Annamalai University

Department of Computer Science and Engineering

BE (CSE) - IV Semester

(A & B-Batch)

18CSPC405 – Python Programming

Lecture Notes (Unit 1- 5)

Course Teachers

Dr. S. Pasupathy (A-Batch)

Dr. M. Balasubramanian (B-Batch)

Associate Professors

Faculty of Engineering and Technology
Annamalai University, AnnamalaiNagar-608002

Table of Contents

18CSPC405	PYTHON PROGRAMMING	L	T	P	C
		3	0	0	3

Course Objectives:

- To understand and be able to use the basic programming principles such as data types, variable, conditionals, loops, recursion and function calls.
- To learn how to use basic data structures such as List, Dictionary and be able to manipulate text files and images.
- To understand the process and will acquire skills necessary to effectively attempt a programming problem and implement it with a specific programming language – Python.

UNIT - I Introduction

Elementary Programming, Selections and Loops: History of Python – Getting Started with Python – Programming Style – Writing a Simple Program – Reading Input from the Console – Identifiers – Variables, Assignment Statements, and Expressions – Simultaneous Assignments – Named Constants – Numeric Data Types and Operators – Type Conversions and Rounding–Introduction – Boolean Types, Values, and Expressions – if Statements – Two-Way if-else Statements – Nested if and Multi-Way if-elif-else Statements – Logical Operators – Conditional Expressions – Operator Precedence and Associativity – Detecting the Location of an Object Case Study: Computing Body Mass Index – The while Loop – The for Loop – Nested Loops – Keywords break and continue – Case Studies: Displaying Prime Numbers and Random Walk.

UNIT - II Python Function

Mathematical Functions, Strings and User Defined Functions: Simple and Mathematical Python Built–in Functions – Strings and Characters – Introduction to Objects and Methods – Formatting Numbers and Strings – Drawing Various Shapes
– Drawing with Colors and Fonts – Defining a Function – Calling a Function – Functions with/without Return Values – Positional and Keyword Arguments – Passing Arguments by Reference Values – Modularizing Code – The Scope of Variables – Default Arguments – Returning Multiple Values –Function Abstraction and Stepwise Refinement – Case Study: Generating Random ASCII Characters.

UNIT - III Class and Object

Introduction to Object – Oriented Programming – Basic principles of Object – Oriented Programming in Python – Class definition, Inheritance, Composition, Operator Overloading and Object creation – Python special Unit – Python Object System – Object representation, Attribute binding, Memory Management, and Special properties of classes including properties, Slots and Private attributes.

UNIT - IV Files and Exception Handling

Files, Exception Handling and Network Programming: Introduction –Text Input and Output – File Dialogs – –Exception Handling – Raising Exceptions – Processing Exceptions Using Exception Objects – Defining Custom Exception Classes – Binary IO Using Pickling – Case Studies: Counting Each Letter in a File and Retrieving Data from the Web–Client Server Architecture–sockets – Creating and executing TCP and UDP Client Server Unit – Twisted Framework – FTP – Usenets – Newsgroup - Emails – SMTP – POP3.

UNIT - V Database and GUI

Database and GUI Programming: DBM database – SQL database – GUI Programming using Tkinter: Introduction – Getting Started with Tkinter – Processing Events – The Widget Classes – Canvas – The Geometry Managers – Displaying Images – Menus – Popup Menus – Mouse, Key Events, and Bindings – List boxes – Animations – Scrollbars – Standard Dialog Boxes–Grids.

TEXT BOOKS:

1. Mark Lutz, "Learning Python, Powerful OOPs", O'Reilly, 2011.
2. Guttag, John, "Introduction to Computation and Programming Using Python", MIT Press, 2013.

REFERENCES:

1. Jennifer Campbell, Paul Gries, Jason montajo, Greg Wilson, "Practical Programming An Introduction To Computer Science Using Python" The Pragmatic Bookshelf, 2009.
2. Wesley J Chun "Core Python Applications Programming", Prentice Hall,
3. 2012.
4. Jeeva Jose, "Taming Python by Programming", Khanna Publishing
5. House,1st edition,2017.
6. J.Jose, "Introduction to Computing and Problem Solving with Python",
7. Khanna Publications, 1st edition, 2015.
8. Reema Thareja, "Python Programming", Pearson, 1st edition, 2017.

Course Outcomes:

At the end of this course, the students will be able to

1. Understand basic concepts of Conditional and Looping Statements in python programming.
2. Solve large program in an easy way using Modules concepts.
3. Apply the concepts of Object Oriented programming including encapsulation, inheritance and polymorphism as used in Python.
4. Simulate the commonly used operations in file system and able to develop application program to communicate from one end system to another end.
5. Develop menu driven program using GUI interface and to gain knowledge about how to store and retrieve data.

Mapping of Course Outcomes with Programme Outcomes (Ratings: 1-Low, 2-Moderate and 3-High)

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	1	0	1	0	0	0	0	0	0	0	0	0
CO2	2	1	0	0	1	0	0	0	0	0	0	0
CO3	1	2	0	0	1	0	0	0	0	0	0	0
CO4	1	2	2	1	0	0	0	0	0	0	0	0
CO5	1	2	3	1	2	0	0	0	1	0	0	2

18CSPC405 – PYTHON PROGRAMMING
Unit I – INTRODUCTION

INTRODUCTION

Python is a widely used general-purpose, high level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.
Python is a programming language that lets you work quickly and integrate systems more efficiently.

Reading input from console

It's to take input from the user and hence manipulate it or simply display it. input() function is used to take input from the user.

```
# Python program to illustrate
# getting input from user
name = input("Enter your name: ")

# user entered the name 'harssh'
print("hello", name)
```

Output:

```
hello harssh
```

 # program to get input from user

```
# accepting integer from the user
num1 = int(input("Enter num1: "))
num2 = int(input("Enter num2: "))
num3 = num1 * num2
print("Product is: ", num3)
```

Output:

```
Enter num1: 8 Enter num2: 6 ('Product is: ', 48)
```

INTERACTIVE MODE PROGRAMMING

Invoking the interpreter without passing a script file as a parameter brings up the following prompt.

```
$ python
Python 2.4.3 (#1, Nov 11 2010, 13:34:43)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Type the following text at the Python prompt and press the Enter −

```
>>> print "Hello, Python!"
```

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!");**. However in Python version 2.4.3, this produces the following result −

Hello, Python!

SCRIPT MODE PROGRAMMING

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have extension **.py**. Type the following source code in a test.py file −

```
print "Hello, Python!"
```

We assume that you have Python interpreter set in PATH variable. Now, try to run this program as follows −

```
$ python test.py
```

This produces the following result −

Hello, Python!

Let us try another way to execute a Python script. Here is the modified test.py file −

```
#!/usr/bin/python

print "Hello, Python!"
```

We assume that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows −

```
$ chmod +x test.py     # This is to make file executable
$./test.py
```

This produces the following result −

Hello, Python!

PYTHON IDENTIFIERS

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

RESERVED WORDS

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	Exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example −

```
if True:
   print "True"
else:
   print "False"
```

However, the following block generates an error −

```
if True:
print "Answer"
print "True"
else:
print "Answer"
print "False"
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block. The following example has various statement blocks

PYTHON - VARIABLE TYPES

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

```
#!/usr/bin/python

counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string

print counter
print miles
print name
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively. This produces the following result −

```
100
```

1000.0
John

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example −

a = b = c = 1

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

a,b,c = 1,2,"john"

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

STANDARD DATA TYPES

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

Assignment is fundamental to Python; it is how the objects created by an expression are preserved. We'll look at the basic assignment statement, plus the augmented assignment statement. Later, in Multiple Assignment Statement, we'll look at multiple assignment.

Basic Assignment

We create and change variables primarily with the *assignment* statement. This statement provides an expression and a variable name which will be used to label the value of the expression.

variable = expression

Here's a short script that contains some examples of assignment statements.

Example example3.py

```
#!/usr/bin/env python
```

```
# Computer the value of a block of stock
shares= 150
price= 3 + 5.0/8.0
value= shares * price
print value
```

SIMULTANEOUS ASSIGNMENT

There is an alternative form of the assignment statement that allows us to calculate several values all at the same time. It looks like this:

<var>, <var>, ..., <var> = <expr>, <expr>, ..., <expr>

This is called simultaneous assignment. Semantically, this tells Python to evaluate all the expressions on the right-hand side and then assign these values to the corresponding variables named on the left-hand side. Here's an example.
Here sum would get the sum of x and y and diff would get the difference.
This form of assignment seems strange at first, but it can prove remarkably useful. Here's an example. Suppose you have two variables x and y and you want to swap the values. That is, you want the value currently stored in x to be in y and the value that is currently in y to be stored in x. At first, you might think this could be done with two simple assignments.

CONSTANTS

A constant is a type of variable that holds values, which cannot be changed. In reality, we rarely use constants in Python. Constants are usually declared and assigned on a different module/file. Example:

```
#Declare constants in a separate file called constant.py
PI = 3.14
GRAVITY = 9.8
```

Then, they are imported to the main file.

```
#inside main.py we import the constants
import constant
print(constant.PI)
print(constant.GRAVITY)
```

TYPES OF OPERATOR

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators

- Membership Operators
- Identity Operators

Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) −	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Python Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Operator	Description	Example

==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Assignment Operators

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a

-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in the binary format their values will be 0011 1100 and 0000 1101 respectively. Following table lists out the bitwise operators supported by Python language with an example each in those, we use the above two variables (a and b) as operands −

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Python Logical Operators

Operator	Description	Example
and Logical	If both the operands are true then condition becomes	(a and b)

AND	true.	is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

TYPE CONVERSION IN PYTHON

Python defines type conversion functions to directly convert one data type to another which is useful in day to day and competitive programming. This article is aimed at providing the information about certain conversion functions.

1. int(a,base) : This function converts **any data type to integer**. 'Base' specifies the **base in which string is** if data type is string.
2. float() : This function is used to convert **any data type to a floating point number**
3. ord() : This function is used to convert a **character to integer.**
4. hex() : This function is to convert **integer to hexadecimal string**.
5. oct() : This function is to convert **integer to octal string**.

BOOL() IN PYTHON

The **bool()** method is used to return or convert a value to a Boolean value i.e., True or False, using the standard truth testing procedure.

Syntax:

```
bool([x])
```

The bool() method in general takes only one parameter(here x), on which the standard truth testing procedure can be applied. **If no parameter is passed, then by default it returns False**. So, passing a parameter is optional. It can return one of the two values.

- It returns True if the parameter or value passed is True.
- It returns False if the parameter or value passed is False.

Here are few cases, in which Python's bool() method returns false. Except these all other values return True.

- If a False value is passed.
- If None is passed.
- If an empty sequence is passed, such as (), [], '', etc
- If Zero is passed in any numeric type, such as 0, 0.0 etc
- If an empty mapping is passed, such as {}.
- If Objects of Classes having __bool__() or __len()__ method, returning 0 or False

LOOP STATEMENT

Python programming language provides following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	nested loops You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Click the following links to check their detail.

Let us go through the loop control statements briefly

Sr.No.	Control Statement & Description
1	break statement Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	pass statement The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

There are following logical operators supported by Python language. Assume variable **a** holds True and variable **b** holds False then

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is False.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is True.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is True.

Assignment Operators:

Assignment operators are used in Python to assign values to variables.

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c
		= c / ac /= a is equivalent to c = c / a

%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Logical Operators:

Logical operators are and, or, not operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Bitwise Operators:

Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

PRECEDENCE OF PYTHON OPERATORS

The combination of values, variables, operators and function calls is termed as an expression.

Python interpreter can evaluate a valid expression.

For example:

```
>>> 5 - 7
-2
```

Here 5 - 7 is an expression. There can be more than one operator in an expression.

To evaluate these type of expressions there is a rule of precedence in Python. It guides the order in which operation are carried out.

Operator Precedence:

When an expression contains **more than one operator, the order of evaluation** depends on the order of operations.

Computing Body Mass Index In Python

```
#Define the constants
METER   = 100

#Read the inputs from user
height  = float(input("Enter your height in Centimeters: "))
weight  = float(input("Enter your weight in Kg: "))

temp    = height / METER
#Calculate the BMI
bmi = weight / (temp*temp)

#Display the result
print("Your Body Mass Index is: ","%d"%(bmi))
```

Output

```
C:\Python\programs>python program.py
Enter your height in Centimeters: 105
Enter your weight in Kg: 69
Your Body Mass Index is: 62

C:\Python\programs>
```

THE ELSE AND ELIF CLAUSES

Now you know how to use an if statement to conditionally execute a single statement or a block of several statements. It's time to find out what else you can do.

Sometimes, you want to evaluate a condition and take one path if it is true but specify an alternative path if it is not. This is accomplished with an else clause:

