(ob5 + ob6)
```

```
Ob1.a=1
Ob2.a=2
Ob3.a=”sai”
Ob4.a=”kumar”
Ob5.a=[ 2,5,6,2    ]
Ob6.a=[  34.6,12 ]
```

OUTPUT:

```
>>>
3
saikumar
[2, 5, 6, 2, 34.6, 12]
```

Case Study An ATM:

```
class ATM:
        def __init__(self):
                self.balance=0
                print("new account created")
        def deposit(self):
                amount=int(input("enter amount to deposit"))
                self.balance=self.balance+amount
```

```
                print("new balance is:",self.balance)
        def withdraw(self):
                amount=int(input("enter amount to withdraw"))
                if self.balance<amount:
                        print("Insufficient Balance")
                else:
                        self.balance=self.balance-amount
                        print("new balance is:",self.balance)
        def enquiry(self):
                print("Balance is:",self.balance)

a=ATM()
a.deposit()
a.withdraw()
a.enquiry()
```

OUTPUT:

```
>>>
new account created
enter amount to deposit15000
new balance is: 15000
enter amount to withdraw5648
new balance is: 9352
Balance is: 9352
```

Adding and retrieving dynamic attributes of classes:

Dynamic attributes in Python are terminologies for attributes that are defined at runtime, after creating the objects or instances.

Example:

```
class EMP:
        employee = True
e1 = EMP()
e2 = EMP()
e1.employee = False
e2.name = "SAI KUMAR"    #DYNAMIC ATTRIBUTE
print(e1.employee)
```

```
print(e2.employee)
print(e2.name)
print(e1.name)      # this will raise an error  as name is a dynamic attribute created only for
                    #the e2 object
```

UNIT 5 PART -1

EXCEPTION HANDLING

Errors and Exceptions: The programs that we write may behave abnormally or unexpectedly because of some errors and/or exceptions.

Errors:

- The two common types of errors that we very often encounter are *syntax errors* and *logic errors*.

  Syntax errors: And syntax errors, arises due to poor understanding of the language. *Syntax errors* occur when we violate the rules of Python and they are the most common kind of error that we get while learning a new language.

  Example :

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

  if you run this program we will get syntax error like below,

```
File "1.py", line 2
while i<=10
^
SyntaxError: invalid syntax
```

  Logical errors: While logic errors occur due to poor understanding of problem and its solution. *Logic error* specifies all those type of errors in which the program executes but gives incorrect results. Logical error may occur due to wrong algorithm or logic to solve a particular program.

- However, such errors can be detected at the time of testing.

Exceptions:

• Even if a statement is syntactically correct, it may still cause an error when executed. • Such errors that occur at run-time (or during execution) are known as *exceptions*. • An exception is an event, which occurs during the execution of a program and disrupts the normal flow of the program's instructions.

• Exceptions are run-time anomalies or unusual conditions (such as divide by zero, accessing arrays out of its bounds, running out of memory or disk space, overflow, and underflow) that a program may encounter during execution.

• Like errors, exceptions can also be categorized as synchronous and asynchronous exceptions.

• While synchronous exceptions (like divide by zero, array index out of bound, etc.) can be controlled by the program

• Asynchronous exceptions (like an interrupt from the keyboard, hardware malfunction, or disk failure), on the other hand, are caused by events that are beyond the control of the program.

• When an exception occurs in a program, the program must raise the exception. After that it must handle the exception or the program will be immediately terminated. • if exceptions are not handled by programs, then error messages are generated..

Example:

```
num=int(input("enter numerator"))
den=int(input("enter denominator"))
quo=num/den
print(quo)
```

output:

```
C:\Users\PP>python excep.py
enter numerator1
enter denominator0
```

```
Traceback (most recent call last):
 File "excep.py", line 3, in <module>
 quo=num/den
ZeroDivisionError: division by zero
```

Handling Exceptions:

We can handle exceptions in our program by using try block and except block. A critical operation which can raise exception is placed inside the try block and the code that handles exception is written in except block.

The syntax for try–except block can be given as

```
try:
        statements
except ExceptionName:
        statements
```

Example:

```
num=int(input("Numerator: "))
deno=int(input("Denominator: "))
try:
        quo=num/deno
        print("QUOTIENT: ",quo)
except ZeroDivisionError:
        print("Denominator can't be zero")
```

Output:

```
Numerator: 10
Denominator: 0
```

Denominator can't be zero

Multiple except blocks:

Python allows you to have multiple except blocks for a single try block. The block which matches with the exception generated will get executed.

syntax:

```
try:
    You do your operations here;
    ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
    If there is any exception from the given exception list,
    then execute this block.
    ......................
else:
    If there is no exception then execute this block.
```

e.g.

