UNIT I Topics and Sub Topics
Introduction to Python Programming: How a Program Works, Using 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. **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. **Lists and Tuples**: Sequences, Introduction to Lists, List slicing, Finding Items in Lists with the in Operator, List Methods and Useful Built-in Functions, Copying Lists, Processing Lists

UNIT I

Introduction

Python is object-oriented programming language that was developed by Guido vas Rossum and first released in 1991. It is a multi-paradigm high-level general-purpose scripting language.

- Python supports multiple programming paradigms (models) like structured programming, object-oriented programming and functional programming. So it is called a multi-paradigm language.
- Python is user friendly and can be used to develop application software like text editors, music players etc. so, it is called high level language.
- Python has several built-in libraries. There are also many third-party libraries with the help of all these libraries we develop almost anything with python. So, it is called General purpose language.
- Finally, python programs are executed line by line(interpreted). So, it is called a Scripting Language.

Features:

1. **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
2. **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
3. **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
4. **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
5. **Interactive Mode:** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
6. **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
7. **Extendable:** You can add low-level modules to the Python interpreter. These

modules enable programmers to add to or customize their tools to be more efficient.

8. **Databases:** Python provides interfaces to all major commercial databases.

GUI Programming: Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

9. **Scalable:** Python provides a better structure and support for large programs than shell scripting.

How a Program Works

A computer's CPU can only understand instructions that are written in machine language. Because people find it very difficult to write entire programs in machine language, other programming languages have been invented.

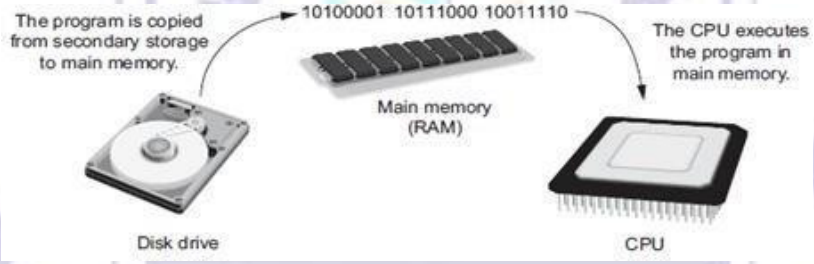
Programs are usually stored on a secondary storage device such as a disk drive. When you install a program on your computer, the program is typically copied to your computer's disk drive from a CD-ROM, or perhaps downloaded from a website.

Although a program can be stored on a secondary storage device such as a disk drive, it has to be copied into main memory, or RAM, each time the CPU executes it.

For example, suppose you have a word processing program on your computer's disk. To execute the program you use the mouse to double-click the program's icon. This causes the program to be copied from the disk into main memory.

Then, the computer's CPU executes the copy of the program that is in main memory.

This process is illustrated in the following Figure.



When a CPU executes the instructions in a program, it is engaged in a process that is known as the fetch-decode-execute cycle. This cycle, which consists of three steps, is repeated for each instruction in the program. The steps are:

1. **Fetch:** A program is a long sequence of machine language instructions. The first step of the cycle is to fetch, or read, the next instruction from memory into the CPU.
2. **Decode** A machine language instruction is a binary number that represents a command that tells the CPU to perform an operation. In this step the CPU decodes the instruction that was just fetched from memory, to determine which operation it should perform.
3. **Execute** The last step in the cycle is to execute, or perform, the operation.

Keywords:

Each high-level language has its own set of predefined words that the programmer must use to write a program. The words that make up a high-level programming language are known as key words or reserved words. Each key word has a specific meaning, and cannot be used for any other purpose.

The Python key words

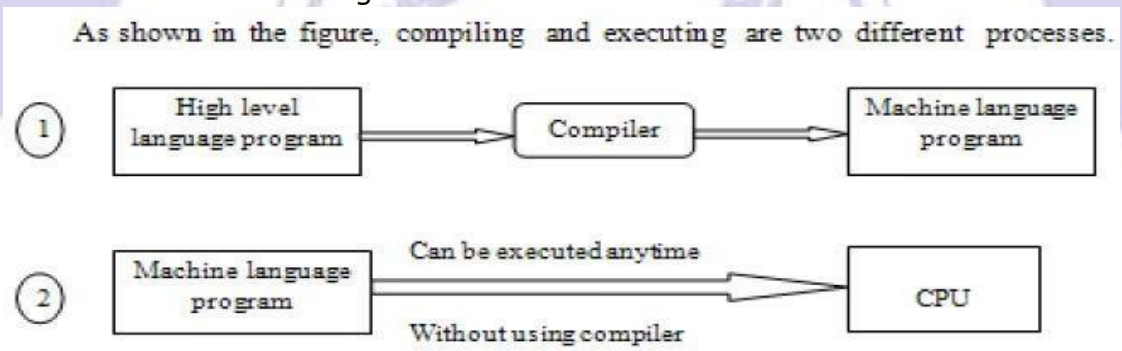
and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	def
for	lambda	return		

Compilers and Interpreters:

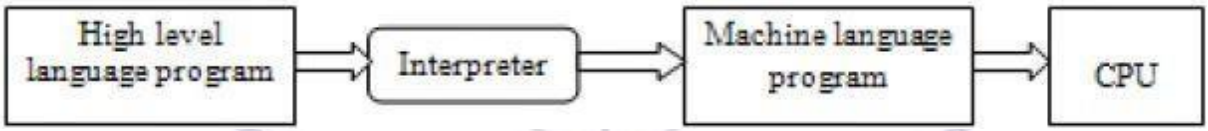
Because the CPU understands only machine language instructions, programs that are written in a high-level language must be translated into machine language. Depending on the language that a program has been written in, the programmer will use either a compiler or an interpreter to make the translation.

Compiler: A compiler is a program that translates a high-level language program into a separate machine language program. The machine language program can then be executed any time it is needed.

This is shown in the below figure.



Interpreter: The Python language uses an **interpreter**, which is a program that both translates and executes the instructions in a high-level language program. As the interpreter reads each individual instruction in the program, it converts it to machine language instructions and then immediately executes them. This process repeats for every instruction in the program. This is shown in the below figure.



Data Type Conversion:

When you perform a math operation on two operands, the data type of the result will depend on the data type of the operands. Python follows these rules when evaluating mathematical expressions:

- When an operation is performed on two int values, the result will be an int.
- When an operation is performed on two float values, the result will be a float.
- When an operation is performed on an int and a float, the int value will be temporarily converted to a float and the result of the operation will be a float. (An expression that uses operands of different data types is called a mixed-type expression.)

Example(typeconvert.py)

```
def main():
    year_sal=float(input("Enter yearly salary:"))
    monthly_sal=year_sal/12
    print("Monthly salary is:",monthly_sal)
main()
```

Output:

```
Enter yearly salary: 65721
Monthly salary is: 5476.75
```

More About Data Output:

1. Breaking Long Statements into Multiple Lines:

Python allows you to break a statement into multiple lines by using the line continuation character, which is a backslash (\). You simply type the backslash character at the point you want to break the statement, and then press the Enter key. Here is a print function call that is broken into two lines with the line continuation character:

Example:

```
print('We sold', units_sold, \
      'for a total of', sales_amount)
```

The line continuation character that appears at the end of the first line tells the interpreter that the statement is continued on the next line.

2. Suppressing the print Function's Ending Newline:

The print function normally displays a line of output. For example, the following three tatements will produce three lines of output:

```
print('One')
print('Two')
print('Three')
```

Each of the statements shown here displays a string and then prints a newline character. You do not see the newline character, but when it is displayed, it causes the output to advance to the next line.If you do not want the print function to start a new line of output when it finishes displaying its output, you can pass the special argument end=' ' to the function, as shown in the following code:

```
print('One', end=' ')
print('Two', end=' ')
print('Three')
```

Notice that in the first two statements, the argument end=' ' is passed to the print function.This specifies that the print function should print a space instead of a newline character at the end of its output. Here is the output of these statements: **One Two**

Three

3. Specifying an Item Separator:

When multiple arguments are passed to the print function, they are automatically separated by a space when they are displayed on the screen. Here is an example, demonstrated in interactive mode:

```
>>> print('One', 'Two', 'Three')        press Enter
        One Two Three
    >>>
```

if you do not want a space printed between the items, you can pass the argument sep=''

to the print function, as shown here:

```
>>> print('One', 'Two', 'Three', sep='*')  press Enter
        OneTwoThree
>>>
```

You can also use this special argument to specify a character other than the space to separate multiple items. Here is an example:

```
>>> print('One', 'Two', 'Three', sep='*')  press Enter
        One*Two*Three
>>>
```

4. Escape Characters:

An escape character is a special character that is preceded with a backslash (\), appearing inside a string literal. When a string literal that contains escape characters is printed, the escape characters are treated as special commands that are embedded in the string.

For example, \n is the newline escape character. When the \n escape character is printed,it isn't displayed on the screen. Instead, it causes output to advance to the next line. For example, look at the following statement:

```
print('One\nTwo\nThree')
```

When this statement executes, it displays

```
One
Two
Three
```

Escape Character	Effect
\n	Causes output to be advanced to the next line.
\t	Causes output to skip over to the next horizontal tab position.
\'	Causes a single quote mark to be printed.

\"	Causes a double quote mark to be printed.
\\	Causes a backslash character to be printed.

---

5. Displaying Multiple Items with the + Operator:

Normally, the + operator is used to add two numbers. When the + operator is used with two strings, however, it performs string concatenation. This means that it appends one string to another. For example, look at the following statement:

```
print('This is ' + 'one string.')
```

This statement will print

```
This is one string.
```

String concatenation can be useful for breaking up a string literal so a lengthy call to the print function can span multiple lines. Here is an example:

```
print('Enter the amount of ' + \
'sales for each day and ' + \
'press Enter.')
```

This statement will display the following:

```
Enter the amount of sales for each day and press Enter.
```

SEETHANA DEGREE COLLEGE BELA ADILABAD (TS)

Designing a Program

Programs must be carefully designed before they are written. During the design process, programmers use tools such as pseudo code and flowcharts to create models of programs.

The Program Development Cycle:

The process of creating a program that works correctly typically requires the five phases. The entire process is known as the **program development cycle.**

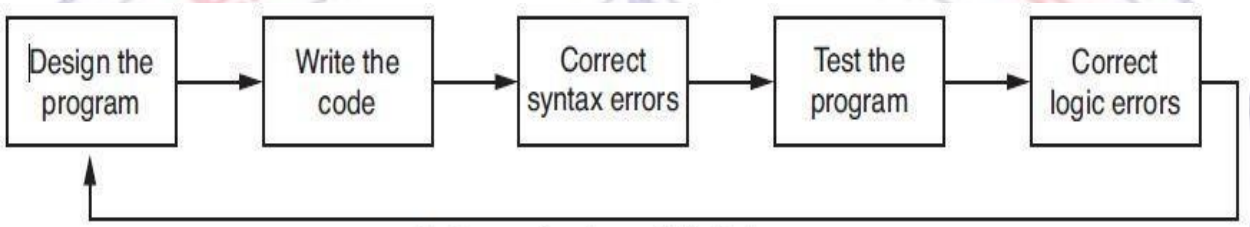


Fig. Program Development Life Cycle

Let's take a closer look at each stage in the cycle.

1. **Design the Program:** All professional programmers will tell you that a program should be carefully designed before the code is actually written. When programmers begin a new project, they never jump right in and start writing code as the first step. They start by creating a design of the program.
2. **Write the Code:** After designing the program, the programmer begins writing code in a high-level language such as Python. Each language has its own rules, known as syntax that must be followed when writing a program. A language's syntax rules dictate things such as how key words, operators, and punctuation characters can be used. A syntax error occurs if the programmer violates any of these rules.
3. **Correct Syntax Errors:** If the program contains a syntax error, or even a simple mistake such as a misspelled keyword, the compiler or interpreter will display an error message indicating what the error is. Virtually all code contains syntax errors when it is first written, so the programmer will typically spend some time correcting these. Once all of the syntax errors and simple typing mistakes have been corrected, the program can be compiled and translated into a machine language program (or executed by an interpreter, depending on the language being used).
4. **Test the Program:** Once the code is in an executable form, it is then tested to determine whether any logic errors exist. A *logic error* is a mistake that does not prevent the program from running, but causes it to produce incorrect results. (Mathematical mistakes are common causes of logic errors.)
5. **Correct Logic Errors:** If the program produces incorrect results, the programmer

debugs the code. This means that the programmer finds and corrects logic errors in the program. Sometimes during this process, the programmer discovers that the program's original design must be changed. In this event, the program development cycle starts over, and continues until no errors can be found

More about the Design Process:

The process of designing a program is arguably the most important part of the cycle. The process of designing a program can be summarized in the following two steps:

1. Understand the task that the program is to perform:

It is essential that you understand what a program is supposed to do before you can determine the steps that the program will perform. Typically, a professional programmer gains this understanding by working directly with the customer. We use the term customer to describe the person, group, or organization that is asking you to write a program.

To get a sense of what a program is supposed to do, the programmer usually interviews the customer. During the interview, the customer will describe the task that the program should perform, and the programmer will ask questions to uncover as many details as possible about the task.

The programmer studies the information that was gathered from the customer during the interviews and creates a list of different software requirements. A software requirement is simply a single task that the program must perform in order to satisfy the customer. Once the customer agrees that the list of requirements is complete, the programmer can move to the next phase.

2. Determine the steps that must be taken to perform the task:

Once you understand the task that the program will perform, you begin by breaking down the task into a series of steps. Dividing a task into set of well- defined logical steps is called Algorithm. The steps in this algorithm are sequentially ordered. Step1 should be performed before step2 , and so on. Once an algorithm is created, which lists all of the logical steps that must be taken to perform the task.

Algorithm: For adding two numbers

Step1: Take two values

Step2: add two values

Step3: display the result

Of course, this algorithm is not ready to be executed on the computer. The steps in this list have to be translated into code. Programmers commonly use two tools to help them accomplish this: **pseudocode** and **flowcharts**.