```
if <expr>:
    <statement(s)>
else:
    <statement(s)>
```

If <expr> is true, the first suite is executed, and the second is skipped. If <expr> is false, the first suite is skipped and the second is executed. Either way, execution then resumes after the second suite. Both suites are defined by indentation, as described above.

ONE-LINE IF STATEMENTS

It is customary to write if <expr> on one line and <statement> indented on the following line like this:

```
if <expr>:
```

```
    <statement>
```
But it is permissible to write an entire if statement on one line. The following is functionally equivalent to the example above:

```
if <expr>: <statement>
```
There can even be more than one <statement> on the same line, separated by semicolons:

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```
But what does this mean? There are two possible interpretations:

1. If <expr> is true, execute <statement_1>.

   Then, execute <statement_2> ... <statement_n> unconditionally, irrespective of whether <expr> is true or not.

2. If <expr> is true, execute all of <statement_1> ... <statement_n>. Otherwise, don't execute any of them.

Python takes the latter interpretation. The semicolon separating the <statements> has higher precedence than the colon following <expr>—in computer lingo, the semicolon is said to bind more tightly than the colon. Thus, the <statements> are treated as a suite, and either all of them are executed, or none of them are:

BREAK & CONTINUE STATEMENTS

Python provides **break** and **continue** statements to handle such situations and to have good control on your loop.

It will discuss the *break, continue* and *pass* statements available in Python.

The *break* Statement:

The **break** statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

Example:

```
#!/usr/bin/python

for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print 'Current Letter :', letter

var = 10                    # Second Example
```

```
while var > 0:
   print 'Current variable value :', var
   var = var -1
   if var == 5:
      break

print "Good bye!"
```

This will produce the following result:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

The *continue* Statement:

The **continue** statement in Python returns the control to the beginning of the while loop. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The **continue** statement can be used in both *while* and *for* loops.

Example:

```
#!/usr/bin/python

for letter in 'Python':     # First Example
   if letter == 'h':
      continue
   print 'Current Letter :', letter

var = 10                    # Second Example
while var > 0:
   var = var -1
   if var == 5:
      continue
   print 'Current variable value :', var
print "Good bye!"
```

This will produce following result:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
```

```
Current Letter : n
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Good bye!
```

The *else* Statement Used with Loops

Python supports to have an **else** statement associated with a loop statements.

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

Example:

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.

```
#!/usr/bin/python

for num in range(10,20):  #to iterate between 10 to 20
  for i in range(2,num): #to iterate on the factors of the number
    if num%i == 0:      #to determine the first factor
      j=num/i #to calculate the second factor
      print '%d equals %d * %d' % (num,i,j)
      break #to move to the next number, the #first FOR
  else:       # else part of the loop
    print num, 'is a prime number'
```

This will produce following result:

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

Similar way you can use **else** statement with **while** loop.

The *pass* Statement:

The **pass** statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

Example:

```
#!/usr/bin/python

for letter in 'Python':
   if letter == 'h':
      pass
      print 'This is pass block'
   print 'Current Letter :', letter

print "Good bye!"
```

This will produce following result:

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

DISPLAY PRIME NUMBERS

```
# Python program to display all the prime numbers within an interval

lower = 900
upper = 1000

print("Prime numbers between", lower, "and", upper, "are:")

for num in range(lower, upper + 1):
   # all prime numbers are greater than 1
   if num > 1:
      for i in range(2, num):
         if (num % i) == 0:
            break
      else:
         print(num)
```

Output

```
Prime numbers between 900 and 1000 are:
907
911
919
929
937
941
947
953
967
971
977
983
991
997
```

One-dimensional random walk An elementary example of a random walk is the random walk on the integer number line, which starts at 0 and at each step moves +1 or ?1 with equal probability.

So lets try to implement the 1-D random walk in python.

```
filter_none
edit
play_arrow
brightness_4
# Python code for 1-D random walk.
import random
import numpy as np
import matplotlib.pyplot as plt

# Probability to move up or down
prob = [0.05, 0.95]

# statically defining the starting position
start = 2
positions = [start]
  # creating the random points
rr = np.random.random(1000)
downp = rr < prob[0]
upp = rr > prob[1]

for idownp, iupp in zip(downp, upp):
    down = idownp and positions[-1] > 1
    up = iupp and positions[-1] < 4
    positions.append(positions[-1] - down + up)
```

```
# plotting down the graph of the random walk in 1D
plt.plot(positions)
plt.show()
```

Prime Numbers

A prime number is an integer number greater than 1 whose only factors are 1 and itself. The first few prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23 and 29.

To check whether a given number (num) is a prime number, first divide it by 2. If the result is an integer number (i.e., num is divisible by 2 or remainder of num/2 is zero or num % 2 is 0) then num is not a prime number. If not, try to divide it by prime numbers 3, 5, 7, 11, … , up to num-1 or divide it by 3, 4, 5, …. , up to num-1. If num is not divisible up to num-1 then num is a prime number.

a) Coding: (using for statement)

```
# To display all the prime numbers within a given interval [Lower, Upper]

lower = int(input("Enter Lower Limit: "))
upper = int(input("Enter Upper Limit: "))

print("Prime numbers between", lower, "and", upper, "are:")

for num in range(lower, upper + 1):
    if num > 1:
        for i in range(2, num):
            if (num % i) == 0:
                break
        else:
            print(num)
```

Result:

```
Enter Lower Limit: 20
Enter Upper Limit: 50
Prime numbers between 20 and 50 are:
23
29
31
37
41
43
47
>>>
```

b) Coding: (using while statement)

Python program to display all the prime numbers within a given interval [Lower, Upper] using while statement

```
lower = int(input("Enter Lower Limit: "))
upper = int(input("Enter Upper Limit: "))

print("Prime numbers between", lower, "and", upper, "are:")

num=lower
while(num<=upper):
    if num > 1:
        for i in range(2, num):
            if (num % i) == 0:
                break
        else:
            print(num)
    num=num+1
```

Result:

```
Enter Lower Limit: 50
Enter Upper Limit: 100
Prime numbers between 50 and 100 are:
53
59
61
67
71
73
79
83
89
97
>>>
```

SOLVING QUADRATIC EQUATION

In algebra, a quadratic equation is any equation of the form $ax^2 + bx + c = 0$, where *x* is unknown and *a*, *b*, and *c* are known numbers. The numbers *a*, *b*, and *c* are the coefficients of the equation.

Examples: $x^2 + 5x + 6 = 0$, $4x^2 + 5x + 8 = 0$.

The values of x that satisfy the equation are called solutions or roots of the equation. A quadratic equation has always two roots. It can be found by computing $d = b^2 - 4ac$. The value of d is zero or +ve or -ve.

i) If d = 0 then then the roots are real and equal. They are root1 = -b/(2a) and root2 = -b/(2a).

ii) If d > 0 then the roots are real and different. They are root1=-b+√d/(2a) and root2=-b-√d/(2a).

iii) If d < 0 then the roots are complex or imaginary. Roots are of the form p + iq and p - iq, where p is the real part of the root and q is the imaginary part of the root. The real part of the root is p = -b/(2a) and the imaginary part is q = √-d/(2a).

Coding:

```
# Finding the roots of quadratic equation ax^2 + bx + c = 0

# import math and complex math modules

import math,cmath

a,b,c= (input("Enter a, b and c: ")).split()
a,b,c =[int(a),int(b),int(c)]

d = (b**2) - (4*a*c)

if d==0:
    sol1 = -b/(2*a)
    sol2 = -b/(2*a)
    print("Roots are real and equal")

elif d>0:
    sol1 = (-b-math.sqrt(d))/(2*a)
    sol2 = (-b+math.sqrt(d))/(2*a)
    print("Roots are real and different")

elif d<0:
    sol1 = (-b-cmath.sqrt(d))/(2*a)
    sol2 = (-b+cmath.sqrt(d))/(2*a)
    print("Roots are imaginary")

print('{0} and {1}'.format(sol1,sol2))
```

Result:

```
Enter a, b and c: 1 4 4
Roots are real and equal
-2.0 and -2.0
>>>
Enter a, b and c: 1 5 6
Roots are real and different
-3.0 and -2.0
>>>
Enter a, b and c: 1 2 3
Roots are imaginary
(-1-1.4142135623730951j) and (-1+1.4142135623730951j)
>>>
```

18CSPC405 – PYTHON PROGRAMMING

Unit II - Python Functions

PYTHON MATHEMATICAL FUNCTIONS

List of Functions in Python Math Module

Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
copysign(x, y)	Returns x with the sign of y
fabs(x)	Returns the absolute value of x
factorial(x)	Returns the factorial of x
floor(x)	Returns the largest integer less than or equal to x
fmod(x, y)	Returns the remainder when x is divided by y
frexp(x)	Returns the mantissa and exponent of x as the pair (m, e)
fsum(iterable)	Returns an accurate floating point sum of values in the iterable
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number)
isinf(x)	Returns True if x is a positive or negative infinity
isnan(x)	Returns True if x is a NaN
ldexp(x, i)	Returns x * (2**i)
modf(x)	Returns the fractional and integer parts of x
trunc(x)	Returns the truncated integer value of x
exp(x)	Returns e**x

STRING AND USER DEFINED FUNCTIONS

User defined function. In **Python**, a **user-defined function's** declaration begins with the keyword def and followed by the **function** name. The **function** may take arguments(s) as input within the opening and closing parentheses, just after the **function** name followed by a colon.In Python, a user-defined function's declaration begins with the keyword def and followed by the function name.

- The function may take arguments(s) as input within the opening and closing parentheses, just after the function name followed by a colon.
- After defining the function name and arguments(s) a block of program statement(s) start at the next line and these statement(s) must be indented.

Advantages of user defined functions:

- ❖ The programmer can write their own functions which are known as user defined function. These functions can create by using def keyword.

Example:

add(), sub()

ELEMENTS OF USER DEFINED FUNCTIONS

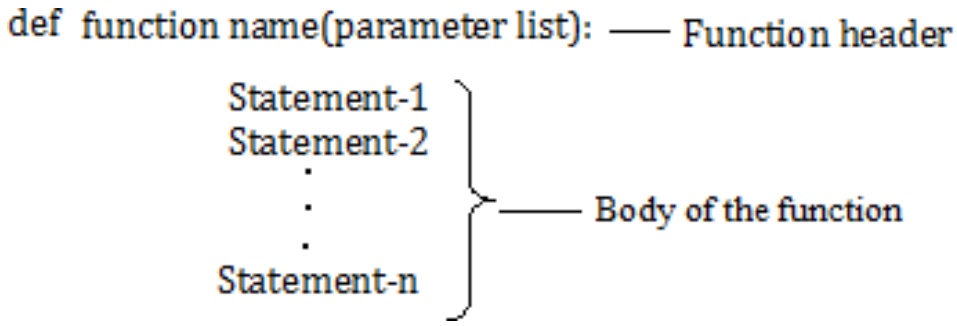
There are two elements in user defined function.

1. function definition
2. function call

FUNCTION DEFINITION

A **function definition specifies** the name of a new function and the sequence of statements that execute when the function is called.

Syntax:



- ❖ def is a keyword that indicates that this is a function definition.
- ❖ The rules for function names are the same as for variable names
- ❖ The first line of the function definition is called the header; the rest is called body of function.
- ❖ The header has to end with a colon and the body has to be indented. The function contains any number of statements.
- ❖ parameter list contains list of values used inside the function.

example:

```
def add():
    a=eval(input("enter  a  value"))
    b=eval(input("enter  a  value"))
    c=a+b
    print(c)
```

The `math` module is a standard module in Python and is always available. To use mathematical functions under this module, you have to import the module using `import math`.

It gives access to the underlying C library functions.

List of Functions in Python Math Module

Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
copysign(x, y)	Returns x with the sign of y
fabs(x)	Returns the absolute value of x
factorial(x)	Returns the factorial of x
floor(x)	Returns the largest integer less than or equal to x
fmod(x, y)	Returns the remainder when x is divided by y
frexp(x)	Returns the mantissa and exponent of x as the pair (m, e)
fsum(iterable)	Returns an accurate floating point sum of values in the iterable
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number)
isinf(x)	Returns True if x is a positive or negative infinity
isnan(x)	Returns True if x is a NaN
ldexp(x, i)	Returns x * (2**i)
modf(x)	Returns the fractional and integer parts of x
trunc(x)	Returns the truncated integer value of x
exp(x)	Returns e**x
expm1(x)	Returns e**x - 1

Strings:

- String is defined as a continues set of characters represented in quotation marks (either single quotes (‘) or double quotes (“).
- An individual character in a string is accessed using a subscript (index).
- The subscript should always be an integer (positive or negative).
- A subscript starts from 0 to n-1.
- Strings are immutable i.e. the contents of the string cannot be changed after it is created.
- Python will get the input at run time by default as a string.