```
string = input("Enter a String:")
try:
    num = int(input("Enter a number"))
    print(string+num)
except TypeError as e:
    print(e)
except ValueError as e:
    print(e)
```

(OR)

```
string = input("Enter a String:")
```

```
try:
    num = int(input("Enter a number"))
    print(string+num)
except (TypeError,ValueError) as e:
    print(e)
```

OUTPUT :

```
>>>
Enter a String:hai
Enter a number3
Can't convert 'int' object to str implicitly
>>>
Enter a String:hai
Enter a numberbye
invalid literal for int() with base 10: 'bye'
```

Raising Exceptions:

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**.**

The raise statement allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception)

You can Explicitly raise an exception using the raise keyword.

The general syntax for the raise statement is

raise [Exception [, args [, traceback]]]

Here, Exception is the name of exception to be raised. *args* is optional and specifies a value for the exception argument. If args is not specified, then the exception argument is None. The final argument, *traceback*, is also optional and if present, is the traceback object used for the exception.

```
num=int(input("enter numerator"))
```

```
den=int(input("enter denominator"))
try:
    quo=num/den
    raise Exception("I want an exception anyway")
    print(quo)

except ZeroDivisionError:
    print("Denominator cant be zero")
```

OUTPUT:

```
>>>
enter numerator4
enter denominator0
Denominator cant be zero
>>>
enter numerator4
enter denominator2
Traceback (most recent call last):
  File "C:\Python32\raisex.py", line 5, in <module>
    raise Exception("I want an exception anyway")
Exception: I want an exception anyway
```

Defining clean-up actions(The finally Block)

The finally block is always executed before leaving the try block. This means that the statements written in finally block are executed irrespective of whether an exception has occurred or not.

Syntax:

```
try:
        Write your operations here
        ......................
```

Due to any exception, operations written here will be skipped

finally:

This would always be executed.

.....

e.g.

```
num=int(input("enter numerator"))

den=int(input("enter denominator"))

try:

   quo=num/den

   print(quo)


except ZeroDivisionError:

   print("Denominator cant be zero")

else:

   print("This line is executed when there is no exception")


finally:

    print("TASK DONE")
```

OUTPUT:

```
>>>

enter numerator4

enter denominator2

2.0

This line is executed when there is no exception

TASK DONE
```

>>>

enter numerator4

enter denominator0

Denominator cant be zero

TASK DONE

Built-in and User-defined Exceptions:

Built-in Exceptions:

Exceptions that are already defined in python are called built in or pre-defined exception. In the table listed some built-in exceptions

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
IndexError KeyError	Raised when an index is not found in a sequence. Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.

IOError Raised when an input/ output operation fails

SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.

User –defined exception:

Python allows programmers to create their own exceptions by creating a new exception class. The new exception class is derived from the base class Exception which is predefined in python.

Example:

```
class myerror(Exception):
    def __init__(self,val):
        self.val=val
try:
    raise  myerror(10)
except  myerror  as  e:
    print("user defined exception generated with value",e.val)
```

OUTPUT:

user defined exception generated with value 10

UNIT 5

Graphical User Interface:

Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below.

- Tkinter
- wxPython
- JPython

Tkinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps –

- Import the *Tkinter* module.
- Create the GUI application main window.
- Add one or more of the above-mentioned widgets to the GUI application.
- Enter the main event loop to take action against each event triggered by the user.

Example:

```
From import Tkinter *
top = Tkinter.Tk()
# Code to add widgets will go here...
top.mainloop()
```

This would create a following window –

Tkinter Widgets:

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table –

- Button
- Canvas
- Check button
- Entry
- Frame
- Label
- List box
- Menu button
- Menu
- Message
- Radio button
- Scale
- Scrollbar
- Text
- Top level.
- Spin box
- Paned Window
- Label Frame
- Tk Message Box

Standard attributes for widgets

- Dimensions
- Colors
- Fonts
- Relief styles
- Bitmaps
- Cursors

Geometry Management:

All Tkinter widgets have access to specific geometry management methods, Tkinter exposes the following geometry manager classes: pack, grid, and place. •

- The pack() Method - This geometry manager organizes widgets in blocks before placing them in the parent widget.
- The grid() Method - This geometry manager organizes widgets in a table-like

structure in the parent widget.

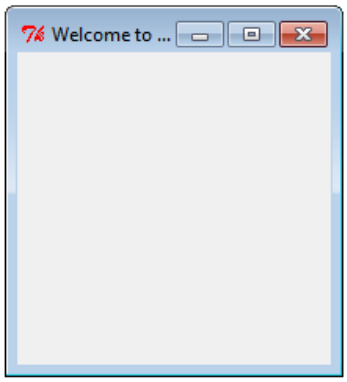
- The place() Method -This geometry manager organizes widgets by placing them in a specific position in the parent widget.

1) Creation of a window/widget

First, we will import Tkinter package and create a window and set its title. The last line which calls mainloop function, this function calls the endless loop of the window, so the window will wait for any user interaction till we close it. If you forget to call the mainloop function, nothing will appear to the user.

Program:

```
from tkinter import *
window = Tk()
window.title("Welcome to tkinter")
window.mainloop()
```



2) Creating a window with specific dimensions and a label

To add a label to our previous example, we will create a label using the label class like this:

```
lbl = Label(window, text="Hello")
```

Then we will set its position on the form using the grid function and give it the location like this:

```
lbl.grid(column=0, row=0)
```

So the complete code will be like this:

Program

```
from tkinter import *
window = Tk()
window.geometry("500x600")
window.title("CSE")
lbl = Label(window, text="HelloWorld",font=("Arial Bold", 50))
lbl.grid(column=0, row=0)
window.mainloop()
```

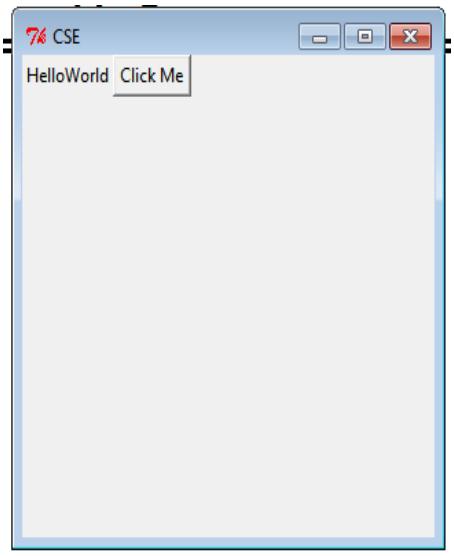
3)Adding a Button to window/widget

Let's start by adding the button to the window, the button is created and added to the window the same as the label:

```
btn = Button(window, text="Click Me")
btn.grid(column=1, row=0)
```

Program:

```
from tkinter import *
window = Tk()
window.geometry("300x300")
window.title("CSE")
lbl = Label(window, text="HelloWorld")
lbl.grid(column=0, row=0)
btn = Button(window, text="Click Me")
btn.grid(column=1, row=0)
window.mainloop()
```



4)Creating 2 text fields to enter 2 numbers, and a button when clicked gives sum of the 2 numbers and displays it in 3rd text field

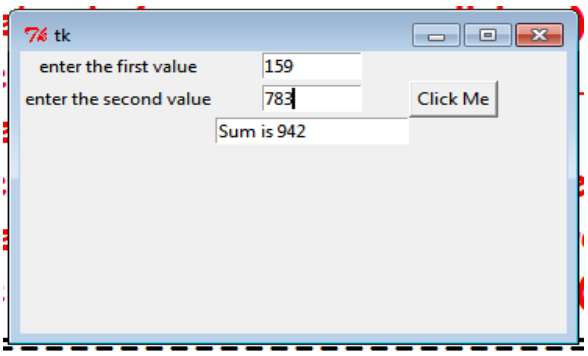
You can create a textbox using Tkinter Entry class like this:

= Entry(window,width=10)

Then you can add it to the window using grid function as usual

Program:

```
from tkinter import *
window = Tk()
window.geometry('350x200')
lbl1 = Label(window, text="enter the first value")
lbl1.grid(column=0, row=0)
lbl2 = Label(window, text="enter the second value")
lbl2.grid(column=0, row=1)
txt1 = Entry(window,width=10)
txt1.grid(column=1, row=0)
txt2 = Entry(window,width=10)
txt2.grid(column=1, row=1)
txt3 = Entry(window,width=20)
txt3.grid(column=1, row=2)
def clicked():
        res=int(txt1.get())+int(txt2.get())
        txt3.insert(0,"Sum is {}".format(res))
btn = Button(window, text="Click Me", command=clicked)
btn.grid(column=2, row=1)
window.mainloop()
```


tk
enter the first value
159
enter the second value
783
Click Me
Sum is 942

5) Creating 2 checkboxes

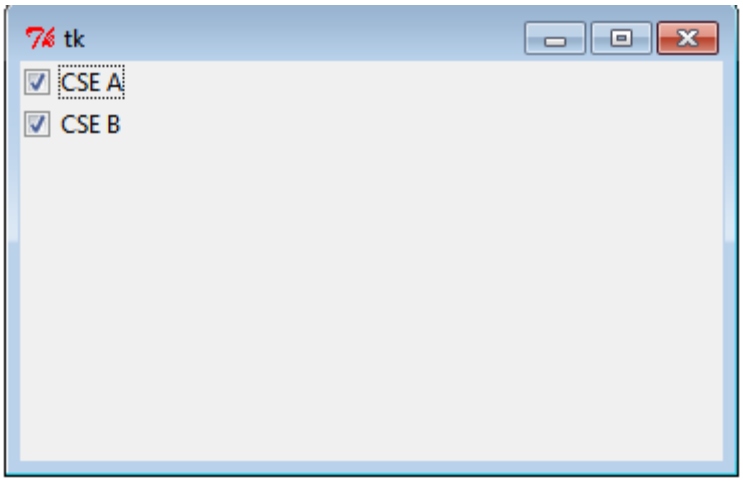
To create a checkbutton, you can use Checkbutton class like this:

```
chk = Checkbutton(window, text='Choose')
```

Here we create a variable of type BooleanVar which is not a standard Python variable, it's a Tkinter variable, and then we pass it to the Checkbutton class to set the check state as the highlighted line in the above example. You can set the Boolean value to false to make it unchecked.

the following program creates a check box.

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
window.geometry('350x200')
chk_state = BooleanVar()
chk_state.set(True) #set check state
chk1 = Checkbutton(window, text='CSE A', var=chk_state)
chk2 = Checkbutton(window, text='CSE B', var=chk_state)
chk1.grid(column=0, row=0)
chk2.grid(column=0, row=1)
window.mainloop()
```



6)Creating 3 radio buttons

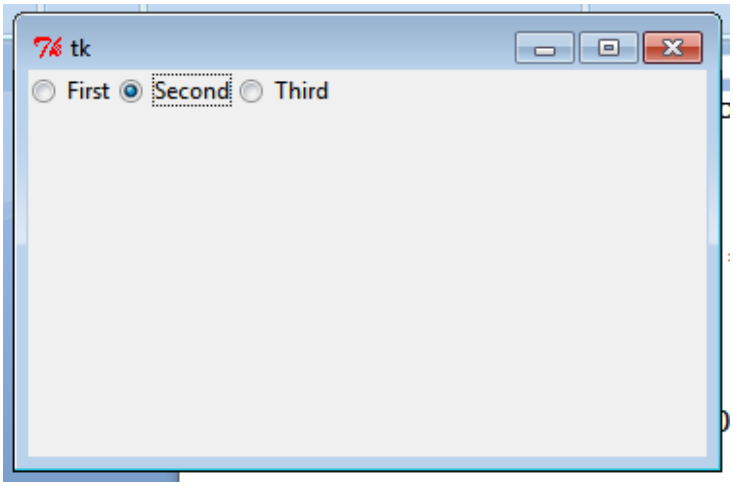
To add radio buttons, simply you can use RadioButton class like this:

rad1 = Radiobutton(window,text='First', value=1)

Note that you should set the value for every radio button with a different value, otherwise, they won't work.

the following program creates a check box

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
window.geometry('350x200')
rad1 = Radiobutton(window,text='First', value=1)
rad2 = Radiobutton(window,text='Second', value=2)
rad3 = Radiobutton(window,text='Third', value=3)
rad1.grid(column=0, row=0)
rad2.grid(column=1, row=0)
rad3.grid(column=2, row=0)
window.mainloop()
```



7) Creating a message box on clicking a button

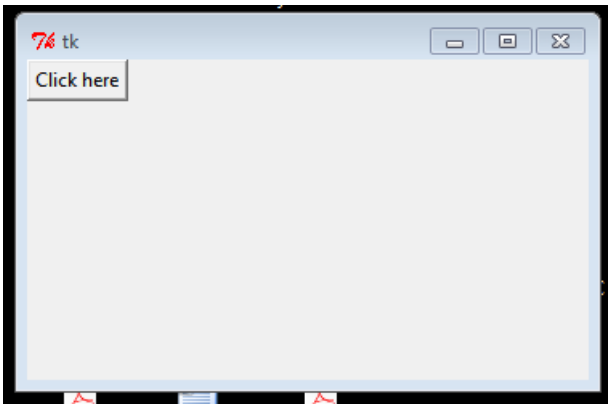
You can show a warning message or error message the same way. The only thing that needs to be changed is the message function.

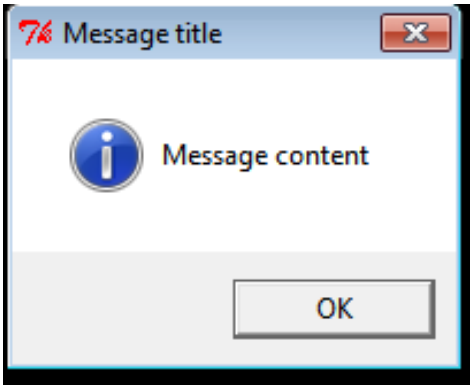
You can show a warning message or error message the same way. The only thing that needs to be changed is the message function

messagebox.showwarning('Message title', 'Message content') #shows warning message

messagebox.showerror('Message title', 'Message content')

