JAMAL MOHAMED COLLEGE

(AUTONOMOUS) TIRUCHIRAPPALLI.

DEPARTMENT OF COMPUTER APPLICATIONS

TRICHY-20

PYTHON PROGRAMMING – 20UCA5CC11

PREPARED BY

Dr. S. PEERBASHA & Mr. Y. MOHAMMED IQBAL

Mr. P.U. MANIMARAN III BCA-B

Department of Computer Applications

UNIT-1

TOPIC-1 INTRODUCTION

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- Web Development (Server-Side),
- Software Development,
- Mathematics,
- System Scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Good to know

- The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
- In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax Compared To Other Programming Languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Using Python:

- ✓ The Python interpreter can run Python programs that are saved in files or interactively execute Python statements that are typed at the keyboard.
- ✓ Python comes with a program named IDLE that simplifies the process of writing, executing, and testing programs.

Interpreted Mode:

- ✓ When we install the Python language on your computer, one of the items that is installed is the Python interpreter.
- ✓ The Python interpreter is a program that can read Python programming statements and execute them.

We can use the interpreter in two modes: interactive mode and script mode.

1. In interactive mode, the interpreter waits for you to type Python statements on the keyboard. Once you type a statement, the interpreter executes it and then waits for you to type another statement.
2. In script mode, the interpreter reads the contents of a file that contains Python statements. Such a file is known as a Python program or a Python script. The interpreter executes each statement in the Python program as it reads it.

Interactive Mode:

Once we install python in our system you start the interpreter in interactive mode by going to the operating system's command line and typing the following command "python".

There we can type our program like

print('Welcome to our department')

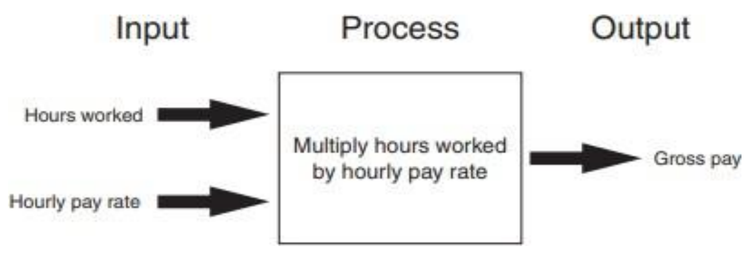
TOPIC-2 INPUT, PROCESSING AND OUTPUT:

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.



```
hoursWorker = 4

hourlyPayRate = 250

grossPay = hoursWorked*hourlyPayRate

print(grossPay)
```

TOPIC-2 Displaying Output with the print Function

- ✓ A Function is a piece of prewritten code that performs an operation.
- ✓ And print is the fundamental built-in function this statement is used to display the output.

Example of print function:-

```
>>>print(“Hello Students”)
```

Output:-

```
Hello Students
```

1. Apostrophe

- ✓ To add Apostrophe in our print statement we have enclose the statement in double quotes. For Example

```
>>>print(“I’m Dr. S. Peerbasha”)
```

Output:-

```
I’m Dr. S. Peerbasha
```

2. Double Quotes

- ✓ To add double quotes in our print statement we have enclose the statement in Apostrophe.
- ✓ For Example

```
>>>print(‘I love “Cars” so much’)
```

Output:-

```
I love “Cars” so much
```

3. Both Apostrophe and Double Quotes

- ✓ If you want to add both the Apostrophe and the double quotes in the print statement we have to enclose the statement in triple quotes either “ “ “ or ‘ ‘ ‘ . For example

```
>>>print(“”” I’m a “Web Developer” “””)
```

Output:-

```
I’m a “Web Developer”
```

Or

```
>>>print(‘’’ I’m a “Game Developer” ‘’’)
```

Output:-

```
I’m a “Game Developer”
```

TOPIC-3 Comments

- ✓ Comments are short notes placed in different parts of a program, explaininghow those parts of the program work.
- ✓ In python a comment statement begins with # symbol.
- ✓ When python interpreter sees the # symbol the complete line will be ignored.

For Example, in the below program comment in the first and the last line of the program

Comment1.py

```
#We can add comments anywhere in the program
print(“Cat”)
print(“Dog”)
#These comment lines will be ignored
```

Output

```
Cat
Dog
```

We can add comment line anywhere in the program

Comment2.py

```
Print(“Continental Gt 650”)

#Comment line in the middle of the program

print(“Interceptor 650”)    #Comment line in the same line of the program code
```

Output

```
Continental Gt 650

Interceptor 650
```

TOPIC-4 Variables

- ✓ Programs use variables to store data in memory.
- ✓ A variable is a name that represents a value in the computer’s memory.
- ✓ For example, a program that calculates the sum of two numbers

```
a = 10

b = 20

c = a + b

print(“Sum of a and b is ”, c )
```

Output

```
Sum of a and b is 30
```

- ✓ In the above example we are storing a value to both **a** and **b**.
- ✓ When a variable represents a value in the computer’s memory, we say that the variable references the value.
- ✓ And in the above example we are assigning a value to a variable with the help of assignment operator.
- ✓ An assignment statement is written in the following general format:

variable = expression

- ✓ In an assignment statement, the variable that is receiving the assignment must appear on the left side of the = operator.
- ✓ As shown in the above example, an error occurs if the item on the left side of the = operator is not a variable:

```
>>>30 = value
SyntaxError: can't assign to literal
```

Variable Reassignment

Value of a variable can be reassigned for Example:

```
a = 10
print("Now value of a is ", a)
a = 20
print("Now value of a is ", a)
```

Output:

```
Now value of a is 10
Now value of a is 20
```

Variable Naming Rule

Although you are allowed to make up your own names for variables, you must follow these rules:

- You 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-z, A-Z, or an underscore character (_).
- After the first character you may use the letters a-z or A-Z, the digits 0 through 9, or underscores.

- Uppercase and lowercase characters are distinct. This means the variable name BcaDept is not the same as Bcadept.

In addition to following these rules, we should always choose names for our variables that give an indication of what they are used for. For example, if we want to student age means we can declare the variable as:

studentAge (or) student_age

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

TOPIC-5 Reading Input from the Keyboard

- ✓ Most of the programs will need to read input from the user and then perform an operation on that input.
- ✓ The input function reads a piece of data that has been entered at the keyboard and returns that piece of data, as a string, back to the program.
- ✓ When a program reads data from the keyboard, usually it stores that data in a variable so it can be used later by the program.

Syntax:-

variable = input(prompt)

Variable is used to store data.

Input is used to get data from the user through keyboard.

prompt is a string that is displayed on the screen. The string’s purpose is to instruct the user to enter a value.

Example

```
>>>Name = input(“Enter your name: “)
Enter your name: Maran
```

If we give a value and enter the value will be stored in the variable. Here name Maran will be stored in the variable called Name.

Or if we want to get a specify datatype value from user we can get it like giving

```
>>>age = int(input(“Enter your age: “))
Enter your age: 20
```

```
value = float(input(“Enter a float value: “))
Print(“Float value is : ”value)
```

Output:

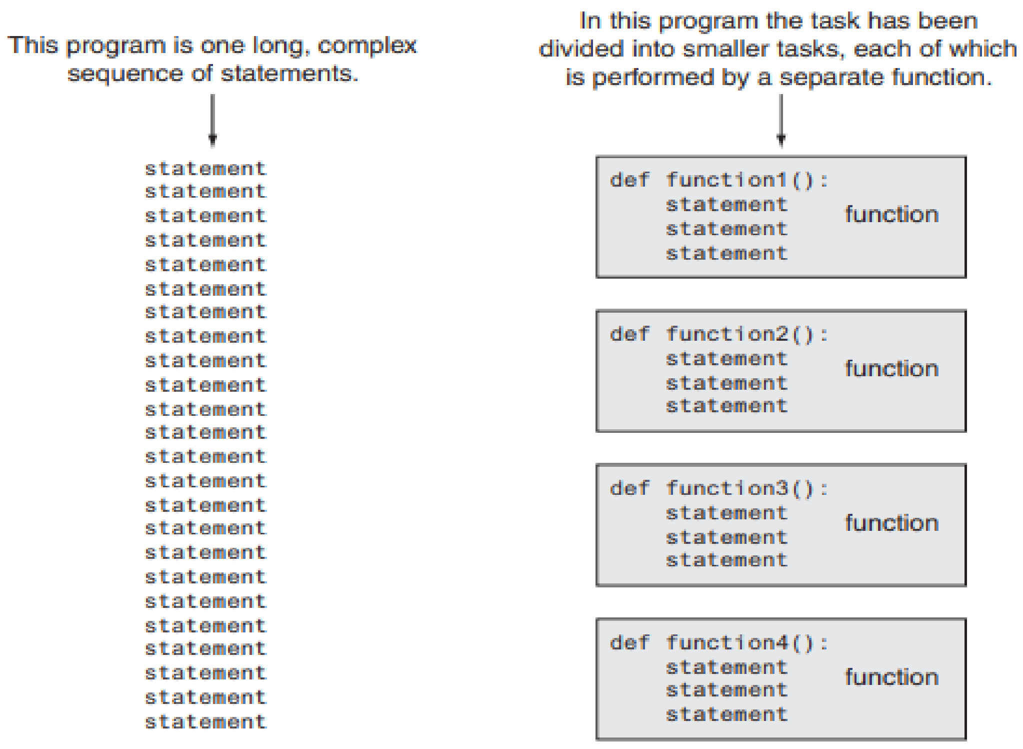
```
Enter a float value: 21.1
Float value is : 21.1
```

If we give a character or a string it will throw an error

TOPIC-6 Functions

- ✓ 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.


This program is one long, complex sequence of statements.
In this program the task has been divided into smaller tasks, each of which is performed by a separate function.
statement
def function1():
def function2():
def function3():
def function4():
function

Benefits of Modularizing a Program with Functions

A program benefits in the following ways when it is broken down into 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, then be executed any time it is needed.

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.

TOPIC-7 Defining and Calling a Void Function

Python requires that you follow the same rules that you follow when naming variables which was on the page 4.

Defining a Function

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

```
def functionName():
    Statement
    Statement
    Statements…
```

- The header of the function should start with the def keyword, 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.
- Notice in the general format that all of the statements in the block are indented.
- This indentation is required, because the Python interpreter uses it to tell where the block begins and ends.

For example

```
def studentDetails():
print(“Name: Dr. S. Peerbasha”)
print(“Age: 32”)
print("Department: Computer Applications")
```

This code defines a function named studentDetails.

The message function contains a block with three 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, you must call it. This is how we would call the studentDetails function:

```
studentDetails()
```

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

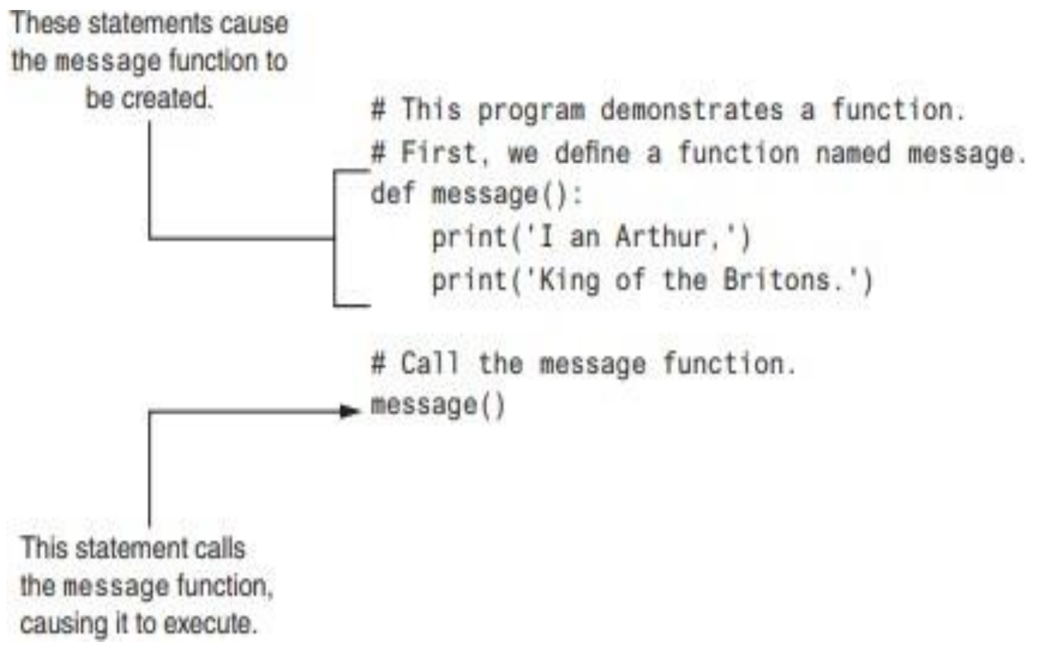
Example

Function1.py

```
#defining a function
def studentDetails():
print("Name: Zenitsu")
print("Age: 18")
print("Department: BCA")
#calling a function
studentDetails()
```

Output:

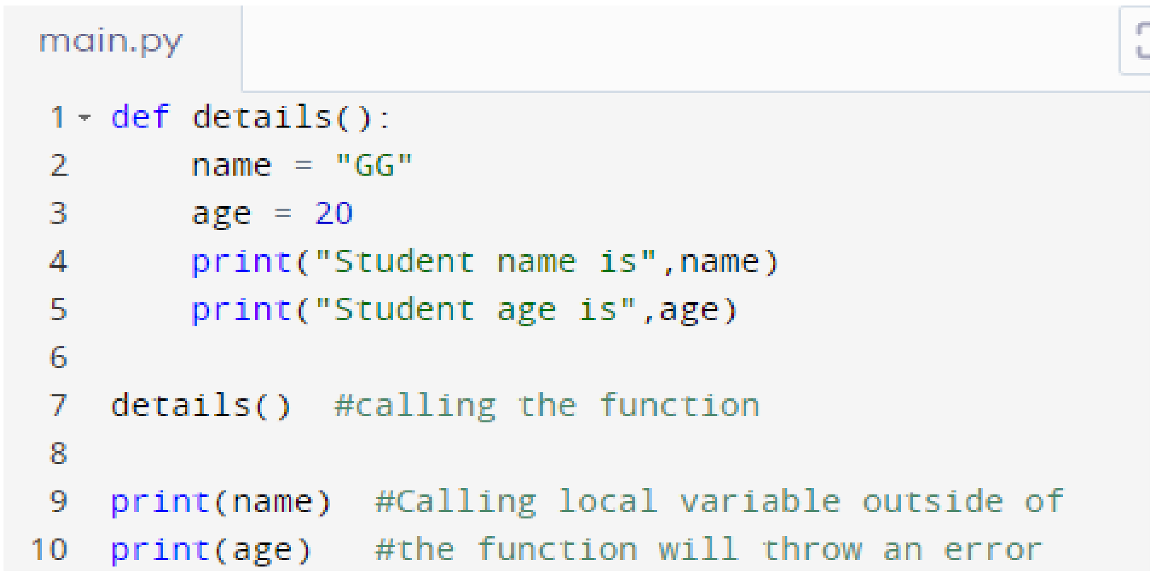
```
Name: Zenitsu
Age: 18
Department: BCA
```


These statements cause the message function to be created.
This statement calls the message function, causing it to execute.

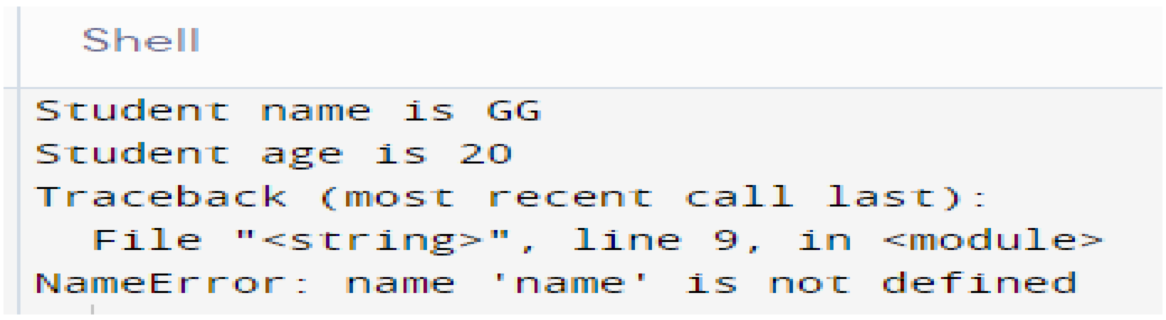
```
# This program demonstrates a function.
# First, we define a function named message.
def message():
    print('I an Arthur,')
    print('King of the Britons.')

# Call the message function.
message()
```

TOPIC-8 Local Variables

- ✓ A local variable is created inside a function and 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.
- ✓ Anytime you assign a value to a variable inside a function, you create a local variable.
- ✓ A local variable belongs to the function in which it is created, and only statements inside that function can access the variable.


main.py

```
def details():
    name = "GG"
    age = 20
    print("Student name is",name)
    print("Student age is",age)

details()  #calling the function

print(name)  #Calling local variable outside of
print(age)   #the function will throw an error
```



```
Student name is GG
Student age is 20
Traceback (most recent call last):
  File "<string>", line 9, in <module>
NameError: name 'name' is not defined
```

- ✓ In the above program we have declared a function called details in that function.
- ✓ We have declared two local variable called name & age and printing them in a print statement.
- ✓ After that we are calling the function which will print the statements inside the function.
- ✓ Then we are calling the local variable outside the function which will raise an NameError that name is not defined.

TOPIC-9: Passing Arguments to Functions

- ✓ Sometimes it is useful not only to call a function, but also to send one or more pieces of data into the function.
- ✓ Pieces of data that are sent into a function are known as arguments.
- ✓ The function can use its arguments in calculations or other operations.
- ✓ If you want a function to receive arguments when it is called, you must equip the function with one or more parameter variables.
- ✓ A parameter variable, often simply called a parameter, is a special variable that is assigned the value of an argument when a function is called.
- ✓ Here is an example of a function that has a parameter variable:

Example

```
def add(a, b):
   c = a + b
print("Sum of given number is: ", c)
```

add(10, 20)

Output:

Sum of given number is: 30

In the above program we declared a function called adds and passed two arguments and performed some addition operation and displayed the output.