- ❖ Python does not support character data type. A string of size 1 can be treated as characters.

1. single quotes (' ')
2. double quotes (" ")
3. triple quotes(“”” “”””)

Operations on string:

1. Indexing
2. Slicing
3. Concatenation
4. Repetitions
5. Member ship

String A	H	E	L	L	O
Positive Index	0	1	2	3	4
Negative Index	-5	-4	-3	-2	-1

indexing	>>>a=”HELLO” >>>print(a[0]) >>>H >>>print(a[-1]) >>>O	❖ Positive indexing helps in accessing the string from the beginning ❖ Negative subscript helps in accessing the string from the end.
Slicing:	Print[0:4] – HELL Print[:3] – HEL Print[0:]- HELLO	The Slice[n : m] operator extracts sub string from the strings. A segment of a string is called a slice.
Concatenation	a=”save ” b=”eart h” print(a+ b) saveeart h	The + operator joins the text on both sides of the operator.
Repetitions:	a=”panimalar ” print(3*a) panimalarpanimalar panimalar	The * operator repeats the string on the left hand side times the value on right hand side.

Membership:	>>> s="good morning"
	>>>"m"
	in s
	True
	>>> "a" not
	in s True
		Using membership operators to check a particular character is in string or not. Returns true if present

ACCESSING CHARACTERS IN PYTHON

In Python, individual characters of a String can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character and so on. While accessing an index out of the range will cause an **IndexError**. Only Integers are allowed to be passed as an index, float or other types will cause a **TypeError**.

INTRODUCTION TO OBJECT AND METHODS

What Is Object-Oriented Programming (OOP)?

Object-oriented Programming, or *OOP* for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual *objects*.

For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running. Or an email with properties like recipient list, subject, body, etc., and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things like cars as well as relations between things like companies and employees, students and teachers, etc. OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

Another common programming paradigm is *procedural programming* which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks, which flow sequentially in order to complete a task.

FORMATTING NUMBERS AND STRINGS

Basic formatting

Simple positional formatting is probably the most common use-case. Use it if the order of your arguments is not likely to change and you only have very few elements you want to concatenate.

Since the elements are not represented by something as descriptive as a name this simple style should only be used to format a relatively small number of elements.

Old

```
'%s %s' % ('one', 'two')
```

New

```
'{} {}'.format('one', 'two')
```

Output

```
one two
```

Old

```
'%d %d' % (1, 2)
```

New

```
'{} {}'.format(1, 2)
```

Output

```
1 2
```

With new style formatting it is possible (and in Python 2.6 even mandatory) to give placeholders an explicit positional index.

This allows for re-arranging the order of display without changing the arguments.

This operation is not available with old-style formatting.

New

```
'{1} {0}'.format('one', 'two')
```

Output

```
two one
```

Value conversion

The new-style simple formatter calls by default the __format__() method of an object for its representation. If you just want to render the output of str(...) or repr(...) you can use the !s or !r conversion flags.

In %-style you usually use %s for the string representation but there is %r for a repr(...) conversion.

Setup

```
class Data(object):

    def __str__(self):
        return 'str'

    def __repr__(self):
        return 'repr'
```

Old

```
'%s %r' % (Data(), Data())
```

New

```
'{0!s} {0!r}'.format(Data())
```

Output

```
str repr
```

In Python 3 there exists an additional conversion flag that uses the output of repr(...) but uses ascii(...) instead.

Setup

```
class Data(object):

    def __repr__(self):
        return 'räpr'
```

Old

```
'%r %a' % (Data(), Data())
```

New

```
'{0!r} {0!a}'.format(Data())
```

Output

```
räpr r\xe4pr
```

Padding and aligning strings

By default values are formatted to take up only as many characters as needed to represent the content. It is however also possible to define that a value should be padded to a specific length.

Unfortunately the default alignment differs between old and new style formatting. The old style defaults to right aligned while for new style it's left.

Align right:

Old

```
'%10s' % ('test',)
```

New

```
'{:>10}'.format('test')
```

Output

```
      test
```

Align left:

Old

```
'%-10s' % ('test',)
```

New

```
'{:10}'.format('test')
```

Output

```
test
```

Again, new style formatting surpasses the old variant by providing more control over how values are padded and aligned.

You are able to choose the padding character:

```
This operation is not available with old-style formatting.
```

New

```
'{:_<10}'.format('test')
```

Output

```
test______
```

And also center align values:

```
This operation is not available with old-style formatting.
```

New

```
'{:^10}'.format('test')
```

Output

```
  test
```

When using center alignment where the length of the string leads to an uneven split of the padding characters the extra character will be placed on the right side:

This operation is not available with old-style formatting.

New

```
'{:^6}'.format('zip')
```

Output

```
 zip
```

Truncating long strings

Inverse to padding it is also possible to truncate overly long values to a specific number of characters.

The number behind a . in the format specifies the precision of the output. For strings that means that the output is truncated to the specified length. In our example this would be 5 characters.

Old

```
'%.5s' % ('xylophone',)
```

New

```
'{:.5}'.format('xylophone')
```

Output

```
xylop
```

Combining truncating and padding

It is also possible to combine truncating and padding:

Old

```
'%-10.5s' % ('xylophone',)
```

New

```
'{:10.5}'.format('xylophone')
```

Output

```
xylop
```

Numbers

Of course it is also possible to format numbers.

Integers:

Old

```
'%d' % (42,)
```

New

```
'{:d}'.format(42)
```

Output

```
42
```

Floats:

Old

```
'%f' % (3.141592653589793,)
```

New

```
'{:f}'.format(3.141592653589793)
```

Output

```
3.141593
```

Padding numbers

Similar to strings numbers can also be constrained to a specific width.

Old

```
'%4d' % (42,)
```

New

```
'{:4d}'.format(42)
```

Output

```
  42
```

Again similar to truncating strings the precision for floating point numbers limits the number of positions after the decimal point.

For floating points the padding value represents the length of the complete output. In the example below we want our output to have at least 6 characters with 2 after the decimal point.

Old

```
'%06.2f' % (3.141592653589793,)
```

New

```
'{:06.2f}'.format(3.141592653589793)
```

Output

```
003.14
```

For integer values providing a precision doesn't make much sense and is actually forbidden in the new style (it will result in a ValueError).

Old

```
'%04d' % (42,)
```

New

```
'{:04d}'.format(42)
```

Output

```
0042
```

Signed numbers

By default only negative numbers are prefixed with a sign. This can be changed of course.

Old

```
'%+d' % (42,)
```

New

```
'{:+d}'.format(42)
```

Output

```
+42
```

Use a space character to indicate that negative numbers should be prefixed with a minus symbol and a leading space should be used for positive ones.

Old

```
'% d' % ((- 23),)
```

New

```
'{: d}'.format((- 23))
```

Output

```
-23
```

Old

```
'% d' % (42,)
```

New

```
'{: d}'.format(42)
```

Output

```
 42
```

New style formatting is also able to control the position of the sign symbol relative to the padding.

This operation is not available with old-style formatting.

New

```
'{:=5d}'.format((- 23))
```

Output

```
-  23
```

New

```
'{:=+5d}'.format(23)
```

Output

```
+  23
```

Named placeholders

Both formatting styles support named placeholders.

Setup

```
data = {'first': 'Hodor', 'last': 'Hodor!'}
```

Old

```
'%(first)s %(last)s' % data
```

New

```
'{first} {last}'.format(**data)
```

Output

```
Hodor Hodor!
```

.format() also accepts keyword arguments.

This operation is not available with old-style formatting.

New

```
'{first} {last}'.format(first='Hodor', last='Hodor!')
```

Output

```
Hodor Hodor!
```

STRING OPERATIONS

A string is a data type used in programming, such as an integer and floating point numbers, but it is used to represent text rather than numbers. It is comprised of a set of characters. Character array is used to store a string.
Examples: “Annamalai”, “2345”

The main operations on string are: length of a string (counting number of characters in the string), reversing a string, concatenating two strings together and comparing two strings. In Python, the arithmetic operator ‘+’ is used for string concatenation and the relational operators ‘<’, ‘<=’, ‘>’, ‘>=’ ‘==” are used for string comparison. In Python, a function is defined using the def keyword.

Coding:

```
# String Operations: String Length, String Reverse, String Concatenation and String Comparison

def strlen(str):
    counter = 0
    while str[counter:]:
        counter += 1
    return counter
```

```
 def strrev(str):
      rstr=""
l=strlen(str)
while l>0:
    rstr = rstr + str[l-1]
    l=l-1
return rstr

    def strcat(st1,st2):
        return(st1+st2)

    def strcmp(st1,st2):
        if(st1==st2):
            print(st1 + " and " + st2 + " are same")
        elif (st1>st2):
            print(st1 + " comes after " + st2 +" in the Dictionary")
        else:
            print(st1 + " comes before " + st2 +" in the Dictionary")

    print("String Functions:\n    1. String Length \n    2. String Reverse \n    3. String Concatenation\n    4.
    String Comparison\n")
    n=int(input("Enter your Choice: "))
    if(n==1):
        str = input("Enter a String: ")
        print("Length of the string is:",strlen(str))
    elif (n==2):
        str = input("Enter a String: ")
        print("Reversed String is:", strrev(str))
    elif (n==3):
        str1 = input("Enter the first String: ")
        str2 = input("Enter the second String: ")
        print("Concatenated string is:", strcat(str1,str2))
    elif (n==4):
        str1 = input("Enter the first String: ")
        str2 = input("Enter the second String: ")
        strcmp(str1,str2)
    else:
        print("Invalid Choice")
```

Result:

```
String Functions:
   1. String Length
   2. String Reverse
   3. String Concatenation
   4. String Comparison

Enter your Choice: 1
Enter a String: annamalai
Length of the string is: 9
>>>
String Functions:
   1. String Length
   2. String Reverse
   3. String Concatenation
   4. String Comparison

Enter your Choice: 2
```

```
Enter a String: university
Reversed String is: ytisrevinu

>>>
String Functions:
   1. String Length
   2. String Reverse
   3. String Concatenation
   4. String Comparison

Enter your Choice: 3
Enter the first String: annamalai
Enter the second String: university
Concatenated string is: annamalaiuniversity
>>>

String Functions:
   1. String Length
   2. String Reverse
   3. String Concatenation
   4. String Comparison

Enter your Choice: 4
Enter the first String: anna
Enter the second String: malai
anna comes before malai in the Dictionary
>>>
```

DRAWING VARIOUS SHAPES AND COLOURS AND FONTS

PyGame and Python programming tutorial video, we cover how to draw shapes with PyGame's built in drawing functionality. We can do things like draw specific pixels, lines, circles, rectangles, and any polygon we want by simply specifying the points to draw between. Let's get started!

```
import pygame

pygame.init()

white = (255,255,255)
black = (0,0,0)

red = (255,0,0)
green = (0,255,0)
blue = (0,0,255)

gameDisplay = pygame.display.set_mode((800,600))
gameDisplay.fill(black)
```

Typical stuff above, now let's cover what would be used to draw a pixel:

```
pixAr = pygame.PixelArray(gameDisplay)
pixAr[10][20] = green
```

Alright, so what have we done above? What we're doing is assigning the entire pixel array to a value, referencing it using pygame.PixelArray. So what this function does is it returns the pixel array of the specified surface (which is the entire display in our case). Then, we're able to modify it. So, we specify pixAr[10][20], which means the pixel residing at (10,20), then we're able to re-assign it. In our case, we call it green.

```
pygame.draw.line(gameDisplay, blue, (100,200), (300,450),5)
```

Drawing lines, above, is easy enough. The function just asks where do we want to draw it, what color do we want it, and then we specify the two coordinate pairs that we want to draw a line between.

```
pygame.draw.rect(gameDisplay, red, (400,400,50,25))
```

We've already extensively covered the drawing of rectangles in this series, but this specific "drawing things" tutorial wouldn't be complete without it. This function asks where to draw, what color, and then asks for a final tuple that contains: the top right x and y, followed by width, then height.

```
pygame.draw.circle(gameDisplay, white, (150,150), 75)
```

Here we draw a circle. This function asks where to draw, what color, what is the center point of the circle, and what is the radius. There is another parameter that you can add which is width.

```
pygame.draw.polygon(gameDisplay, green, ((25,75),(76,125),(250,375),(400,25),(60,540)))
```