```
from tkinter import *
window = Tk()
window.geometry('350x200')
def clicked():
    messagebox.showinfo('Message title ', 'Message content')
    # messagebox.showerror('Message title ', 'Message content')
    # messagebox.show('Message title ', 'Message content')
btn = Button(window,text='Click here', command=clicked)
btn.grid(column=0,row=0)
window.mainloop()
```



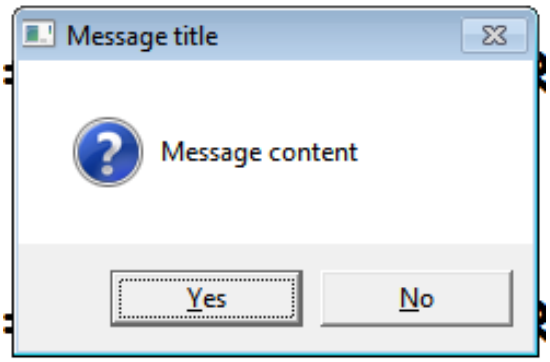

Message title
Message content
OK

8) Creating various message boxes

To show a yes no message box to the user, you can use one of the following messagebox Functions.

- If you click OK or yes or retry, it will return True value, but if you choose no or cancel, it will return False.
- The only function that returns one of three values is askyesnocancel function, it returns True or False or None.

```
from tkinter import messagebox
res = messagebox.askquestion('Message title','Message content')
res = messagebox.askyesno('Message title','Message content')
res = messagebox.askyesnocancel('Message title','Message content')
res = messagebox.askokcancel('Message title','Message content')
res = messagebox.askretrycancel('Message title','Message content')
```



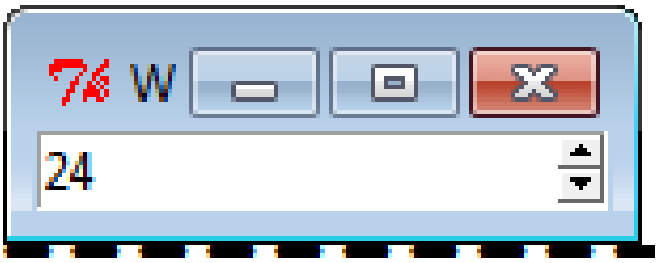
9) Creating a Spinbox

To create a Spinbox widget, you can use Spinbox class like this:

spin = Spinbox(window, from_=0, to=100)

Here we create a Spinbox widget and we pass the from_ and to parameters to specify the numbers range for the Spinbox.