While calling the function we have passed two values as parameter to the function.

Like this we can pass any type of value to the function.

Example 2

def studentDetails(firstName, lastName):

print(“Student First name is: “, firstName)

print(“Student Last name is: “, lastName)

studentDetails(“Micheal”, “Rayappan”) #calling function with string values

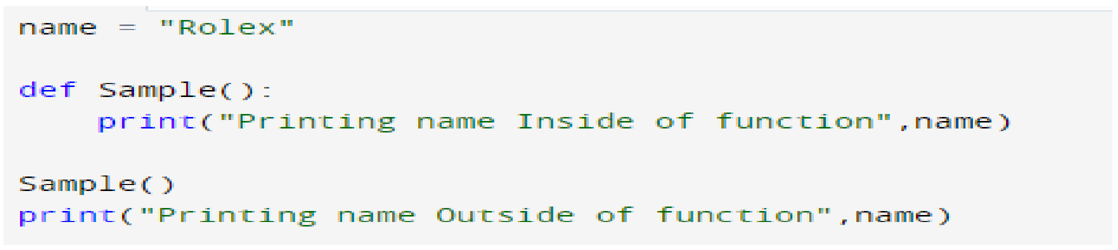
Output

Student First name is: Micheal

Student Last name is: Rayappan

TOPIC-10 Global Variables and Global Constants

- ✓ Local variable can be accessed only inside the function that created it.
- ✓ When a variable is declared outside the function it can accessed anywhere in the program this is called global variable.



```
name = "Rolex"

def Sample():
    print("Printing name Inside of function",name)

Sample()
print("Printing name Outside of function",name)
```

Output

```
Printing name Inside of function Rolex

Printing name Outside of function Rolex
```

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.
- 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, you have to be aware of all the other parts of the program that access the global variable.

In most cases, you should create variables locally and pass them as arguments to the functions that need to access them.

Global Constant

Although you should try to avoid the use of global variables, it is permissible to use global constants in a program. A global constant is a global name that references a value that cannot be changed. Because a global constant's value cannot be change execution, you do not have to worry about many of the potential hazards that are associated with the use of global variables.

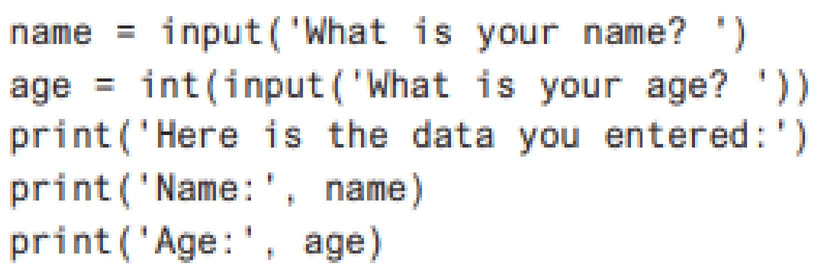
Although the Python language does not allow you to create true global constants, you can simulate them with global variables. If you do not declare a global variable with the global keyword inside a function, then you cannot change the variable's assignment inside that function. The following In the Spotlight section demonstrates how global variables can be used in Python to simulate global constants.

UNIT-II

Decision Structures and Boolean Logic

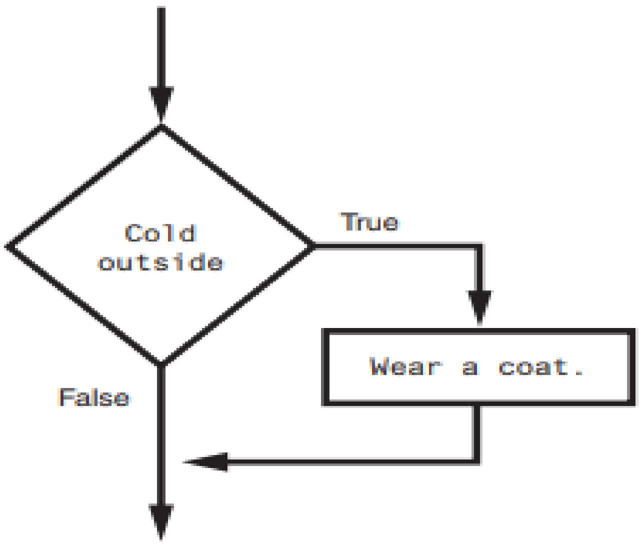
TOPIC-1 The if Statement

- ✓ The if statement is used to create a decision structure, which allows a program to have more than one path of execution.
- ✓ The if statement causes one or more statements to execute only when a Boolean expression is true.
- ✓ A control structure is a logical design that controls the order in which a set of statements execute.
- ✓ A sequence structure is a set of statements that execute in the order in which they appear.
- ✓ For example, the following code is a sequence structure because the statements execute from top to bottom



```
name = input('What is your name? ')
age = int(input('What is your age? '))
print('Here is the data you entered:')
print('Name:', name)
print('Age:', age)
```

- ✓ In a decision structure's simplest form, a specific action is performed only if a certain condition exists.
- ✓ If the condition does not exist, the action is not performed.
- ✓ The below flowchart shows how the logic of an everyday decision can be diagrammed as a decision structure.
- ✓ The diamond symbol represents a true/false condition.
- ✓ If the condition is true, we follow one path, which leads to an action being performed. If the condition is false, we follow another path, which skips the action.



- ✓ In the flowchart, the diamond symbol indicates some condition that must be tested.
- ✓ In this case, we are determining whether the condition Cold outside is true or false.
- ✓ If this condition is true, the action Wear a coat is performed.
- ✓ If the condition is false, the action is skipped.
- ✓ The action is conditionally executed because it is performed only when a certain condition is true.
- ✓ In Python, we use the if statement to write a single alternative decision structure.

Syntax:-

```
if condition:
   statement-1
   .........
   statement-n
```

Boolean Expressions and Relational Operators

- ✓ The if statement tests an expression to determine whether it is true or false.
- ✓ The expressions that are tested by the if statements are called Boolean expressions.
- ✓ Typically, the Boolean expression that is tested by an if statement is formed with a relational operator.

- ✓ A relational operator determines whether a specific relationship exists between two values.
- ✓ For example, the greater than operator (>) determines whether one value is greater than another.
- ✓ The equal to operator (==) determines whether two values are equal.

>Greater than

<Lesser than

>= Greater than or equal to

<= Lesser than or equal to

!=Not equal to

Example:

Age = 18

If Age >= 18: #condition will become true and the statements inside if block execute

print(“The example that almost used million time”)

print(“You are eligible to vote”)

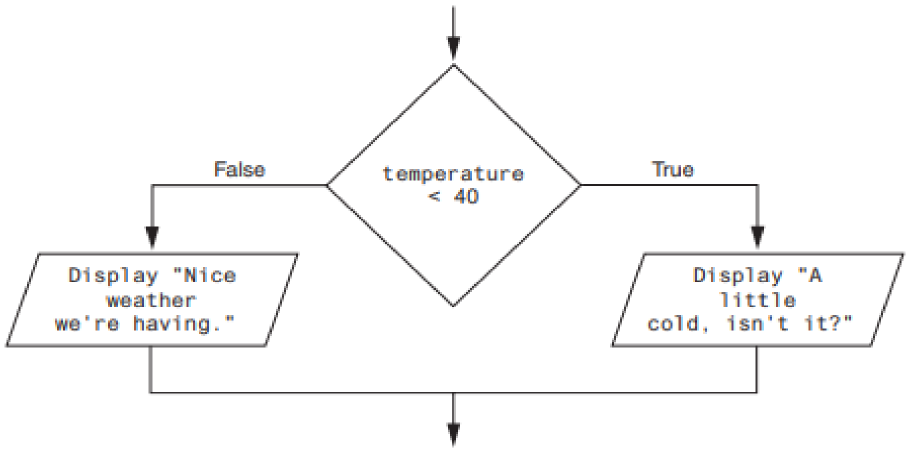
Output:

The example that almost used million time

You are eligible to vote

TOPIC-2 The if-else Statement

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


False
temperature < 40
True
Display "Nice weather we're having."
Display "A little cold, isn't it?"

- ✓ The decision structure in the flowchart tests the condition temperature < 40.
- ✓ If this condition is true, the message "A little cold, isn't it?" is displayed.
- ✓ If the condition is false, the statement message "Nice weather we're having." is displayed.

Syntax

```
If condition:
     Statement
     Statement
     etc
else:
     statement
     statements
     etc
```

Example1

if temperature < 40:

```
print("A little cold, isn't it?")
else:
print("Nice weather we're having.")
```

Example2

```
time = 8.50
if time <= 8.40:
print(“You are allowed”)
else:
print(“What time is it?”)
print(“Attendance will not be given”)
```

Output:

```
What time is it?
Attendance will not be given
```

In the above program the time we have declared is 8.50 but the condition will be true only if the time is lesser than or equal to 8.40. So, the output will be else block.

TOPI-3 Comparing String

- ✓ Python allows you to compare strings.
- ✓ This allows you to create decision structures that test the value of a string.
- ✓ We saw in the preceding examples how numbers can be compared in a decision structure.
- ✓ You can also compare strings.
  For example, look at the following code.

Example 1

```
Name1 = “Jones”

Name2 = “Sones”

If Name1 == Name2:

print(“The names are same.”)

else:

print(“The names are Not same.”)
```

Output:

```
The names are Not same.
```

The == operator compares Name1 and Name2 to determine whether they are equal. Because the strings 'Jones' and 'Sones' are not equal, the else clause will display the message 'The names are NOT the same.'

Example 2

```
password = input("Enter your password: ")

if password == "Shivaji Cool":

print("Hello Sir Welcome")

else:

print("You got few more chance")
```

Output:

```
Enter your password: I am Salman          #1st try

You got few more chance

Enter your password: Shivaji Cool          #2nd try

Hello Sir Welcome
```

We 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.

It will perform the operation with the ASCII value of the character. For Example

```
If 'a' < 'b':
print("a is less than b")
```

Output:

```
a is less than b
```

Because the ASCII value of a is 97 and b is 98. 97 is less than 98 so the condition is true.

TOPIC-4 Nested Decision Structures and the if-elif-else Statement

- ✓ To test more than one condition, a decision structure can be nested inside another decision structure.

Nested If else:

- ✓ If you create if-else statements inside other if-else statements, we will call them nested if-else statements in python.
- ✓ The nested if-else statements are useful when you want to execute the block of code/statements inside other if-else statements.

Syntax:-

```
if condition:
    if condition:
        statement(s)
    else:
        statement(s)
else:
    statement(s)
```

Example

```
x = 30
y = 10
if x >= y:
print("x is greater than or equals to y")
    if x == y:
print("x is equals to y")
    else:
print("x is greater than y")
else:
print("x is less than y")
```

Output

```
x is greater than or equals to y
x is greater than y
```

TOPIC-5 The if-elif-else Statement

- ✓ The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.
- ✓ Similar to the else, the elif statement is optional.
- ✓ However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

Syntax:

```
if expression1:

     statement

elif expression2:

     statement                                    #we can add n number of elif
block

elif expression3:

      statement

else:

     statement
```

Example

```
var = 100
if var == 200:
print("1 - Got a true expression value”)
elif var == 150:
print("2 - Got a true expression value")
elif var == 100:
print("3 - Got a true expression value")
else:
print("4 - Got a false expression value")
```

Output

```
3 – Got a true expression value
```

TOPIC-6 Logical Operators

- ✓ The logical and operator and the logical or operator allow you to connect multiple Boolean expressions to create a compound expression.
- ✓ The logical not operator reverses the truth of a Boolean expression.

and:-

- ✓ The and operator connects two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true.

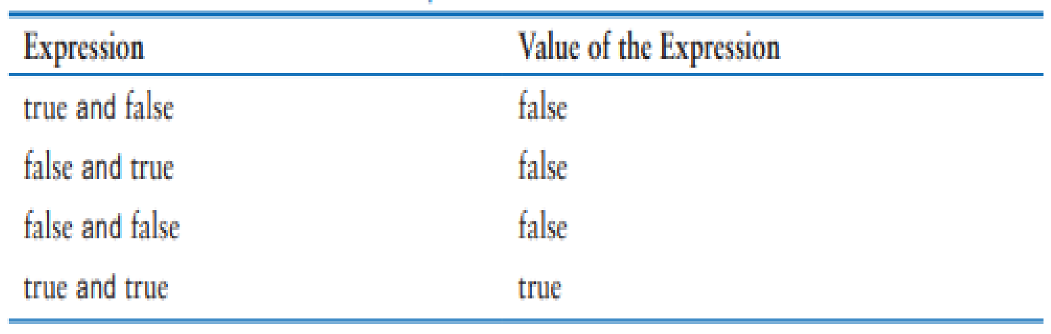
Example

```
Age = 20

If Age >= 18 and Age <=60:              #Age is greater than 18 = true

print("Age is valid")                   #Age is less than 60 = true

else:                                   #true and true is true

print("Age is not valid")
```

Output

```
Age is valid
```

In the above program we have checked whether the Age is greater than or equal to 18 and less than or equal to 60. It satisfies both the condition so the condition will become true.



Expression	Value of the Expression
true and false	false
false and true	false
false and false	false
true and true	true

or:-

- ✓ The or operator takes two Boolean expressions as operands and creates a compound Boolean expression that is true when either of the subexpressions is true.
- ✓ The following is an example of an if statement that uses the or operator

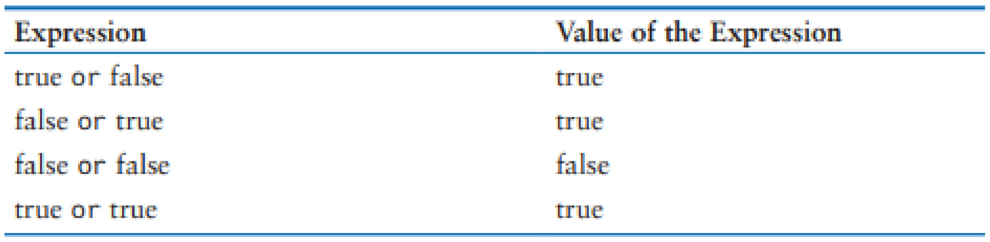
Example

```
Mark = 105
```

```
If Mark<1 or Mark>100:

print(“Mark is not valid”)

else:

print(“Mark is valid”)
```

Output

```
Mark is not valid
```



Expression	Value of the Expression
true or false	true
false or true	true
false or false	false
true or true	true

not:-

- ✓ The not operator is a unary operator that takes a Boolean expression as its operand and reverses its logical value.
- ✓ In other words, if the expression is true, the not operator returns false, and if the expression is false, the not operator returns true.
- ✓ The following is an if statement using the not operator

Example

```
if not(temperature > 100):

print('This is below the maximum temperature.')
```

First, the expression (temperature > 100) is tested and a value of either true or false is the result. Then the not operator is applied to that value. If the expression (temperature > 100) is true, the not operator returns false. If the expression (temperature > 100) is false, the not operator returns true.

True -> False

False -> True

TOPIC-7 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.
- ✓ We have worked with int, float, and str (string) variables.
- ✓ In addition to these data types, Python also provides a bool data type.
- ✓ The bool data type allows you to create variables that may reference one of two possible values: True or False.
- ✓ Here are examples of how we assign values to a bool variable

isHungry = false

isSleepy = true

- ✓ 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.

TOPIC-8 Introduction to Repetition Structures

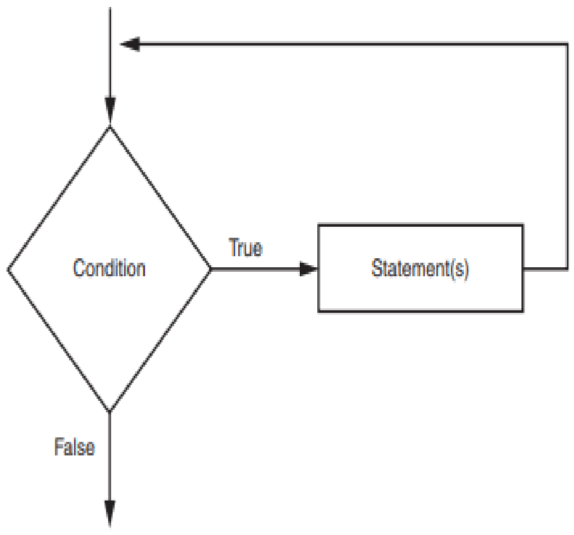
- ✓ A repetition structure causes a statement or set of statements to execute repeatedly.
- ✓ Programmers commonly have to write code that performs the same task over and over so we have to write the code again and again this will make our program a long sequence of statement.
- ✓ In this approach there are several disadvantages, including the following:

- The duplicated code makes the program large.
- Writing a long sequence of statements can be time consuming.
- If part of the duplicated code has to be corrected or changed, then the correction or change has to be done many times.

- Instead of writing the same sequence of statements over and over, a better way to repeatedly perform an operation is to write the code for the operation once, then place that code in a structure that makes the computer repeat it as many times as necessary.
- This can be done with a repetition structure, which is more commonly known as a loop

TOPIC-9 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.



Syntax:-

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

- ✓ For simplicity, we will refer to the first line as the while clause.
- ✓ The while clause begins with the word while, followed by a Boolean condition that will be evaluated as either true or false.
- ✓ A colon appears after the condition.
- ✓ Beginning at the next line is a block of statements.
- ✓ When the while loop executes, the condition is tested.
- ✓ If the condition is true, the statements that appear in the block following the while clause are executed, and the loop starts over.
- ✓ If the condition is false, the program exits the loop.