Finally, we have polygons. This function asks where to draw, what color, and then asks for a long tuple, of tuples, containing the points of the polygon.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    pygame.display.update()
```

DEFINING A FUNCTIONS

FUNCTIONS:

- Function is a sub program which consists of set of instructions used to perform a specific task.
- A large program is divided into basic building blocks called function.

NEED FOR FUNCTION

- When the program is too complex and large they are divided into parts. Each part is separately coded and combined into single program. Each subprogram is called as function

Built in function	description
>>>max(3,4) 4	# returns largest element
>>>min(3,4) 3	# returns smallest element

>>>len("hello") 5	#returns length of an object
>>>range(2,8,1) [2, 3, 4, 5, 6, 7]	#returns range of given values
>>>round(7.8) – 8.0	#returns rounded integer of the given number
>>>chr(5) \x05'	#returns a character (a string) from an integer
>>>float(5) 5.0	#returns float number from string or integer
>>>int(5.0) 5	# returns integer from string or float
>>>pow(3,5) 243	#returns power of given number
>>>type(5.6) <type 'float'>	#returns data type of object to which it belongs
>>>t=tuple([4,6.0,7]) (4, 6.0, 7)	# to create tuple of items from list
>>>print("good morning") Good morning	# displays the given object

- ❖ Debugging, Testing and maintenance becomes easy when the program is divided into subprograms.
- ❖ Functions are used to avoid rewriting same code again and again in a program.
- ❖ Function provides code re-usability
- ❖ The length of the program is reduced.

TYPES OF FUNCTION:

1. Built in function
2. user defined function

BUILT IN FUNCTION

- ❖ Built in functions are the functions that are already created and stored in python. These built in functions are always available for usage and accessed by a programmer.
- ❖ It cannot be modified.

FUNCTION PROTOTYPES:

Based on arguments and return type functions are classified into 4 types.

1. Function without arguments and without return type
2. Function with arguments and without return type
3. Function without arguments and with return type
4. Function with arguments and with return type

1. **Function without arguments and without return type**
   In this type no argument is passed through the function call and no output is return to main function The sub function will read the input values perform the operation and print the result in the same block

2. **Function with arguments and without return type**
   Arguments are passed through the function call but output is not return to the main function

3. **Function without arguments and with return type**
   In this type no argument is passed through the function call but output is return to the main function.

4. **Function with arguments and with return type**
   In this type arguments are passed through the function call and output is return to the main function

without return type	
Without argument	**With argument**
def add(): a=int(input("enter a")) b=int(input("enter b")) c=a+b print(c) add()	def add(a,b): c=a+b print(c) a=int(input("enter a")) b=int(input("enter b")) add(a,b)
OUTPUT: enter a 5 enter b 10 15	**OUTPUT:** enter a 5 enter b 10 15

with return type	
Without argument	**With argument**
def add(): a=int(input("enter a")) b=int(input("enter b")) c=a+b return c c=add() print(c)	def add(a,b): c=a+b return c a=int(input("enter a")) b=int(input("enter b")) c=add(a,b) print(c)
OUTPUT: enter a 5 enter b 10 15	**OUTPUT:** enter a 5 enter b 10 15

PARAMETERS AND ARGUMENTS:

Parameters:

- Parameters are the value(s) provided in the parenthesis when we write function header.
- These are the values required by function to work.
- If there is more than one value required, all of them will be listed in parameter list separated by **comma.**

Example: def add(a,b):

Arguments :

- Arguments are the value(s) provided in function call statement.
- List of arguments should be supplied in same way as parameters are listed.
- Bounding of parameters to arguments is done 1:1, and so there should be same number and type of arguments as mentioned in parameter list.

Example: add(a,b)

Return Statement:

- The **return statement is used to exit a function** and go back to the place from

where it was called.

- If the return statement has no arguments, then it will not return any values. But exits from function.

Syntax:	**Example:**

return variable	def add(a,b):
	  c=a+b return
	  c
	x=5
	y=4
	c=add(a,b)
	print(c)

ARGUMENTS TYPES:

You can call a function by using the following types of formal arguments:

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

Required Arguments:

The number of arguments in the function call should match exactly with the function definition.

Example	**Output:**
def student(name, roll): print(name,roll) student("george",98)	George 98
def student(name, roll): print(name,roll) student(101,"rithika")	101 rithika
def student(name, roll): print(name,roll) student(101)	student() missing 1 required positional argument: 'name'
def student(name, roll): print(name,roll) student()	student() missing 2 required positional arguments: 'roll' and 'name'

Keyword Arguments:

Python interpreter is able to use the keywords provided to match the values with parameters even though if they are arranged in out of order.

Example	**Output:**
def student(name, roll,mark):	bala

print(name,roll,mark) student (mark=90,roll=102,name="bala")	102 90

TURTLE COMMANDS

"Turtle" is a python feature like a drawing board, which lets you command a turtle to draw all over it!

You can use functions like turtle.forward(...) and turtle.left(...) which can move the turtle around.

Before you can use turtle, you have to import it. We recommend playing around with it in the interactive interpreter first, as there is an extra bit of work required to make it work from files. Just go to your terminal and type:

```
import turtle
```

Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzeig, Seymour Papert and Cynthia Solomon in 1967.

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an import turtle, give it the command turtle.forward(15), and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command turtle.right(25), and it rotates in-place 25 degrees clockwise.

The object-oriented interface uses essentially two+two classes:

The TurtleScreen class defines graphics windows as a playground for the drawing turtles. Its constructor needs a tkinter.Canvas or a ScrolledCanvas as argument. It should be used

def student(name, roll,mark): print(name,roll,mark) student (name="bala", roll=102,mark=90)	bala 102 90

when turtle is used as part of some application.

The function Screen() returns a singleton object of a TurtleScreen subclass. This function should be used when turtle is used as a standalone tool for doing graphics. As a singleton object, inheriting from its class is not possible.

All methods of TurtleScreen/Screen also exist as functions, i.e. as part of the procedure-oriented interface.

RawTurtle (alias: RawPen) defines Turtle objects which draw on a TurtleScreen. Its constructor needs a Canvas, ScrolledCanvas or TurtleScreen as argument, so the RawTurtle objects know where to draw.

Derived from RawTurtle is the subclass Turtle (alias: Pen), which draws on "the" Screen instance which is automatically created, if not already present.

All methods of RawTurtle/Turtle also exist as functions, i.e. part of the procedure-oriented interface.

The procedural interface provides functions which are derived from the methods of the classes Screen and Turtle. They have the same names as the corresponding methods. A screen object is automatically created whenever a function derived from a Screen method is called. An (unnamed) turtle object is automatically created whenever any of the functions derived from a Turtle method is called.

To use multiple turtles on a screen one has to use the object-oriented interface.

METHODS OF RAWTURTLE/TURTLE AND CORRESPONDING FUNCTIONS

Most of the examples in this section refer to a Turtle instance called turtle.

Turtle motion
turtle.**forward**(*distance*)

turtle.**fd**(*distance*)

Parameters

distance – a number (integer or float)

Move the turtle forward by the specified *distance*, in the direction the turtle is headed.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

turtle.**back**(*distance*)

turtle.**bk**(*distance*)

turtle.**backward**(*distance*)

Parameters

distance – a number

Move the turtle backward by *distance*, opposite to the direction the turtle is headed. Do not change the turtle's heading.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

turtle.**right**(*angle*)

turtle.**rt**(*angle*)

Parameters

angle – a number (integer or float)

Turn turtle right by *angle* units. (Units are by default degrees, but can be set via the degrees() and radians() functions.) Angle orientation depends on the turtle mode, see mode().

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

turtle.**left**(*angle*)

turtle.**lt**(*angle*)

Parameters

angle – a number (integer or float)

Turn turtle left by *angle* units. (Units are by default degrees, but can be set via the degrees() and radians() functions.) Angle orientation depends on the turtle mode, see mode().

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.**goto**(*x*, *y=None*)

turtle.**setpos**(*x*, *y=None*)

turtle.**setposition**(*x*, *y=None*)

Parameters

- **x** – a number or a pair/vector of numbers
- **y** – a number or None

If *y* is None, *x* must be a pair of coordinates or a Vec2D (e.g. as returned by pos()).

Move turtle to an absolute position. If the pen is down, draw line. Do not change the turtle's orientation.

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.**setx**(*x*)

Parameters

x – a number (integer or float)

Set the turtle's first coordinate to *x*, leave second coordinate unchanged.

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
```

```
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

Parameters

y – a number (integer or float)

Set the turtle's second coordinate to *y*, leave first coordinate unchanged.

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.**setheading**(*to_angle*)

turtle.**seth**(*to_angle*)

Parameters

to_angle – a number (integer or float)

Set the orientation of the turtle to *to_angle*. Here are some common directions in degrees:

standard mode	logo mode
0 – east	0 - north
90 – north	90 - east
180 – west	180 - south
270 – south	270 - west

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.**home**()

Move turtle to the origin – coordinates (0,0) – and set its heading to its start-orientation (which depends on the mode, see mode()).

```
>>> turtle.heading()
90.0
```

```
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

DEFAULT ARGUMENTS:

Assumes a default value if a value is not provided in the function call for that argument.

Example	Output:
def student(name=”raja”, roll=101,mark=50): print(name,roll,mark) student (mark=90,roll=102,name="bala")	bala 102 90
def student(name, roll,mark): print(name,roll,mark) student (name="bala", roll=102)	bala 102 50
def student(name, roll,mark): print(name,roll,mark) student ()	bala 102 90

Variable length Arguments

If we want to specify more arguments than specified while defining the function, variable length arguments are used. It is denoted by * symbol before parameter.

Example	Output:
def student(name,*mark): print(name,mark) student (“bala”,90)	bala 102 90
def student(name, roll,mark): print("name,roll,mark) student (“raja”,90,70,80)	raja 90 70 80

GENERATE RANDOM ASCII CHARACTERS IN PYTHON

- **Generate a random string of a fixed length.**
- Generate a random string with **lower case** and **upper case**.

- Generate a random **alphanumeric string** with letters and numbers.
- Generate a random string password which contains the **letters**, **digits**, and **special characters**.
- Use the **UUID** and **secrets** module to **generate a secure random string** for a sensitive application.

Call string.ascii_letters to get a string of the lowercase alphabet followed by the uppercase alphabet. Use random.choice(seq) with the string of alphabets as seq to generate a random letter from the string.

lower_upper_alphabet = string.ascii_letters

random_letter = random.choice(lower_upper_alphabet)
Generate random letter

print(random_letter)
O U T P U T

y

GENERATE A RANDOM STRING OF FIXED LENGTH

To generate a random string we need to use the following two Python modules.

The string module contains various string constant which contains the ASCII characters of all cases. The string module contains separate constants for lowercase, uppercase letters, digits, and special characters.

random module to perform the random generations.

Let see the **steps to generate a random string** of a fixed length of n.

Use the string constant **string.ascii_lowercase** to get all the lowercase letters in a single string.
The string.ascii_lowercase constant contains all lowercase letters.
I.e., 'abcdefghijklmnopqrstuvwxyz'

Run for loop n number of times to pick a single character from a string constant using a random.choice() function and add it to the string variable using a join function**.**

Note: – The **random.choice()** function used to choose a single character from a list.

For example, Suppose you want a random string of length 6 then we can execute a random.choice() function 6 times to pick a single letter from the string.ascii_lowercase and add it to the string variable. Let see the code now.

```
import random
import string
def randomString(stringLength=10):
    """Generate a random string of fixed length """
    letters = string.ascii_lowercase
    return ''.join(random.choice(letters) for i in range(stringLength))
print ("Random String is ", randomString() )
print ("Random String is ", randomString(10) )
print ("Random String is ", randomString(10) )
```

Output:

```
Random String is  ptmihemlzj
Random String is  dbgxpggrez
Random String is  wkhhaghero
```

Generate a random string of specific letters only

```
## Generate a random string of specific characters
def randString(length=5):
    #put your letters in the following string
    your_letters='abcdefghi'
    return ''.join((random.choice(your_letters) for i in range(length)))

print ("Random String with specific letters ", randString() )
print ("Random String with specific letters ", randString(5) )
```

Output:

```
Random String with specific letters  agbfh

Random String with specific letters  deifd
```

Generate random string with letters and digits in Python

Many times we need a random string that contains both letters and digit. For example, you want to generate a random string like **ab23cd**, **jkml98**, **87thki**.