1. Pseudocode:

The word "pseudo" means fake, so pseudocode is fake code. It is an informal language that has no syntax rules, and is not meant to be compiled or executed. Instead, programmers use pseudocode to create models of programs. Because programmers do not have to worry about syntax errors while writing pseudocode, they can focus all of their attention on the program's design. Once a satisfactory design has been created with pseudocode, the pseudocode can be translated directly to actual code.

Pseudocode : for adding two numbers: Declare three variables a, b, sum

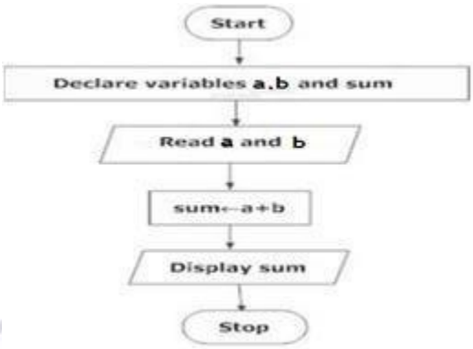
Input the values for a,b variables

sum ← a+b

Print sum

2. **Flow chart:** Flowcharting is another tool that programmers use to design programs. A flowchart is a diagram that graphically depicts the steps that take place in a program.

Flowchart for adding two numbers:



Input, Processing, and Output

Computer programs typically perform the following three-step process:

1. Input is received.
2. Some process is performed on the input.
3. Output is produced.

Input is any data that the program receives while it is running. One common form of input is data that is typed on the keyboard. Once input is received, some process, such as a mathematical calculation, is usually performed on it. The results of the process are then sent out of the program as output.

The below Figure illustrates these three steps in the pay calculating program. The number of hours worked and the hourly pay rate are provided as input. The program processes this data by multiplying the hours worked by the hourly pay rate. The results of the calculation are then displayed on the screen as output.

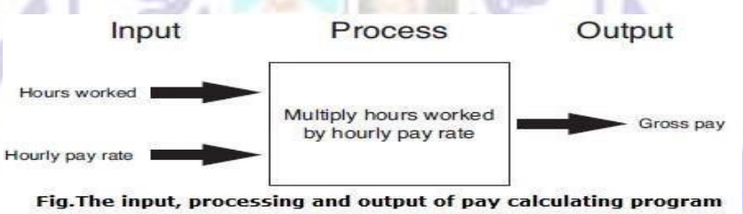


Fig.The input, processing and output of pay calculating program

Comments

Comments are short notes placed in different parts of a program, explaining how those parts of the program work. Although comments are a critical part of a program, they are ignored by the Python interpreter. Comments are intended for any person reading a program's code, not the computer.

In Python you begin a comment with the # character. When the Python interpreter sees a # character, it ignores everything from that character to the end of the line.

Ex: comments.py

```
# This program displays a person's #
name and address.
    print('Mahesh') print('Gandhi
    Nagar') print('Rajanna
    Sircilla')
```

In the above program, First two lines are Comment lines which are ignored by the python interpreter. So, the output will be:

Output:

Mahesh Gandhi

Nagar Rajanna

Sircilla

Variables

Programs use variables to access and manipulate data that is stored in memory. A variable is a name that represents a value in the computer's memory. For example, a program that calculates the **sales tax** on a purchase might use the variable name **tax** to represent that value in memory. When a variable represents a value in the computer's memory, we say that the variable references the value.

Variable naming rules: Although we are allowed to make up our own names for variables, we must follow these rules.

- We cannot use one of python's keywords as a variable name.
- A variable name cannot contain spaces.
- The first character must be one of the letters a through z , A through Z or an underscore character (_).
- After the first character we may use the letters a through , A through Z, the Digits 0 through 9 or underscore.
- Uppercase and Lower case letters are distinct.

In addition to following these rules, you should always choose names for your variables that give an indication of what they are used for. For example, a variable that holds the temperature might be named temperature, and a variable that holds a car's speed might be named speed. You may be tempted to give variables names like x and b2, but names like these give no clue as to what the variable's purpose is.

Sample Variable Names

Variable Name	Legal or Illegal?
`units_per_day`	Legal
`dayOfWeek`	Legal
`3dGraph`	Illegal. Variable names cannot begin with a digit.
`June1997`	Legal
`Mixture#3`	Illegal. Variable names may only use letters, digits, or underscores.

Creating Variables with Assignment Statements

We use an assignment statement to create a variable and make it reference a piece of data. Here is an example of an assignment statement: age = 25

An assignment statement is written in the following general format:

Variable = expression

The equal sign (=) is known as the assignment operator. In the general format, variable is the name of a variable and expression is a value, or any piece of code that results in a value. After an assignment statement executes, the variable listed on the left side of the (=) operator will reference the value given on the right side of the (=) operator.

Ex: (variable_demo1.py)

```
# This program demonstrates a variable.
room = 503
print('I am staying in room number', room)
```

Program output:

```
I am staying in room number 503
```

Variable Reassignment : Variables are called "variable" because they can reference different values while a program is running. When you assign a value to a variable, the variable will reference that value until you assign it a different value.

Ex: variable_demo2.py

```
# This program demonstrates variable reassignment.
# Assign a value to the dollars variable.
```

Program output:

```
dollars = 2.75                                  I have 2.75 in my account.
print('I have', dollars, 'in my account.')      But now I have 99.95 in my account! #
Reassign dollars so it references a different value.
dollars = 99.95
print('But now I have', dollars, 'in my account!')
```

Data Types and Literals

Python uses data types to categorize values in memory. When an integer is stored in memory, it is classified as an int, and when a real number is stored in memory, it is classified as a float. In addition to the int and float data types, Python also has a data type named str, which is used for storing strings in memory. Python has five standard data types:

- Numbers (int , float)
- String
- List
- Tuple
- Dictionary

A number that is written into a program's code is called a numeric literal. When the Python interpreter reads a numeric literal in a program's code, it determines its data type according to the following rules:

- A numeric literal that is written as a whole number with no decimal point is considered an int.

  Examples are 7, 124, and 9.

A numeric literal that is written with a decimal point is considered afloat. Examples are 1.5, 3.14159, and 5.0.

- A String literal that is written as enclosing with double quotation marks or single quotation marks is considered a string.

  Examples are "python", 'programming', and "language"

Ex: different_variable.py

```
# Create variables to reference different types of values.
rollno = 100                                    Program output:
name = " vikas "                                Roll No: 100
fee = 5000.00                                   Name: vikas
print("Roll No :",rollno)                       Fee: 5000.00
print("Name:",name)
print("Fee:",fee)
```

Python Basic Operators

Python has several operators that can be used to perform mathematical calculations. They are

1. **Arithmetic Operators or Math operators:** Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

Symbol	operation	Description
+	Addition	Adds two numbers
-	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies one number by another
/	Division	Divides one number by another and gives the result as a floating-point number
%	Modulus	Divides one number by another and gives the remainder
**	Exponent	Raises a number to a power

//	Floor Division	Divides one number by another and gives the result as an integer

2. **Comparison (Relational) Operators:** Comparison operators are used to compare values. It either returns True or False according to the condition.

Symbol	operation	Description
==	Equal to	If the values of two operands are equal, then the condition becomes true.
!=	Not equal to	If values of two operands are not equal, then condition becomes true.
>	Greater than	If the value of left operand is greater than the value of right operand, then condition becomes true.
<	Less than	If the value of left operand is less than the value of right operand, then condition becomes true.
>=	Greater than or equal to	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.
<=	Less than or equal to	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

1. **Assignment Operators:** Assignment operators are used in Python to assign values to variables.

   Ex: x=10; here, x is a variable and it is assigned with 10.

2. **Logical Operators:** These operators are used to form compound conditions by combining two or more relations.

Symbol	operation	Description
and	Logical AND	If both the operands are true then condition becomes true.
or	Logical OR	if either of the operands is true then condition becomes true.
not	Logical NOT	if operand is false (complements the operand) then condition is true

3. **Bitwise Operators:** Bitwise operator works on bits and performs bit by bit operation.

Symbol	operation	Description
&	Bitwise AND	Operator copies a bit to the result if it exists in both operands
|	Bitwise OR	It copies a bit if it exists in either operand.
^	Bitwise XOR	It copies the bit if it is set in one operand but not both.
~	Bitwise NOT	It is unary and has the effect of 'flipping' bits.
>>	Bitwise right shift	The left operands value is moved right by the number of bits specified by the right operand.
<<	Bitwise left shift	The left operands value is moved left by the number of bits specified by the right operand.

4. **Membership Operators:** Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

Symbol	Description

in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

1. **Identity Operators:** Identity operators compare the memory locations of two objects.

Symbol	Description
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

Python Operators Precedence

Precedence is the priority order of an operator, if there are two or more operators in an expression then the operator of highest priority will be executed first then higher, and then high.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Control Structures

Normally statements in a program are executed sequentially i.e, in the order in which they are written. This is called sequential execution. Transferring control to a desired location in a program is possible through control structure. Python allows many kind of control structures which include:

1. Decision structures
   - i) Simple if
   - ii) If-else
   - iii) If-elif-else
   - iv) Nested decision structure
2. Repetition structures
   - i) While loop
   - ii) For loop

1. Decision Structures:

The if statement is used to create a decision structure, which allows a program to have more than one path of execution. If statements are present in different forms in python.

1. **Simple if:** The if statement causes one or more statements to execute only when a Boolean expression or condition is true.

General format of if statement

```
if condition:
    statement
statement
etc
```

Flow chart

Example(simpleif.py)

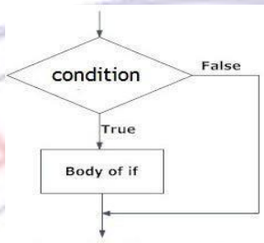
```
n=int(input("Enter a value: "))
if(n>0):
print("positive number")
```

2. **If-else:** An if-else statement will execute one block of statements if its condition is true, or another block if its condition is false.

General format of if–else statement

```
if condition:
        statement
        statement
          etc.
else:
        statement
        statement
        etc.
```

Flow chart



Example (exifelse.py)

```
a=int(input("Enter a value: "))
b=int(input("Enter b value: "))
    if(a>b):
        print("a is big")
else:
        print("b is big")
```

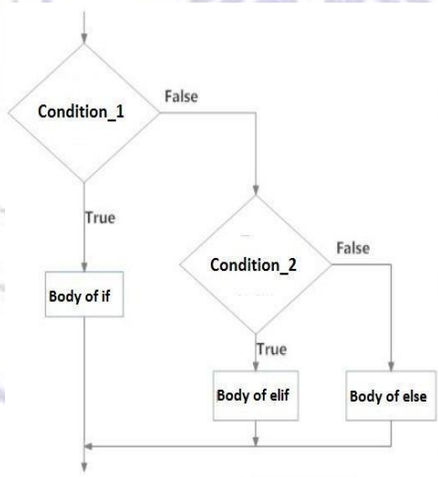
3. **The if-elif-else Statement:** Python provides a special version of the decision structure known as the if-elif-else statement. When this statement executes, condition_1 is tested. If condition_1 is true, the block of statements that immediately follow is executed, up to the elif clause. The rest of the structure is ignored. If condition_1 is false, however, the program jumps to the very next elif

clause and tests condition_2. If it is true, the block of statements that immediately follow is executed, up to the next elif clause. The rest of the structure is then ignored. This process continues until a condition is found to be true, or no more elif clauses are left. If no condition is true, the block of statements following the else clause is executed.

General format of **if – elif-else statement**

```
if condition_1:
    statement
    statement
    etc.
elif condition_2:
    statement
    statement
    etc.
.
.
else:
    statement
    statement
    etc.
```

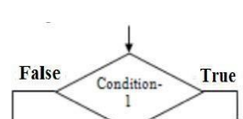
Flow chart



Exam ple(exifelif.py)

```
a=int(input("Enter a value: "))
b=int(input("Enter b value: "))
c=int(input("Enter c value: "))
if(a>b and a>c):
    print("a is big")
elif(b>c):
    print("b is big")
else:
    print("c is big")
```

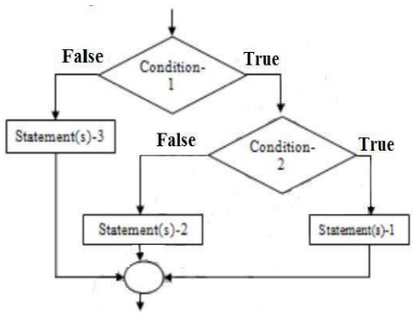
4. **Nested Decision Structures:** To test more than one condition, a decision structure can be nested inside another decision structure. Writing of if-else statement in another if or else statement is called Nested decision structure. In this, if the condition1 is false statement(s)-3 will be executed; otherwise it continues to perform condition2. If condition2 is true, statement(s)-1 will be executed. Otherwise statement(s)-2 will be executed.



General format of decision structure:

```
if condition 1:
    if condition 2:
        Statement(s)-1
    else:
        Statement(s)-2
else:
    Statement(s)-3
```

Flow chart



Example(nested.py) Nested

```
if(a>b):
    if(a>c):
    print("a is big")
    else:
    print("c is big")
else:
if(b>c):
    Print("b  is  big")
else:
    Print("c is big")
```

Repetition Structures:

A repetition structure causes a statement or set of statements to execute repeatedly. There are two categories of loops:

1. condition-controlled loop
2. count-controlled loop

A condition-controlled loop uses a true/false condition to control the number of times that it repeats. A count-controlled loop repeats a specific number of times. In Python you use the while statement to write a condition-controlled loop, and you use the for statement to write a count-controlled loop.

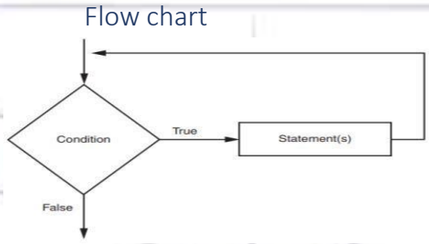
1. **The while loop: a condition-controlled loop:** A condition-controlled loop causes a statement or set of statements to repeat as long as a condition is true. In Python you use the while statement to write a condition-controlled loop. The while loop gets its name from the way it works: while a condition is true, do some task.

   The loop has two parts: (1) a condition that is tested for a true or false value, and (2) a statement or set of statements that is repeated as long as the condition is true.