Example

```
count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1
```

Output

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
```

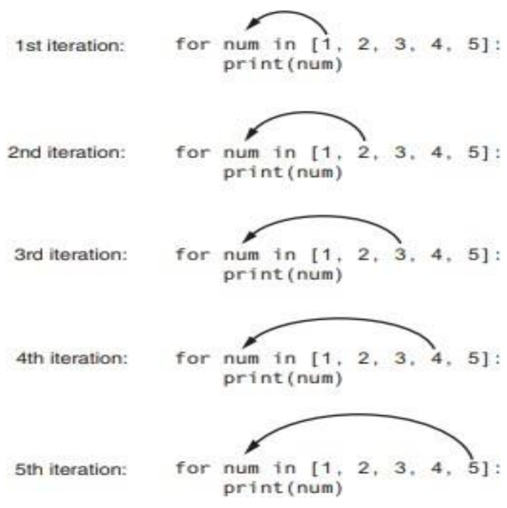
TOPIC-10 The for Loop: A Count-Controlled Loop

- A count-controlled loop iterates a specific number of times.
- In Python, you use the for statement to write a count-controlled loop.
- A for loop is used for iterating over a sequence (that is a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Syntax:-

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



```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

Example:-

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

Output

```
apple
banana
cherry
```

Break

With the break statement we can stop the loop before it has looped through all the items

Example 2

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

Output

```
apple
banana
```

Continue

With the continue statement we can stop the current iteration of the loop, and continue with the next.

Example 2

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

Output

```
apple
cherry
```

Range

To loop through a set of code a specified number of times, we can use the range() function, The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example 1

```
for x in range(6): #start from zero to end but not including 6
  print(x)
```

Output

```
1
2
3
4
5
```

Example 2

```
for x in range(2, 6):  #start from the specified number (2) to end but not including 6
  print(x)
```

Output

```
2
3
4
5
```

Example 3

```
for x in range(3, 31, 3):
  print(x)
```

Output

```
3
6
9
12
15
```

```
18
21
24
27
30
```

start from the specified number (3) to end but not including 31 and the third value indicate the increment of number here we have given 3 which means the value would go from 3, 6, to 30.

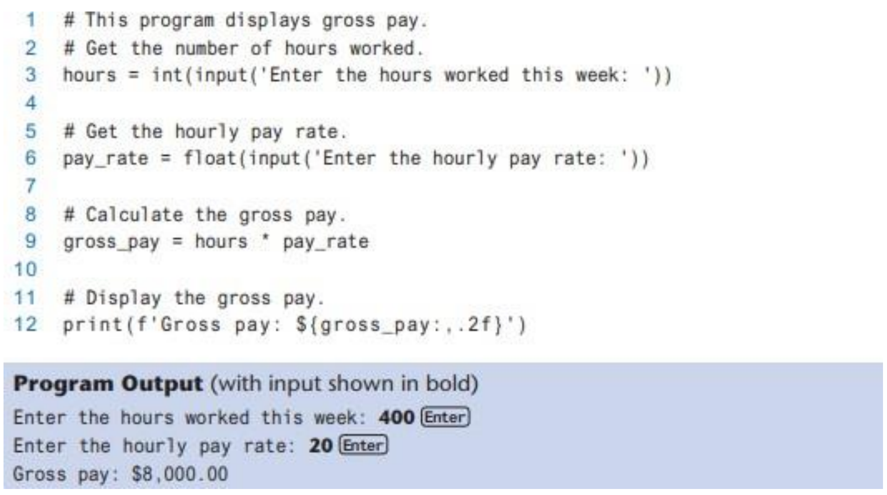
TOPIC-11 Sentinels

- ✓ A sentinel is a special value that marks the end of a sequence of values.
- ✓ Consider the following scenario: You are designing a program that will use a loop to process a long sequence of values. At the time you are designing the program, you do not know the number of values that will be in the sequence. In fact, the number of values in the sequence could be different each time the program is executed. What is the best way to design such a loop?
- ✓ Here are some techniques with the disadvantages of using them when processing a long list of values:
  - Simply ask the user, at the end of each loop iteration, if there is another value to process. If the sequence of values is long, however, asking this question at the end of each loop iteration might make the program cumbersome for the user.
  - Ask the user at the beginning of the program how many items are in the sequence. This might also inconvenience the user, however. If the sequence is very long, and the user does not know the number of items it contains, it will require the user to count them.
  - When processing a long sequence of values with a loop, perhaps a better technique is to use a sentinel. A sentinel is a special value that marks the end of a sequence of items. When a program reads the sentinel value, it knows it has reached the end of the sequence, so the loop terminates.

TOPIC-12 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. For example, look at the payroll program and notice what happens in the sample run when the user gives bad data as input.

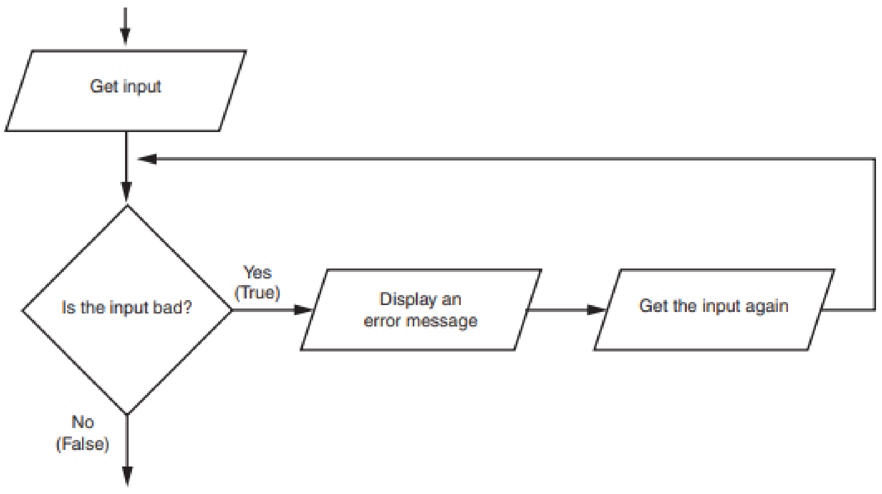


```python
# This program displays gross pay.
# Get the number of hours worked.
hours = int(input('Enter the hours worked this week: '))

# Get the hourly pay rate.
pay_rate = float(input('Enter the hourly pay rate: '))

# Calculate the gross pay.
gross_pay = hours * pay_rate

# Display the gross pay.
print(f'Gross pay: ${gross_pay:,.2f}')
```

Program Output (with input shown in bold)

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

Did you spot the bad data that was provided as input? The person receiving the paycheck will be pleasantly surprised, because in the sample run the payroll clerk entered 400 as the number of hours worked. The clerk probably meant to enter 40, because there are not 400 hours in a week. The computer, however, is unaware of this fact, and the program processed the bad data just as if it were good data. Can you think of other types of input that can be given to this program that will result in bad output? One example is a negative number entered for the hours worked; another is an invalid hourly pay rate.



Let’s consider an example. Suppose you are designing a program that reads a test score and you want to make sure the user does not enter a value less than 0. The following code shows how you can use an input validation loop to reject any input value that is less than 0.

Example

```
score = int(input('Enter a test score: ')
while score < 0:
print('ERROR: The score cannot be negative.')
score = int(input('Enter the correct score: '))
```

This code first prompts the user to enter a test score (this is the priming read), then the while loop executes. Recall that the while loop is a pre-test loop, which means it tests the expression score < 0 before performing an iteration. If the user entered a valid test score, this expression will be false, and the loop will not iterate. If the test score is invalid, however, the expression will be true, and the loop’s block of statements will execute. The loop displays an error message and prompts the user to enter the correct test score. The loop will continue to iterate until the user enters a valid test score.

TOPIC-13 Nested Loops

- ✓ A nested loop is a loop inside the body of the outer loop.
- ✓ The inner or outer loop can be any type, such as a while loop or for loop.
- ✓ For example, the outer for loop can contain a while loop and vice versa.

- The outer loop can contain more than one inner loop. There is no limitation on the chaining of loops.
- In the nested loop, the number of iterations will be equal to the number of iterations in the outer loop multiplied by the iterations in the inner loop.

- In each iteration of the outer loop inner loop execute all its iteration. For each iteration of an outer loop the inner loop re-start and completes its execution before the outer loop can continue to its next iteration
- Nested loops are typically used for working with multidimensional data structures, such as printing two-dimensional arrays, iterating a list that contains a nested list.

Syntax:-

```
# outer for loop
for element in
sequence:
  # inner for loop
for element in
sequence:
body of inner for loop
```

Example

```
for i in range(1, 11):
   for j in range(1, 11):
print(i * j, end=' ')
print()
```

In the above example, we are using a for loop inside a for loop. In this example, we are printing a multiplication table of the first ten numbers.

- The outer for loop uses the range() function to iterate over the first ten numbers
- The inner for loop will execute ten times for each outer number
- In the body of the inner loop, we will print the multiplication of the outer number and current number

- The inner loop is nothing but a body of an outer loop.

Output

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

- In this program, the outer for loop is iterate numbers from 1 to 10. The range() return 10 numbers. So total number of iteration of the outer loop is 10.
- In the first iteration of the nested loop, the number is 1. In the next, it 2. and so on till 10.
- Next, For each iteration of the outer loop, the inner loop will execute ten times. The inner loop will also execute ten times because we are printing multiplication table up to ten.
- In each iteration of an inner loop, we calculated the multiplication of two numbers.

Nested Loop to Print Pattern

Another most common use of nested loop is to print various star and number patterns.

Example

```
rows = 5
for i in range(1, rows + 1):
   for j in range(1, i + 1):
print("*", end=" ")
print('')
```

Output

```
*
* *
* * *
* * * *
* * * * *
```

- In this program, the outer loop is the number of rows print.
- The number of rows is five, so the outer loop will execute five times
- Next, the inner loop is the total number of columns in each row.
- For each iteration of the outer loop, the columns count gets incremented by 1
- In the first iteration of the outer loop, the column count is 1, in the next it 2. and so on.
- The inner loop iteration is equal to the count of columns.
- In each iteration of an inner loop, we print star

Nested Loop to print rectangular pattern

```
rows = int(input('How many rows? '))
cols = int(input('How many columns? '))
for r in range(rows):
    for c in range(cols):
print('*',  end='')
print()
```

Output

```
How many rows? 5
How many columns? 10
**********
**********
**********
**********
**********
```

JAMAL MOHAMED COLLEGE (AUTONOMOUS)

TIRUCHIRAPPALLI-20

DEPARTMENT OF COMPUTER APPLICATIONS

PYTHON PROGRAMMING – 20UCA5CC11

PREPARED BY

Dr. S. Peerbasha, Mr. Y. Mohammed Iqbal

&

Mr. P.U. Manimaran

Department Of Computer Applications

Jamal Mohamed College (Autonomous)

Trichy-20

UNIT-III

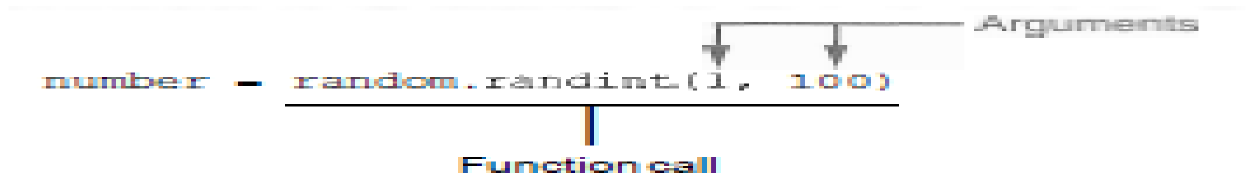
Value Returning Functions and Modules: Introduction to Value Returning Functions: Generating Random Numbers – Writing your own value returning functions – The math module – Storing Functions in Modules – Files and exceptions – Introduction to File Input and Output – Using Loops to Process Files – Processing Records - Exceptions

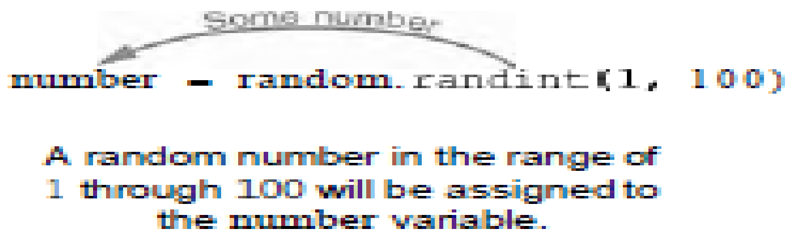
TOPIC-1:- INTRODUCTION TO VALUE-RETURNING FUNCTIONS AND MODULES:

- A value-returning function is a function that returns a value back to the part of the program that called it.
- Python, as well as most other programming languages, provides a library of prewritten functions that perform commonly needed tasks.
- These libraries typically contain a function that generates random numbers.
- *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.

TOPIC-2:- 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 statistical programs that must randomly select data for analysis.
- Random numbers are commonly used in computer security to encrypt sensitive data.
- Python provides several library functions for working with random numbers. Example
- import random
- This statement causes the interpreter to load the contents of the random module into memory.
- This makes all of the functions in the random module available to your program.
- The first random-number generating function that we will discuss is named randint.
- Because the randint function is in the random module.
- The following statement shows an example of how you might call the randint function.
  number = random.randint(1, 100)




Some number
number = random.randint(1, 100)
A random number in the range of 1 through 100 will be assigned to the number variable.

Sample Program

```
# This program displays a random number in the range of 1 through 10.
import random
def main ( ) :
# Get a random number.
number= random.randint(1, 10)
# Display the number.
print 'The number is ' , number
# Call the main function.
main( )
```

Program Output

```
The number is 7
```

number= random.randrange(5, 10)

- When this statement executes, a random number in the range of 5 through 9 will be assigned to number.
- The following statement specifies a starting value, an ending limit, and a step value:
  number= random.randrange(0, 101, 10)
- In this statement the randrange function returns a randomly selected value from the following sequence of numbers:
  [0, 10, 20, 30, 40, 50, 60, 70, 80, 9O, 100]
- Both the randint and the randrange functions return an integer number.
- The random function returns, however, 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 (but not including 1.0).

Here is an example:

number= random.random()

The uniform function also returns a random floating-point number, but allows you to specify the range of values to select from. Here is an example:

number= random.uniform(l.0, 10.0)

In this statement the uniform function returns a random floating-point number in the range of 1.0 through 10.0 and assigns it to the number variable.

TOPIC-3: WRITING 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.
- You write a value-returning function in the same way that you write a simple function, with one exception: a value-returning function must have a return statement.

Here is the general format of a value-returning function definition in Python:

```
def function-name ( ) :
statement
statement
etc.
return expression
```

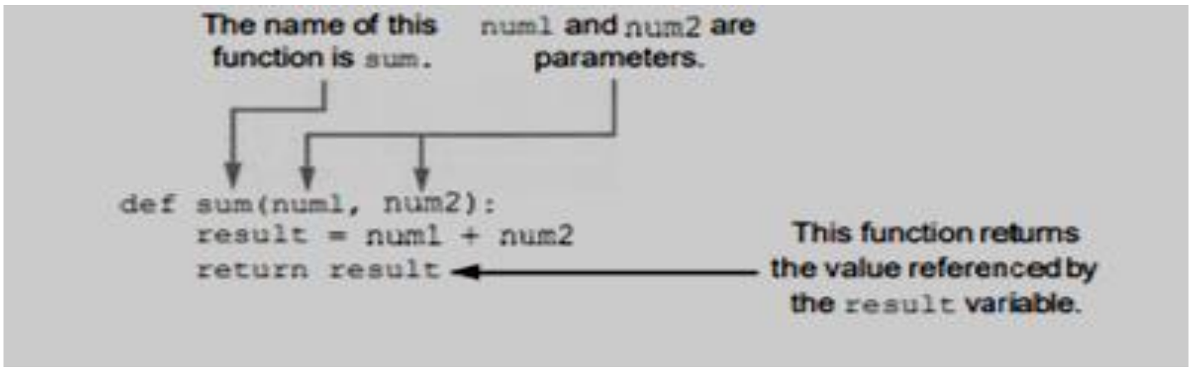
Here is a simple example of a value-returning function:

Example

```
def sum(num1, num2 ) :
result = numl + num2
return result
value = sum(10,20)
print(value)
```

Output

```
30
```


The name of this function is sum.
num1 and num2 are parameters.
This function returns the value referenced by the result variable.

```
def sum(num1, num2):
    result = num1 + num2
    return result
```

Returning Multiple Values

- The examples of value-returning functions that we have looked at so far return a single value.
- In Python, however, you are not limited to returning only one value.
- You can specify multiple expressions separated by commas after the return statement, as shown in this general format:

return expressionl, expression2, etc.