We need to use the string.ascii_letters and **string.digits** constants two get the combinations of letters and digits in our random string. Now, Let see how to generate a random string with letters and digits.

```
import random

import string

def randomStringDigits(stringLength=6):

    """Generate a random string of letters and digits """

    lettersAndDigits = string.ascii_letters + string.digits

    return ''.join(random.choice(lettersAndDigits) for i in range(stringLength))

print ("Generating a Random String including letters and digits")

print ("First Random String is  ", randomStringDigits(8))

print ("Second Random String is ", randomStringDigits(8))

print ("Third Random String is  ", randomStringDigits(8))
```

Output:

```
Generating a Random String including letters and digits

First Random String is   zp5bK7Ah

Second Random String is  w48Nr78j

Third Random String is   Heq11gr4
```

CLASS AND OBJECTS:

A class in C++/Java/Python is the building block that leads to object-oriented programming. It is an user defined data-type which has data members and member functions. Data members are the data variables and member functions are the functions used to manipulate these variables. A class is defined in python using the keyword class followed by name of the class.

An object is an instance of a class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) then the memory is allocated.

The __init__() function in the class is executed automatically every time the class is being used to create a new object. It is called as a constructor in object oriented terminology. This function is used to initialize the data members of the class.

In most of the object-oriented programming (OOP) languages access specifiers are used to limit the access to the variables and functions of a class. Most of the OOP languages use three types of access specifiers, they are: private, public and protected. In Python, all the variables and member functions of a class are public by default. Adding a prefix __ (double underscore) to the member variable or function makes them to be private.

Coding:

```
# To find the Euclidean distance between two pints in a three dimensional space using
#     class and objects.
# Class called point consists of  a constructor (__init__())  and two functions
#     distancefromorgin() and distance()

import math
class point():
    def __init__(self,a,b,c):
        self.x=a
        self.y=b
        self.z=c
    def distancefromorigin(self):
        return ((self.x ** 2) + (self.y ** 2) +(self.z ** 2)) ** 0.5
    def distance(self, point2):
        xdiff = self.x-point2.x
        ydiff = self.y-point2.y
        zdiff = self.z-point2.z
        dist = math.sqrt (xdiff**2 + ydiff**2+ zdiff**2)
        return dist

x1,y1,z1= (input("Enter the coordinates of a first point P1(x1,y1,z1): ")).split()
x1,y1,z1 =[int(x1),int(y1), int(z1)]
x2,y2,z2= (input("Enter the coordinates of a second point P2(x2,y2,z2): ")).split()
```

```
x2,y2,z2 =[int(x2),int(y2),int(z2)]

# P1 and P2 are  objects  of point class

p1 = point(x1,y1,z1)
p2 = point(x2,y2,z2)
print('Distance from origin to P1:', p1.distancefromorigin())
print('Distance from origin to P2:', p2.distancefromorigin())
print('Distance from P1 to P2:',p1.distance(p2))
```

Result:

```
Enter the coordinates of a first point P1(x1,y1,z1): 1 2 3
1 2 3
Enter the coordinates of a second point P2(x2,y2,z2): 2 3 4
2 3 4
Distance from origin to P1: 3.7416573867739413
Distance from origin to P2: 5.385164807134504
Distance from P1 to P2: 1.7320508075688772
>>>
```

CLASS

- A class will have attributes and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.
- The key word is “class”

Syntax

```
class class_name:

    statement(s);
```

OBJECT

- An object is an instance of class.
- A class may have any number of objects.
- Objects can pass message between them.

Syntax

```
object_name = class_name(parameters);
```

Attributes

- Attributes are the identifiers which can hold the data.
- These attributes are used inside the class methods.
- A class may have any number of attributes.

METHODS

- Methods are nothing but functions which can do some work.
- Methods will accept parameters as input and if required return the output.

- Methods can be identified by paranthesis.

Example

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

The __init__() method will be called automatically every time the class is being used to create a new object.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

INITIALISATION USING SELF

SELF

The self in keyword in Python is used to all the instances in a class. By using the self keyword, one can easily access all the instances defined within a class, including its methods and attributes.

init

__init__ is one of the reserved methods in Python. In object oriented programming, it is known as a constructor. The __init__ method can be called when an object is created from the class, and access is required to initialize the attributes of the class.

CLASS DEFINITION & INHERITANCE

CREATING CLASSES

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows.

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

- The class has a documentation string, which can be accessed via *ClassName.__doc__*.
- The *class_suite* consists of all the component statements defining class members, data attributes and functions.

Example

Following is the example of a simple Python class.

```
class Employee:
   'Common base class for all employees'
   empCount = 0
   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1
   def displayCount(self):
     print "Total Employee %d" % Employee.empCount
   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- The variable *empCount* is a class variable whose value is shared among all instances of a this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

PYTHON - PUBLIC, PRIVATE AND PROTECTED ACCESS MODIFIERS

Classical object-oriented languages, such as C++ and Java, control the access to class resources by public, private and protected keywords. Private members of a class are denied access from the environment outside the class. They can be handled only from within the class.

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

Protected members of a class are accessible from within the class and are also available to its sub-classes. No other environment is permitted access to it. This enables specific resources of the parent class to be inherited by the child class.

Python doesn't have any mechanism that effectively restricts access to any instance variable or method. Python prescribes a convention of prefixing the name of the variable/method with single or double underscore to emulate the behaviour of protected and private access specifiers.

All members in a Python class are **public** by default. Any member can be accessed from outside the class environment.

CREATING INSTANCE OBJECTS

To create instances of a class, you call the class using class name and pass in whatever arguments its *__init__* method accepts.

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows.

```
emp1.displayEmployee()
emp2.displayEmployee()
```

```
print "Total Employee %d" % Employee.empCount
```

INHERITANCE:

Inheritance is a mechanism in which one class (derived class) acquires the property of another class (base class). With inheritance, we can reuse the variables and methods of the existing class. The existing class is called base class and the new class is called derived class. Hence, inheritance facilitates reusability and is an important concept of object oriented programming. Types of inheritance are: single inheritance, multiple inheritance, multi-level inheritance, hierarchical inheritance and hybrid inheritance.

Single inheritance enables a derived class to use the variables and functions defined in an existing class. In multiple inheritance, derived class inherits the characteristics and features from more than one existing classes.

In python, syntax for defining single inheritance is class **z(x)**, where x is the name of the base class and z is the name of the derived class. Similarly, multiple inheritance is defined using the syntax **class z(x, y)**, where x and y are the names of base classes and z is the name of the derived class.

Advantages of inheritance

Inheritance is the capability of one class to derive or inherit the properties from some another class. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.

- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Coding:

```
# person is a base class
class person:
    def __init__(self, n, a):
        self.name = n
        self.age = a
  # employee is the class derived from person using single inheritance

class employee(person):
    def __init__(self, n,a, d,s):
        person.__init__(self,n,a)
        self.designation=d
        self.salary=s


    def show(self):
        print("Employee Details: ")
        print("  Name: ",self.name,"\n  Age:",self.age, "\n  Designation:",self.designation, "\n
Salary:",self.salary)

#  student is a  base class
class student:
    def __init__(self, id, rno):
        self.studentId = id
        self.roomno=rno

#  resident is a class derived from person and student using multiple inheritance
class resident(person, student):
    def __init__(self, n, a, id,rno):
        person.__init__(self, n, a)
        student.__init__(self, id,rno)

    def show(self):
        print("Resident Details:")
        print("  Name:", self.name,"\n  Age: ",self.age, "\n  Id:" ,self.studentId,"\n  Room no.:",self.roomno)
  # Creating objects of employee and resident classes
e1 =employee("Arun",35,"Data analyst",50000)
r1 = resident("John", 30, 201900025,203)
e1.show()
r1.show()
```

Result:

```
Employee Details:
  Name:  Arun
```

```
  Age: 35
  Designation: Data analyst
  Salary: 50000
Resident Details:
  Name: John
  Age:  30
  Id: 201900025
  Room no.: 203
>>>
```

COMPOSITION

Composition means that an object knows another object, and explicitly delegates some tasks to it. While inheritance is implicit, composition is explicit: in Python, however, things are far more interesting than this =).

First of all let us implement classic composition, which simply makes an object part of the other as an attribute.

The primary goal of composition is to relax the coupling between objects. This little example shows that now SecurityDoor is an object and no more a Door, which means that the internal structure of Door is not copied.

For this very simple example both Door and SecurityDoor are not big classes, but in a real system objects can very complex; this means that their allocation consumes a lot of memory and if a system contains thousands or millions of objects that could be an issue.

OPERATOR OVERLOADING

Operator overloading is an important concept in object oriented programming. It is a type of polymorphism in which a user defined meaning can be given to an operator in addition to the predefined meaning for the operator.

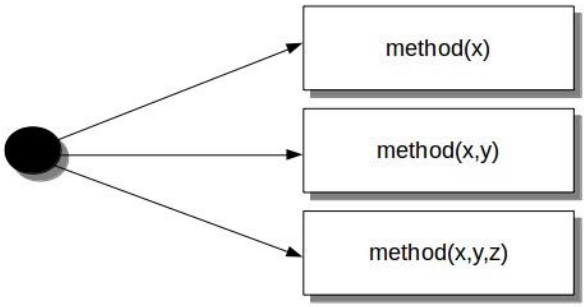
Operator overloading allow us to redefine the way operator works for user-defined types such as objects. It cannot be used for built-in types such as int, float, char etc., For example, '+' operator can be overloaded to perform addition of two objects of distance class.

Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator on objects, the magic method __add__() is automatically invoked in which the meaning/operation for + operator is defined for user defined objects.

METHOD OVERLOADING

In Python you can define a method in such a way that there are multiple ways to call it.

Depending on the function definition, it can be called with zero, one, two or more parameters.



Coding:

```
# distance is a class. Distance is measured in terms of feet and inches

class distance:
    def __init__(self, f,i):
        self.feet=f
        self.inches=i
    # overloading of binary operator > to compare two distances
    def __gt__(self,d):
        if(self.feet>d.feet):
          return(True)
        elif((self.feet==d.feet) and (self.inches>d.inches)):
            return(True)
        else:
            return(False)
# overloading of binary operator + to add two distances
    def __add__(self, d):
        i=self.inches + d.inches
        f=self.feet + d.feet
        if(i>=12):
            i=i-12
            f=f+1
        return distance(f,i)
```

```
        # displaying the distance
    def show(self):
        print("Feet= ", self.feet, "Inches= ",self.inches)
  a,b= (input("Enter feet and inches of distance1: ")).split()
a,b =[int(a),int(b)]
c,d= (input("Enter feet and inches of distance2: ")).split()
c,d =[int(c),int(d)]
d1 = distance(a,b)
d2 = distance(c,d)
if(d1>d2):
    print("Distance1 is greater than Distance2")
else:
    print("Distance2 is greater or equal to Distance1")
d3=d1+d2
print("Sum of the two Distance is:")
d3.show()
```

Result:

```
Enter feet and inches of distance1: 8 4
8 4
Enter feet and inches of distance2: 6 9
6 9
Distance1 is greater than Distance2
Sum of the two Distance is:
Feet=  15 Inches=  1
>>>
```

METHOD OVERRIDING IN PYTHON

Overriding is the ability of a class to change the implementation of a method provided by one of its ancestors. Overriding is a very important part of OOP since it is the feature that makes inheritance exploit its full power. Through method overriding a class may "copy" another class, avoiding duplicated code, and at the same time enhance or customize part of it. Method overriding is thus a strict part of the inheritance mechanism.

As for most OOP languages, in Python inheritance works through implicit delegation: when the object cannot satisfy a request, it first tries to forward the request to its ancestors, following the specific language rules in the case of multiple inheritance.

An example:

```
class Parent(object):
    def __init__(self):
        self.value = 5
    def get_value(self):
        return self.value

class Child(Parent):
    pass
```

As you can see the Child class is empty, but since it inherits from Parent Python takes charge of routing all method calls. So you may use the get_value() method of Child objects and everything works as expected.