```
from tkinter import *
window = Tk()
window.title("Welcome to tkinter")
spin = Spinbox(window, from_=0, to=100)
spin.grid(column=0,row=0)
window.mainloop()
```



Introduction to programming concepts of scratch

Programming is core of computer science, it's worth taking some time to really get to grips with programming concepts and one of the main tools used in schools to teach these concepts, Scratch.

Programming simply refers to the art of writing instructions (algorithms) to tell a computer what to do. Scratch is a visual programming language that provides an ideal learning environment for doing this. Originally developed by America's Massachusetts Institute of Technology, Scratch is a simple, visual programming language. Colour coded blocks of code simply snap together. Many media rich programs can be made using Scratch, including games, animations and interactive stories. Scratch is almost certainly the most widely used software for teaching programming to Key Stage 2 and Key Stage 3 (learners from 8 to 14 years).

Scratch is a great tool for developing the programming skills of learners, since it allows all manner of different programs to be built. In order to help develop the knowledge and understanding that go with these skills though, it's important to be familiar with some key programming concepts that underpin the Scratch programming environment and are applicable to any programming language. Using screenshots, we will understand the scratch concepts.

Sprites

The most important thing in any Scratch program are the sprites. Sprites are the graphical objects or characters that perform a function in your program. The default sprite in Scratch is the cat, which can easily be changed. Sprites by themselves won't do anything of course, without coding!

Sequences

In order to make a program in any programming language, you need to think through the sequence of steps.

Iteration (looping)

Iteration simply refers to the repetition of a series of instructions. This is accomplished in Scratch using the repeat, repeat until or forever blocks.

Conditional statements

A conditional statement is a set of rules performed if a certain condition is met. In Scratch, the if and if-else blocks check for a condition.

Variables

A variable stores specific information. The most common variables in computer games for example, are score and timer.

Lists (arrays)

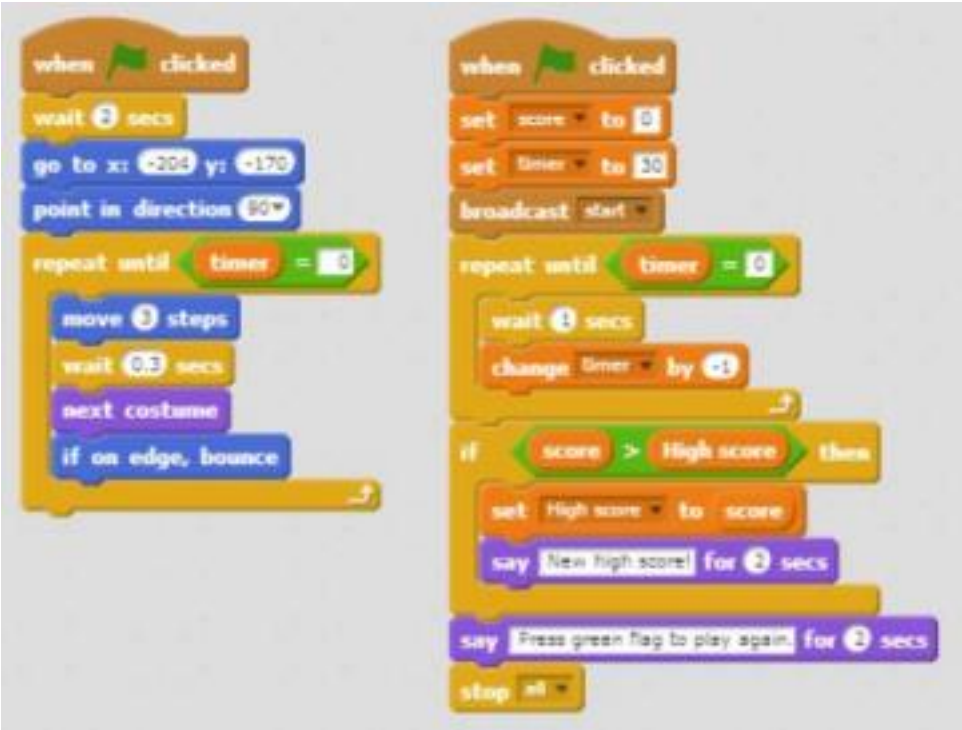
A list is a tool that can be used to store multiple pieces of information at once.

Event Handling

When key pressed and when sprite clicked are examples of event handling. These blocks allow the sprite to respond to events triggered by the user or other parts of the program.

Threads

A thread just refers to the flow of a particular sequence of code within a program. A thread cannot run on its own, but runs within a program. When two threads launch at the same time it is called parallel execution.



Coordination & Synchronisation

The broadcast and when I receive blocks can coordinate the actions of multiple sprites.

They work by getting sprites to cooperate by exchanging messages with one another. A common example is when one sprite touches another sprite, which then broadcasts a new level.

Keyboard input

This is a way of interacting with the user. The ask and wait prompts users to type. The answer block stores the keyboard input.

Boolean logic

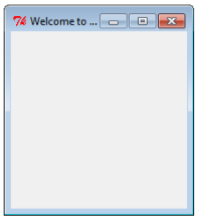
Boolean logic is a form of algebra in which all values are reduced to either true or false. The and, or, not statements are examples of Boolean logic.

User interface design

Interactive user interfaces can be designed in Scratch using clickable sprites to create buttons.

1) Creation of a window/widget

```
from tkinter import *

window = Tk()

window.title("Welcome to tkinter")

window.mainloop()
```

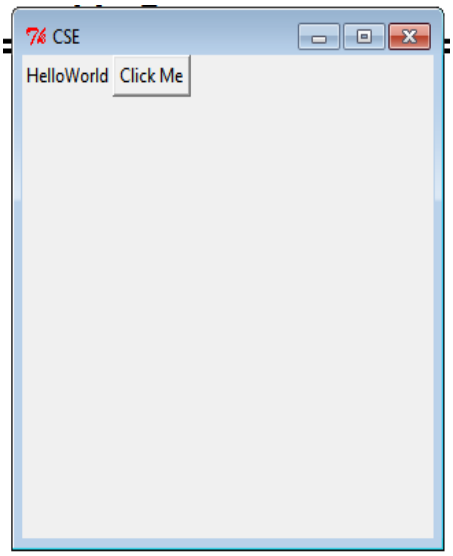


2) Creating a window with specific dimensions and a label