- As an example, look at the following definition for a function named get - name.
- The function prompts the user to enter his or her first and last names.
- These names are stored in two local variables: first and last.
- The return statement returns both of the variables.

```
def get-name():
   # Get the user's first and last names.
first = raw-input( 'Enter your first name: ' )
last = raw-input( 'Enter your last name: ' )
   # Return both names.
return first, last
```

- When you call this function in an assignment statement, you need to use two variables on the left side of the = operator. Here is an example:

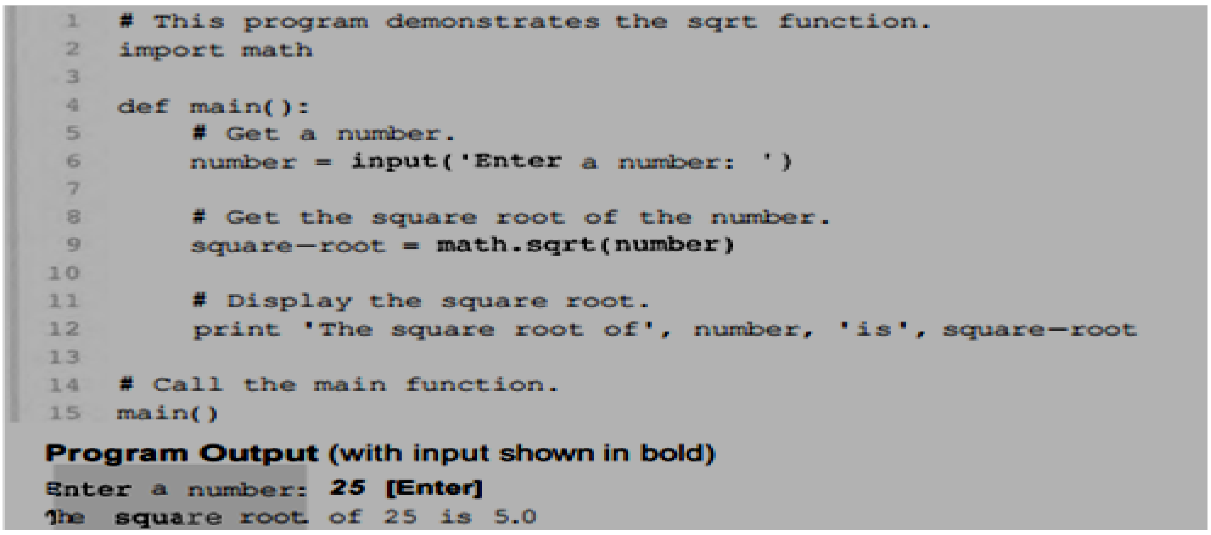
first-name, last-name = get-name()

TOPIC-4: MATH MODULE

- The Python standard library's math module contains numerous functions that can be used in mathematical calculations.
- 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.
- For example, one of the functions is named sqrt.
- The sqrt function accepts an argument and returns the square root of the argument.
- Here is an example of how it is used:

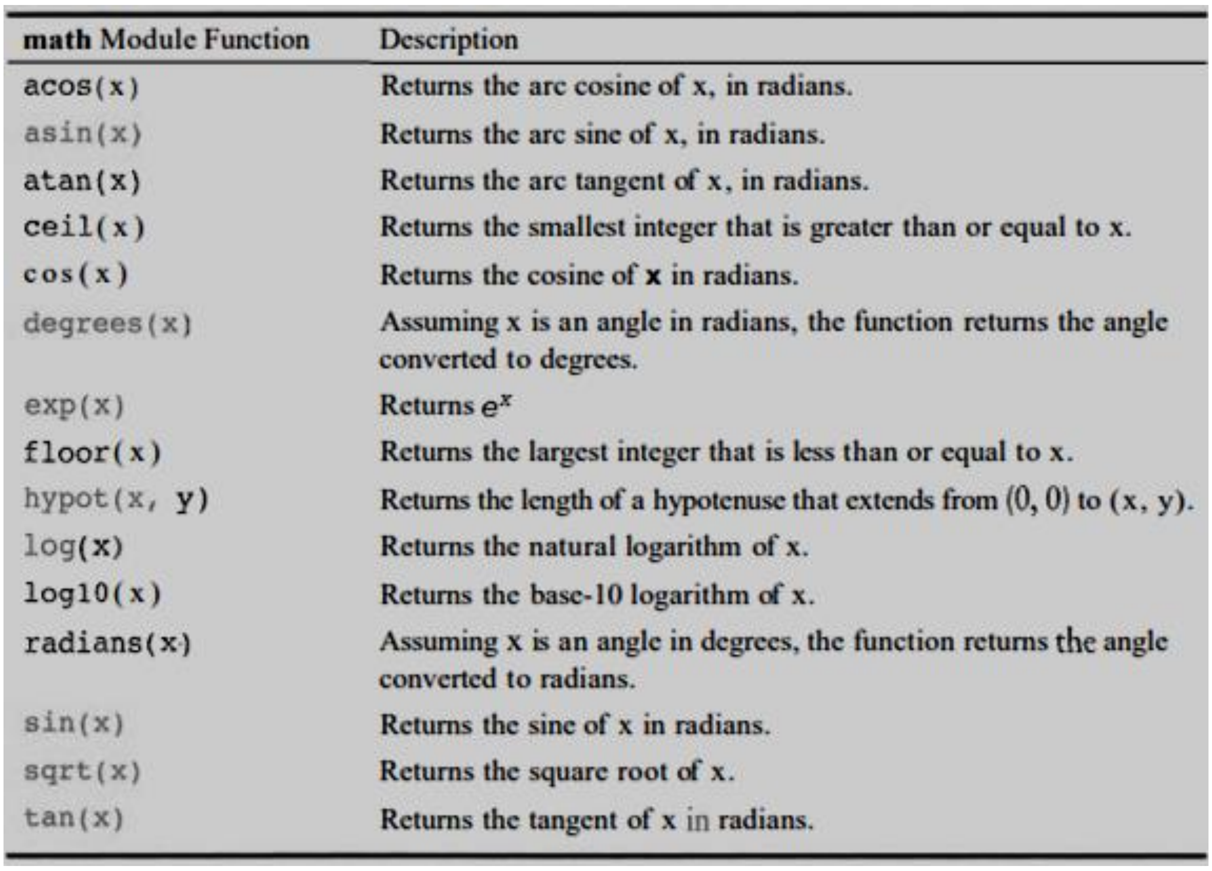
result = math.sqrt(l6)



```
# This program demonstrates the sqrt function.
import math

def main():
    # Get a number.
    number = input('Enter a number: ')

    # Get the square root of the number.
    square-root = math.sqrt(number)

    # Display the square root.
    print 'The square root of', number, 'is', square-root

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter a number: 25 [Enter]
The square root of 25 is 5.0
```



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.

TOPIC-5:- STORING FUNCTIONS IN MODULES

- A module is a file that contains Python code.
- Large programs are easier to debug and maintain when they are divided into modules.
- As your programs become larger and more complex, the need to organize your code becomes greater.
- You have already learned that a large and complex program should be divided into functions that each performs a specific task.
- As you write more and more functions in a program, you should consider organizing the functions by storing them 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.
- The circle module contains two function definitions: area (which returns the area of a circle), and circumference (which returns the circumference of a circle).

```
# The circle module has functions that perform calculations related to circles.
import math
 # The area function accepts a circle's radius as an argument and returns the area of the circle.
def area(radius):
return math.pi * radius**2
# The circumference function accepts a circle's radius and returns the circle's circumference.
def circumference(radius):
return 2 * math.pi * radius
```

The rectangle module contains two function definitions: area (which returns the area of a rectangle), and perimeter (which returns the perimeter of a rectangle.)

```
# The rectangle module has functions that perform calculations related to rectangles.
# The area function accepts a rectangle's width and # length as arguments and returns the rectangle's area.
def area(width, length):
return width * length
# The perimeter function accepts a rectangle's width and length as arguments and returns the rectangle's perimeter.
def perimeter(width, length):
return 2 * (width + length)
```

- Notice both of these files contain function definitions, but they do not contain code that calls the functions. That will be done by the program or programs that import these modules.
- Before continuing, we should mention the following things about module names:
- A module’s file name should end in .py. If the module’s file name does not end in .py, you will not be able to import it into other programs.
- A module’s name cannot be the same as a Python keyword. An error would occur, for example, if you named a module for.
- To use these modules in a program, you import them with the import statement. Here is an example of how we would import the circle module:

```
import circle

import rectangle
```

for example

```
circle.area(20)

circle.circumference(10)

rectangle.perimeter(23)
```

FILES AND EXCEPTIONS

TOPIC-6 INTRODUCTION TO FILE INPUT AND OUTPUT

- When a program needs to save data for later use, it writes the data in a file.
- The data can be read from the file at a later time.
- The programs you have written so far require the user to renter data each time the program runs, because data that is stored in RAM (referenced by variables) disappears once the program stops running.
- If a program is to retain data between the times it runs, it must have a way of saving it. Data is saved in a file, which is usually stored on a computer's disk.
- Once the data is saved in a file, it will remain there after the program stops running.
- Data that is stored in a file can be retrieved and used at a later time.
- Most of the commercial software packages that you use on a day-to-day basis store data in files.

The following are a few examples.

Word processors.

- Word processing programs are used to write letters, memos, reports, and other documents.
- The documents are then saved in files so they can be edited and printed.

Web browsers.

- Sometimes when you visit a Web page, the browser stores a small file known as a cookie on your computer.
- Cookies typically contain information about the browsing session, such as the contents of a shopping cart.

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

- 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.
- 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).
- 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.

Opening a File

- 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:

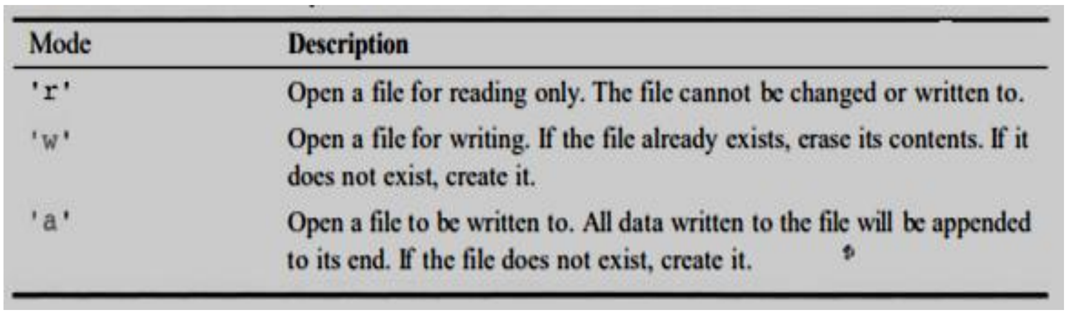
file-variable = open (filename, mode)

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.



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.

For example, suppose the file customers.txt contains customer data, and we want to open for reading. Here is an example of how we would call the open function:

customer-file = open('cusomters.txt', 'r')

Writing Data to a File

- A method is a function that belongs to an object, and performs some operation using that object.
- Once you have opened a file, you use the file object's methods to perform operations on the file.

- For example, 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 I a mode) or an error will occur.
- Let's assume that customer - file references a file object, and the file was opened for writing with the w mode.
- Here is an example of how we would write the string 'Charles Pace' to the file:

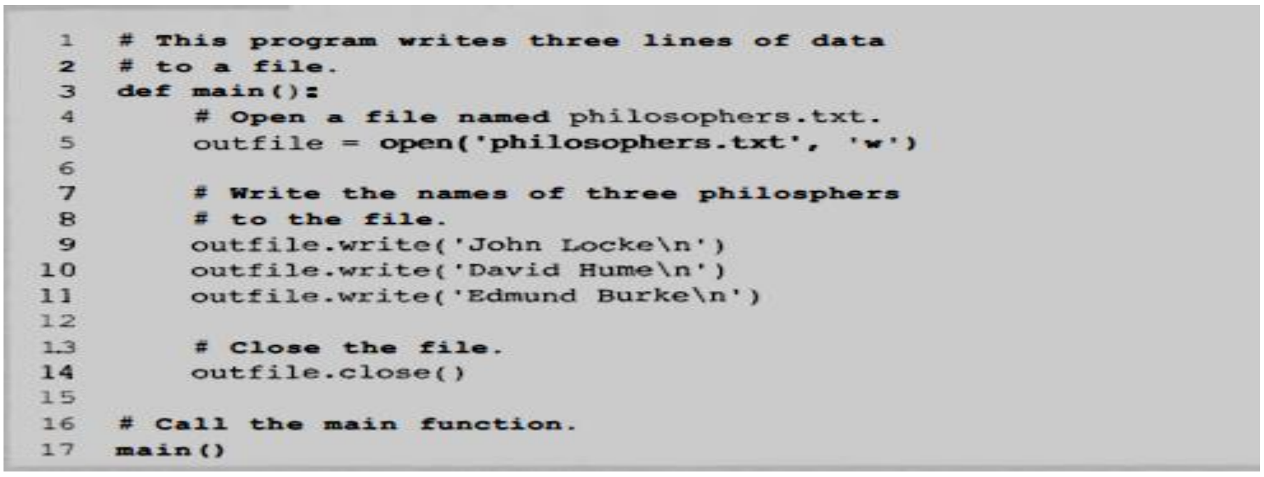
  customer-file.write('Charles Pace')

  The following code shows another example:

  name = 'Charles Pace'

  customer-file.write(name)

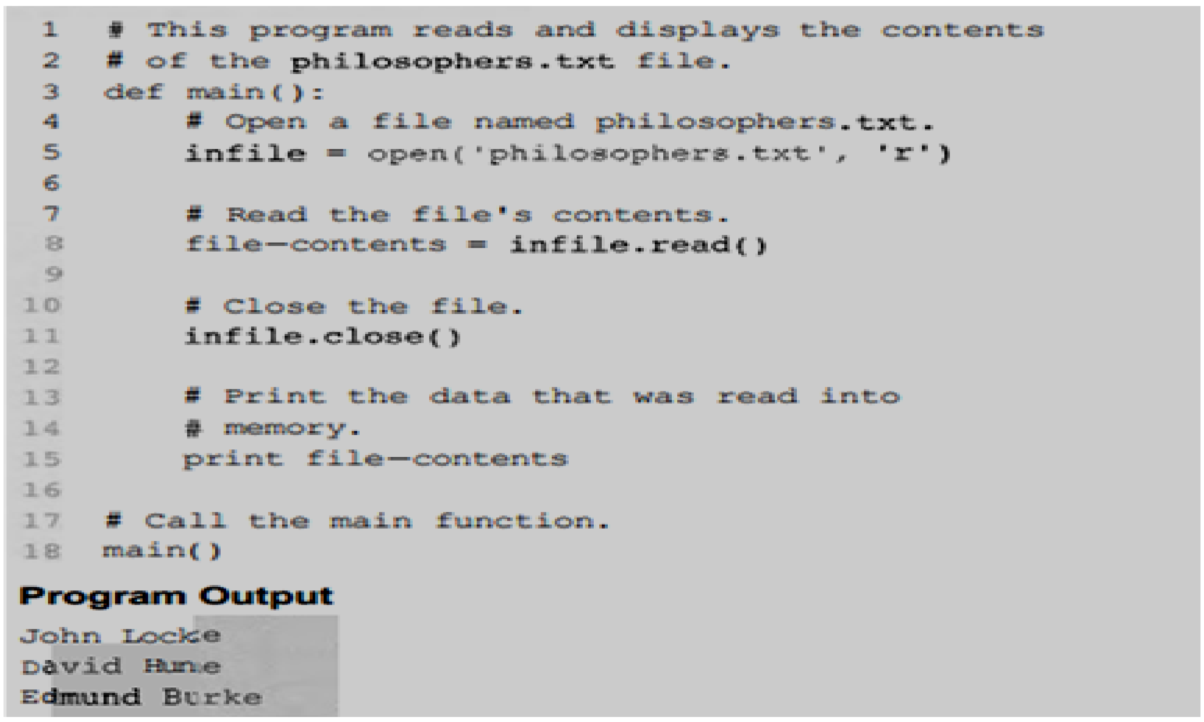
- The second statement writes the value referenced by the name variable to the file associated with customer - file.
- In this case, it would write the string 'Charles Pace' to the file.



```
# This program writes three lines of data
# to a file.
def main():
    # Open a file named philosophers.txt.
    outfile = open('philosophers.txt', 'w')

    # Write the names of three philosphers
    # to the file.
    outfile.write('John Locke\n')
    outfile.write('David Hume\n')
    outfile.write('Edmund Burke\n')

    # Close the file.
    outfile.close()

# Call the main function.
main()
```

Reading Data from a File

- If a file has been opened for reading (using the r mode) you 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.



```
# This program reads and displays the contents
# of the philosophers.txt file.
def main():
    # Open a file named philosophers.txt.
    infile = open('philosophers.txt', 'r')

    # Read the file's contents.
    file—contents = infile.read()

    # Close the file.
    infile.close()

    # Print the data that was read into
    # memory.
    print file—contents

# Call the main function.
main()
```

Program Output

```
John Locke
David Hume
Edmund Burke
```

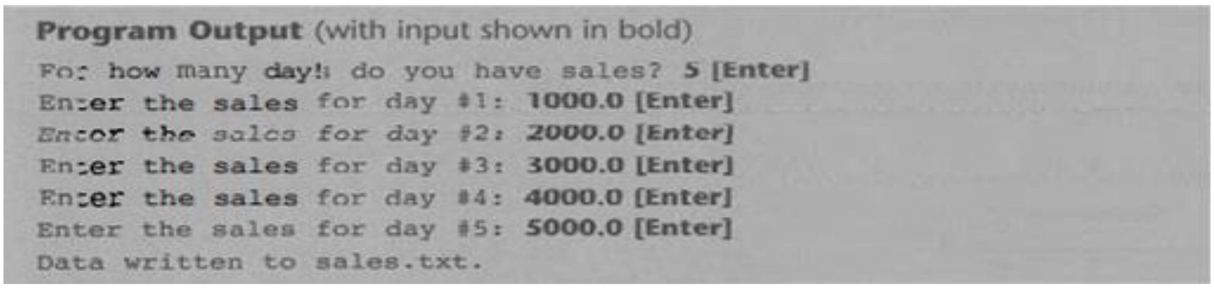
TOPIC-7 USING LOOP TO PROCESS FILE

- Files usually hold large amounts of data, and programs typically use a loop to process the data in a file.
- 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, the first 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.
- It shows the contents of the sales. txt file containing the data entered by the user in the sample run.