```
>>> c = Child()
>>> c.get_value()
5
```

PYTHON SPECIAL UNIT

The unittest unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

To achieve this, unittest supports some important concepts in an object-oriented way:

test fixture

A *test fixture* represents the preparation needed to perform one or more tests, and any associated cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case

A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.

test suite

A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner

A *test runner* is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

OBJECT REPRESENATION

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method *__del__()*, called a

destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.

ATTRIBUTE BINDING

Attributes of class objects

You normally specify an attribute of a class object by binding a value to an identifier within the class body. For example:

```
class C1(object):
    x = 23
print(C1.x)                    # prints: 23
```

The class object *C1* has an attribute named *x*, bound to the value 23, and *C1.x* refers to that attribute.

You can also bind or unbind class attributes outside the class body. For example:

```
class C2(object): pass
C2.x = 23
print(C2.x)                    # prints: 23
```

Your program is usually more readable if you bind, and thus create, class attributes only with statements inside the class body. However, rebinding them elsewhere may be necessary if you want to carry state information at a class, rather than instance, level; Python lets you do that, if you wish. There is no difference between a class attribute created in the class body, and one created or rebound outside the body by assigning to an attribute.

As we'll discuss shortly, all instances of the class share all of the class's attributes.

The class statement implicitly sets some class attributes. Attribute __name__ is the *classname* identifier string used in the class statement. Attribute __bases__ is the tuple of class objects given as the base classes in the class statement. For example, using the class *C1* we just created:

```
print(C1.__name__, C1.__bases__)
```

```
# prints: C1 (<type 'object'>,)
```

A class also has an attribute __dict__, the mapping object that the class uses to hold other attributes (AKA its *namespace*); in classes, this mapping is read-only.

In statements that are directly in a class's body, references to attributes of the class must use a simple name, not a fully qualified name. For example:

```
class C3(object):
    x = 23
    y = x + 22                  # must use just x, not C3.x
```

However, in statements in *methods* defined in a class body, references to attributes of the class must use a fully qualified name, not a simple name. For example:

```
class C4(object):
    x = 23
    def amethod(self):
        print(C4.x)  # must use C4.x or self.x, not just x!
```

Note that attribute references (i.e., an expression like *C.s*) have semantics richer than those of attribute bindings.

Function definitions in a class body

Most class bodies include def statements, since functions (known as *methods* in this context) are important attributes for most class objects. A def statement in a class body obeys the rules presented in "Functions". In addition, a method defined in a class body has a mandatory first parameter, conventionally named self, that refers to the instance on which you call the method. The self parameter plays a special role in method calls, as covered in "Bound and Unbound Methods".

Here's an example of a class that includes a method definition:

```
class C5(object):
```

```
def hello(self):
    print('Hello')
```

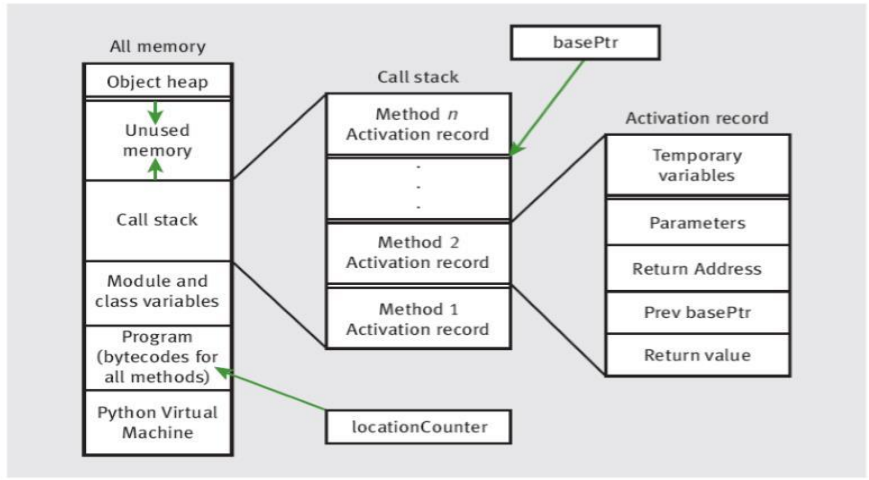
MEMORY MANAGEMENT IN PYTHON

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions. In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

- PyMem_MALLOC(size)
- PyMem_NEW(type, size)
- PyMem_REALLOC(ptr, size)
- PyMem_RESIZE(ptr, type, size)
- PyMem_FREE(ptr)
- PyMem_DEL(ptr)



The architecture of a run-time environment

Fundamentals of Python: From First Programs Through Data Structures

---

Special properties of classes

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then MyClass.i and MyClass.f are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of MyClass.i by assignment. __doc__ is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable x.

The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named __init__(), like this:

```
def __init__(self):
    self.data = []
```

When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the __init__() method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to __init__(). For example,

```
>>>
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

SLOTS AND PRIVATE ATTRIBUTES

Slots

slots provide a special mechanism to reduce the size of objects. It is a concept of memory optimisation on objects.

Avoiding Dynamically Created Attributes

The attributes of objects are stored in a dictionary "__dict__". Like any other dictionary, a dictionary used for attribute storage doesn't have a fixed number of elements. In other words, you can add elements to dictionaries after they have been defined, as we have seen in our chapter on dictionaries. This is the reason, why you can dynamically add attributes to objects of classes that we have created so far:

```
>>> class A(object):
...     pass
```

```
...
>>> a = A()
>>> a.x = 66
>>> a.y = "dynamically created attribute"
```

The dictionary containing the attributes of "a" can be accessed like this:

```
>>> a.__dict__
{'y': 'dynamically created attribute', 'x': 66}
```

You might have wondered that you can dynamically add attributes to the classes, we have defined so far, but that you can't do this with built-in classes like 'int', or 'list':

```
>>> x = 42
>>> x.a = "not possible to do it"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'a'
>>>
>>> lst = [34, 999, 1001]
>>> lst.a = "forget it"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'a'
```

18CSPC405 – PYTHON PROGRAMMING

Unit IV - FILES AND EXCEPTION HANDLING

A file is a collection of data stored in one unit, identified by a filename. It can be a text document, picture/image, audio, audio-video or other collection of data. The common format/extensions for text documents are .doc, .docx (Microsoft word documents), odt (Libre Office open document text), .pdf (Adobe portable document format) , rtf (Microsoft rich text format), .tex (LaTeX text), .txt (Microsoft Notepad text). The image file formats are: .jpg, .tiff, .gif, .png, bmp. The commonly used audio formats are: .wav and .mp3. The audio-video format includes .avi, .mp4, .mkv, .mov, .flv, .wmv etc.,

Python supports file handling and allow users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. Python file functions open() and close() are used for opening and closing a file. The read() and write() functions are used for reading and writing text (or numeric or binary data) from/to the file, respectively. File opening modes in Python are: r (read), w (write), a (append), rb (reading binary data), wb(writing binary data).

OUTPUT USING THE PRINT() FUNCTION

To output your data to the screen, use the print () function. You can write print (argument) and this will print the argument in the next line when you press the ENTER key.

Definitions to remember: An argument is a value you pass to a function when calling it. A value is a letter or a number. A variable is a name that refers to a value. It begins with a letter. An assignment statement creates new variables and gives them values.

This syntax is valid in both Python 3.x and Python 2.x. For example, if your data is "Guido," you can put "Guido" inside the parentheses ( ) after print.

```
>>>print("Guido")
Guido
```

Sample Program

a) Coding: (Reading and Writing Text File)

```
# File copy – content of a text file (input.txt) is copied to another text file (output.txt)
infile=open("/media/yughu/D/input.txt","r")
outfile=open("/media/yughu/D/output.txt","w")
lines = chars = 0
for line in infile:
    lines += 1
```

```
        chars += len(line)
        outfile.write(line)
    print(lines, "lines copied,",chars, "characters copied")
    infile.close()
    outfile.close()
```

Result:

11 lines copied, 82 characters copied

b) Coding: (Reding and Writing Numeric Data File)

 # sum of all the numbers in the input file (input.dat) is computed and
 # it is written to the output file (output.dat)

```
    infile=open("/media/yughu/D/input.dat","r")
    outfile=open("/media/yughu/D/output.dat","w")
    sum=0
    s = infile.read()
    numbers = [int(x) for x in s.split()]
    print("The numbers are:")
    print(numbers)
    for num in numbers:
      sum=sum+num
    sum=str(sum)
    outfile.write("Sum is ")
    outfile.write(sum)
    infile.close()
    outfile.close()
```

Result:

The numbers are:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

FILE DIALOGUES

Creating Menus

When you enter the menu bar editor, it displays a new untitled menu.

How to Create A New Menu

Choose File New (or the New button on the toolbar).

The menu bar editor displays a new menu with an end-of-menu marker (>).

In the Label property, enter the text of the first menu label and select which type of menu item it is**.**

How to Insert New Menu Items

Select the item that is to be under the new item. To place an item at the bottom of the menu, select the end-of-menu marker.

Choose Edit Insert. The editor places the new item above the previous selection with the label UNTITLED.

EXCEPTION HANDLING

Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution such as division by zero (ZeroDivisionError), opening a file for reading that does not exist (IOError), indentation is not specified properly (IndentationError) etc., In general, an exception breaks the normal flow of execution. Exception handling enables a program to deal with exceptions and continue its normal execution.

A try statement in Python can have more than one except clause to handle different exceptions. The statement can also have an optional else and/or finally statement. The try-except syntax is:

```
try:
        <body>
except <ExceptionType1>:
        <handler1>
           ...
           ...
except <ExceptionTypeN>:
         <handlerN>
except:
         <handlerExcept>
else:
        <process_else>
finally:
         <process_finally>
```

The multiple excepts are similar to elifs in python. When an exception occurs, it is checked to match an exception in an except clause after the try clause one by one (sequentially). If a match is found, the handler for the matching case is executed and the rest of the except clauses are skipped. Note that the <ExceptionType> in the last except clause may be omitted. If the exception does not match any of the exception types before the last except clause, the <handlerExcept> for the last except clause is executed.

A try statement may have an optional else clause, which is executed if no exception is raised in the try body. A try statement may have an optional finally clause, which is intended to define cleanup actions that must be performed under all circumstances.

Sample Coding:

```
# Hadling exceptions that occurs at runtime such as division by zero, syntax error and

# raising and handling the exception.

try:

   number1, number2 = eval(input("Enter two numbers separated by a comma: "))
   result = number1 / number2
   print("Result is", result)
   if(number1==0):
     raise RuntimeError()
         except ZeroDivisionError:
            print("Division by Zero")
         except SyntaxError:
            print("A comma may be Missing in the Input")
         except RuntimeError:
            print("May be Meaningless")
except:
        print("Something Wrong in the Input")
     else:
        print("No Exceptions")
     finally:
        print("Finally Clause is Executed")
```

Result:

```
Enter two numbers separated by a comma: 4,0
        4,0
        Division by Zero
        Finally Clause is Executed
        >>>
Enter two numbers separated by a comma: 5 6
        5 6
        A comma may be Missing in the Input
        Finally Clause is Executed
        >>>
Enter two numbers separated by a comma: 0,9
        0,9
        Result is 0.0
        May be Meaningless
        Finally Clause is Executed
```

```
        >>>
Enter two numbers separated by a comma: 6
        6
        Something Wrong in the Input
        Finally Clause is Executed
        >>>
Enter two numbers separated by a comma: 12,4
        12,4
        Result is 3.0
        No Exceptions
        Finally Clause is Eexecuted
```

TEXT INPUT AND OUTPUT

Opening a file creates a file object.

In this example, the variable f refers to the new file object.

```
>>> f = open("test.dat","w")
>>> print f
<open file 'test.dat', mode 'w' at fe820>
```

The open function takes two arguments. The first is the name of the file, and the second is the mode. Mode "w" means that we are opening the file for writing.

If there is no file named test.dat, it will be created. If there already is one, it will be replaced by the file we are writing.

When we print the file object, we see the name of the file, the mode, and the location of the object.

To put data in the file we invoke the write method on the file object:

```
>>> f.write("Now is the time")
>>> f.write("to close the file")
```

Closing the file tells the system that we are done writing and makes the file available for reading:

```
>>> f.close()
```

Now we can open the file again, this time for reading, and read the contents into a string. This time, the mode argument is "r" for reading:

```
>>> f = open("test.dat","r")
```

If we try to open a file that doesn't exist, we get an error:

```
>>> f = open("test.cat","r")
```

EXCEPTION HANDLING – PROGRAM ERRORS AND EXCEPTION HANDLING

- Types of program errors
- Syntax, semantic, and logical errors
- Compile time and runtime errors
- Test drivers
- Debugging techniques
- Exception handling
- The most common types of exceptions
- The throws clause and the throw statement
- Catching exceptions by means of the try-catch construct
- Propagation of exceptions
- Exceptions when reading from a file

Types of program errors

We distinguish between the following types of errors:

Syntax errors: errors due to the fact that the syntax of the language is not respected.
Semantic errors: errors due to an improper use of program statements.
Logical errors: errors due to the fact that the specification is not respected.

From the point of view of when errors are detected, we distinguish:

Compile time errors: syntax errors and static semantic errors indicated by the compiler.
Runtime errors: dynamic semantic errors, and logical errors, that cannot be detected by the compiler.
Syntax errors: Syntax errors are due to the fact that the syntax of the Java language is not respected.
Semantic errors: Semantic errors indicate an improper use of Java statements.
Logical errors: Logical errors are caused by the fact that the software specification is not respected. The program is compiled and executed without errors, but does not generate the requested result.