General format of while

```
while condition:
    statement
    statement
    etc.
```

Flow chart



Example: (exwhile.py)

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

2. **The for loop: a count-controlled loop:** A count-controlled loop iterates a specific number of times. In Python you use the for statement to write a count- controlled loop. In Python, the for statement is designed to work with a sequence of data items. When the statement executes, it iterates once for each item in the sequence.

   General format of for loop:

```
for variable in [value1, value2, etc.]:
    statement
      statement
      etc.
```

In the for clause, variable is the name of a variable. Inside the brackets a sequence of values appears, with a comma separating each value. The for statement executes in the following manner: The variable is assigned the first value in the list, and then the statements that appear in the block are executed. Then, variable is assigned the next value in the list, and the statements in the block are executed again. This continues until variable has been assigned the last value in the list.

Example: (exfor.py):

```
list=[1,2,3,4]
for i in list:
    print(i)
```

Using the range Function with the for Loop:

Python provides a built-in function named range that simplifies the process of writing a count-controlled for loop. The range function creates a type of object known as an iterable. An iterable is an object which is similar to a list. It contains a sequence of values that can be iterated over with something like a loop.

Here is an example of a for loop that uses the range function: for

```
num in range(5):
    print(num)
```

Notice that instead of using a list of values, we call to the range function passing 5 as an argument. In this statement the range function will generate an iterable sequence of integers in the range of 0 up to (but not including) 5. So, the above code prints : 0 to 4 numbers.

If you pass two arguments to the range function, the first argument is used as the starting value of the sequence and the second argument is used as the ending limit. Here is an example:

Example:

```
for num in range(1, 5):
    print(num)
```

Output:

```
1
2
3
4
```

If you pass a third argument to the range function, that argument is used as step value. Instead of increasing by 1, each successive number in the sequence will increase by the step value. Here is an example:

Example:

```
for num in range(1, 10, 2):
    print(num)
```

Output:

```
1
3
5
7
9
```

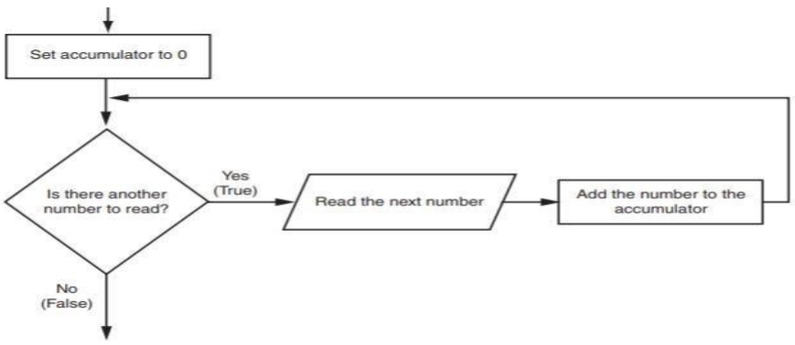
Calculating a Running Total:

A running total is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an accumulator. Many programming tasks require you to calculate the total of a series of numbers. For example, suppose you are writing a program that calculates the sum of numbers entered by the user.

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.0

Logic of Running Total:



The above flowchart shows the general logic of a loop that calculates a running total. When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop. Notice that the first step in the flowchart is to set the accumulator variable to 0. This is a critical step. Each time the loop reads a number, it adds it to the accumulator. If the accumulator starts with any value other than 0, it will not contain the correct total when the loop finishes.

Example program for calculating Running Total:

```
# this program calculates the series of numbers entered by the user. def
main( ):
   tot=0
   for i in range(1,4):
         num=int(input("Enter a number:"))
         tot=tot+num
   print("sum of 3 numbers is:"+tot)
```

Output:

```
Enter a number: 5
Enter a number: 5
Enter a number: 2
sum of 3 numbers is: 12
```

Comparing Strings:

Python allows you to compare strings. This allows you to create decision structures that test the value of a string.

For example, look at the following code:

```
name1 = 'Vikas'
name2 = 'Vikky'
if name1 == name2:
      print('The names are the same.')
else:
      print('The names are NOT the same.')
```

The == operator compares name1 and name2 to determine whether they are equal. Because the strings 'Vikas' and 'Vikky' are not equal, the else clause will display the message 'The names are NOT the same.

In addition to determining whether strings are equal or not equal, you can also determine whether one string is greater than or less than another string. This is a useful capability because programmers commonly need to design programs that sort strings in some order.

Boolean Variables:

A Boolean variable can reference one of two values: True or False. Boolean variables are commonly used as flags, which indicate whether specific conditions exist. Boolean variables are most commonly used as flags. A flag is a variable that signals when some condition exists in the program. When the flag variable is set to False, it indicates the condition does not exist. When the flag variable is set to True, it means the condition does exist.

Here are examples of how we assign values to a boolean variable:

```
hungry = True
sleepy = False
```

Input Validation Loops:

Input validation is the process of inspecting data that has been input to a program, to make sure it is valid before it is used in a computation. Input validation is commonly done with a loop that iterates as long as an input variable references bad data.

One of the most famous sayings among computer programmers is "garbage in, garbage out. This saying, sometimes abbreviated as GIGO, refers to the fact that computers cannot tell the difference between good data and bad data. If a user provides bad data as input to a program, the program will process that bad data and, as a result, will produce bad data as output. Consider the following example.

Example (payroll.py):

```
def main():
    hours = int(input('Enter the hours worked this week: '))
    pay_rate = float(input('Enter the hourly pay rate: '))
    gross_pay = hours * pay_rate
    print('Gross pay: $', format(gross_pay, ',.2f'))

main()
```

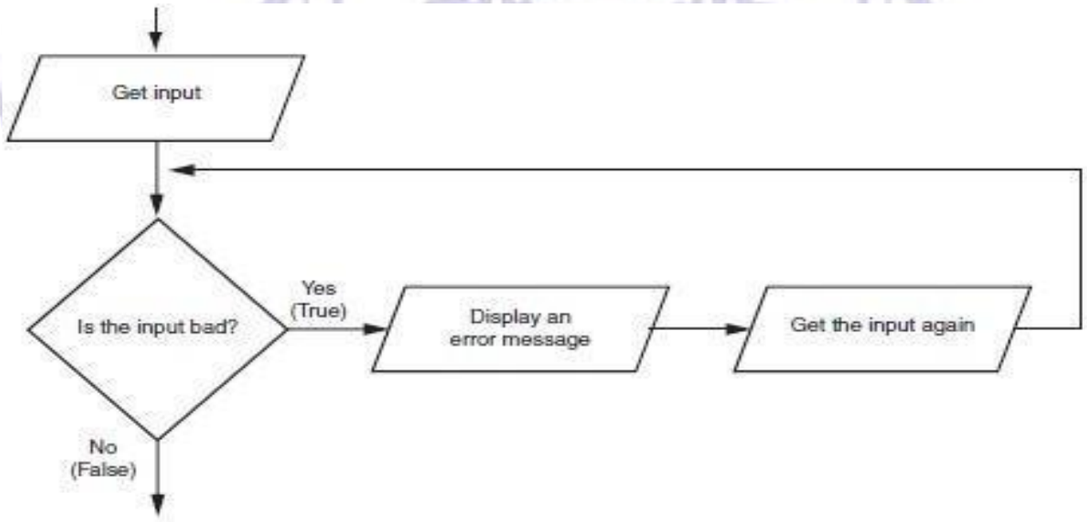
Program Output:

```
Enter the hours worked this week: 400
Enter the hourly pay rate: 20
The gross pay is $8,000.00
```

In the above example, the user or clerk entered 400 instead of 40 hours worked in the week. There are no 400 hours in a week. So, it produces wrong result.

For this reason, you should design your programs in such a way that bad input is never accepted. When input is given to a program, it should be inspected before it is processed. If the input is invalid, the program should discard it and prompt the user to enter the correct data. This process is known as input validation.

Logic containing an input validation loop:



Nested Loops:

A loop that is inside another loop is called a nested loop.

Syntax:

```
for iterating_var in sequence: for
    iterating_var in sequence:
        Statement(s)
    Statement(s)
```

Example: (pattern.py)

```
def main():
```

Output:

```
1
```

```
for i in range(1,6):                1  2
    for j in range(1,i+1):          1  2  3
        print(j,end=" ")            1  2  3  4
    print(end='\n')                 1  2  3  4  5
main()
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

LIST AND TUPLES

Sequences:

A sequence is an object that contains multiple items of data. The items that are in a sequence are stored one after the other. Python provides various ways to perform operations on the items that are stored in sequences. There are several different types of sequential objects in python. They are

i) Lists
ii) Tuples
iii) Dictionaries
iv) Sets

There are two fundamental sequence types: lists & tuples. Both lists and tuples are sequences that can hold various types of data. The difference between lists and tuples is simple. A list is mutable, which means that a program can change its contents, but a tuple is immutable, which means that once it is created, its contents cannot be changed.

Lists

Introduction to lists:

A list is an object that contains multiple data items. Lists are mutable, which means that their contents can be changed during a program execution. Lists are dynamic data structures, meaning that items may be added it them or removed from them. We can use indexing, slicing and various methods to work with lists in a program.

Creating a list:

A list is an object that contains multiple data items. Each item that is stored in a list is called an element.

Example:

```
Numbers=[10,20,30,40,50]
Names=[‘srinu’,’vikas’,’ajay’]
Stud_info=[1,’vikas’,1200.00]
```

Printing or accessing list Elements: Example:

stud_info=[1,’vikas’,1200.00]

- Printing all elements from a list
  Print(stud_info) **Output:** [1,’vikas’,1200.00]
- Accessing individual elements from lilst
  Print(stud_info[1]) **Output:** vikas
- Accessing individual elements from list using –ve index value
  Print(stud_info[-1]) **Output**:1200.00 **[Note: -1 referes last element]**
  Print(stud_info[-2]) **Output:** vikas

List ():

Python has a built-in function called list() That can convert certain types of objects to a

list

Example:

i) num1=list(range(5)) -> [0,1,2,3,4]

ii) num2=list(range(1,5)) -> [,1,2,3,4]

iii) num3=list(range(0,50,5)) -> [0,5,10,15,20,25,30,35,40,45]

The repetition operator:

The repetition operator makes multiple copies of a list and joins them alltogether. Here is the general format.

List *n

In the general format, list is a list and n is the number of copies to make

Example:

```
>>> nums=[1,2,3]*3
>>> print(nums)
```

Output: [1,2,3,1,2,3,1,2,3]

when the operand on the left side of the * symbol is a sequence (such as a list) and the operand on the right side is an integer, it becomes the repetition operator.

Iterating over a list with the for loop:

In python, we can use for loop to access all elements from a list object. Following program explains how to access all the elements from the list using for loop.

Example:

```
nums=[10,20,30,40,50]
    For i in nums:
    print(i)
```

Output: 10,20,30,40,50

Accessing list elements using Indexing:

Another way that we can access the individual elements in a list is with an index. Each element in a list has an index that specifies its position in the list. Indexing starts at '0'. So the index of the first element is '0', the index of the second element is 1 and so on. The index of the last element in a list is number of elements in the list minus one.

Example:

Accessing individual elements from the list using it's index values: i=0

```
nums=[10,20,30,40,50]
    while(i<5):
      print(nums[i])
      i=i+1
```

Output: 10 20 30 40 50

The 'len' function:

The python has a built-in function named **len()** that returns the length of a sequence, such as a list

Example:

```
nums=[10,20,30,40,50]
length=len(nums)
print(length)
```

Output:

5

Lists are mutable:

Lists in python are mutable, which means their elements can be changed.

Example1:

```
nums=[10,20,30,40,50]
nums[0]=5
Print(num)
```

Output: 5 20 30 40 50

Example2:

```
NUM_DAYS=5
def main():
    sales=[0]*NUM_DAYS
    print("Enter the sales for each day:")
    for i in range(0, NUM_DAYS):
        print("Day", i+1, ":", end=" ")
        sales[i]=float(input())
    print("Here are the values we entered ") for i
    in range (NUM_DAYS):
        print(sales[i])
main()
```

Output:

```
Enter the sales for each day:      Here are values we entered
day1:1000                          1000.0
day2:2000                          2000.0
day3:3000                          3000.0
day4:4000                          4000.0
day5:5000                          5000.0
```

Concatenating lists

Concatenating means joining two things together. We can use the '+' operator to concatenate two lists.

Example1:

```
>>>List1=[10,20,30]
>>>List2=[40,50,60]
>>>List3= list1+list2
 >>>Print(list3)
```

Output: 10 20 30 40 50 60

Example2:

```
>>>List1=['vikas', 'mahesh', 'srikanth']
>>>List1+=['vishnu', 'kranthi','vijay']
>>>print(List1)
```

Output:

[vikas,mahesh,srikanth,vishnu,kranthi,vijay]

List slicing:

A slice is a span of items that are taken from a sequence. When we take a slice from list, we get span of elements from within the list. To get a slice of a list, we write an expression in the following general format

List_name[start:end]

In the general format, start is the index of the first element in the slice, and end is the index marking the end of the slice. The expression returns a list containing a copy of the elements from start upto end

Example1:

```
>>>days=['mon','tue','wed','thurs','fri','sat','sun']
>>>Print(days[2:5]
```

Output: [wed, thurs, fri]

Example2:

```
>>>nums=[10,20,30,40,50,60,70,80]
```

>>>print(nums[:4] **Output:** 10 20 30 40

Example3:

>>>nums=[10,20,30,40,50,60,70,80]

print(nums[2:] **Output:** 30 40 50 60 70 80

Example4:

nums=[10,20,30,40,50,60,70,80]

print(nums[:-5]) **Output:** 10 20 30

Finding items in lists with the 'in' operator:

In python, we can use the **in** operator to determine whether an item is contained in a list. Here is the general format of an expression written with the 'in' operator to search for an item in a list. Here is the general format.

item in list

In the general format, item is the item for which we searching and list is a list. The expression returns true if item is found in the list or false otherwise

Example:

```
def main():
    stud_names=['vishnu', 'kranthi', 'srikanth']
    search=input("Enter a name that ur searching for:") if
    search in stud_names:
    print(search, " is found in the list") else:
    print(search, " is not found in the list") main()
```

Output:

Enter a name that ur searching for: srikanth

srikanth is found in the list

List methods and useful built-in functions

Python provides some built-in functions that are useful for working with lists.

1. **Append(item)** : It Adds an item to the end of the list.

   General Format: list.append(item)

   Example: Output:

   >>>names=['vikas','kranthi'] ['vikas','kranthi','ajay']

   >>>names.append('ajay')

   >>>print(names)

2. **index(item):** It Returns the index of the first element whose value is equal to item. A value error exception is raised if item is not found in list.

   General Format: list.index(item)

   Example: Output:

   >>>names=['vikas','kranthi','ajay'] kranthi

   >>>print(names[1])

3. **insert(index, item):**It Inserts item into the list at the specified index. When an item is inserted into a list the list expanded in size to accommodate the new item. The item was previously at the specified index and all the items after it are shifted by one position towards the end of the list.

   General Format: list.insert(index,item)

   Example: Output:

   >>>names=['vikas','kranthi','ajay'] ['vikas','aravind','kranthi','ajay']

   >>>names.insert(1,'aravind')

   >>>print(names)

4. **sort():**It Sort the items in the list so they appear in ascending order.

   General Format: list.sort()

   Example:

   >>>names=[‘vikas’,’aravind’,’kranthi’,’ajay’]

   >>>names.sort()

   >>>print(names)

   Output:

   [‘ajay’,’aravind’,’kranthi’,’vikas’]

5. **remove(item):**It Removes the first occurrence of the item from the list. A value error exception raised if item is not found in the list.

   General Format: list.remove(item)

Example:

Output:

>>>names=[‘vikas’,’aravind’,’kranthi’,’ajay’]

[‘vikas’,’aravind’,’kranthi’]

>>>names.remove(‘ajay’)

>>>print(names)

6. **reverse():**It Reverses the order of the items in the list.

   General Format: list.reverse()

   Example:

   Output:

   >>>names=[‘vikas’,’aravind’,’kranthi’,’ajay’]

   [‘ajay’,’kranthi’,’aravind’,’vikas’]

   >>>names.reverse()

   >>>print(names)

The del statement

The remove method that removes only a specified item from a list, If that item is in the list. Some situations might require that we remove an element from a specific index, regardless of the item that is stored at that index. This can be done with the del statement

Example(listdel.py):

```
def main():
    list=[1,2,3,4,5,6,7,8]
    print("Before deleting the element in the list:") print(list)
    del list[2]
    print("after deleting an item from the list")
    print(list)
main()
```

Output:

Before deleting the element in the list:

[1, 2, 3, 4, 5, 6, 7, 8]

after deleting an item from the list [1,

2, 4, 5, 6, 7, 8]

The min and max functions

Python has two built-in functions named min and max that work with sequences.

1. **min():** The min() function accepts a sequence, such as a list, as an argument and returns the item that has the lowest value in the sequence

Example:

Output

>>>num_list=[17,15,9,2,18,20]

minimum value:2

>>>print(“minimum value:”,min(num_list))

2. **max():** The max() function accepts a sequence, such as a list, as an argument and returns the item that has the highest value in the sequence

Example: Output

```
>>>num_list=[17,15,9,2 ,18,20]
>>>print("maximum value:",max(num_list))
```

maxiimum value:20

Copying lists

To make a copy of a list, we must copy the list's elements

Example: Output:

```
>>>list1=[1,2,3,4]
>>>list2=list1
>>>print(list2)
```

[1,2,3,4]

[**Note:** In the above program, the list1 and list2 variables reference the same list in memory]

Suppose we wish to make a copy of the list, so that list1 and list2 reference two separate but identical lists. One way to do this with a loop that copies each element of the list

Example: Output:

```
def main():
    list1=[1,2,3,4]
    list2=[]
    for item in list1:
        list2.append(item)
    print("after appending elements from list1 to list2:")
    print(list2)

main()
```

after appending elements from list1 to list2:
[1, 2, 3, 4]

To accomplish the same task of above program, simply we have to use concatenation operator

Example: Output:

```
>>>list1=[1,2,3,4]
>>>list2=[]
>>>list2=list2+list1
>>>print(list2)
```

[1,2,3,4]

Processing lists:

us to process the data held in list in different ways.

Python allows
Example:

```
NUM_EMP=3
hours=[0]*NUM_EMP
for item in range(NUM_EMP):
    print("Enter the hour worked by employee:",item+1,':',sep=' ',end=' ')
    hours[item]=float(input())
    pay_rate=float(input("Enter hourly pay rate:"))
for item in range(NUM_EMP):
    gross_pay=hours[item]*pay_rate
    print("Gross pay for employee",item+1,':',format(gross_pay),'.2f')
main()
```

Output:

Enter the hour worked by employee: 1 : 5

Enter hourly pay rate:100

Enter the hour worked by employee: 2 : 6

Enter hourly pay rate:100

Enter the hour worked by employee: 3 : 4

Enter hourly pay rate:100

Gross pay for employee 1 : 500.00 Gross pay for

employee 2 : 600.00 Gross pay for employee 3 :

400.00 **Totaling the values in a list**

Assuming a list contains numeric values, to calculate the total of those values we use a loop with an accumulator variable. The loop steps through the list, adding the value of each element to the accumulator.

Example:

```
def main():
    tot=0
    list=[1,2,3,4,5,6]
    for i in range(len(list)):
        tot=tot+list[i]
    print("total:", tot)
main()
```

Output

```
total:21
```

Averaging the value in a list:

The first step in calculating the average of the values in a list is to get the total of the value. The second step is to divide the total by the number of elements in the list

Example:

```
def main():
    marks=[80,80,80,80,80,80]
    total=0
    for i in marks:
        total=total+i
    average=total/len(marks)
    print("average of elements in the list:",format(average,'.2f'))
main()
```

Output

```
average of elements in the list:80
```

Passing a list an argument to a function

We can easily pass a list an argument to a function. This gives us the ability to put many of the operations that we perform on a list in their own functions.

Example:

```
def main():
    list1=[1,2,3,4,5]
    get_list(list1)
def get_list(value_list):
    total=0
    for i in value_list:
        total=total+i
    print("total:",total)

main()
```

Output:

```
total: 15
```

Returning a list from a function

A function can return a reference to a list. This gives us the ability to write a function that creates a list and adds elements to it and then returns a reference to the list.so, other

parts of the programs can work with it.

Example:

```
def main():
    numbers=get_list()
    print(numbers)
def get_list():
    list1=[]
    choice='y'
    while(choice=='y' or choice=='Y'):
        item=int(input("Enter an element:"))
        list1.append(item)
        choice=input("Do u want to add another element (y/n):")
    return list1
main()
```

Output:

```
Enter an element:10
Do u want to add another element (y/n):y
Enter an element:20
Do u want to add another element (y/n):y
Enter an element:30
Do u want to add another element (y/n):n
[10, 20, 30]
```

Working with lists and files

Some tasks may require us to save the contents of a list to a file so the data can be used at a later time. Like wise some situations may require us to read the data from a file into a list. For example

Suppose we have a file that contains a set of values that appear in random order and we want to sort the values. One technique for sorting the values in the file would be to read them into a list. Call the list's sort method. And then write the values in the list back to the file.

Saving the contents of a list to a file is a straight forward procedure. In fact, python file objects have a method named **writelines** that writes an entire list to a file. A drawback to the writelines method, However, is that it doesn't automatically write a new line("\n") at the end of each item.

Example:

```
def main()
 cities=['thangallapally','sircilla','hyd','warangal']
 list_write=open('cities.txt','w')
 list_write.writelines(cities) list_write.close()
 print("file successfully copied from list")
main()
```

Output:

```
file successfully copied from list
```

[**Note:** After this program executes, the cities.txt file will contain the following line- 'thangallapally','sircilla','hyd','warangal']

An alternative approach is to use the for loop to iterate through the list. Writing each element with a terminating new line character

Example:

```
def main():
    cities=['thangallapally','sircilla','hyd','warangal']
    list_write=open('cities.txt','w')
    for item in cities:
        list_write.write(item+'\n')
        list_write.close()
main()
```

Example: Reading data from cities.txt file:

```
def main():
    list_write=open('cities.txt','r')
    cities=list_write.readline() index=0
```

while(index<len(cities)):

UNIT II Topics and Sub Topics
Tuples- operations on tuples, Strings: Basic String Operations, String Slicing, Testing, Searching, and Manipulating Strings. **Dictionaries and Sets**: Dictionaries, Sets- operations on sets and Dictionaries. **Functions**: Introduction, Defining and Calling a Void Function, Designing a Program to Use Functions, Local Variables, Passing Arguments to Functions, Global Variables and Global Constants, ValueReturning Functions- Generating Random Numbers, Writing Our Own Value-Returning Functions, The math Module, Storing Functions in Modules. **File and Exceptions**: Introduction to File Input and Output, Using Loops to Process Files, Processing Records, Exceptions.

cities[index]=cities[index].rstrip('\n') index=index+1

print(cities)

Output:

['thangallapally','sircilla','hyd','warangal']

Tuples: A tuple is a sequence very much like a list the primary difference between tuples and lists is that tuples are immutable. That means that once a tuple is created, it can't be changed. When we create a tuple, we enclose it elements in a set of parenthesis.

Example:

```
>>>nums=(1,2,3,4,5)
>>>print(nums)
```

Output:

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

Accessing individual elements from a tuple:

Example:

```
>>> names=('vikas', 'kranthi', 'shiva')
>>> for item in names:
        print(item)
```

Output:

```
vikas
kranthi
shiva
```

Accessing individual elements from tuple using index: Example:

```
>>> names=('vikas', 'kranthi', 'shiva')
>>> for i in range(len(names)):
        print(names[i])
```

Output:

```
vikas
kranthi
shiva
```

[**Note:** if a tuple has single item we should place , (comma) after the item otherwise it can be treated as simple datatype]

Converting between lists and tuples:

Example:

```
>>>nums=(1,2,3,4)
>>>num=list(nums)
>>>print(nums)
>>>print(num)
```

Output:

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

Srting:-

String is a set of characters. Python provides a wide verity of tools and programming techniques that we can used to examine and manipulate strings.

Accessing the individual characters in a string:

python provides two techniques that we can use in python to access the individual characters in a string.

1. using the for loop
2. indexing

1. **Using the for loop:** one of the easiest ways to access the individual characters in a string is to use the for loop. Here is the general format.

   for variable in string:

   statements etc.

In the general format, variable is the name of a variable and string is either a string literal or a variable that references a string. Each time the loop iterates, variable will reference a copy of a character in string, beginning with the first character.

Example1: Output:

```
>>> name='Ram'
>>> for ch in name:
      print(ch)
```

```
R
a
m
```

Example2: Output:

```
def main():
    count=0
    sentence=input('enter sentence:')
    for ch in sentence:
      if(ch=='i'):
        count=count+1
    print("count of 'i' in a string:",count)
main()
```

```
enter sentence: animation
count of 'i' in a string: 2
```

2. **Indexing:** another way that we can access individual characters in a string is with an index. Each character in a string has an index that specifies its position in the string. Indexing starts at zero, so the index of first character is zero, the index of the

second character is 1 and so on. The index of the last character in a string is less than the no.of characters in the string.

Example: Output:

```
>>> name='Ram'
>>> for i in range(len(name)):
      print(name[i])
```

```
R
a
m
```

IndexError exception:

an indexError exception will occur if we try to use an index that is out of range for a particular string.

Example:

```
>>> name='Ram'
>>>print(name[4])
```

The above code raises IndexError exception because of an index i.e out of range for a particular string 'Ram'.

The length function: len()

The length function (len()) can also be used to get the length of a string.

Example: Output:

```
>>>name='Ram'
>>>print(len(name))
```

```
3
```

String concatenation:

A common operation that performed on string is concatenation or appending one string to the end of another string. For concatenation operation we use the operator called plus(+)

Example1:

```
>>> name="srikanth"+" kumar"
>>> print(name)
```

Output:

srikanth kumar

Example2:

```
>>>firstname="srikanth"
>>>name=firstname+" kumar"
>>>print(name)
```

Output:

srikanth kumar

Example3:

```
>>>name="srikanth" + "  " +"kumar"
>>>print(name)
```

Output:

srikanth kumar

Strings are immutable:

In python strings are immutable, which means that once they are created, can't be changed. Some operations, such as concatenation, give the impression that they modify strings but in reality they don't.

Example:(immute.py)

```
def main():
    name="srikanth"
    name=name+" kumar" print(name)

main()
```

Output:

srikanth kumar

String slicing:

When we take a slice from a string, we get a span of characters from within the string. String slices are also called as substrings

To get a slice of a string, we write an expression in the following general format

String[start:end]

In the general format start is the index of the first character in the slice, and end is the index marking the end of the slice.

Example:

```
>>> sentence="vikas degree college"
>>> print(sentence[ :5])
```

vikas

```
>>> print(sentence[6:13])
```

degree

```
>>> print(sentence[13: ])
```

college

```
>>>
```

Testing, searching and manipulating strings: Testing strings with in and not in:

In python, we can use the in operator to determine whether one string is contained in another string.

General format: string1 in string2

String1 and string2 can be either string literals or variables referencing strings. The expression returns true if string1 is found in string2

Example:(testing.py)

```
sentence="India is my country"
substring="country"

if substring in sentence:
```

Output:

string is found

```
    print("string is found")
else:
    print("string is not found")
```

String Methods:

Strings in python have numerous methods which are used for performing the following types of operations.

- Testing the values of strings
- Performing various modifications
- Searching for substring and replacing sequences of characters.

Stringvar.method(arguments)

In general format, stringvar is a variable that references a string, method is the name of the method that being called, arguments is one or more arguments being passed to the method.

String testing methods

1. **isalnum():** It returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.

   General format: stringvar.isalnum()

   Example:

   ```
   >>> input_string="vikas"
   >>> if(input_string.isalnum()):
           print("string contains alphanumeric constants")
   ```

   Output:

   string contains alphanumeric

2. **isalpha():** It return true if the string contains only alphabetic letters, and is at least one character in length. Returns false otherwise.

   General format: stringvar.isalpha()

   Example:

   ```
   >>> input_string="vikas"
   >>> if(input_string.isalpha()):
           print("string contains alphabetic letters")
   ```

   Output:

   string contains alphanumeric constants

3. **isdigit():** It return true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.

   General format: stringvar.isdigit()

   Example:

   ```
   >>> input_string="7091"
   >>> if(input_string.isdigit()):
   print("string contains digits")
   ```

   Output:

   string contains digits

4. **islower():** It return true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.

   General format: stringvar.islower()

   Example:

   ```
   >>> input_string="vikas"
   >>> if(input_string.islower()):
           print("string contains lower case letters")
   ```

   Output:

   string contains lower case letters

5. **isupper():** It return true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.

   General format: stringvar.isupper()

Example: Output:

```
>>> input_string="vikas"
>>> if(input_string.isupper()):
```

string contains upper case letters

6. **isspace():** It return true if the string contains only write space characters, and is at least one character in length. Returns false otherwise (white space characters are spaces, new lines \n, tabs\t).

   General format: stringvar.isspace()

Example: Output:

```
>>> input_string="\n"
>>> if(input_string.isspace()):
        print("string contains white space characters")
```

string contains white space characters

Searching and replacing:

Programs commonly need to search for substrings or strings that appear within other strings. To perform searching and replacing operations on strings, python provides following methods.

1. **endswith(substring):** The substring argument is a string. The method returns true if the string ends with substring.

   General format: stringvar.endswith(substring)

   Example:

```
>>> name="string.py"
>>> if(name.endswith('.py')):
        print("python file")
```

Output

python file

2. **find(substring):** The substring argument is a string. The method returns the lowest index in the string where substring is found. If substring is not found, the method returns -1.

   General format: stringvar.find(substring)

   Example:

```
>>> name="vikas degree college"
>>> position=name.find("degree")
>>> if(position!=-1):
        print("substring found at:",position)
```

Output

substring found at:6

3. **replace(old,new):** The old and new arguments are both strings. The method returns a copy of the string with all instances of old replaced by new.

   General format: stringvar.replace(old,new)

Example: Output:

```
>>> name="vikas degree college"
>>> print(name.replace('degree','PG'))
```

vikas PG college

4. **startwith(substring):** The substring argument is a string. The method returns true if the string starts with substring.

   General format: stringvar.startwith(substring)

Example:

```
>>> name="string.py"
>>> if(name.endswith('string')):
        print("string type of file")
```

Output

string type of file

String Example program

Creating login.py module: (login.py):

```
def login_generate(firstname,lastname,idnumber):
    set1=firstname[0:3]
    set2=lastname[0:3]
    set3=idnumber[-3: ]
    login=set1+set2+set3
    return login

def valid_password(password):
    correct_length=False
    has_upper=False
    has_lower=False
    has_digit=False
    valid=False
    if(len(password)>=7):
        correct_length=True
        for ch in password:
            if(ch.isupper()):
                has_upper=True
            if(ch.islower()):
                has_lower=True
            if(ch.isdigit()):
                has_digit=True
            if(correct_length and has_upper and has_lower and has_digit):
                valid=True
                 return valid
```

Dictionary and sets

Dictionary:

- The dictionary is an object which stores a collection of data. Each element that is stored in a dictionary has two parts: a key and a value.
- Dictionary elements are commonly referred to as key value pairs.
- When we want to retrieve a specific value from a dictionary. We can use the key that is associated with that value.

Creating a dictionary:

We can create a dictionary by enclosing the elements inside a set of curly braces({}). An element consist of a key, followed by a colon(:), followed by a value. The elements are separated by commas(,)

Example:

```
phonebook={'vikas':'9848657683','vishnu':'9987876745','vishal':'9912345578'}
```

The above statement creates a dictionary and assigns it to the phonebook variable.
The dictionary contains the following three elements

- The first element is 'vikas':'9848657683'. In this element the key is vishnu and the value is 9848657683.
- The second element is 'vishnu':'9987876745'. In this element the key is vishnu

and the value is 9987876745.

- The third element is 'vishal':'9912345578'. In this element the key is vishal and the value is 9912345578.

Rules for creating a Dictionary:

- The value in a dictionary can be objects of any type, but the keys must be immutable objects.
- Keys can be strings, integers, floating point values or tuples.
- Keys can't be lists or any other type of immutable objects.

Retrieving a value from a dictionary

The elements in a dictionary are not stored in any particular order. So, the order in which the elements are displayed is different than the order in which they were created. As a result, we can't use a numeric index to retrieve a value by its position from a dictionary instead. We use a key to retrieve a value.

Example: To retrieve all elements from phonebook

```
>>> phonebook={'vikas':'9848657683','vishnu':'9987876745','vishal':'9912345578'}
>>> print(phonebook)
```

Output:

```
{'vikas': '9848657683', 'vishal': '9912345578', 'vishnu': '9987876745'}
```

To retrieve a value from a dictionary, we simply write an expression in the following

General format:

```
dictionary_name[key]
```

In the general format, dictionary_name is the variable that references the dictionary and key is a key. If the key exists in the dictionary, the expression returns the value that is associated with the key. If the key doesn't exist, keyError exception is raised

Example:

```
>>> phonebook={'vikas':'9848657683','vishnu':'9987876745','vishal':'9912345578'}
>>> print('vikas')
```

Output: 9848657683

Adding elements to an existing dictionary

Dictionaries are mutable objects. We can add new key-value pairs to a dictionary with an assignment statement in the following general format:

```
Dictionary-name[key]=value
```

In the general format dictionary name is the variable that references the dictionary and key is a key. If key already exists in the dictionary, it associated value will be changed to value. If the key doesn't exist, it will be added to the dictionary along with value as its associated value

Example1:

```
>>> phonebook={'vikas':'9848657683','vishal':'9987876745','vishnu':'9912345578'}
>>> phonebook['vishnu']='9556677880'
>>> print(phonebook)
```

Output: {'vikas': '9848657683', 'vishnu': '9556677880', 'vishal': '9987876745'}

Deleting elements:

We can delete an existing key-value pair from a dictionary with the del statement

General format:

```
del dictionary-name[key]
```

In the general format, dictionary-name is the variable that references the dictionary and key is a key. After the statement executes, the key and its associated value will be deleted from the dictionary. If the key doesn't exist, a KeyError exception is raised

Example:

```
>>> phonebook={'vikas':'9848657683','vishal':'9987876745','vishnu':'9912345578'}
>>>del phonebook['vishal']                    Output:
>>>print(phonebook)                {'vikas':'9848657683','vishnu':'9912345578'}
```

To prevent a KeyError exception from being raised, we should use the **in** operator to determine whether a key exist before we try to delete and its associated value.

Example:

```
>>> phonebook={'vikas':'9848657683','vishal':'9987876745','vishnu':'9912345578'}

>>> if('vishal' in phonebook):                    Output:
      del    phonebook['vishal']      {'vikas':'9848657683','vishnu':'9912345578'}
      print(phonebook)
```

Using the in and not in operator to test for a value in a dictionary:

A KeyError exception is raised if we try to retrieve a value from a dictionary using a non-existent key. To prevent such an exception, we can use the in operator or not-in operator to retrieve a value

Example:

```
>>> phonebook={'vikas':'9848657683','vishnu':'9987876745','vishal':'9912345578'}
>>> if('vishal' in phonebook):                    Output:
        print(phonebook['vishal'])              9912345578
```

Not-in operator: this is used to determine whether a key doesn't exist in a dictionary.

Example:

```
>>> phonebook={'vikas':'9848657683','vishnu':'9987876745','vishal':'9912345578'}
>>> if('virat' not in phonebook):
        print("key is not found")
```

Getting the no.of elements in the dictionary:

We can use the built-in len functions to get the no.of elements in a dictionary.

Example:

```
>>> phonebook={'vikas':'9848657683','vishal':'9987876745','vishnu':'9912345578'}
>>> num_elements=len(phonebook)                Output:
>>> print(num_elements)                          3
```

Mixing data types in a dictionary:

The keys in a dictionary must be immutable objects, but their associated values can be any type of object. That can be lists, strings, integers, floating point values.

Example1:

```
>>> names={'vikas':[90.95,89],'vishal':[89,78,76],'vishnu':[67,87,90]}
>>> print(names)
```

Output: {'vikas': [90.95, 89], 'vishal': [89, 78, 76], 'vishnu': [67, 87, 90]}

Example2:

The values that are stored in a single dictionary can be of different types. For example,One element's value might be a string, anther element's value might be an integer. The keys can be of different types too, as long as they are immutable.

```
>>> mixed_up={'one':1,2:'two',(1,2):[1,2]}                Output:
>>> print(mixed_up)                          {(1, 2): [1, 2], 'one': 1, 2: 'two'}
```

Creating an empty dictionary:

Sometimes we need to create an empty dictionary and then add an empty list as the

program executes. We can use an empty set of curly braces to create an empty dictionary

Example:

```
>>> names={}
>>> names['vikas']=1
>>> names['vishal']=2
>>> print(names)
```

Output:

```
{'vikas': 1, 'vishal': 2, 'vishnu': 3}
```

Using the for loop to iterate over a dictionary:

We can use the for loop to iterate over all the keys in a dictionary.

General format:

```
for var in dictionary:
    statement
    statement
    etc.
```

In the general format, var is the name of a variable and dictionary is the name of the dictionary. This loop iterates once for each element in the dictionary. Each time the loop iterates var is assigned a key.

Example1:

```
>>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}
>>> for key in age:
    print(key)
```

Output:

```
vikas
vishal
vishnu
```

Example2:

```
>>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}
>>> for key in age:
    print(age[key])
```

Output:

```
20
21
23
```

Dictionary methods:

Dictionary objects have several methods that are described below

1. **clear():**The clear method deletes all the elements in a dictionary leaving the dictionary empty.

   General format: dictionary.clear()

   Example:

   ```
   >>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}
   >>> age.clear()
   >>> print(age)
   ```

   Output:

   ```
   {}
   ```

2. **get():**The get method is used to get the value associated with a specified key. If the key is not found, the method doesn't raise an exception. Instead, returns a default value.

   General format: dictionary.get(key,default)

   Example:

   ```
   >>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}
   >>> value=age.get('vishal','key not found')
   >>> print(value)
   ```

   Output:

   ```
   21
   ```

3. **items():**The items method returns all of a dictionary's keys and their associated values. They are returned as a special type of sequence known as a dictionary view. Each element in the dictionary view is a tuple. Each tuple contains a key and its associated value.

   General format: dictionary.items()

Example1:

```
>>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}
>>> age_info=age.items()
>>> print(age_info)
```

Output:

```
dict_items([('vikas', 20), ('vishal', 21), ('vishnu', 23)])
```

We can use the for loop to iterate over the tuples in the sequence.

Example2:

```
>>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}
>>> for key,value in age.items():
        print(key,value)
```

Output:

```
vikas 20
vishal 21
vishnu 23
```

4. **keys():**The keys() method returns all of a dictionary's keys as a dictionary view(tuple), which is a type of sequence. Each element in the dictionary view is a key from the dictionary.

   General format: dictionary.keys()

   Example:

   ```
   >>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}        Output:
   >>> print(age.keys())          dict_keys(['vikas', 'vishal', 'vishnu'])
   ```

5. **pop():**The pop method returns the value associated with a specified key and removes that key- value pair from the dictionary. If the key is not found, the method returns a default value.

   General format: dictionary.pop(key,default)

   Example1: Output:

   ```
   >>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}        21
   >>> print(age.pop('vishal','key not found'))
   ```

   Example2: Output:

   ```
   >>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}        key not found
   >>> print(age.pop('vikky','key not found'))
   ```

6. **popitem():**The popitem method returns a randomly selected key-value pair, and it removes that key-value pair from the dictionary. The key-value pair returned as a tuple.

   General format: dictionary.popitem()

   We can use an assignment statement to assign the returned key and value to individual variables.

   General format:

   k,v= dictionary.popitem()

   Example:

   ```
   >>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}
   >>> key,value=age.popitem()          Output:
   >>> print(key,value)                 vikas 20
   ```

7. **values():**The values method returns all a dictionary's values (without their keys) as a dictionary view, which is a type of sequence. Each element in the dictionary view is a value from the dictionary.

   General format: dictionary.values()

   Example1: Output:

   ```
   >>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}        dict_values([20, 21, 23])
   >>> print(age.values())
   ```

   We can use for loop to iterate over the sequence that is returned from the values

method.

Example2:

```
>>> age={'vikas': 20, 'vishal': 21, 'vishnu':23}
>>> for values in age.values():
        print(values)
```

Output:

```
20
21
23
```

Dictionary Example program : Program: (Birthdays.py)

```
Look_up=1
add=2
change=3
delete=4
Quit=5
def main():
    birthdays={ }
    choice=get_menulist()
    if(choice==1):
        look_up(birthdays)
    elif(choice==2):
        add(birthdays)
    elif(choice==3):
        change(birthdays)
    elif(choice==4):
        delete(birthdays)
    else:
        print("end of a program")

                def add(birthdays):
name=input("enter ur frnd name:")
    bday=input("Enter ur frnd birthday:")
    if name not in birthdays:
        birthdays[name]=bday
    else:

        print("name already exist")
    print(birthdays)

def change(birthdays):
    name=input("enter ur frnd name:")
    bday=input("Enter birthday")
```

Output:

```
iend name and their birthdays:

1.look_up
2.add
3.change
4.delete
5.quit
Enter valid choice:2
enter ur frnd name:vikas
Enter ur frnd birthday:10-11-1998
{'vikas': '10-11-1998'}
```

```
    f name in birthdays:
        birthday[name]=bday
    else:
        print("name not found")
    print(birthdays)

def get_menulist():
    choice=0
    print("Friend name and their birthdays:")
    print("----------")
    print("1.look_up")
    print("2.add")
    print("3.change")
    print("4.delete")
    print("5.quit")
    while(choice<Look_up or choice>Quit):
        choice=int(input("Enter valid choice:"))
    return choice
def look_up(birthdays):
    name=input("enter name of ur frnd:")
    bday=birthdays.get(name,'not found')
    print(bday)

def delete(birthdays):
    name=input("enter ur frnd name:")
    if name in birthdays:
del birthdays[name]
name=input("enter ur frnd name:")
    bday=input("Enter ur frnd birthday:")
    if name not in birthdays:
        birthdays[name]=bday
    else:

        print("name already exist")
    print(birthdays)

def change(birthdays):
    name=input("enter ur frnd name:")
    bday=input("Enter birthday")
```