Example

```
def main( ) :
    # Get the number of days.
num-days = input('For how many days do ' + \ 7 'you have sales? ' )
sales-file = open('sales.txtl, 'w')
    # Get the amount of sales for each day and write it to the file.
for count in range( 1, num-days + 1 ) :
# Get the sales for a day.
```

```
sales = input( 'Enter the sales for day 8' + \ str(count) + ': ')
 # Write the sales amount to the file.
            sales-file.write(str(sales) + '\n')
# Close the file.
      sales-file.close()
print 'Data written to sa1es.txt.'

# Call the main function.
main ( )
```



Program Output (with input shown in bold)

```
For how many days do you have sales? 5 [Enter]
Enter the sales for day #1: 1000.0 [Enter]
Enter the sales for day #2: 2000.0 [Enter]
Enter the sales for day #3: 3000.0 [Enter]
Enter the sales for day #4: 4000.0 [Enter]
Enter the sales for day #5: 5000.0 [Enter]
Data written to sales.txt.
```

- 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:

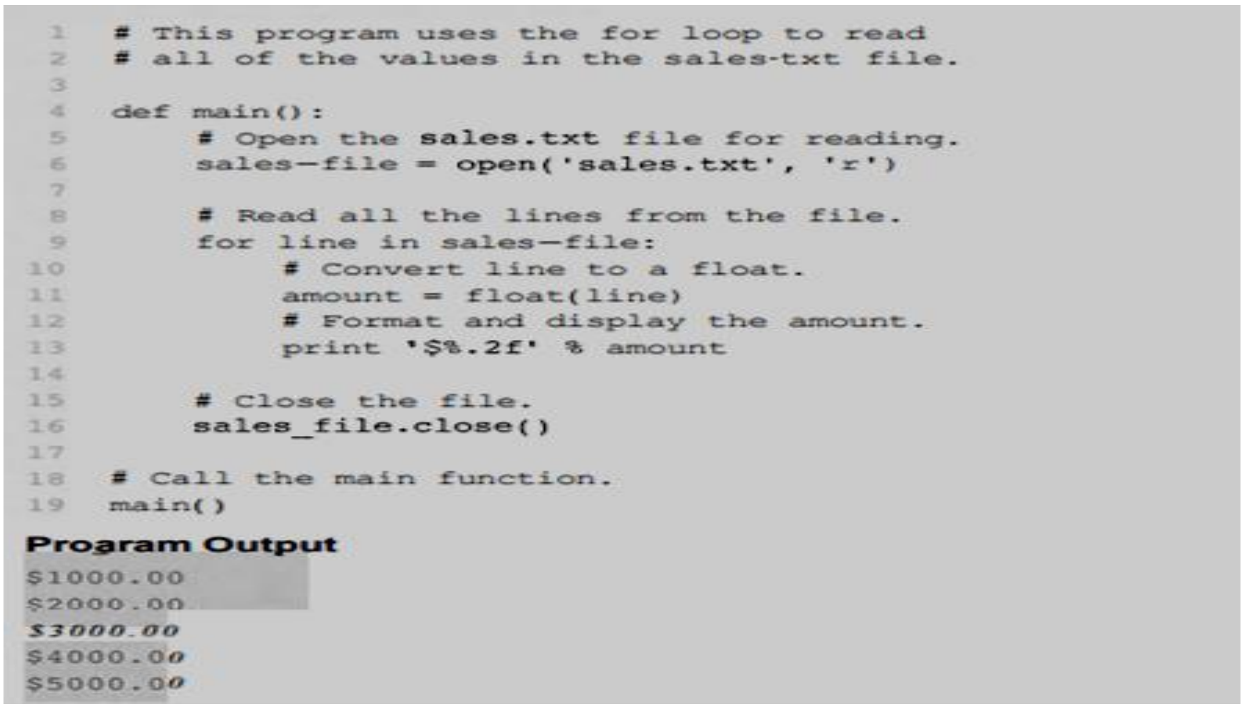
for variable in file-object:

statement

statement

etc.

Example



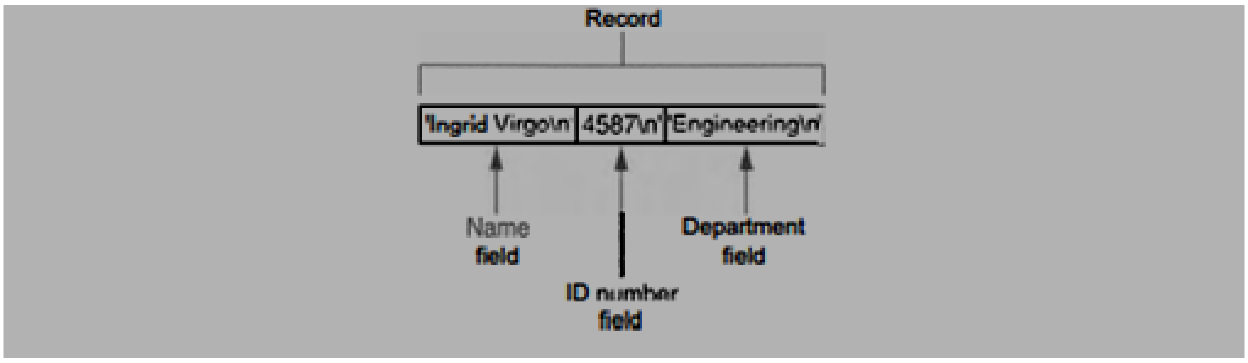
```
# This program uses the for loop to read
# all of the values in the sales-txt file.

def main():
    # Open the sales.txt file for reading.
    sales—file = open('sales.txt', 'r')

    # Read all the lines from the file.
    for line in sales—file:
        # Convert line to a float.
        amount = float(line)
        # Format and display the amount.
        print '$%.2f' % amount

    # Close the file.
    sales_file.close()

# Call the main function.
main()
```

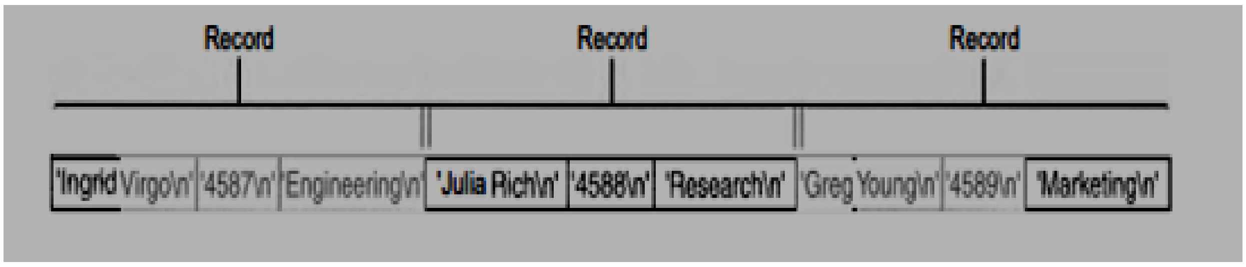
Program Output

```
$1000.00
$2000.00
$3000.00
$4000.00
$5000.00
```

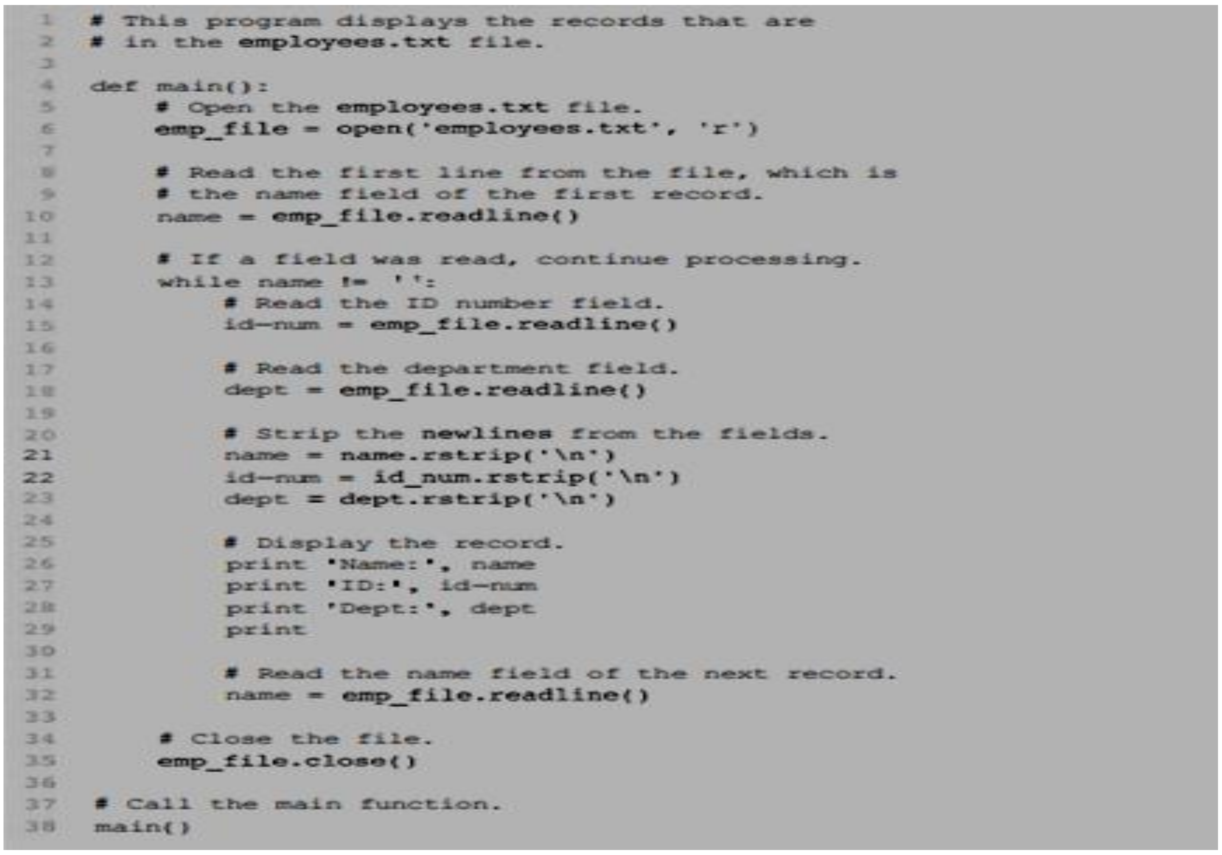
TOPIC-8 PROCESSING RECORDS

- The data that is stored in a file is frequently organized in records.
- A record is a complete set of data about an item, and a field is an individual piece of data within a record. 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.





Example



```
# This program displays the records that are
# in the employees.txt file.

def main():
    # Open the employees.txt file.
    emp_file = open('employees.txt', 'r')

    # Read the first line from the file, which is
    # the name field of the first record.
    name = emp_file.readline()

    # If a field was read, continue processing.
    while name != '':
        # Read the ID number field.
        id—num = emp_file.readline()

        # Read the department field.
        dept = emp_file.readline()

        # Strip the newlines from the fields.
        name = name.rstrip('\n')
        id—num = id_num.rstrip('\n')
        dept = dept.rstrip('\n')

        # Display the record.
        print 'Name:', name
        print 'ID:', id—num
        print 'Dept:', dept
        print

        # Read the name field of the next record.
        name = emp_file.readline()

    # Close the file.
    emp_file.close()

# Call the main function.
main()
```

Name: Ingrid Virgo

ID: 4587

Dept: Engineering

Name: Julia Rich

ID: 4588

Dept: Research Name:

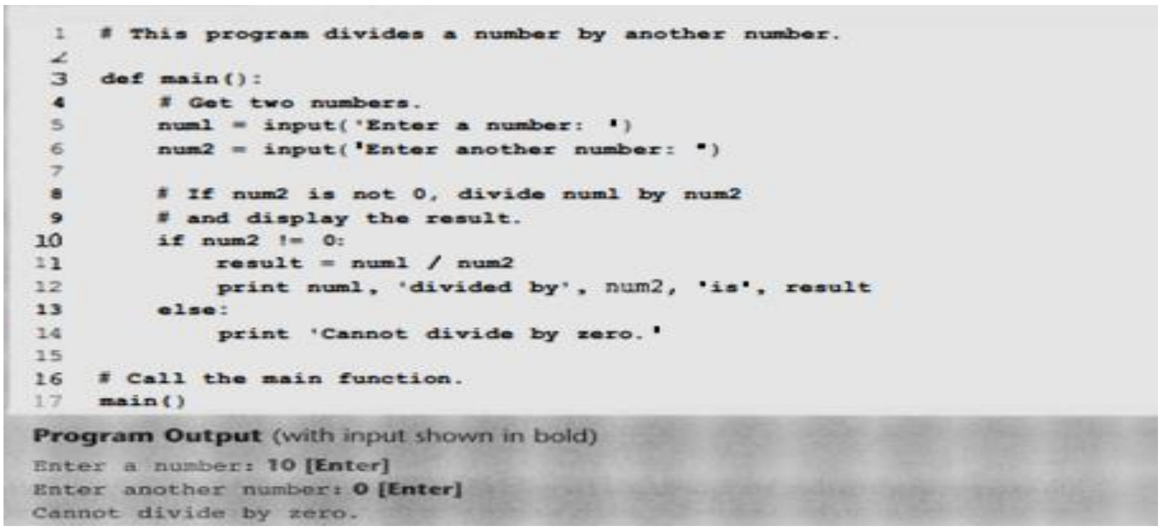
Name: Greg Young

ID: 4589

Dept : Marketing

TOPIC-9 EXCEPTIONS

- An exception is an error that occurs while a program is running, causing the program to abruptly halt. You can use the try/except statement to gracefully handle exceptions. You can prevent many exceptions from being raised by carefully coding your program.
- For example, the below program shows how division by 0 can be prevented with a simple if statement.
- Rather than allowing the exception to be raised, the program tests the value of num2, and displays an error message if the value is 0.
- This is an example of gracefully avoiding an exception.



```
# This program divides a number by another number.

def main():
    # Get two numbers.
    num1 = input('Enter a number: ')
    num2 = input('Enter another number: ')

    # If num2 is not 0, divide num1 by num2
    # and display the result.
    if num2 != 0:
        result = num1 / num2
        print num1, 'divided by', num2, 'is', result
    else:
        print 'Cannot divide by zero.'

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter a number: 10 [Enter]
Enter another number: 0 [Enter]
Cannot divide by zero.
```

- Python, like most modern programming languages, allows you 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.

exceptExceptionName :

statement

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 block.

- The try block is one or more statements that can potentially raise an exception.
- After the try block, 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 block begin to execute.
- The following describes what happens next:
- If a statement in the try block raises an exception that is specified by the Exception Name 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 block raises an exception that is not specified by the Exception Name in an except clause, then the program will halt with a traceback error message.
- If the statements in the try block 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.
- Like this we can catch multiple exceptions by using different catch blocks for a single try block.

UNIT – IV

Lists and Tuples: Sequences – Introduction to Lists – List Slicing – Finding items in list with the "in" operator – List Methods and Useful Built-in functions - #Copying Lists# - Processing Lists – Two-Dimensional Lists – Tuples – More about Strings – Basic String Operations – String Slicing – Testing, Searching and Manipulating Strings – Dictionaries and Sets: Dictionaries – Sets – Serializing Objects

TOPIC-1 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's execution.
- Lists are dynamic data structures, meaning that items may be added to them or removed from them.
- You can use indexing, slicing, and various methods to work with lists in a program.
  even-numbers = [2, 4, 6, 8, 10]
- The items that are enclosed in brackets and separated by commas are the list elements.
  names = ['Molly', 'Steven', 'Will', 'Alicia', 'Adriana']
- This statement creates a list of five strings.
  numbers = [5, 10, 15, 20]
  print numbers

  When the print statement executes, it will display the elements of the list like this:

  [5, 10, 15, 20]
  numbers = range(5)
- In this statement the range function will return a list of integers in the range of 0 up to (but not including) 5.
- This statement will assign the list 0, 1, 2, 3, 4 1 to the numbers variable. Here is another example:
  numbers = range(1, 10, 2)
- When you pass three arguments to the range function, the first argument is the list's starting value, the second argument is the list's ending limit, and the third argument is the step value.
- This statement will assign the list [1, 3, 5, 7, 9] to the numbers variable
- You can use the repetition operator (") to easily create a list with a specific number of elements, each with the same value. Here is an example:
  numbers = [O] * 5
- Alicia This statement will create a list with five elements, with each element holding the value 0.
- This statement will assign the list [0, 0, 0, 0, 0] to the numbers variable.

Iterating Over a List with the for Loop

```
numbers = [99, 100, 101, 102]

for n in numbers:

print n
```

If we run this code, it will print:

```
99

100

101

102
```

Indexing:-

- Indexing works with lists just as it does with strings.
- Each element in a list has an index which 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 forth.
- The index of the last element in a list is 1 less than the number of elements in the list.
- For example, the following statement creates a list with 4 elements:

```
my_list = [10, 20, 30, 40]
```

- The indexes of the elements in this list are 0, 1, 2, and 3.
- We can print the elements of the list with the following statement:

```
printmy_list[0], my_list[l], my_list[2], my_list[3]
```

The following loop also prints the elements of the list:

```
index = 0

while index < 4:

printmy_list[index]

index += 1
```

Negative Indexing

- You can also use negative indexes with lists, to identify element positions relative to the end of the list.
- The Python interpreter adds negative indexes to the length of the list to determine the element position.

- The index -1 identifies the last element in a list, -2 identify the next to last element, and so forth.
- The following code shows an example:

```
my_list = [10, 20, 30, 40]

print my_list[-1], my_list[-2], my_list[-3], my_list[-4]
```

This print statement will display:

```
40 30 20 10
```

- An IndexError exception will be raised if you use an invalid index with a list.
- For example

```
My_list = [10, 20, 30, 40]

index = 0

while index < 5:

print my-list[index]

index += 1
```

- The last time that this loop iterates, the index variable will be assigned the value 5, which is an invalid index for the list.
- As a result, the print statement will cause an IndexError exception to be raised.
- When you pass a list as an argument, the len function returns the number of elements in the list.

```
myLlist = [10, 20, 30, 40]

index = 0

while index <len(my-list) :

print my-list[index]

index += 1
```

TOPIC-2: LIST SLICING

- Slicing operations work with lists just as they do with strings.
- For example, suppose we create the following list:

```
days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

- The following statement uses a slicing expression to get the elements from indexes 2 up to, but not including, 5:

mid-days = days[2:5]

- After this statement executes the mid - days variable will reference the following list:

['Tuesday', 'Wednesday', 'Thursday'].

TOPIC-3 FINDING ITEMS IN A LIST WITH ‘IN’ AND ‘NOT IN‘

- You can use the in operator to determine whether an item is contained in a list.