```
from tkinter import *
window = Tk()
window.geometry("500x600")
window.title("CSE")
lbl = Label(window, text="HelloWorld",font=("Arial Bold", 50))
lbl.grid(column=0, row=0)
window.mainloop()
```

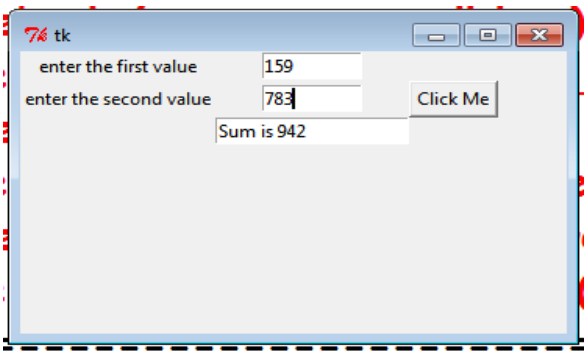
3)Adding a Button to window/widget

```
from tkinter import *

window = Tk()

window.geometry("300x300")

window.title("CSE")

lbl = Label(window, text="HelloWorld")

lbl.grid(column=0, row=0)

btn = Button(window, text="Click Me")
btn.grid(column=1, row=0)

window.mainloop()
```



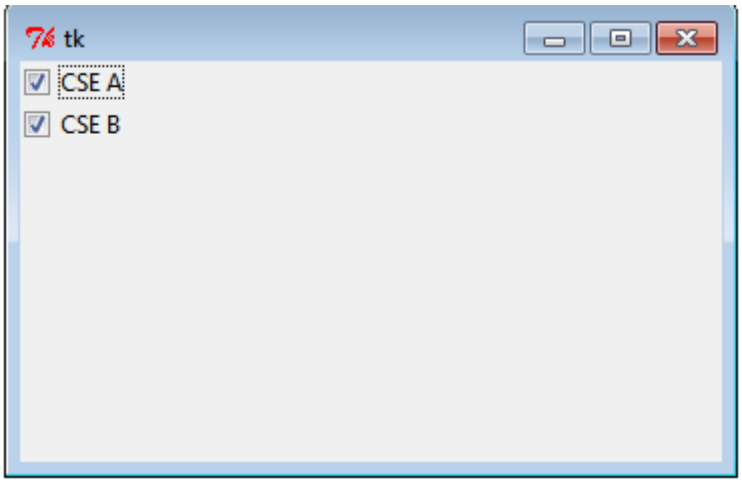
4)Creating 2 text fields to enter 2 numbers, and a button when clicked gives sum of the 2 numbers and displays it in 3rd text field

```
from tkinter import *

window = Tk()

window.geometry('350x200')

lbl1 = Label(window, text="enter the first value")

lbl1.grid(column=0, row=0)

lbl2 = Label(window, text="enter the second value")

lbl2.grid(column=0, row=1)

txt1 = Entry(window,width=10)

txt1.grid(column=1, row=0)

txt2 = Entry(window,width=10)

txt2.grid(column=1, row=1)

txt3 = Entry(window,width=20)

txt3.grid(column=1, row=2)

def clicked():

        res=int(txt1.get())+int(txt2.get())

        txt3.insert(0,"Sum is {}".format(res))

btn = Button(window, text="Click Me", command=clicked)

btn.grid(column=2, row=1)

window.mainloop()
```



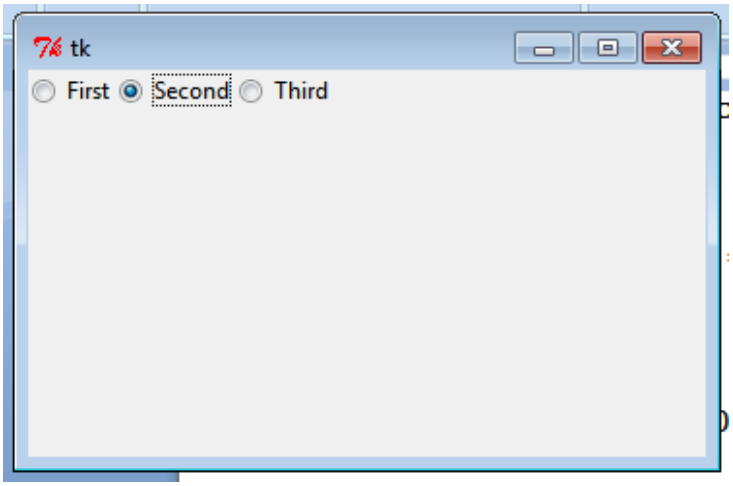
5) Creating 2 checkboxes

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
window.geometry('350x200')
chk_state = BooleanVar()
chk_state.set(True) #set check state
chk1 = Checkbutton(window, text='CSE A', var=chk_state)
chk2 = Checkbutton(window, text='CSE B', var=chk_state)
chk1.grid(column=0, row=0)
chk2.grid(column=0, row=1)
window.mainloop()
```