```
    f name in birthdays:
        birthday[name]=bday
    else:
        print("name not found")
    print(birthdays)

def get_menulist():
    choice=0
    print("Friend name and their birthdays:")
    print("-----------------------------------------------------------")
    print("1.look_up")
    print("2.add")
    print("3.change")
    print("4.delete")
    print("5.quit")
    while(choice<Look_up or choice>Quit):
        choice=int(input("Enter valid choice:"))
    return choice
def look_up(birthdays):
    name=input("enter name of ur frnd:")
    bday=birthdays.get(name,'not found')
    print(bday)

def delete(birthdays):
    name=input("enter ur frnd name:")
    if name in birthdays:
del birthdays[name]
```

Adding elements to a set:

Sets are mutable objects. So, we can add items to them and remove items from them. We use the add method to add an element to a set

Example:

```
>>> myset=set()
>>> myset.add(1)
>>> myset.add(2)
>>> myset.add(3)
>>> print(myset)
```

Output:

```
{1, 2, 3}
```

We can add group of elements to a set all at one time, with the update(). When we call the update method as an argument, we pass an objects that contains iterable

elements, such as a list, string or another set.

Example1:

```
>>> myset=set([1,2,3])
>>> myset.update([4,5,6])
>>> print(myset)
```

Output:

```
{1, 2, 3, 4, 5, 6}
```

Example2:

```
>>> myset=set([1,2,3])
>>> myset.update("vikas")
>>> print(myset)
```

Output:

```
{1, 2, 3, 'a', 'i', 's', 'k', 'v'}
```

Removing elements from a set:

we can remove an item from a set with either the remove method or the discard method. We pass the item that we want to remove as an argument to either method, and that item is removed from the set. The only difference between the two methods is how they behave when the specified item is not found in the set. The remove method raises a KeyError exception, but the discard method doesn't raise an exception.

Example1:

```
>>> myset=set([1,2,3])
>>> myset.remove(2)
>>> print(myset)
```

Output:

```
{1, 3}
```

we can clear the elements of a set by calling the **clear** method.

Example2:

```
>>> myset=set([1,2,3])
>>> myset.clear()
>>> print(myset)
```

Output:

```
set()
```

KLERTHANA DEGREE COLLEGE BELA ADILABAD (TS)

Getting the number of elements in a set:

As with lists, tuples and dictionaries, we can use the len function to get the number of elements in a set.

Example: Output:

```
>>>myset=set([1,2,3,4])          4
>>>print(len(myset))
```

Using the for loop to iterate a set:

we can use the for loop to iterate all the elements in a set

General format:

```
for var in set:
    statement(s)
    etc
```

In the general format 'var' is the name of a variable and set is the name of a set .This loop iterates once for each element in the set and each time the loop iterates 'var' is assigned an element.

Example: Output:

```
>>> myset=set(['vikas','vishal','vishnu'])      vikas
>>> for name in myset:                          vishal
    print(name)                                  vishnu
```

Using the 'in' and 'not in ' operators to test for a value in a set we can use **in** operator to determine whether a value exists in a set **Example:**

```
>>> myset=set(['vikas','vishal','vishnu'])      Output:
>>> if('vishal' in myset):                      Element found
        print("Element found")
```

we can also use **not in** operator to determine if a value doesn't exist in a set.

```
>>> myset=set(['vikas','vishal','vishnu'])      Output:
>>> if('vikky' not in myset):                   Element not found
        print("Element not found")
```

Finding the union of sets:

The union of two sets is a set that contains all the elements of both sets. In python, we can call the union method to get the union of two sets.

General format:

```
set1.union(set2)
```

In the general format, set1 and set2 are sets. This method returns a set that contains the elements of both set1 and set2.

Example: Output:

```
>>> set1=set([1,2,3])                {1, 2, 3, 4}
>>> set2=set([2,3,4])
>>> set3=set1.union(set2)
>>> print(set3)
```

We can also use the '|' operator to find the union of two sets.

General format:

set1|(set2)

In the general format set1 & set2 are sets. The expression returns a set that contains the elements of both set1 and set2.

Example: Output:

```
>>> set1=set([1,2,3])                {1, 2, 3, 4}
>>> set2=set([2,3,4])
>>> set3=set1|set2
>>> print(set3)
```

Finding the intersection of sets:

The intersection of two sets is a set that contains only the elements that are found in both sets. In python, we can call the intersection method to get the intersection of two sets.

General format:

set1.intersction(set2)

In the general format set1 and set2 are sets the method returns a set that contains the elements that are found in both set1 and set2

Example: Output:

```
>>> set1=set([1,2,3])                {2, 3}
>>> set2=set([2,3,4])
>>> set3=set1.intersection(set2)
>>> print(set3)
```

We can also use **&** operator to find the intersection of two sets.

General format:

set1 & set2

In the general format, set1 and set2 are sets.The expression returns a set that contains the elements that are found in both set1 and set2.

Example: Output:

```
>>> set1=set([1,2,3])                {2, 3}
>>> set2=set([2,3,4])
>>> set3=set1 & set2)
>>> print(set3)
```

Finding the difference of sets:

The difference of set1 and set2 are the elements that appear in set1 but don't appear in set2. In python, we can call the difference method to get the difference of two sets.

General format:

set1.difference(set2)

In the general format, set1 and set2 are sets. The method returns a set that contains the elements that are found in set1 but not in set2

Example:

```
>>> set1=set([1,2,3])
>>> set2=set([2,3,4])
>>> set3=set1.difference(set2)
>>> print(set3)
```

Output:

```
{1}
```

We can also uses the '**-**' operator to find the difference of two sets.

General format:

set1-set2

In the general format set1 and set2 are sets. The method returns a set that contains the elements that are found in set1 but not in set2

```
>>> print(set1>=set2
```

We can also use '<=' to determine whether one set is a subset of anther.

General format:

set2<=set1

In the general format set1 and set2 are sets. The expression returns true if set2 is subset of set1. Otherwise, returns fasle.

Example:

```
>>> set1=set([1,2,3,4])
>>> set2=set([2,3])
>>> print(set2<=set1)
```

Output:

```
True
```

Supersets: In python we can call the issuperset method to determine whether one set is a superset of another.

General format:

set1.issuperset(set2)

In the general format, set1 and set2 are sets, the method returns true if set1 is superset of set2. Otherwise it returns false.

Example:

```
>>> set1=set([1,2,3,4])
>>> set2=set([2,3])
>>> print(set1.issuperset(set2))
```

Output:

```
True
```

We can also use '>=' operator to determine whether one set is a super set of another.

General format: set1>=set

In the general format set1 and set2 are sets. The expression returns true if set1 is superset of set2. Otherwise, returns fasle.

Example: Output:

```
>>> set1=set([1,2,3,4])
>>> set2=set([2,3])
```

True

To open a file for binary writing we use 'wb' as the mode when we call the open function.

Example for opening a mydata.txt in binary writing mode:

```
output_file=open('vikas.txt','wb')
```

Once we have opened a file for binary writing. We call the pickle's module's dump function.

General format:

```
pickle.dump(object, file)
```

In the general format, object is a variable that references the object. We want to pickle, and file is a variable that references a file object. After the function executes the object referred by object will be serialized and return to the file.

Example:(pickle.py)

```
import pickle
phonebook={'vikas':'9848657683','vishal':'9987876745','vishnu':'9912345578'}
output_file=open('vikas.txt','wb')
pickle.dump(phonebook,output_file)
output_file.close()
print("contents are copied")
```

Output:

contents are copied

To open a file for binary reading, we use 'rb' as the mode when we call the open function.

Example for opening a mydata.txt in reading binary mode:

```
input_file=open('vikas.txt','rb')
```

Once we have opened a file for binary reading. We call the pickle module's load function

General format: pickle.load(file)

in the general format, object is a variable, and file is a variable that references a file object. After the function executes, the object variable will references an object that was retrieved from the file and unpickled.

Example:(unpickle.py)

```
import pickle
input_file=open('vikas.txt','rb')
pb=pickle.load(input_file)
input_file.close()
print(pb)
```

Output:

{'vikas': '9848657683', 'vishnu': '9912345578', 'vishal': '9987876745'}

Functions

Function: A function is a group of statements that exist within a program for the purpose of performing a specific task. Instead of writing a large program as one long sequence of statements, it can be written as several small functions, each one performing a specific part of the task. These small functions can then be executed in the desired order to perform the overall task. This approach is sometimes called divide and conquer because a large task is divided into several smaller tasks that are easily performed.

Benefits of using Functions (or) Benefits of Modularizing a Program with Functions:

1. **Simpler Code:** A program's code tends to be simpler and easier to understand when it is broken down into functions. Several small functions are much easier to read than one long sequence of statements.
2. **Code Reuse:** Functions also reduce the duplication of code within a program. If a specific operation is performed in several places in a program, a function can be written once to perform that operation, and then be executed any time it is needed. This benefit of using functions is known as code reuse because you are writing the code to perform a task once and then reusing it each time you need to perform the task.
3. **Better Testing:** When each task within a program is contained in its own function, testing and debugging becomes simpler. Programmers can test each function in a program individually, to determine whether it correctly performs its operation. This makes it easier to isolate and fix errors.
4. **Faster Development:** Suppose a programmer or a team of programmers is developing multiple programs. They discover that each of the programs perform several common tasks, such as asking for a username and a password, displaying the current time, and so on. It doesn't make sense to write the code for these tasks multiple times. Instead, functions can be written for the commonly needed tasks, and those functions can be incorporated into each program that needs them.
5. **Easier Facilitation of Teamwork:** Functions also make it easier for programmers to work in teams. When a program is developed as a set of functions that each performs an individual task, then different programmers can be assigned the job of writing different functions.

Function Names:

Just as we name the variables that we use in a program, we also name the functions. A function's name should be descriptive enough so that anyone reading our code can reasonably guess what the function does.

Rules for defining function names:

- You cannot use one of Python's key words as a function name.
- A function name cannot contain spaces.
- The first character must be one of the letters a through z, A through Z, or an underscore character (_).

- After the first character we may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct.

Defining and Calling a void Function

Function: A function is a group of statements that exist within a program for the purpose of performing a specific task.

Defining function: To create a function we write its definition. Here is the general format of a function definition in Python:

```
def function_name():
    statement
    statement
    etc.
```

The first line is known as the function header. It marks the beginning of the function definition. The function header begins with the key word def, followed by the name of the function, followed by a set of parentheses, followed by a colon.

Beginning at the next line is a set of statements known as a block. A block is simply a set of statements that belong together as a group. These statements are performed any time the function is executed.

Example for defining a function:

```
def display( ):
    print("I am in display method")
    print("I will be executed when I called")
```

This code defines a function named display. The display function contains a block with two statements. Executing the function will cause these statements to execute.

Calling a Function: A function definition specifies what a function does, but it does not cause the function to execute. To execute a function, we must call it.

Example for calling a function:

```
display( )
```

When a function is called, the interpreter jumps to that function and executes the statements in its block.

Example(simplefunction.py):

```
def display( ):
    print("I am in display method")
    print("I will be executed when I called")
display( )
```

Output:

```
I am in display method
I will be executed when I called
```

Indentation in Python

In Python, each line in a block must be indented. When you indent the lines in a block, make sure each line begins with the same number of spaces. Otherwise an error will occur. In an editor there are two ways to indent a line: (1) by pressing the Tab key at the beginning of the line, or (2) by using the spacebar to insert spaces at the beginning of the line. You can use either tabs or spaces when indenting the lines in a block, but

don't use both. Doing so may confuse the Python interpreter and cause an error.

Programmers typically draw a separate flowchart for each function in a program. For IDLE, as well as most other Python editors, automatically indents the lines in a block. When we type the colon at the end of a function header, all of the lines typed afterward will automatically be indented. After we have typed the last line of the block we press the Backspace key to get out of the automatic indentation.

Designing a Program to Use Functions

Programmers commonly use a technique known as top-down design to break down an algorithm into functions. Flow charts are common tools for designing programs. In a flowchart, a function call is shown with a rectangle that has vertical bars at each side. The example shown below shows how we would represent a call to the message

function.

Consider the following example:

(two_functions.py):

```
# This program has two functions. First we
# define the main function.
def main( ):
    print "I have a message for you"
    message( )
    print "Goodbye!"
# Next we define the message function.
def message( ):
    print "I am Arthur"
    print "King of Britons"
# Call the main function.
main( )
```

Output:

```
I have a message for you.
I am Arthur,
King of Britons
Goodbye!
```

example, Below figure shows how the main function and the message function of above program would be flowcharted. When drawing a flowchart for a function, the starting terminal symbol usually shows the name of the function and the ending terminal symbol usually reads Return.

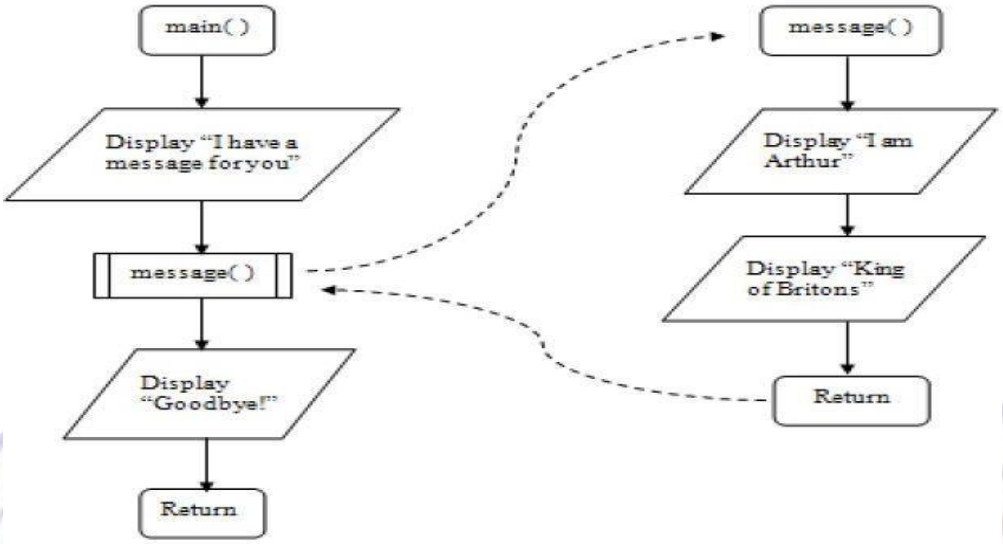
In the below design, we can see how control of a program is transferred to a function when it is called, and then returns to the part of the program that called the function when the function ends.

Programmers commonly use a technique known as top-down design to break down an algorithm into functions. The process of top-down design is performed in the following manner:

- The overall task that the program is to perform is broken down into a series of subtasks.
- Each of the subtasks is examined to determine whether it can be further broken down into more subtasks. This step is repeated until no more subtasks can be identified.

- Once all of the subtasks have been identified, they are written in code.

Flow chart for (two_functions.py)



Local Variables

The variables that are defined inside the functions are called as local variables. They cannot be accessed by statements that are outside the function. Different functions can have local variables with the same names because the functions cannot see each other's local variables.

An error will occur if a statement in one function tries to access a local variable that belongs to another function. Local variables memory will be deallocated when it's function terminates.

Scope and Local Variables: A variable's scope is the part of a program in which the variable may be accessed. A variable is visible only to statements in the variable's scope. A local variable's scope is the function in which the variable is created, no statement outside the function may access the variable.

Example: (local.py):

```
def   main(   ):
    getname( )
    print("hello",name) # it causes error because local variable of getname.
                                # so, "name" variable visible to only getname(  ).
def getname(  ):
   Name = input("Enter ur name:")
main(  )
```

Passing Arguments to Functions

An argument is any piece of data that is passed into a function when the function is called. A parameter is a variable that receives an argument that is passed into a function. The following example explains how to pass arguments to functions.

Example:(arguments.py):

```
def main(  ):
  a=int(input("Enter a number:")
```

```
    b=int(input("Enter b number:")
    print(" Addition of two numbers:")
    add(a,b)

def add(x,y):
    sum=x+y
    Print("addition of x,y is:",sum)

main( )
```

In the above program, a,b are **arguments** which are passed to called function **add** and x,y are **parameters** which are received the values coming from calling function.

Making Changes to Parameters

When an argument is passed to a function in Python, the function parameter variable will reference the argument's value. However, any changes that are made to the parameter variable will not affect the argument. The following program demonstrates it.

Example(changeparameter.py):

```
def main():
    value=70
    print("The argument value in main:",value)
    change_me(value)
    print("after changing the parameter value in change_me:",value)

def change_me(value):
    print("I am going to change the value:",value)
    value=10
    print("the parameter value in change me:",value)

main()
```

In the above program, the reassingnment changes the "value" variable inside the change_me function but it does not affect the value variable in main.

Program Output:

```
The argument value in main: 70
I am going to change the value: 70
the parameter value in change me: 10
after changing the parameter value in change_me: 70
```

Global Variables and Global Constants

Global variables:

When a variable is created by an assignment statement that is written outside all the functions in a program file, the variable is called global variable. A global variable can be accessed by any statement in the program file, including the statements in any function.

Example:(global.py):

```
Value=10    #gloabal variable
def main():
   Print("global variable value is:",value)
main()
```

Output:

```
global variable value is: 10
```

In the above example, "value" is global variable. So, inside the main() we accessed it directly. Global variables can be accessed directly in any functions in the program.

Reasons for most of the programmers avoid using global variables:

Most programmers agree that you should restrict the use of global variables, or not use them at all. The reasons are as follows:

- Global variables make debugging difficult. Any statement in a program file can change the value of a global variable. If you find that the wrong value is being stored in a global variable, you have to track down every statement that accesses it to determine where the bad value is coming from. In a program with thousands of lines of code, this can be difficult.
- Functions that use global variables are usually dependent on those variables. If you want to use such a function in a different program, most likely you will have to redesign it so it does not rely on the global variable.
- Global variables make a program hard to understand. A global variable can be modified by any statement in the program. If you are to understand any part of the program that uses a global variable.

Global constants: A global constant is a global name that references a value that cannot be changed. Because a global constant's value cannot be changed during the program's execution, you do not have to worry about many of the potential hazards that are associated with the use of global variables.

Although the Python language does not allow you to create true global constants, you can simulate them with global variables. If you do not declare a global variable with the global key word inside a function, then you cannot change the variable's assignment inside that function.

Example(global2.py):

```
x=5
y=2 # global constant
def main():
   global x
   x=10 #changing the global variable value
```

Output:

```
sum: 20
subtraction: 8
```

```
    y=10 #local variable to main
    print("sum",(x+y))
    show()
def show():
print("subtraction:",x-y) #y is global constant it's value is 2 and x is global variable.
    main()
```

Value-Returning Functions:

A value-returning function is a special type of function. It is like a simple function in the following ways.

- It is a group of statements that perform a specific task.
- When you want to execute the function, you call it.

When a value-returning function finishes, however, it returns a value back to the part of the program that called it. The value that is returned from a function can be used like any other value: it can be assigned to a variable, displayed on the screen, used in a mathematical expression.

Writing Your Own Value-Returning Functions:

A value-returning function has a return statement that returns a value back to the part of the program that called it. Here is the general format of a value-returning function definition in Python:

```
def function_name():
    statement
    statement
    etc.
    return expression
```

The first line is known as the function header. The function header begins with the key word def, followed by the name of the function, followed by a set of parentheses, followed by a colon.

Beginning at the next line is a set of statements known as a block. In value-returning function block one of the statement should be a **return** statement.

The value of the **expression** that follows the keyword return will be sent back to the part of the program that called the function. This can be any value, variable, or expression that has a value

Example(returnfunction.py):

```
def main():`
    a=int(input("enter a number:"))
    b=int(input("enter b number:"))
    c=add(a,b)
    print("sum:",c)
def add(a,b):
    c=a+b
    return c
main()
```

Output:

```
enter a number: 5
enter b number: 5
sum: 10
```

In the above program, **add** function returns a value which is assigned in 'c' variable of main function.

Standard Library Functions and the import Statement

Python, as well as most other programming languages, comes with a standard library of functions that have already been written for you. These functions, known as library functions, make a programmer's job easier because they perform many of the tasks that programmers commonly need to perform. In fact, you have already used several of Python's library functions. Some of the functions that you have used are print, input, and range. Python has many other library functions.

Some of Python's library functions are built into the Python interpreter. If you want to use one of these built-in functions in a program, you simply call the function. This is the case with the print, input, range etc.

Many of the functions in the standard library, however, are stored in files that are known as modules. These modules, which are copied to your computer when you install Python, help organize the standard library functions. For example, functions for performing math operations are stored together in a module, functions for working with files are stored together in another module, and so on.

In order to call a function that is stored in a module, you have to write an import statement at the top of your program. An import statement tells the interpreter the name of the module that contains the function. For example, one of the Python standard modules is named math. The math module contains various mathematical functions that work with floating-point numbers. If you want to use any of the math module's functions in a program, you should write the following import statement at the top of the program:

```
import math
```

This statement causes the interpreter to load the contents of the math module into memory and makes all the functions in the math module available to the program.

Because you do not see the internal workings of library functions, many programmers think of them as black boxes. The term "black box" is used to describe any mechanism that accepts input, performs some operation (that cannot be seen) using the input, and produces output.

Fig. A library function viewed as a black box

Generating Random Numbers

Random numbers are useful for lots of different programming tasks. The following are just a few examples.

- Random numbers are commonly used in games. For example, computer games that let the player roll dice use random numbers to represent the values of the dice.
- Random numbers are useful in simulation programs. In some simulations, the computer must randomly decide how a person, animal, insect, or other living being will behave. Formulas can be constructed in which a random number is used to determine various actions and events that take place in the program.

- Random numbers are useful in statistical programs that must randomly select data for analysis.
- Random numbers are commonly used in computer security to encrypt sensitive data.

Library functions for working with random numbers:

Python provides several library functions for working with random numbers. These functions are stored in a module named random in the standard library. To use any of these functions you first need to write this import statement at the top of your program:

```
import random
```

The following are the library functions for working with random numbers:

1. **randint**( ): This function is used to generate random integer numbers. Because the randint function is in the random module, we will need to use dot notation to refer to it in our program. In dot notation, the function's name is **random.randint**. On the left side of the dot (period) is the name of the module, and on the right side of the dot is the name of the function.

   The following interactive session depicts using randint function.

   Example:

   ```
   >>> import random
   >>> number=random.randint(1,100)
   >>> print(number)
   7
   >>>
   ```

   In the above example, randint function returns a random integer number between 1 to 100

2. **randrange**( ):The randrange function takes the same arguments as the range function.The difference is that the randrange function does not return a list of values. instead, it returns a randomly selected value from a sequence of values.

   Example of randrange function which having single argument:

   ```
   >>> import random
   >>> number=random.randrange(10)
   >>> print(number)
   9
   >>>
   ```

   In the above interactive session, The randrange function will return a randomly selected number from the sequence of values 0 up to, but not including, the ending limit 10.

   Example of randrange function which having two arguments:

   ```
   >>> import random
   >>> number=random.randrange(5,20)
   >>> print(number)
   8
   ```

In the above interactive session, The randrange function will return a randomly selected number from the sequence of values 5 up to, but not including, the ending limit 20.

Example of randrange function which having three arguments:

```
>>> import random
>>> number=random.randrange(0,101,10)
>>> print(number)
31
>>>
```

In the above interactive session, the randrange function returns a randomly selected value from the following sequence of numbers:

[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

3. **random**( ): The random function returns a random floating-point number. You do not pass any arguments to the random function. When you call it, it returns a random floating point number in the range of 0.0 up to 1.0.

   The following interactive session depicts the random() function:

```
>>> import random
>>> number=random.random()
>>> print(number)
0.8106100877945103
>>>
```

4. **uniform**( ) : The uniform function also returns a random floating-point number, but allows you to specify the range of values to select from.

   The following interactive session depicts the uniform() function:

```
>>> import random
>>> number=random.uniform(0.1,1.2)
>>> print(number)
0.8829074535237236
>>>
```

The math Module

The math module in the Python standard library contains several functions that are useful for performing mathematical operations. These functions typically accept one or more values as arguments, perform a mathematical operation using the arguments, and return the result. To use the functions of math module in our program, we have to import it. The following table lists the functions in math module.

Math Module	Function Description
acos(x)	Returns the arc cosine of x, in radians.
asin(x)	Returns the arc sine of x, in radians.
atan(x)	Returns the arc tangent of x, in radians.
ceil(x)	Returns the smallest integer that is greater than or equal to x.

cos(x)	Returns the cosine of x in radians.
degrees(x)	Assuming x is an angle in radians, the function returns the angle converted to degrees.
exp(x)	Returns e^x
floor(x)	Returns the largest integer that is less than or equal to x.
hypot(x, y)	Returns the length of a hypotenuse that extends from (0, 0) to (x, y).
log(x)	Returns the natural logarithm of x.
log10(x)	Returns the base-10 logarithm of x.
radians(x)	Assuming x is an angle in degrees, the function returns the angle converted to radians.
sin(x)	Returns the sine of x in radians.
sqrt(x)	Returns the square root of x.
tan(x)	Returns the tangent of x in radians.

Example:(exmath.py)

```
import math
def main():
    print("sin(x) :",format(math.sin(30),'.2f'))
    print("cos(x) :",format(math.cos(90),'.2f'))
    print("tan(x) :",format(math.tan(45),'.2f'))
    print("floor(x) :",math.floor(3.7))
    print("ceil(x) :",math.ceil(3.7))
    print("log(x) :",format(math.log(8),'.2f'))
    print("log10(x) :",format(math.log10(100),'.2f'))
    print("sqrt(x) :",math.sqrt(16))
main()
```

Output:

```
sin(x) : -0.99
cos(x) : -0.45
tan(x) : 1.62
floor(x) : 3
ceil(x) : 4
log(x) : 2.08
log10(x) : 2.00
sqrt(x) : 4.0
```

Storing Functions in Modules

A module is simply a file that contains Python code. When you break a program into modules, each module should contain functions that perform related tasks. Modules make it easier to reuse the same code in more than one program. If you have written a set of functions that are needed in several different programs, you can place those functions in a module. Then, you can import the module in each program that needs to call one of the functions.

For example, following are the two modules:

Example1 (module): circle.py

```
import math
def area(radius):
    return math.pi*radius**2
def circumference(radius):
    return 2*math.pi*radius
```

Example2 (module): rectangle.py

```
def area(length,breadth):
    return length*breadth
def perimeter(length,breadth):
    return 2*(length+breadth)
```

The above two modules has certain functions that are useful to calculate area of circle, circumference of a circle, area of rectangle and perimeter of a rectangle.

Let us use those functions from circle and rectangle modules in our following program:

Exmodule.py import

circle import

rectangle def

main():

```
    print("area of circle is:",format(circle.area(3.0),'.2f'))
    print("area of rectangle is:",rectangle.area(2,3))
    print("perimeter of rectangle is:",rectangle.perimeter(3,2))
main()
```

Output:

```
area of circle is: 28.27
area of rectangle is: 6
perimeter of rectangle is: 10
```

file:

Introduction to File Input and Output:

Programmers usually refer to the process of saving data in a file as "writing data to" the file. When a piece of data is written to a file, it is copied from a variable in RAM to the file. The term output file is used to describe a file that data is written to. It is called an output file because the program stores output in it.

The process of retrieving data from a file is known as "reading data from" the file. When a piece of data is read from a file, it is copied from the file into RAM, and referenced by a variable. The term input file is used to describe a file that data is read from. It is called an input file because the program gets input from the file.

There are always three steps that must be taken when a file is used by a program.

1. **Open the file**: Opening a file creates a connection between the file and the program. Opening an output file usually creates the file on the disk and allows the program to write data to it. Opening an input file allows the program to read data from the file.
2. **Process the file**: In this step data is either written to the file (if it is an output file) or read from the file (if it is an input file).
3. **Close the file**: When the program is finished using the file, the file must be closed. Closing a file disconnects the file from the program.

Types of Files:

In general, there are two types of files: text and binary. Python allows you to work with text and binary files.

A text file contains data that has been encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. As a result, the file may be opened and viewed in a text editor such as Notepad.

A binary file contains data that has not been converted to text. The data that is stored in a binary file is intended only for a program to read. As a consequence, you cannot view the contents of a binary file with a text editor.

File Access Methods:

Most programming languages provide two different ways to access data stored in a file: sequential access and direct access.

In sequential access file, you access data from the beginning of the file to the end of the file. If you want to read a piece of data that is stored at the very end of the file, you have to read all of the data that comes before it—you cannot jump directly to the desired data. This is similar to the way cassette tape players work. If you want to listen to the last song on a cassette tape, you have to either fast-forward over all of the songs that come before it or listen to them. There is no way to jump directly to a specific song.

In direct access file (which is also known as a random access file), you can jump directly to any piece of data in the file without reading the data that comes before it. This is similar to the way a CD player or an MP3 player works. You can jump directly to any song that you want to listen to.

File operations:

1. **Opening a File**: We use the open function in Python to open a file. The open function creates a file object and associates it with a file on the disk. Here is the general format of how the open function is used:

```
file_variable = open(filename, mode)
```

In the general format:

- **file_variable -** is the name of the variable that will reference the file object.
- **Filename -** is a string specifying the name of the file.
- **Mode -** is a string specifying the mode (reading, writing, etc.) in which the file will be opened.

The following table lists the some of file opening modes for working with files.

Mode	Description
'r'	Open a file for reading only. The file cannot be changed or written to.
'w'	Open a file for writing. If the file already exists, erase its contents. If it does not exist, create it.
'a'	Open a file to be written to. All data written to the file will be appended to its end. If the file does not exist, create it.

2. Writing Data to a File: Once you have opened a file, we use the file object's methods to perform operations on the file. If we want to write data to a file, file objects have a method named **write** that can be used to write data to a file. Here is the general format of how you call the write method:

```
file_variable.write(string)
```

In the format, file_variable is a variable that references a file object, and string is a string that will be written to the file. The file must be opened for writing (using the 'w' or 'a' mode). Otherwise, an error will occur.

4. **Reading Data from a File:** If a file has been opened for reading (using the 'r' mode) we can use the file object's **read method** to read its entire contents into memory. When you call the read method, it returns the file's contents as a string. Here is the general format of how we call the **read method**:

   file_contents=file_variable.read( )

   In the format, file_contents is a variable which stores the entire file contents returned by file object's read method.

5. **Closing a File:** After all operations are performed on a file, we should close it. To do so, we should call file object's **close method**. Closing a file disconnects the file from the program. In some systems, failure to close an output file can cause a loss of data. Here is the general format of how we call the **close method**:

   file_variable.close( )

The following two example programs for writing and reading data from a file named: vikas.txt

filewrite.py

```
def main():
    fv=open('vikas.txt','w')
    fv.write('COURSES OFFERED \n')
    fv.write('----------------------\n')
    fv.write('BSc(MPCs)\n')
    fv.write('BSc(MStCs)\n')
    fv.write('BSc(MCCs)\n')
    fv.close()
print("file successfully copied")
```

fileread.py

```
def main():
    fv=open('vikas.txt','r')
    file_contents=fv.read()
    fv.close()
    print(file_contents)
main()
```

fileread.py output:

```
COURSES OFFERED
----------------------
BSc(MPCs)
BSc(MStCs)
BSc(MCCs)
```

filewrite.py output:

```
file successfully copied
```

Using Loops to Process Files

Although some programs use files to store only small amounts of data, files are typically used to hold large collections of data. When a program uses a file to write or read a large amount of data, a loop is typically involved. For example, look at the code in following Program .

This program gets sales amounts for a series of days from the user and writes those amounts to a file named sales.txt. The user specifies the number of days of sales data he or she needs to enter. In the sample run of the program, the user enters sales amounts for five days.

Example (writefile.py)

```
def main():
    num_days = int(input('For how many days do you have sales? '))
```

```
    sales_file = open('sales.txt', 'w')
    for count in range(1, num_days + 1):
        sales = float(input('Enter the sales for day #', str(count) + ': '))
        sales_file.write(str(sales) + '\n')
    sales_file.close()
    print('Data written to sales.txt.')

main()
```

Program output:

```
For how many days do you have sales? 3
Enter the sales for day #1: 1000
Enter the sales for day #2: 2000
Enter the sales for day #3: 1500
Data written to sales.txt.
```

Reading a File with a Loop and Detecting the End of the File

If we need to write a program that read all of the amounts in the sales.txt file and calculates their total. We can use a loop to read the items in the file, but we need a way of knowing when the end of the file has been reached. In python, the readline() method returns an empty string(` `) when it has attempted to read beyond the end of a file. This makes it possible to write while loop that determines when the end of a file has been reached. Following program reads and displays all of the values in the sales.txt.

Example(readfile.py)

```
def main():
    sales_file = open('sales.txt', 'r')
    line = sales_file.readline()
    while line != '':
        amount = float(line)
        print(format(amount, '.2f'))
        line = sales_file.readline()
    sales_file.close()
main()
```

Output:

```
1000
2000
1500
```

Using Python's for Loop to Read Lines

The Python language also allows you to write a for loop that automatically reads line in a file without testing for any special condition that signals the end of the file. The loop does not require a priming read operation, and it automatically stops when the end of the file has been reached. When you simply want to read the lines in a file, one after the other, this technique is simpler and more elegant than writing a while loop that explicitly tests for an end of the file condition. Here is the general format of the loop:

```
for variable in file_object:
```

statement

statement

etc.

In the general format, variable is the name of a variable and file_object is a variable that references a file object. The loop will iterate once for each line in the file. The first time the loop iterates, variable will reference the first line in the file (as a string), the second time the loop iterates, variable will reference the second line, and so forth.

Example(readfor.py)

```
def main():
    sales_file = open('sales.txt', 'r')
    for line in sales_file:
        amount = float(line)
        print(format(amount, '.2f'))
    sales_file.close()
main()
```

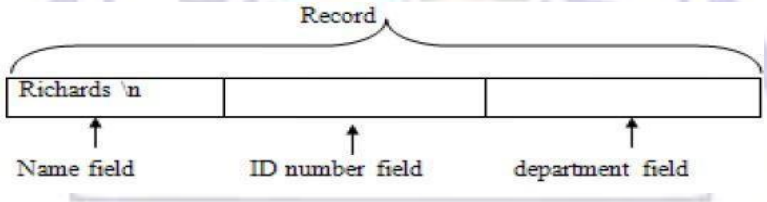
Output:

```
1000
2000
1500
```

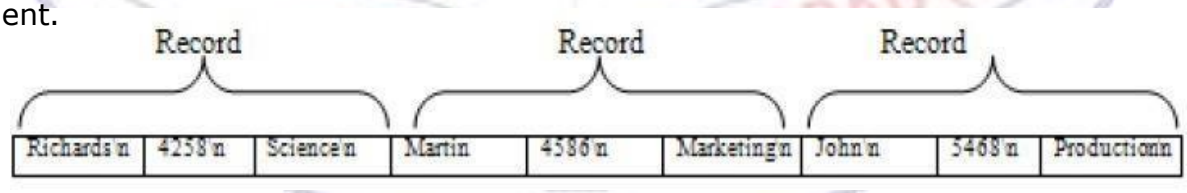
Processing Records

When data is written to a file, it is often organized into records and fields. A record is a

complete set of data that describes one item, and a field is a single piece of data within a record. For example, suppose we want to store data about employees in a file. The file will contain a record for each employee. Each record will be a collection of fields, such as name,ID number, and department. This is illustrated in the following figure.



Each time you write a record to a sequential access file, you write the fields that make up the record, one after the other. For example, The following figure shows a file that contains three employee records. Each record consists of the employee's name, ID number, and department.



The following program shows how employee records can be written to a file.

```
def main():
    num_emps = int(input('How many employee records do you want to create? '))
    emp_file = open('employees.txt', 'w')
    for count in range(1, num_emps + 1):
        print('Enter data for employee #', count, sep='')
```

```
        name = input('Name: ')
        id_num = input('ID number: ')
        dept = input('Department: ')
        emp_file.write(name + '\n')
        emp_file.write(id_num + '\n')
        emp_file.write(dept + '\n')
        print()
    emp_file.close()
    print('Employee records written to employees.txt.')

main()
```

program output:

```
How many employee records do you want to create? 3
Enter data for employee #1
Name: Richards
ID number: 4258
Department: science

Enter data for employee #2
Name: Martin
ID number: 4586
Department: Marketing

Enter data for employee #3
Name: John
ID number: 5468
Department: production
Employee records written to employees.txt.
```

When we read a record from a sequential access file, we read the data for each field, one after the other, until we have read the complete record. The following program demonstrates how we can read the employee records in the employee.txt file.

Example (readfile.py)

```
def main():
    emp_file = open('employees.txt', 'r')
    name = emp_file.readline()
    while name != '':
        id_num = emp_file.readline()
        dept = emp_file.readline()
        name = name.rstrip('\n')
```

Output:

```
Name: Richards
ID: 4258
Dept: science

Name: Martin
ID: 4586
Dept: Marketing
```

```
        id_num = id_num.rstrip('\n')
        dept = dept.rstrip('\n')
        print('Name:', name)
        print('ID:', id_num)
        print('Dept:', dept)
        print()
        name = emp_file.readline()
    emp_file.close()

main()
```

```
Name: John
ID: 5468
Dept: production
```

Exception:

An exception is an error that occurs while a program is running, causing the program to abruptly halt.

Exception handler:

Python, like most modern programming languages, allows us to write code that responds to exceptions when they are raised, and prevents the program from abruptly crashing. Such code is called an **exception handler**, and is written with the try/except statement. There are several ways to write a try/except statement, but the following general format shows the simplest variation:

```
try:
   statement
   statement
   etc.
except ExceptionName:
   statement
   etc.
```

First the key word try appears, followed by a colon. Next, a code block appears which we will refer to as the try suite. The try suite is one or more statements that can potentially raise an exception.

After the try suite, an except clause appears. The except clause begins with the key word except, optionally followed by the name of an exception, and ending with a colon. Beginning on the next line is a block of statements that we will refer to as a handler.

When the try/except statement executes, the statements in the try suite begin to execute. The following describes what happens next:

- If a statement in the try suite raises an exception that is specified by the ExceptionName in an except clause, then the handler that immediately follows the except clause executes. Then, the program resumes execution with the statement immediately following the try/except statement.
- If a statement in the try suite raises an exception that is not specified by the ExceptionName in an except clause, then the program will halt with a traceback error message.

- If the statements in the try suite execute without raising an exception, then any except clauses and handlers in the statement are skipped and the program resumes execution with the statement immediately following the try/except statement.

The following program handles ZeroDivisionError exception:

```
def main():
    try:
        a=int(input("Enter a number:"))
        b=int(input("Enter b number:")) zero division error  c=a/b
        print(c)
    except ZeroDivisionError:
        print("zero division error"
```

Output:

```
Enter a number: 5
Enter b number: 0
```

Short & Long Question

LONG QUESTIONS

1. What is Python? Explain its features.
2. How a program executes? Explain.
3. Explain program development life cycle?
4. What is variable? How do create, assign values into variables in python.(write including naming rules of creating variables)
5. Explain Datatypes & Literals in python.
6. What is an operator? Explain different types of operators?
7. What is operator precedence? Explain precedence of operators.
8. Explain Decision structures.
9. Explain Repetition structure.
10. What is input validation loop? Explain it.
11. What is Function? What are the benefits of function?
12. What is Function? How do you define, calling a function?
13. What is value-returning function? Explain it with an example.
14. Explain the library functions for working with random numbers.
15. Explain the Math module's functions in python.
16. How do you store functions in modules in python.
17. Explain File operations in python.
18. How do you process records using files in python?
19. What is an Exception? Explain about Exception handler.
20. How do you handle exceptions in python with an example?
21. How do you handle multiple exceptions in python?
22. What is List? How do you create, access elements from the list.
23. Explain the different list methods.
    - (or)
24. Explain list built-in methods.
25. What is Tuple? How do you access elements from tuples?
26. What is String? How do you access individual characters from string?
27. What is String? Explain string testing methods.

28. What is String? Explain modification methods.
29. What is String? Explain searching & replacing string.
30. What is Dictionary? How do you create, accessing (or) retrieving values, adding & deleting elements from dictionary.
31. Explain dictionary methods.
32. What is Set? How do you create, adding elements, removing elements from the set?

SHORT QUESTIONS

1. Write Python keywords.
2. What is compiler?
3. What is an interpreter?
4. How do you break long statements into multiple lines in python?
5. What are the available escape characters in Python?
6. What is Algorithm?
7. What is Pseudo code?
8. What is flowchart?
9. How do you mention the comments in Python?
10. How do you compare strings in python?
11. What is Boolean variable?
12. What is Nested loop?
13. Rules for defining function names.
14. What is Local variable? Explain it with an example.
15. Difference between Global variable & Global constants.
16. Difference between Local variable & Global variable.
17. Explain the reasons for most programmers avoid using global variables
18. Define Standard library function.
19. What is import statement in python?
20. Applications of Random numbers.
21. What are the steps involved when you working with the files
22. Explain types of Files.
23. Explain File access methods.
24. What is Default error message?
25. Define else clause in python.
26. Define the finally clause.
27. What is Repetition operator?
28. What is list slicing? Explain with an example.
29. Define Del statement.
30. Define Min & Max functions.
31. Is String immutable? Justify it.
32. What is String slicing?
33. How do you split String?
34. How do you test for a value in a dictionary using in & not in operator?

35.How do you find the number of elements in a dictionary?
36. Define union function on sets.
37.Define intersection function on Sets.
38. Define difference function on Sets.
39 .Define symmetry difference onsets.
40.Explain about subsets & supersets.
41.What is Recursion?
42 .What is Direct & indirect recursion


KEERTHANA DEGREE COLLEGE
BELA
ADILABAD (TS)