```
# This program demonstrates the ‘in’ operator used with a list.
def main( ) :
# Create a list of product numbers.
prod_nums = ['V475', 'F987', 'Q143', 'R688']
#Get a product number to search for.
search = raw-input('Enter a product number: ')
# Determine whether the product number is in the list.
if search in prod_nums:
print search, 'was found in the list.'
else:
print search, 'was not found in the list. '
# Call the main function.
main( )
```

Output

```
Enter a product number: Q143
Q143 was found in the list
Enter a product number: 8000
8000 was found in the list
```

You can use the not in operator to determine whether an item is not in a list.

Here is an example:

```
if search not in prod_nums:
print search, 'was not found in the list.'
else:
```

print search, 'was found in the list. '

- List Are Mutable.
- Unlike strings, lists in Python are mutable, which means their elements can be changed.
- Consequently, an expression in the form list [index] can appear on the left side of an assignment operator.
- The following code shows an example:

```
numbers = [1, 2, 3, 4, 5]
print numbers
numbers[0] = 99
print numbers
```

The statement in line 2 will display:

```
[1, 2, 3, 4, 5]
```

The statement in line 3 assigns 99 to numbers [0].

This changes the first value in the list to 99.

When the statement in line 4 executes it will display:

```
[99, 2, 3,4,5]
```

TOPIC-4 LIST METHODS

1. append()

- To add items in a list at the end

Syntax:

samplelist.append(Item)

Example:

mylist.append("Apple")

2. index()

- It returns the index of the first occurrence element in a list.

Syntax:

index(Item)

Example:

index("Apple")

3. **insert()**
    - It is used to insert items in the list at the specified index.

Syntax:

index(Item)

Example:

index("Apple")

4. **sort()**
    - It sorts the items in the list in ascending order (from lowest to highest).

Syntax:

sort()

5. **remove()**
    - It removes the first occurrence of item from the list.

Syntax:

remove(Item)

Example:

remove("Apple")

6. **reverse()**
    - It reverses the order of the item in a list.

Syntax:

Samplelist.reverse()

Example:

My_list.reverse()

Example

```
# This program demonstrates how the append
# method can be used to add items to a list.
def main ( ) :
   # First, create an empty list.
name-list = []
   # Create a variable to control the loop.
```

```
again = 'Y'
   # Add some names to the list.
whileagain.upper() == 'Y':
        # Get a name from the user.
name = raw-input ( ' Enter a name: ' )
        # Append the name to the list.
        name-list.append(name)
        # Add another one?
print 'Do you want to add another name?'
again = raw-input( 'Y = yes, anything else = no: ' )
  # Display the names that were entered.
print 'Here are the names you entered.'
for name in name-list:
print name
      # Call the main function.
main()
```

Output

```
Enter a name: Dr. S. Peerbasha
Do you want to add another name?
'Y = yes, anything else = no: y

Enter a name: Mr. Y. Mohammed Iqbal
Do you want to add another name?
'Y = yes, anything else = no: y

Enter a name: Mr. P.U. Manimaran
Do you want to add another name?
'Y = yes, anything else = no: n
```

```
Here are the names you entered
Dr. S. Peerbasha
Mr. Y. Mohammed Iqbal
Mr. P.U. Manimaran
```

Example 2

```
name_list = [‘Peerbasha’, ‘Iqbal’, ‘Basheer’, ‘Shabeer’]
name_list[3] = ‘Kamal’
print(name_list)                O/P:- [‘Peerbasha’, ‘Iqbal’, ‘Basheer’, ‘Kamal’]
name_list.index(‘Basheer’)      O/P:- 2
name_list.insert(0,’Jamal’)     O/P :- [‘Jamal’, ‘Peerbasha’, ‘Iqbal’, ‘Basheer’, ‘Kamal’]
name_list.sort()                O/P::- [‘Basheer’,’Iqbal’,’Jamal’,’Kamal’,’Peerbasha’]
name_list.remove(‘Peerbasha’)
print(name_list)                O/P:-:- [‘Basheer’,’Iqbal’,’Jamal’,’Kamal’]
name_list.reverse()             O/P:- [‘Kamal’,’Jamal’,’Iqbal’,’Basheer]
```

TOPIC-5: PYTHON BUILT IN FUNCTIONS

Function	**Description**
print()	Displayed function.
input ()	Get value from user.
int()	Convert integer data type.
float()	Convert float data type.
str()	Convert string data type.
type()	Displayed types of variables.
round()	Returns the nearest integer to its input.
max()	Returns the maximum value in a list.
min()	Returns the minimum value in a list.
sum()	Returns the sum of value in a list.
pow()	Return the power of values.

TOPIC-6 COPYING LIST

- A list can be copied into another list.

Example

```
List1 = [1,2,3,4]

List2 = List1

print(List1)

print(List2)
```

Output

```
[1,2,3,4]

[1,2,3,4]
```

TOPIC-7 PROCESSING LISTS

- Working with lists is very common.
- Python is very adept at processing lists

```
>>> a = [1, 2, 3, 4, 5]
>>> sum(a)
15
>>> a[0:3]
[1, 2, 3]
>>> a * 2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> min(a)
1
```

TOPIC-8 TWO DIMENSIONAL LISTS

- Lists can contain other lists as elements.
- A typical use of such nested (or multidimensional) lists is to represent **tables** of values consisting of information arranged in **rows** and **columns**.
- To identify a particular table element, we specify *two* indices—by convention, the first identifies the element's row, the second the element's column.
- Lists that require two indices to identify an element are called **two-dimensional lists** (or **double-indexed lists** or **double-subscripted lists**).
- Multidimensional lists can have more than two indices.
- Here, we introduce two-dimensional lists.

Creating a Two-Dimensional List

Consider a two-dimensional list with three rows and four columns (i.e., a 3-by-4 list) that might represent the grades of three students who each took four exams in a course:

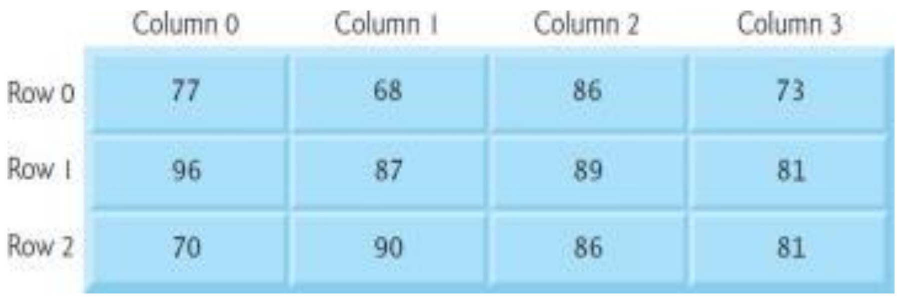
In [1]: a = [[77, 68, 86, 73], [96, 87, 89, 81], [70, 90, 86, 81]]

Writing the list as follows makes its row and column tabular structure clearer:

```
a = [[77, 68, 86, 73],  # first student's grades
     [96, 87, 89, 81],  # second student's grades
     [70, 90, 86, 81]]  # third student's grades
```

Illustrating a Two-Dimensional List

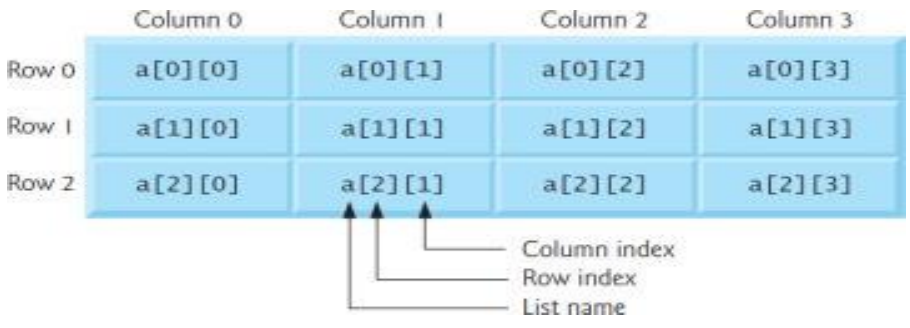
The diagram below shows the list a, with its rows and columns of exam grade values:



	Column 0	Column 1	Column 2	Column 3
Row 0	77	68	86	73
Row 1	96	87	89	81
Row 2	70	90	86	81

Identifying the Elements in a Two-Dimensional List

The following diagram shows the names of list a's elements:



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Every element is identified by a name of the form a[*i*][*j*]—a is the list's name, and *i* and *j* are the indices that uniquely identify each element's row and column, respectively.
- The element names in row 0 all have 0 as the first index.
- The element names in column 3 all have 3 as the second index.

In the two-dimensional list a:

77, 68, 86 and 73 initialize a[0][0], a[0][1], a[0][2] and a[0][3], respectively,

96, 87, 89 and 81 initialize a[1][0], a[1][1], a[1][2] and a[1][3], respectively, and

70, 90, 86 and 81 initialize a[2][0], a[2][1], a[2][2] and a[2][3], respectively.

- A list with *m* rows and *n* columns is called an **m-by-n list** and has *m* × *n* elements.
- The following nested for statement outputs the rows of the preceding two-dimensional list one row at a time:

```
In [2]: for row in a:
   ...:     for item in row:
   ...:         print(item, end=' ')
   ...:     print()
   ...:
77 68 86 73
96 87 89 81
```

TOPIC-9 TUPLES

- Tuples are used to store multiple items in a single variable.
- Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
- A tuple is a collection which is ordered and **unchangeable**.
- Tuples are written with round brackets.

```
mytuple = ("apple", "banana", "cherry")
```

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Tuple Items

- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered

- When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable

- Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates

- Since tuples are indexed, they can have items with the same value:

Example

Tuples allow duplicate values:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Example

Print the second item in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Negative Indexing

Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.

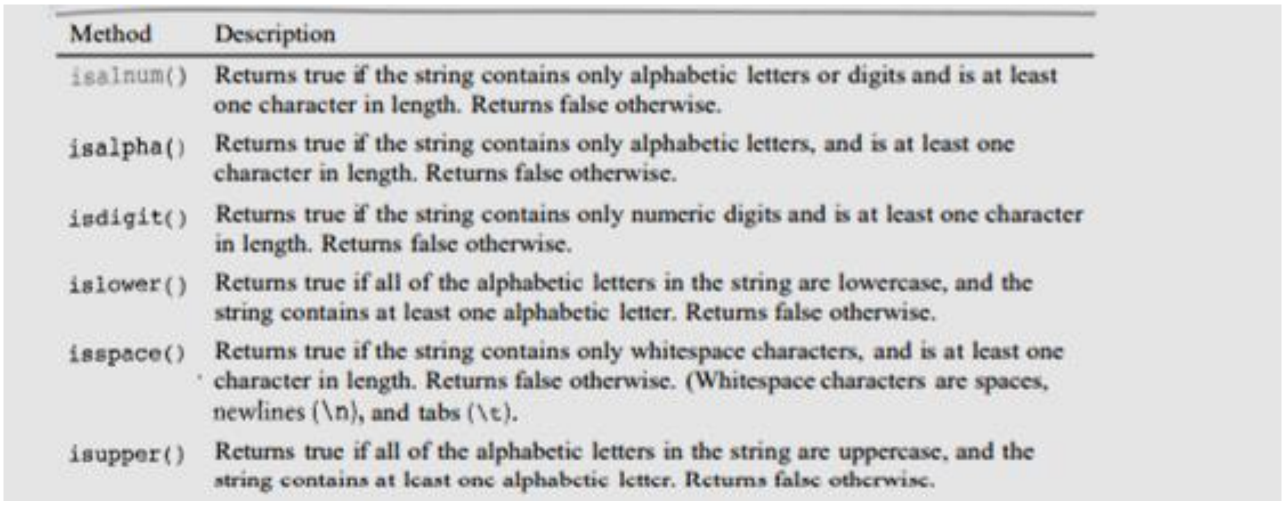
Example

Print the last item of the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

MORE ABOUT STRINGS:

TOPIC-10 BASIC STRING OPERATIONS:

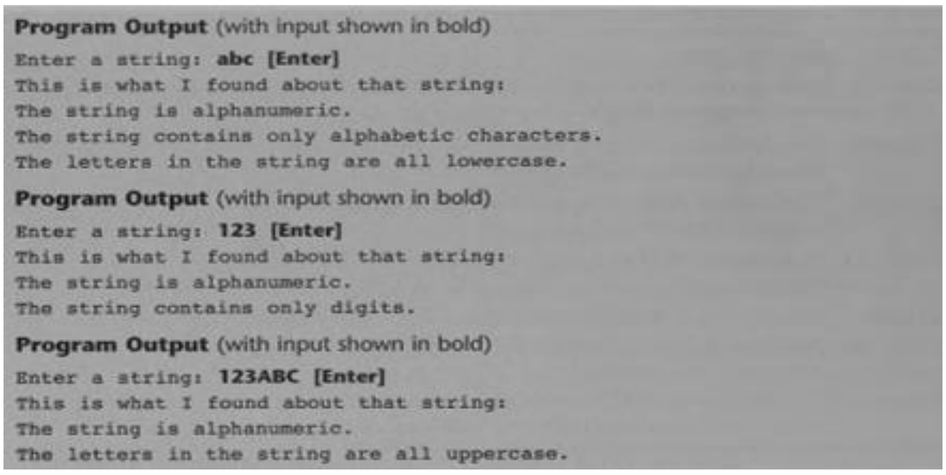


Method	Description
`isalnum()`	Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.
`isalpha()`	Returns true if the string contains only alphabetic letters, and is at least one character in length. Returns false otherwise.
`isdigit()`	Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.
`islower()`	Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.
`isspace()`	Returns true if the string contains only whitespace characters, and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (`\n`), and tabs (`\t`).
`isupper()`	Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.

Program

```
def main( ) :
    # Get a string from the user.
user - string = raw-input('Enter a string: ')
print 'This is what I found about that string: '
    # Test the string.
if user-string.isalnum():
print 'The string is alphanumeric.'
if user-string.isdigit():
print 'The string contains only digits.'
if user-string.isalpha():
print 'The string contains only alphabetic characters.'
if user-string.isspace():
print 'The string contains only whitespace characters.'
if user-string.islower():
print 'The letters in the string are all lowercase. '
if user-string.isupper():
print 'The letters in the string are all uppercase. '


# Call the string.
main( )
```

Output:



Program Output (with input shown in bold)

```
Enter a string: abc [Enter]
This is what I found about that string:
The string is alphanumeric.
The string contains only alphabetic characters.
The letters in the string are all lowercase.
```

Program Output (with input shown in bold)

```
Enter a string: 123 [Enter]
This is what I found about that string:
The string is alphanumeric.
The string contains only digits.
```

Program Output (with input shown in bold)

```
Enter a string: 123ABC [Enter]
This is what I found about that string:
The string is alphanumeric.
The letters in the string are all uppercase.
```

TOPIC-11 STRING SLICING

- A slice is a span of items that are taken from a sequence.
- When you take a slice from a string, you get a span of characters from within the string.
- String slices are also called substrings.
- To get a slice of a string, you 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.
- The expression will return a string containing a copy of the characters from start up to (but not including) end.
- For example, suppose we have the following:

```
full_name = ' Patty Lynn Smith '

middle_name = full_name[6:10]
```

- The second statement assigns the string 'Lynn to the middle_name variable.
- If you leave out the start index in a slicing expression, Python uses 0 as the starting index. Here is an example:

```
full_name = 'Patty Lynn Smith'

first_name = full_name[:5]
```

- The second statement assigns the string ' Lynn ' to first name.
- If you leave out the end index in a slicing expression, Python uses the length of the string as the end index. Here is an example:

```
full_name = 'Patty Lynn Smith'

last_name = full-name[11:]
```

- The second statement assigns the string Smith to first_name.
- What do you think the following code will assign to the my_string variable?

```
full_name = ' Patty Lynn Smith '

my_string = full_name[:]
```

- The second statement assigns the entire string ' Patty Lynn Smith ' to my_string.
- The statement is equivalent to:

```
my_string = full-name[0 : len(ful1_name)]
```

- The slicing examples we have seen so far get slices of consecutive characters from strings.
- Slicing expressions can also have step value, which can cause characters to be skipped in the string.
- Here is an example of code that uses a slicing expression with a step value:

```
letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

print letters[0:26:2]
```

- The third number inside the brackets is the step value.
- A step value of 2, as used in this example, causes the slice to contain every second character from the specified range in the string.
- The code will print the following:

```
ACEGIKMOQSUWY
```

- You can also use negative numbers as indexes in slicing expressions to reference positions relative to the end of the string.
- Here is an example:

```
full_name = ' Patty Lynn Smith '

last_name = full-name[-5:]
```

- Recall that Python adds a negative index to the length of a string to get the position referenced by that index.
- The second statement in this code assigns the string ' Smith ' to the last_name variable.

TOPIC-12 TESTING, SEARCHING AND MANIPULATING STRINGS

TESTING STRING WITH “in” AND “not in”

- In Python you can use the in operator to determine whether one string is contained in another string.
- Here is the general format of an expression using the in operator with two strings:

```
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.
- For example, look at the following code:

```
text = 'Four score and seven years ago'

if 'seven' in text:

        print 'The string "seven" was found.'

else :

        print 'The string "seven" was not found.'
```

This code determines whether the string ' Four score and seven years ago ' contains the string ' seven '. If we run this code it will display:

The string "seven" was found.

You can use the not in operator to determine whether one string is not contained in another string.