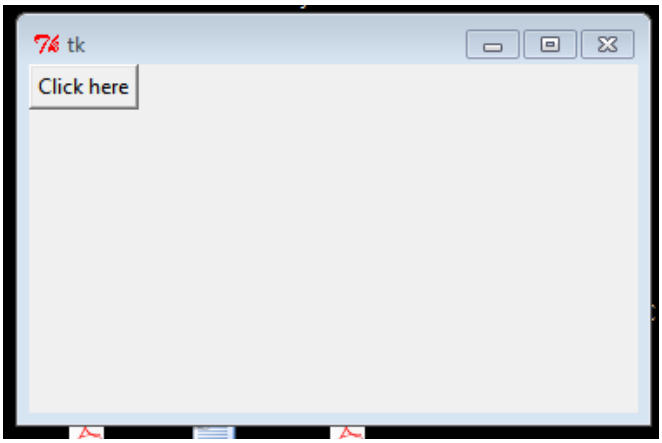
6)Creating 3 radio buttons

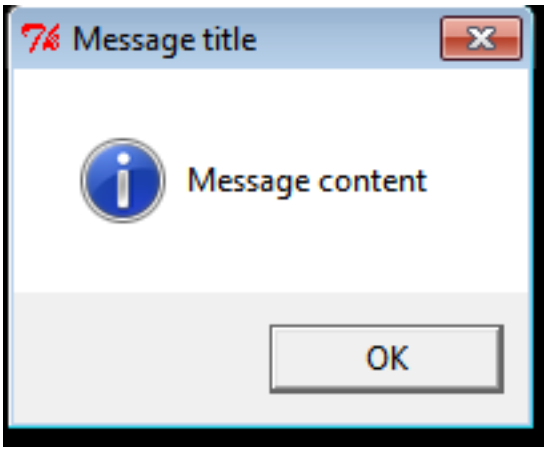
```
from tkinter import *
from tkinter.ttk import *
window = Tk()
window.geometry('350x200')
rad1 = Radiobutton(window,text='First', value=1)
rad2 = Radiobutton(window,text='Second', value=2)
rad3 = Radiobutton(window,text='Third', value=3)
rad1.grid(column=0, row=0)
rad2.grid(column=1, row=0)
rad3.grid(column=2, row=0)
window.mainloop()
```



7) Creating a message box on clicking a button

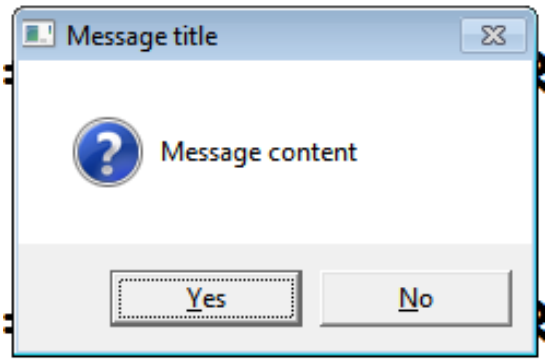
```
from tkinter import *
window = Tk()
window.geometry('350x200')
def clicked():
    messagebox.showinfo('Message title ', 'Message content')
    # messagebox.showerror('Message title ', 'Message content')
    # messagebox.show('Message title ', 'Message content')
btn = Button(window,text='Click here', command=clicked)
btn.grid(column=0,row=0)
window.mainloop()
```





8) Creating various message boxes

```
from tkinter import messagebox

res = messagebox.askquestion('Message title','Message content')

res = messagebox.askyesno('Message title','Message content')

res = messagebox.askyesnocancel('Message title','Message content')

res = messagebox.askokcancel('Message title','Message content')

res = messagebox.askretrycancel('Message title','Message content')
```



9) Creating a Spinbox

```
from tkinter import *
window = Tk()
window.title("Welcome to tkinter")
spin = Spinbox(window, from_=0, to=100)
spin.grid(column=0,row=0)
window.mainloop()
```