TYPES OF EXCEPTION CLASS

- Base Exception
    - Exception
        - Arithmetic Error
            - Floating Point Error
            - Overflow Error
            - Zero Division Error
- Assertion Error

Base Exception

The Base Exception class is, as the name suggests, the base class for all built-in exceptions in Python. Typically, this exception is never raised on its own, and should instead be inherited by other, lesser exception classes that can be raised.

Exception

Exception is the most commonly-inherited exception type (outside of the true base class of Base Exception). In addition, all exception classes that are considered errors are subclasses of the Exception class. In general, any custom exception class you create in your own code should inherit from Exception.

Arithmetic Error

The base class for the variety of arithmetic errors, such as when attempting to divide by zero, or when an arithmetic result would be too large for Python to accurately represent.

Assertion Error

This error is raised when a call to the [assert] statement fails.

BINARY I/O USING PICKLE — PYTHON OBJECT SERIALIZATION

The pickle module implements binary protocols for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream and “unpickling” is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “**serialization”,** “marshalling,” 1 or “flattening”; however, to avoid confusion, the terms used here are “**pickling” and “unpickling**”.

Module Interface

To serialize an object hierarchy, you simply call the dumps() function. Similarly, to de-serialize a data stream, you call the loads () function. However, if you want more control over serialization and de-serialization, you can create a Pickler or an Unpickler object, respectively.

The pickle module provides the following constants:

pickle.HIGHEST_PROTOCOL
An integer, the highest protocol version available. This value can be passed as a protocol value to functions dump() and dumps() as well as the Pickler constructor.

pickle.DEFAULT_PROTOCOL
An integer, the default protocol version used for pickling. May be less than HIGHEST_PROTOCOL. Currently the default protocol is 4, first introduced in Python 3.4 and incompatible with previous versions.

The pickle module provides the following functions to make the pickling process more convenient:

pickle.dump(obj, file, protocol=None, *, fix imports=True, buffer callback=None)
Write the pickled representation of the object obj to the open file object file. This is equivalent to Pickler(file, protocol).dump(obj).

READING AND WRITING TO A BINARY FILE

The open() function opens a file in text format by default. To open a file in binary format, add 'b' to the mode parameter. Hence the "rb" mode opens the file in binary format for reading, while the "wb" mode opens the file in binary format for writing. Unlike text mode files, binary files are not human readable. When opened using any text editor, the data is unrecognizable.

The following code stores a list of numbers in a binary file. The list is first converted in a byte array before writing. The built-in function byte array() returns a byte representation of the object.

Example: Write to a Binary File

```
f=open("binfile.bin", "wb")
num=[5, 10, 15, 20, 25]
arr=byte array(num)
f.write(arr)
f.close()
```

To read the above binary file, the output of the read() method is casted to a list using the list() function.

Example: Reading a Binary File

```
f=open("binfile.bin","rb")
num=list(f.read())
Print (num)
f.close ()
```

CASE STUDIES: COUNTING A CHARACTER IN A FILE

Write a Python program to count the number of each character of a text file.

Inputs:

abc.txt -

German Unity Day: The Day of German Unity (German: Tag der DeutschenEinheit) is the national day of Germany, celebrated on 3 October as a public holiday. It commemorates the anniversary of German reunification in 1990, when the goal of a united Germany that originated in the middle of the 19th century, was fulfilled again. Therefore, the name addresses neither the re-union nor the union, but the unity of Germany. The Day of German Unity on 3 October has been the German national holiday since 1990, when the reunification was formally completed.

Sample Solution:
Python Code:

```
import collections
import pprint
file_input =input('File Name: ')
withopen(file_input,'r')as info:
  count = collections.Counter(info.read().upper())
  value = pprint.pformat(count)
print(value)
Sample output:
File Name:  abc.txt
Counter({' ': 93,
        'E': 64,
        'N': 45,
        'A': 42,
        'T': 40,
        'I': 36,
        'O': 31,
        'R': 29,
        'H': 25,
        'D': 19,
        'M': 17,
        'Y': 17,
        'L': 15,
        'F': 15,
        'U': 14,
        'C': 13,
        'G': 13,
```

'S': 12,
',': 7,
'B': 6,
'W': 5,
'9': 5,
'.': 4,
'P': 4,
'1': 3,
'\n': 2,
'0': 2,
'3': 2,
':': 1,
'-': 1,
'K': 1,
'(': 1,
')': 1,
'V': 1})

CLIENT SERVER ARCHITECTURE IN PYTHON

1. **socket.socket()**: Create a new socket using the given address family, socket type and protocol number.

2. **socket.bind(address)**: Bind the socket to address.

3. **socket.listen(backlog)**: Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5), the minimum value is forced to 0.

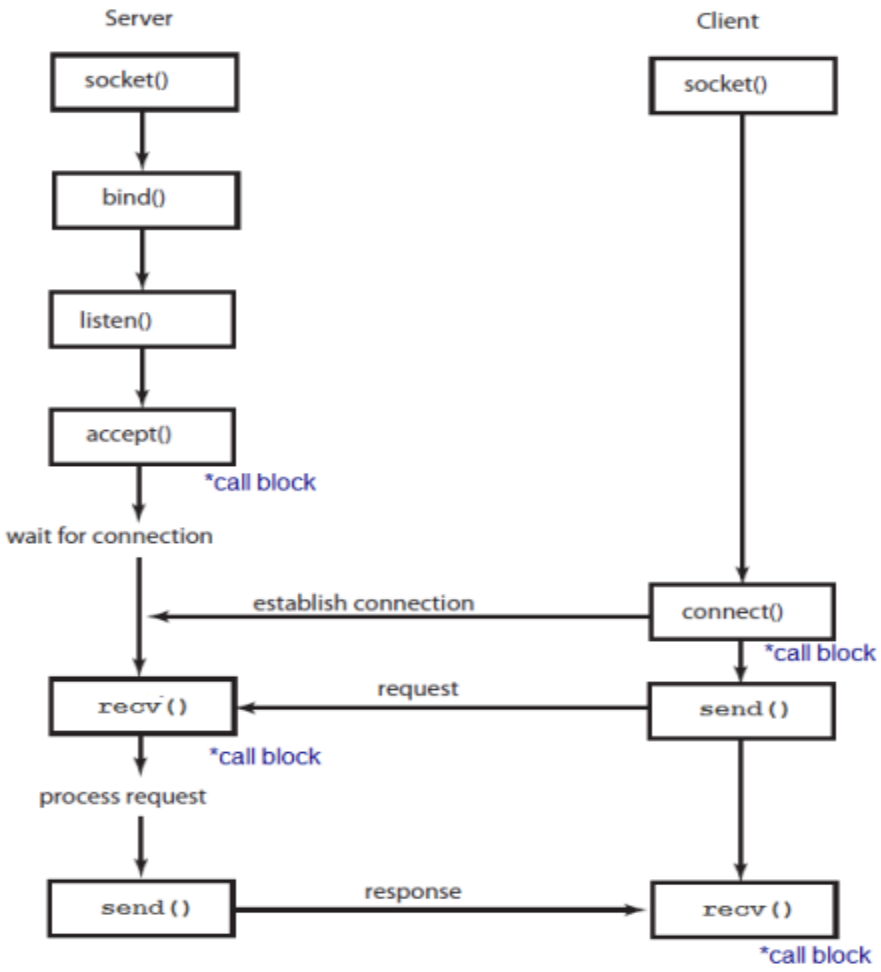
4. **socket.accept()**: The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

At accept(), a new socket is created that is distinct from the named socket. This new socket is used solely for communication with this particular client.

For TCP servers, the socket object used to receive connections is not the same socket used to perform subsequent communication with the client. In particular, the accept() system call returns a new socket object that's actually used for the connection. This allows a server to manage connections from a large number of clients simultaneously.

5. **socket.send(bytes[, flags])**: Send data to the socket. The socket must be connected to a remote socket. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

6. **socket.colse():** Mark the socket closed. all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected, but it is recommended to close() them explicitly.



CLIENT SERVER SOCKET PROGRAMMING

Procedure (Server): The server will host the network using the assigned host and port using which the client will connect to the server. The server will act as the sender and the user will have to input the filename of the file that he/she would like to transmit. The user must make sure that the file that needs to be sent is in the same directory as the "server.py" program.

Procedure (Client): The client program will prompt the user to enter the host address of the server while the port will already be assigned to the port variable. Once the client program has connected to the server it will ask the user for a filename to be used for the file that will be received from the server. Lastly the client program will receive the file and leave it in the same directory under the same filename set as the user.

Server Coding:

```
import socket
s = socket.socket()
host = socket.gethostname()   #Get localhost IP address
port = 8080                   #assign port for session
s.bind((host,port))           #bind the socket to assigned host IP and port
s.listen(1)                   #put the socket into listening mode for 1 client
print(host)
print("Waiting for any incoming connection... ")
conn, addr = s.accept()
print(addr, "Has connected to the server")
filename = input(str("Enter the name of the file to be transmitted: "))
file = open(filename , 'rb')      # Opens a file for reading only in binary format
file_data = file.read(1024)
conn.send(file_data)
print("File has been transmitted successfully")
```

Client Coding:

```
import socket
s = socket.socket()
host = input(str("Please enter the host address of the sender: "))
port = 8080
s.connect((host,port))
print("Connected ... ")
filename = input(str("Please enter a filename for the incoming file: "))
file = open(filename, 'wb')       # Opens a file for writing only in binary format
file_data = s.recv(1024)
file.write(file_data)
file.close()
print("File has been received successfully.")
```

TCP AND UDP IN TRANSPORT LAYER

Layer 3 or the Network layer uses IP or Internet Protocol which being a connection less protocol treats every packet individually and separately leading to lack of reliability during a transmission. For example, when data is sent from one host to another, each packet may take a different path even if it belongs to the same session. This means the packets may/may not arrive in the right order. Therefore, IP relies on the higher layer protocols to provide reliability.

TCP (Transmission Control Protocol):

Transmission Control Protocol (TCP) – a connection-oriented communications protocol that facilitates the exchange of messages between computing devices in a network. It is the most common protocol in networks that use the Internet Protocol (IP); together they are sometimes referred to as TCP/IP.

TCP takes messages from an application/server and divides them into packets, which can then be forwarded by the devices in the network – switches, routers, security gateways – to the destination. TCP numbers each packet and reassembles them prior to handing them off to the application/server recipient. Because it is connection-oriented, it ensures a connection is established and maintained until the exchange between the application/servers sending and receiving the message is complete.

UDP (User Datagram Protocol):

UDP is also a layer 4 protocol but unlike TCP it doesn't provide acknowledgement of the sent packets. Therefore, it isn't reliable and depends on the higher layer protocols for the same. But on the other hand it is simple, scalable and comes with lesser overhead as compared to TCP. It is used in video and voice streaming.

TCP	UDP
Keeps track of lost packets. Makes sure that lost packets are re-sent	Doesn't keep track of lost packets
Adds sequence numbers to packets and reorders any packets that arrive in the wrong order	Doesn't care about packet arrival order
Slower, because of all added additional functionality	Faster, because it lacks any extra features
Requires more computer resources, because the OS needs to keep track of ongoing communication sessions and manage them on a much deeper level	Requires less computer resources
Examples of programs and services that use TCP: - HTTP - HTTPS - FTP - Many computer games	Examples of programs and services that use UDP: - DNS - IP telephony - DHCP - Many computer games

TWISTED NETWORK FRAMEWORK

Twisted is a framework for writing asynchronous, event-driven networked programs in Python -- both clients and servers. In addition to abstractions for low-level system calls like select and socket, it also includes a large number of utility functions and classes, which make writing new servers easy. Twisted includes support for popular network protocols like HTTP and SMTP, support for GUI frameworks like GTK+/GNOME and Tk and many other classes designed to make network programs easy. Whenever possible, Twisted uses Python's introspection facilities to save the client programmer as much work as possible. Even though Twisted is still work in progress, it is already usable for production systems -- it can be used to bring up a Web server, a mail server or an IRC server in a matter of minutes, and require almost no configuration.

Python lends itself to writing frameworks. Python has a simple class model, which facilitates inheritance. It has dynamic typing, which means code needs to assume less. Python also has built-in memory management, which means application code does not need to track ownership. Thus, when writing a new application, a programmer often finds himself writing a framework to make writing this kind of application easier. Twisted evolved from the need to write high-performance interoperable servers in Python, and making them easy to use (and difficult to use incorrectly).

There are three ways to write network programs:

1. Handle each connection in a separate process
2. Handle each connection in a separate thread
3. Use non-blocking system calls to handle all connections in one thread.