Here is an example:

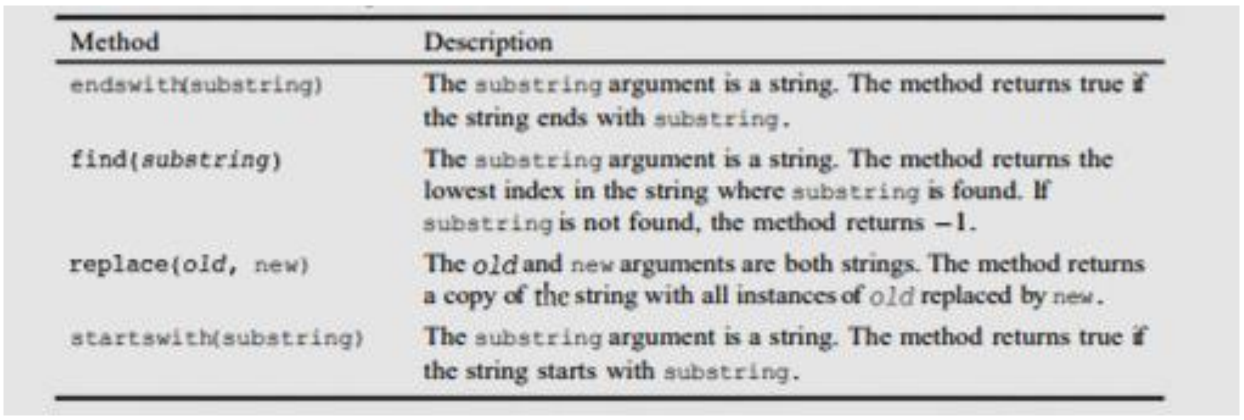
```
names = 'Bill Joanne Susan Chris Juan Katie'

if 'Pierre' not in names:

print 'Pierre was not found.'

else : print 'Pierre was found.'
```

If we run this code it will display:

```
Pierre was not found.
```

SEARCHING AND REPLACING

- Programs commonly need to search for substrings, or strings that appear within other strings.
- For example, suppose you have a document opened in your word processor, and you need to search for a word that appears somewhere in it.
- The word that you are searching for is a substring that appears inside a larger string, the document.



Method	Description
`endswith(substring)`	The `substring` argument is a string. The method returns true if the string ends with `substring`.
`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.
`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`.
`startswith(substring)`	The `substring` argument is a string. The method returns true if the string starts with `substring`.

DICTIONARIES AND SETS

TOPIC-13 DICTIONARIES

- A dictionary is an object that stores a collection of data.
- Each element in a dictionary has two parts: a key and a value.
- You use a key to locate a specific value.

Creating a Dictionary

- You can create a dictionary by enclosing the elements inside a set of curly braces ({}).
- An element consists of a key, followed by a colon, followed by a value.
- The elements are separated by commas.
- The following statement shows an example:

  phonebook = {'Jones':'555−1111', 'JJ':'555−2222', 'JJappa':'555−3333'}

- This statement creates a dictionary and assigns it to the phonebook variable.
- The dictionary contains the following three elements:
- The first element is 'Jones':'555−1111'. In this element, the key is 'Jones' and the value is '555−1111'.
- The second element is 'JJ':'555−2222'. In this element, the key is 'JJ' and the value is '555−2222'.
- The third element is 'JJaapa':'555−3333'. In this element, the key is 'JJaapa' and the value is '555−3333'.

To retrieve a value from a dictionary, you simply write an expression in the following general format:

dictionary_name[key]

For Example

print(Phonebook[‘Jones’])

print(Phonebook[‘JJ’])

print(Phonebook[‘JJaapa’])

Output

```
555-1111

555-2222

555-3333
```

Adding Elements to an Existing Dictionary

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

  dictionary_name[key] = value

  Example

  Phonebook[‘Salman’] = ‘111-1111’   #add new value

Phonebook[‘JJaapa’] = ‘121-1212’ #changing value using key

Deleting Elements

- You can delete an existing key-value pair from a dictionary with the del statement.
- Here is the general format:

deldictionary_name[key]

Example

del Phonebook[‘JJaapa’]

Using the for Loop to Iterate over a Dictionary

- You can use the for loop in the following general format to iterate over all the keys in a dictionary:

for var in dictionary:

statement

statement

etc

Example

for key in Phonebook:

print(key, Phonebook[key])

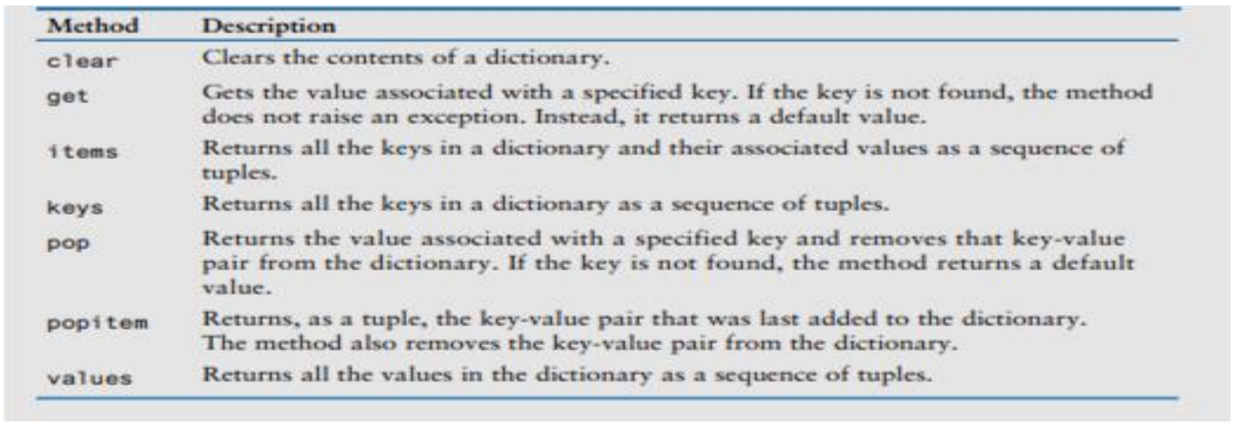
Output:-

Jones 555-1111

JJ 555-2222

Salman 111-1111

SOME DICTIONARY METHODS



Method	Description
clear	Clears the contents of a dictionary.
get	Gets the value associated with a specified key. If the key is not found, the method does not raise an exception. Instead, it returns a default value.
items	Returns all the keys in a dictionary and their associated values as a sequence of tuples.
keys	Returns all the keys in a dictionary as a sequence of tuples.
pop	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.
popitem	Returns, as a tuple, the key-value pair that was last added to the dictionary. The method also removes the key-value pair from the dictionary.
values	Returns all the values in the dictionary as a sequence of tuples.

TOPIC-14 SETS

- A set contains a collection of unique values and works like a mathematical set.
- A set is an object that stores a collection of data in the same way as mathematical sets.
- Here are some important things to know about sets:
    - All the elements in a set must be unique. No two elements can have the same value.
    - Sets are unordered, which means that the elements in a set are not stored in any particular order.
    - The elements that are stored in a set can be of different data types.

Creating a Set

- To create a set, you have to call the built-in set function.
- Here is an example of how you create an empty set:

```
myset = set()
```

- If you pass a string as an argument to the set function, each individual character in the string becomes a member of the set.
- Here is an example:

```
myset = set('abc')
```

- After this statement executes, the myset variable will reference a set containing the elements 'a', 'b', and 'c'.
- Sets cannot contain duplicate elements.
- If you pass an argument containing duplicate elements to the set function, only one of the duplicated elements will appear in the set.
- Here is an example:

```
myset = set('aaabc')
```

- The character 'a' appears multiple times in the string, but it will appear only once in the set.
- After this statement executes, the myset variable will reference a set containing the elements 'a', 'b', and 'c'.
- What if you want to create a set in which each element is a string containing more than one character?
- For example, how would you create a set containing the elements 'one', 'two', and 'three'?
- The following code does not accomplish the task, because you can pass no more than one argument to the set function:

```
# This is an ERROR!

myset = set('one', 'two', 'three')
```

The following does not accomplish the task either:

```
# This does not do what we intend.
```

myset = set('one two three')

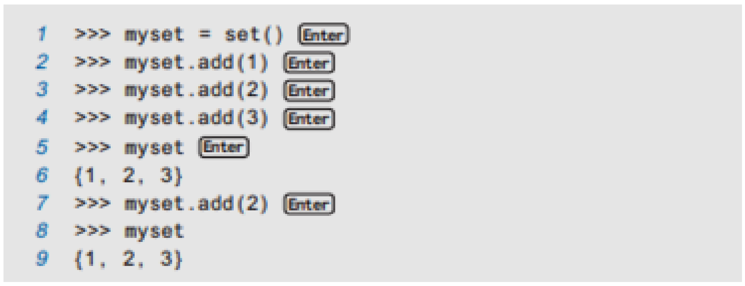
- ❖ After this statement executes, the myset variable will reference a set containing the elements 'o', 'n', 'e', ' ', 't', 'w', 'h', and 'r'.
- ❖ To create the set that we want, we have to pass a list containing the strings ‘one’, ‘two’, and ‘three’ as an argument to the set function.
- ❖ Here is an example:

```
# OK, this works.

myset = set(['one', 'two', 'three'])
```

Adding and Removing Elements

- ❖ Sets are mutable objects, so you can add items to them and remove items from them.
- ❖ You use the add method to add an element to a set.

To add



```
>>> myset = set()
>>> myset.add(1)
>>> myset.add(2)
>>> myset.add(3)
>>> myset
{1, 2, 3}
>>> myset.add(2)
>>> myset
{1, 2, 3}
```

To remove

- You can remove an item from a set with either the remove method or the discard method.
- You pass the item that you 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 does not raise an exception.

```
myset.remove(1)
myset.remove(2)
```

Using the for Loop to Iterate over a Set

- You can use the ‘for loop’ in the following general format to iterate over all the elements in a set:

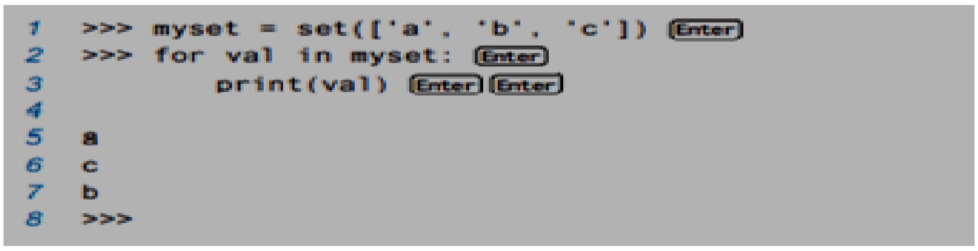
for var in set:

statement

statement

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.
- Each time the loop iterates, var is assigned an element.
- The following interactive session demonstrates



```
>>> myset = set(['a', 'b', 'c']) Enter
>>> for val in myset: Enter
        print(val) Enter Enter

a
c
b
>>>
```

TOPIC-15 SERIALIZING OBJECTS

- Serializing an object is the process of converting the object to a stream of bytes that can be saved to a file for later retrieval.
- In Python, object serialization is called pickling.
- In Python, the process of serializing an object is referred to as pickling.
- The Python standard library provides a module named pickle that has various functions for serializing, or pickling, objects.
- Once you import the pickle module, you perform the following steps to pickle an object:
- You open a file for binary writing.
- You call the pickle module’s dump method to pickle the object and write it to the specified file.
- After you have pickled all the objects that you want to save to the file, you close the file.
- Let’s take a more detailed look at these steps.
- To open a file for binary writing, you use 'wb' as the mode when you call the open function.
- For example, the following statement opens a file named mydata.dat for binary writing:

  outputfile = open('mydata.dat', 'wb')

- Once you have opened a file for binary writing, you call the pickle module’s dump function. Here is the general format of the dump method:

  pickle.dump(object, file)

- In the general format, object is a variable that references the object you want to pickle, and file is a variable that references a file object.
- After the function executes, the object referenced by object will be serialized and written to the file.

- (You can pickle just about any type of object, including lists, tuples, dictionaries, sets, strings, integers, and floating point numbers).
- You can save as many pickled objects as you want to a file.
- When you are finished, you call the file object’s close method to close the file.
- The following interactive session provides a simple demonstration of pickling a dictionary:

```
1  >>> import pickle Enter
2  >>> phonebook = {'Chris' : '555-1111', Enter
```

```
3                    'Katie' : '555-2222', Enter
4                    'Joanne' : '555-3333'} Enter
5  >>> output_file = open('phonebook.dat', 'wb') Enter
6  >>> pickle.dump(phonebook, output_file) Enter
7  >>> output_file.close() Enter
8  >>>
```

Let’s take a closer look at the program:

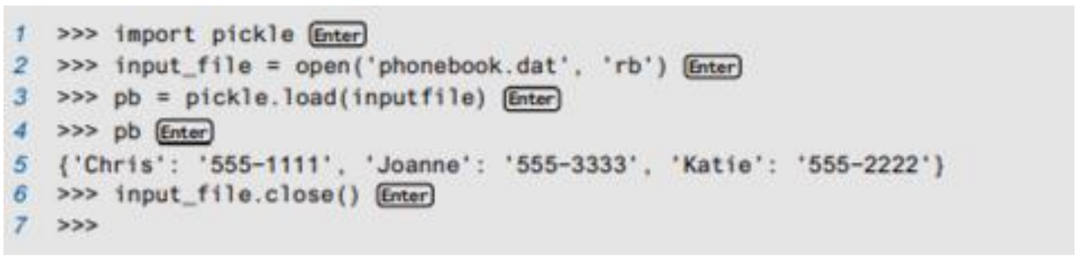
- Line 1 imports the pickle module.
- Lines 2 through 4 create a dictionary containing names (as keys) and phone numbers (as values).
- Line 5 opens a file named phonebook.dat for binary writing.
- Line 6 calls the pickle module’s dump function to serialize the phonebook dictionary and write it to the phonebook.dat file.
- Line 7 closes the phonebook.dat file.
- At some point, you will need to retrieve, or unpickle, the objects that you have pickled. Here are the steps that you perform:
- You open a file for binary reading.
- You call the pickle module’s load function to retrieve an object from the file and unpickle it.
- After you have unpickled all the objects that you want from the file, you close the file.
- To open a file for binary reading, you use 'rb' as the mode when you call the open function.
- For example, the following statement opens a file named mydata.dat for binary reading:

    inputfile = open('mydata.dat', 'rb')

- Once you have opened a file for binary reading, you call the pickle module’s load function.
- Here is the general format of a statement that calls the load function:

    object = 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 reference an object that was retrieved from the file and unpickled.
- You can unpickle as many objects as necessary from the file. (If you try to read past the end of the file, the load function will raise an EOFError exception.)
- When you are finished, you call the file object's close method to close the file.
- The following interactive session provides a simple demonstration of unpickling the phonebook dictionary that was pickled in the previous session:



```
1 >>> import pickle
2 >>> input_file = open('phonebook.dat', 'rb')
3 >>> pb = pickle.load(inputfile)
4 >>> pb
5 {'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
6 >>> input_file.close()
7 >>>
```

Let's take a closer look at the program:

- Line 1 imports the pickle module.
- Line 2 opens a file named phonebook.dat for binary reading.
- Line 3 calls the pickle module's load function to retrieve and unpickle an object from the phonebook.dat file. The resulting object is assigned to the pb variable.
- Line 4 displays the dictionary referenced by the pb variable. The output is shown in line 5.
- Line 6 closes the phonebook.dat file.

UNIT – V

Classes and Object - Oriented Programming

Classes and Object Oriented Programming: Procedural and Object-Oriented Programming – Classes – Working with instances – Techniques for designing classes - Inheritance: Introduction to Inheritance - Polymorphism - Getting MySQL for Python - # import MySQL for Python # - MySQLDb - Connecting with a Database

TOPIC-1: PROCEDURAL AND OBJECT-ORIENTED PROGRAMMING

- Procedural programming is a method of writing software.
- It is a programming practice centered on the procedures or actions that take place in a program.
- Object-oriented programming is centered on objects.
- Objects are created from abstract data types that encapsulate data and functions together.
- There are primarily two methods of programming in use today: procedural and object oriented.
- The earliest programming languages were procedural, meaning a program was made of one or more procedures.
- You can think of a procedure simply as a function that performs a specific task such as gathering input from the user, performing calculations, reading or writing files, displaying output, and so on.
- The programs that you have written so far have been procedural in nature.
- Typically, procedures operate on data items that are separate from the procedures.
- In a procedural program, the data items are commonly passed from one procedure to another.

Object Reusability

- In addition to solving the problems of code and data separation, the use of OOP has also been encouraged by the trend of object reusability.
- An object is not a stand-alone program, but is used by programs that need its services.

TOPIC-2 CLASSES

- A class is code that specifies the data attributes and methods for a particular type of object.
- The programmer determines the data attributes and methods that are necessary, and then creates a class.
- A class is code that specifies the data attributes and methods of a particular type of object.
- Think of a class as a “blueprint” from which objects may be created.
- It serves a similar purpose as the blueprint for a house.

Class Definitions

- To create a class, you write a class definition.
- A class definition is a set of statements that define a class’s methods and data attributes.
- Let’s look at a simple example.
- Suppose we are writing a program to simulate the tossing of a coin.
- In the program, we need to repeatedly toss the coin and each time determine whether it landed heads up or tails up.
- Taking an object-oriented approach, we will write a class named Coin that can perform the behaviours of the coin.

Example

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("Dr. S. Peerbasha", 35)
p1.myfunc()
```

Output

```
Hello my name is John
```

The __init__() Function

- All classes have a function called __init__(), which is always executed when the class is being initiated.
- Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created.

- The __init__() function is called automatically every time the class is being used to create a new object.

Example

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:

  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("Dr. S. Peerbasha", 35)
print(p1.name)
print(p1.age)
```

Output

```
Dr. S. Peerbasha

35
```

The self Parameter

- The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.
- It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class.

The __str__() Function

  - The __str__() function controls what should be returned when the class object is represented as a string.
  - If the __str__() function is not set, the string representation of the object is returned.

Example

The string representation of an object WITHOUT the __str__() function:

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

p1 = Person("Dr. S. Peerbasha", 35)
print(p1)
```

Output

```
<__main__.Person object at 0x146516613100>
```

Example

The string representation of an object WITH the __str__() function.

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

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("Dr. S. Peerbasha", 35)
print(p1)
```

Output

```
Dr. S. Peerbasha(35)
```

TOPIC-3: WORKING WITH INSTANCES

- Each instance of a class has its own set of data attributes.
- When a method uses the self parameter to create an attribute, the attribute belongs to the specific object that self reference.
- We call these attributes instance attributes because they belong to a specific instance of the class.
- It is possible to create many instances of the same class in a program.
- Each instance will then have its own set of attributes.
- For example, The below program creates three instances of the Coin class.
- Each instance has its own _ _sideup attribute.

```
 1  # This program imports the simulation module and
 2  # creates three instances of the Coin class.
 3
 4  import coin
 5
 6  def main():
 7      # Create three objects from the Coin class.
 8      coin1 = coin.Coin()
 9      coin2 = coin.Coin()
10      coin3 = coin.Coin()
11
12      # Display the side of each coin that is facing up.
13      print('I have three coins with these sides up:')
14      print(coin1.get_sideup())
15      print(coin2.get_sideup())
16      print(coin3.get_sideup())
17      print()
18
19      # Toss the coin.
20      print('I am tossing all three coins ...')
21      print()
22      coin1.toss()
23      coin2.toss()
24      coin3.toss()
25
26      # Display the side of each coin that is facing up.
27      print('Now here are the sides that are up:')
28      print(coin1.get_sideup())
29      print(coin2.get_sideup())
30      print(coin3.get_sideup())
31      print()
32
33  # Call the main function.
34  if __name__ == '__main__':
35      main()
```

Program Output

```
I have three coins with these sides up:
Heads
Heads
Heads
I am tossing all three coins ...

Now here are the sides that are up:
Tails
Tails
```

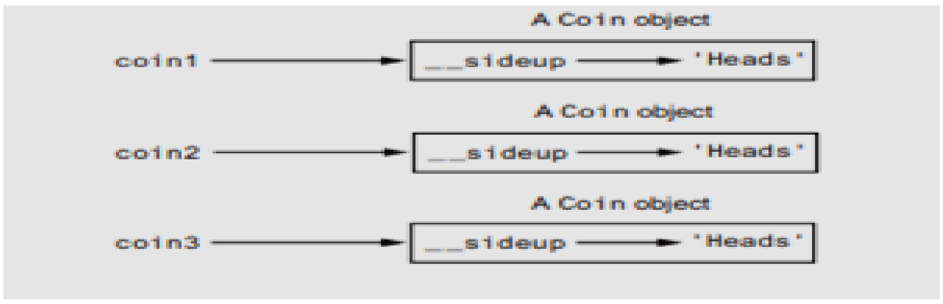
In lines 8 through 10, the following statements create three objects, each an instance of the Coin class:

```
coin1 = coin.Coin()
coin2 = coin.Coin()
```

coin3 = coin.Coin()

- The coin1, coin2, and coin3 variables reference the three objects after these statements execute.
- Notice each object has its own _ _sideup attribute. Lines 14 through 16 display the values returned from each object's get_sideup method.

The coin1, coin2, and coin3 variables reference three Coin objects



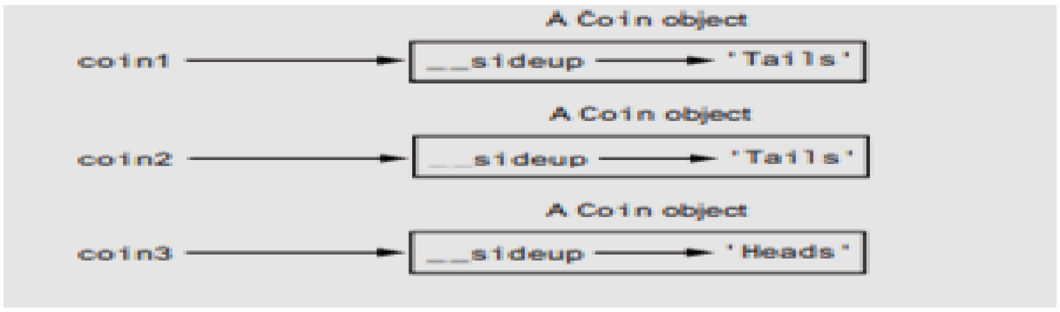
Then, the statements in lines 22 through 24 call each object's toss method:

coin1.toss()

coin2.toss()

coin3.toss()

The objects after the toss method



TOPIC-4 TECHNIQUES FOR DESIGNING CLASSES

- When designing a class, it is often helpful to draw a UML diagram.
- UML stands for Unified Modeling Language.
- It provides a set of standard diagrams for graphically depicting object-oriented systems.
- Below image shows the general layout of a UML diagram for a class.
- Notice the diagram is a box that is divided into three sections.
- The top section is where you write the name of the class.
- The middle section holds a list of the class's data attributes.
- The bottom section holds a list of the class's methods

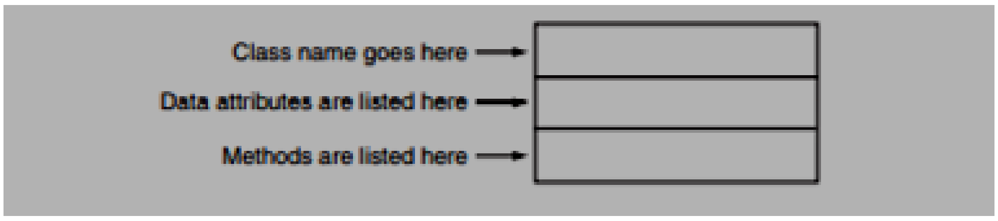


Diagram for the Coin Class

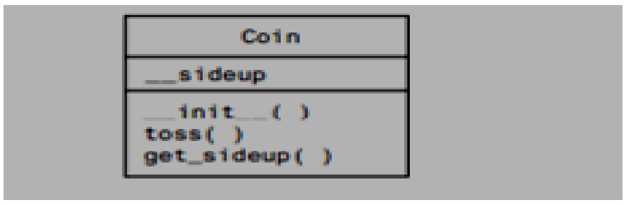
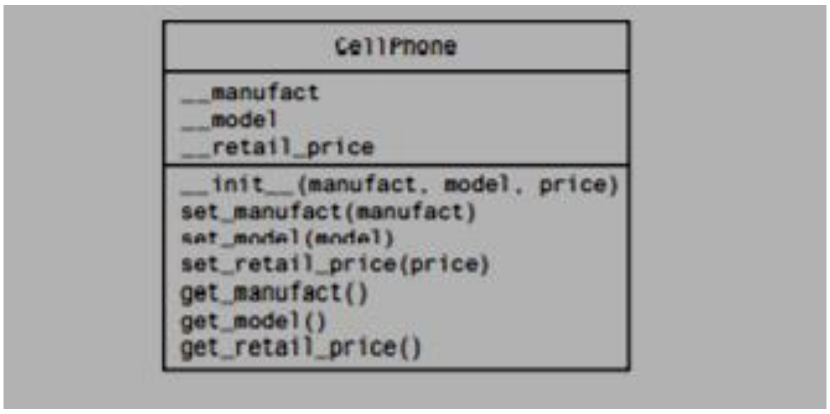


Diagram for the CellPhone class



- When developing an object-oriented program, one of your first tasks is to identify the classes that you will need to create.

- Typically, your goal is to identify the different types of real-world objects that are present in the problem, then create classes for those types of objects within your application.
- Over the years, software professionals have developed numerous techniques for finding the classes in a given problem.
- One simple and popular technique involves the following steps:
    - Get a written description of the problem domain.
    - Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class.
    - Refine the list to include only the classes that are relevant to the problem

TOPIC-5 INHERITANCE

- Inheritance allows a new class to extend an existing class.
- The new class inherits the members of the class it extends.
- Inheritance is an important aspect of the object-oriented paradigm.
- Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class.
- A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.
- In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name.
- Consider the following syntax to inherit a base class into the derived class.

Syntax:

```
class derived-class(base class):
<class-suite>
```

- A class can inherit multiple classes by mentioning all of them inside the bracket.

Consider the following syntax.

class derive-class(<base class 1>, <base class 2>, ... <base class n>):

<class - suite>

Example

```
class Animal:
    def speak(self):
        print("Animal Speaking")
```

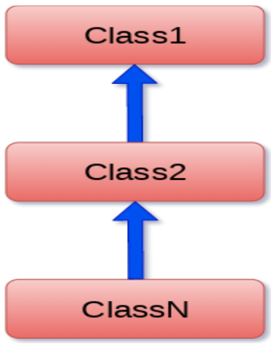
```
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output:-

```
dog barking
Animal Speaking
```

MULTI-LEVEL INHERITANCE

- Multi-Level inheritance is possible in python like other object-oriented languages.
- Multi-level inheritance is archived when a derived class inherits another derived class.
- There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.



Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
```

Example

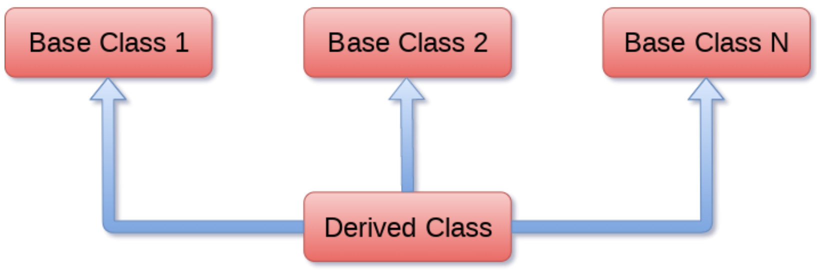
```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output

```
dog barking
Animal Speaking
Eating bread...
```

MULTIPLE INHERITANCE

Python provides us the flexibility to inherit multiple base classes in the child class.



The syntax to perform multiple inheritance is given below.

Syntax

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
. . .
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ...... BaseN):
    <class-suite>
```

Example

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

Output:

```
30
200
0.5
```

METHOD OVERRIDING

- We can provide some specific implementation of the parent class method in our child class.
- When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.
- We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Example

```
class Animal:
    def speak(self):
        print("speaking")
class Dog(Animal):
    def speak(self):
        print("Barking")
d = Dog()
d.speak()
```

Output

```
Barking
```

DATA ABSTRACTION IN PYTHON

- Abstraction is an important aspect of object-oriented programming.
- In python, we can also perform data hiding by adding the double underscore (___) as a prefix to the attribute which is to be hidden.
- After this, the attribute will not be visible outside of the class through the object.

Example

```
class Employee:
    __count = 0;
    def __init__(self):
        Employee.__count = Employee.__count+1
    def display(self):
        print("The number of employees",Employee.__count)
emp = Employee()
emp2 = Employee()
```

```
try:
    print(emp.__count)
finally:
    emp.display()
```

Output

```
The number of employees 2
AttributeError: 'Employee' object has no attribute '__count'
```

TOPIC-5: POLYMORPHISM

- Polymorphism refers to having multiple forms.
- Polymorphism is a programming term that refers to the use of the same function name, but with different signatures, for multiple types.

Two essential ingredients of polymorphic behaviour:

- The ability to define a method in a superclass, then define a method with the same name in a subclass.
- When a subclass method has the same name as a super class method, it is often said that the subclass method overrides the super class method.

The ability to call the correct version of an overridden method, depending on the type of object that is used to call it.

If a subclass object is used to call an overridden method, then the subclass's version of the method is the one that will execute.

If a super class object is used to call an overridden method, then the super class's version of the method is the one that will execute

Example of in-built polymorphic functions:

```
# Python program for demonstrating the in-built poly-morphic functions
# len() function is used for a string
print (len("Javatpoint"))
# len() function is used for a list
print (len([110, 210, 130, 321]))
```

Output:

```
10
4
```

Polymorphism with Inheritance:

- Polymorphism allows us to define methods in Python that are the same as methods in the parent classes.
- In inheritance, the methods of the parent class are passed to the child class.
- It is possible to change a method that a child class has inherited from its parent class.
- This is especially useful when the method that was inherited from the parent doesn't fit the child's class.
- We re-implement such methods in the child classes. This is Method Overriding.

Example

```
class Birds:
    def intro1(self):
        print("There are multiple types of birds in the world.")
    def flight1(self):
        print("Many of these birds can fly but some cannot.")

class sparrow1(Birds):
    def flight1(self):
        print("Sparrows are the bird which can fly.")

class ostrich1(Birds):
    def flight1(self):
        print("Ostriches are the birds which cannot fly.")

obj_birds = Birds()
obj_spr1 = sparrow1()
obj_ost1 = ostrich1()

obj_birds.intro1()
obj_birds.flight1()

obj_spr1.intro1()
obj_spr1.flight1()
```

```
  obj_ost1.intro1()
obj_ost1.flight1()
```

Output:

```
There are multiple types of birds in the world.
Many of these birds can fly but some cannot.
There are multiple types of birds in the world.
Sparrows are the bird which can fly.
There are multiple types of birds in the world.
Ostriches are the birds which cannot fly.
```

TOPIC-6 GETTING MYSQL FOR PYTHON

Install mysql.connector

- To connect the python application with the MySQL database, we must import the mysql.connector module in the program.
- The mysql.connector is not a built-in module that comes with the python installation. We need to install it to get it working.

Execute the following command to install it using pip installer.

```
>  python -m pip install mysql-connector
```

There are the following steps to connect a python application to our database.

1. **Import mysql.connector module**
2. **Create the connection object.**
3. **Create the cursor object**
4. **Execute the query**
5. **Import MySQL Connector**

To test if the installation was successful, or if you already have "MySQL Connector" installed, create a Python page with the following content:

demo_mysql.py:

```
import mysql.connector
```

If the above code was executed with no errors, "MySQL Connector" is installed and ready to be used.

Create Connection

Start by creating a connection to the database.

Use the username and password from your MySQL database:

```
importmysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword"
)

print(mydb)
```

Now you can start querying the database using SQL statements.

Creating a cursor object

- The cursor object can be defined as an abstraction specified in the Python DB-API 2.0.
- It facilitates us to have multiple separate working environments through the same connection to the database.
- We can create the cursor object by calling the 'cursor' function of the connection object.
- The cursor object is an important aspect of executing queries to the databases.

The syntax to create the cursor object is given below.

```
<my_cur>  = conn.cursor()
```

Example

import mysql.connector

#Create the connection object

myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google", data base = "mydb")

#printing the connection object

print(myconn)

#creating the cursor object

cur = myconn.cursor()

print(cur)

Output:

```
<mysql.connector.connection.MySQLConnection object at 0x7faa17a15748>
MySQLCursor: (Nothing executed yet)
```

Creating the new database

The new database can be created by using the following SQL query.

```
>  create database <database-name>
```

Example

```
import mysql.connector
#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google")
#creating the cursor object
cur = myconn.cursor()
try:
    #creating a new database
    cur.execute("create database PythonDB2")
#getting the list of all the databases which will now include the new database PythonDB
    dbs = cur.execute("show databases")
except:
    myconn.rollback()
for x in cur:
        print(x)
myconn.close()
```

Output:

```
('EmployeeDB',)
('PythonDB',)
('Test',)
('TestDB',)
('anshika',)
```

TOPIC-7 MYSQLDb

MySQLdb is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and is built on top of the MySQL C API.

Packages to Install

mysql-connector-python

mysql-python

If using anaconda

conda install -c anaconda mysql-python

conda install -c anaconda mysql-connector-python

else

pip install MySQL-python

pip install MySQL-python-connector

Import-Package

import MYSQLdb

How to connect to a **remote MySQL database using python?**

- **connect():** This method is used for creating a connection to our database it has four arguments:

1. Server Name
2. Database User Name
3. Database Password
4. Database Name

- **cursor():** This method creates a cursor object that is capable of executing SQL queries on the database.
- **execute():** This method is used for executing SQL queries on the database. It takes a sql query(as string) as an argument.
- **fetchone():** This method retrieves the next row of a query result set and returns a single sequence, or None if no more rows are available.
- **lose() :** This method close the database connection.

```
# Module For Connecting To MySQL database
import MySQLdb
# Function for connecting to MySQL database
def mysqlconnect():
 #Trying to connect
    try:
        db_connection= MySQLdb.connect
        ("Hostname","dbusername","password","dbname")
 # If connection is not successful
   except:
       print("Can't connect to database")
       return 0
 # If Connection Is Successful
   print("Connected")

 # Making Cursor Object For Query Execution
   cursor=db_connection.cursor()

 # Executing Query
   cursor.execute("SELECT CURDATE();")

 # Above Query Gives Us The Current Date
 # Fetching Data
   m = cursor.fetchone()

 # Printing Result Of Above
   print("Today's Date Is ",m[0])

 # Closing Database Connection
   db_connection.close()

# Function Call For Connecting To Our Database
mysqlconnect()
```

```
Connected
Today's Date Is  2017-11-14
```

TOPIC-8 CONNECTING WITH A MYSQL DATABASE

- The connect() constructor creates a connection to the MySQL server and returns a MySQLConnection object.
- The following example shows how to connect to the MySQL server:

```
import mysql.connector


cnx = mysql.connector.connect(user='scott', password='password',
                              host='127.0.0.1',
                              database='employees')
cnx.close()
```

It is also possible to create connection objects using **connection.MySQLConnection()** class:

```
from mysql.connector import (connection)
```

```
cnx = connection.MySQLConnection(user='scott', password='password',
                                 host='127.0.0.1',
                                 database='employees')
cnx.close()
```

Both forms (either using the connect() constructor or the class directly) are valid and functionally equal, but using connect() is preferred and used by most examples in this manual.

IMPORTANT QUESTIONS:-

UNIT-III
Section-B:-

1. Explain the Value-Returning Functions with necessary examples.
2. How will you generate random numbers?
3. Write a note on Math Module
4. How will you process the files using loops?

Section-C:-

1. Develop a python program that implements the concept of Files, Input and Output.
2. Develop a python program to make use of the Exception concept.

UNIT-IV
Section-B:-

1. Explain the concept of List Slicing
2. How will you find the items in the list with the 'in' operator?
3. Write a python program to implements the various list methods.
4. Write a note on Two-Dimensional lists.

Section-C:-

1. Create a tuple and perform concatenation, repetition, membership, access items and slicing operations.
2. Write a python program to sort (Ascending and Descending) a Dictionary by value.

UNIT-V
Section-B:-

1. Explain the concept of Inheritance with necessary example program.
2. Explain the various techniques for designing classes
3. Write short note on Polymorphism
4. Explain the procedural and object-oriented programming in detail.

Section-C:-

1. Prepare a Students Mark list using Class.
2. Develop a python program to find the area of a circle using class and objects.
3. Perform various database operations (Create, Insert, Delete, Update) using MQSQL.

There are no secrets to success. It is the result of preparation, hard work and learning from the failure.