USENET

Usenet is a worldwide distributed discussion system available on computers. It was developed from the general-purpose Unix-to-Unix Copy (UUCP) dial-up network architecture. Users read and post messages (called articles or posts, and collectively termed news) to one or more categories, known as newsgroups. Usenet resembles a bulletin board system (BBS) in many respects and is the precursor to Internet forums that are widely used today. Discussions are threaded, as with web forums and BBSs, though posts are stored on the server sequentially. The name comes from the term "users network"

NEWSGROUP – E-MAIL

comp.lang.python

comp.lang.python is a high-volume Usenet open (not moderated) newsgroup for general discussions and questions about Python. You can also access it as a mailing list through python-list.

Pretty much anything Python-related is fair game for discussion, and the group is even fairly tolerant of off-topic digressions; there have been entertaining discussions of topics such as floating point, good software design, and other programming languages such as Lisp and Forth.

Most discussion on comp.lang.python is about developing with Python, not about development of the Python interpreter itself. Some of the core developers still read the list, but most of them don't. Occasionally comp.lang.python suggestions have resulted in an enhancement proposal being written, leading to a new Python feature. If you find a bug in Python, don't send it to comp.lang.python; file a bug report in the issue tracker.

SIMPLE MAIL TRANSFER PROTOCOL (SMTP)

SMTP is a protocol, which handles sending e-mail and routing e-mail between mail servers.

Python provides smtplib module, which defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon.Here is a simple syntax to create one SMTP object, which can later be used to send an e-mail.

```
import smtplib
smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

Here is the detail of the parameters −

host − This is the host running your SMTP server. You can specify IP address of the host or a domain name like tutorialspoint.com. This is optional argument.

port − If you are providing host argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.

local_hostname − If your SMTP server is running on your local machine, then you can specify just localhost as of this option.

An SMTP object has an instance method called sendmail, which is typically used to do the work of mailing a message. It takes three parameters −

The sender − A string with the address of the sender.

The receivers − A list of strings, one for each recipient.

The message − A message as a string formatted as specified in the various RFCs

POP3

To receive email you can write a MUA(Mail User Agent) as the client, and retrieve the email from MDA (Mail Delivery Agent) to the user's computer or mobile phone. The most commonly used protocol for receiving mail is POP protocol. The current version number is 3, commonly known as POP3. Python has a built-in poplib module, which implements POP3 protocol and can be used to receive mail directly.

The POP3 protocol does not receive the original readable message itself, but the encoded text of the message that SMTP sent. So in order to turn the text received by POP3 into a readable email, it is necessary to parse the original text with various classes provided by the email module and turn it into a readable email object. So there are two steps for you to receive email from a pop3 server in Python.

The pop3 protocol is an email protocol to download messages from the email-server. These messages can be stored in the local machine.

Key Points

- POP is an application layer internet standard protocol.
- Since POP supports offline access to the messages, thus requires less internet usage time.
- POP does not allow search facility.
- In order to access the messaged, it is necessary to download them.
- It allows only one mailbox to be created on server.
- It is not suitable for accessing non mail data.

18CSPC405 – Python Programming

Unit V –Database and GUI

The dbm package in Python's built-in library provides a dictionary like an interface DBM style databases. The dbm library is a simple database engine, written by Ken Thompson. DBM stands for DataBase Manager, used by UNIX operating system, the library stores arbitrary data by use of a single key (a primary key) in fixed-size buckets and uses hashing techniques to enable fast retrieval of the data by key.

There are following modules in dbm package:

The dbm.ndbm module provides an interface to the Unix "(n)dbm" library. Dbm objects behave like dictionaries, with keys and values should be stored as bytes. The module doesn't support and the items() and values() methods.

The dbm.dumb module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as dbm.gnu no external library is required. As with other persistent mappings, the keys and values are always stored as bytes.

These modules are internally used by Python's shelve module. As in the case of shelve database, user-specified database name carries '.dir' postfix. The dbm object's whichdb() function tells which implementation of dbm is available on current Python installation.

```
>>> dbm.whichdb('mydbm.db')

'dbm.dumb'

>>> db = dbm.open('mydbm.db','n')

>>> db['name'] = Rajani Deshmukh'

>>> db['address'] = 'Shivajinagar Pune'

>>> db['PIN'] = '431001'

>>> db.close()
```

The open() function allows mode these flags −

Value	Meaning
'r'	Open an existing database for reading only (default)
'w'	Open an existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn't exist
'n'	Always create a new, empty database, open for reading and writing

A dbm object is a dictionary like an object, just as a shelf object. Hence all dictionary operations can be performed. The dbm object can invoke get(),pop(), append(0 and update()

methods. Following code opens 'mydbm.db' with 'r' flag and iterates over the collection of key-value pairs.

```
>>> db = dbm.open('mydbm.db','r')
>>> for k,v in db.items():
print (k,v)
```

Output:

```
b'name' : Rajani Deshmukh'
b'address' : b'Shivajinagar Pune'
b'PIN' : b'431001'
```

dbm objects also provide the following methods −

sync(): Synchronize the on-disk directory and data files. This method is called by the Shelve.sync() method.

close(): Close the dbm database.

gnu dbm objects have the following methods −

firstkey()

It's possible to loop over every key in the database using this method and the nextkey() method. This method returns the starting key.

gdbm.nextkey(key): Returns the key that follows key in the traversal.

gdbm.reorganize(): this function will reorganize the database. gnu dbm objects will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

SQL Database

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.

Database Connection

Before connecting to a MySQL database, make sure of the followings −

- You have created a database TESTDB.
- You have created a table EMPLOYEE in TESTDB.
- This table has fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.
- Python module MySQLdb is installed properly on your machine.
- You have gone through MySQL tutorial to understand MySQL Basics.

Example

Following is the example of connecting with MySQL database "TESTDB"

```
import MySQLdb

# Open database connection

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method

cursor = db.cursor()

# execute SQL query using execute() method.

cursor.execute("SELECT VERSION()"

# Fetch a single row using fetchone() method.

data = cursor.fetchone()

print ("Database version : %s " ,% data)

# disconnect from server

db.close()
```

If a connection is established with the data source, then a Connection Object is returned and saved into **db** for further use, otherwise **db** is set to None. Next, **db** object is used to create a **cursor** object, which in turn is used to execute SQL queries. Finally, before coming out, it ensures that database connection is closed and resources are released.

Creating Database Table

Once a database connection is established, we are ready to create tables or records into the database tables using **execute** method of the created cursor.

Example

Let us create Database table EMPLOYEE −

```
import MySQLdb

# Open database connection

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
```

```
cursor = db.cursor()

# Drop table if it already exist using execute() method.

cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# Create table as per requirement

sql = """CREATE TABLE EMPLOYEE (FIRST_NAME  CHAR(20) NOT NULL,

LAST_NAME  CHAR(20), AGE INT,  SEX CHAR(1), INCOME FLOAT )"""

cursor.execute(sql)

# disconnect from server

db.close()
```

INSERT Operation

It is required when you want to create your records into a database table.

Example

The following example, executes SQL *INSERT* statement to create a record into EMPLOYEE table −

```
import MySQLdb

# Open database connection

db = MySQLdb.connect("localhost","testuser","test123","TESTDB"

# prepare a cursor object using cursor() method

cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.

sql = """INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME) VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""

try:

   # Execute the SQL command

   cursor.execute(sql)
```

```
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
# disconnect from server
db.close()
```

Above example can be written as follows to create SQL queries dynamically −

```
import MySQLdb
# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
# prepare a cursor object using cursor() method
cursor = db.cursor()
# Prepare SQL query to INSERT a record into the database.
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES ('%s', '%s', '%d', '%c', '%d' )" % ('Mac', 'Mohan', 20, 'M', 2000)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
# disconnect from server
```

db.close()

Example

Following code segment is another form of execution where you can pass parameters directly −

.....

user_id = "test123"

password = "password"

con.execute('insert into Login values("%s", "%s")' % (user_id, password))

.....

READ Operation

READ Operation on any database means to fetch some useful information from the database.

Once our database connection is established, you are ready to make a query into this database.

You can use either **fetchone()** method to fetch single record or **fetchall()** method to fetech multiple values from a database table.

- **fetchone()** − It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- **fetchall()** − It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.
- **rowcount** − This is a read-only attribute and returns the number of rows that were affected by an execute() method.

Example

The following procedure queries all the records from EMPLOYEE table having salary more than 1000 −

import MySQLdb

Open database connection

db = MySQLdb.connect("localhost","testuser","test123","TESTDB")

prepare a cursor object using *cursor()* method

cursor = db.cursor()

sql = "SELECT * FROM EMPLOYEE WHERE INCOME > '%d'" % (1000)

```
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Fetch all the rows in a list of lists.
   results = cursor.fetchall()
   for row in results:
      fname = row[0]
      lname = row[1]
      age = row[2]
      sex = row[3]
      income = row[4]
      # Now print fetched result
      print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % (fname, lname, age, sex,
income )
except:
   print "Error: unable to fecth data"
# disconnect from server
db.close()
```

This will produce the following result −

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

Update Operation

UPDATE Operation on any database means to update one or more records, which are already available in the database.

The following procedure updates all the records having SEX as **'M'**. Here, we increase AGE of all the males by one year.

Example

```
import MySQLdb
```

```
# Open database connection

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method

cursor = db.cursor()

# Prepare SQL query to UPDATE required records

sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = '%c'" % ('M')

try:

   # Execute the SQL command

   cursor.execute(sql)

   # Commit your changes in the database

   db.commit()

except:

   # Rollback in case there is any error

   db.rollback()

# disconnect from server

db.close()
```

DELETE Operation

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where AGE is more than 20 −

Example

```
import MySQLdb

# Open database connection

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method

cursor = db.cursor()
```

```
# Prepare SQL query to DELETE required records

sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)

try:

   # Execute the SQL command

   cursor.execute(sql)

   # Commit your changes in the database

   db.commit()

except:

   # Rollback in case there is any error

   db.rollback()

# disconnect from server

db.close()
```

Performing Transactions

Transactions are a mechanism that ensures data consistency. Transactions have the following four properties −

- **Atomicity** − Either a transaction completes or nothing happens at all.
- **Consistency** − A transaction must start in a consistent state and leave the system in a consistent state.
- **Isolation** − Intermediate results of a transaction are not visible outside the current transaction.
- **Durability** − Once a transaction was committed, the effects are persistent, even after a system failure.

The Python DB API 2.0 provides two methods to either *commit* or *rollback* a transaction.

Example

You already know how to implement transactions. Here is again similar example −

```
# Prepare SQL query to DELETE required records

sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)

try:

   # Execute the SQL command
```

```
   cursor.execute(sql)

   # Commit your changes in the database

   db.commit()

except:

   # Rollback in case there is any error

   db.rollback()
```

COMMIT Operation

Commit is the operation, which gives a green signal to database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call **commit** method.

```
db.commit()
```

ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use **rollback()** method.

Here is a simple example to call **rollback()** method.

```
db.rollback()
```

Disconnecting Database

To disconnect Database connection, use close() method.

```
db.close()
```

If the connection to a database is closed by the user with the close() method, any outstanding transactions are rolled back by the DB. However, instead of depending on any of DB lower level implementation details, your application would be better off calling commit or rollback explicitly.

Handling Errors

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.

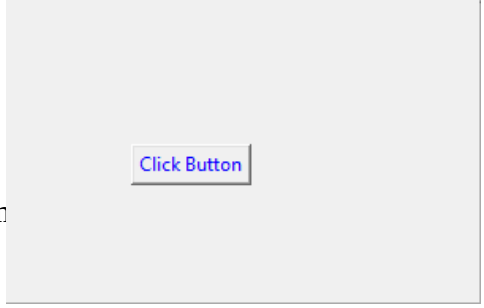
The DB API defines a number of errors that must exist in each database module. The following table lists these exceptions.

Sr.No.	Exception & Description
1	**Warning** Used for non-fatal issues. Must subclass StandardError.
2	**Error** Base class for errors. Must subclass StandardError.
3	**InterfaceError** Used for errors in the database module, not the database itself. Must subclass Error.
4	**DatabaseError** Used for errors in the database. Must subclass Error.
5	**DataError** Subclass of DatabaseError that refers to errors in the data.
6	**OperationalError** Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally outside of the control of the Python scripter.
7	**IntegrityError** Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys.
8	**InternalError** Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.
9	**ProgrammingError** Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you.
10	**NotSupportedError** Subclass of DatabaseError that refers to trying to call unsupported functionality.

Python - GUI Programming (Tkinter)

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

- **Tkinter**
- **wxPython**
- **JPython**



Out of all the GUI methods, Tkinter is the m

Tkinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

Creating a GUI application using Tkinter is an easy task. A GUI application is created using Tkinter as follows:

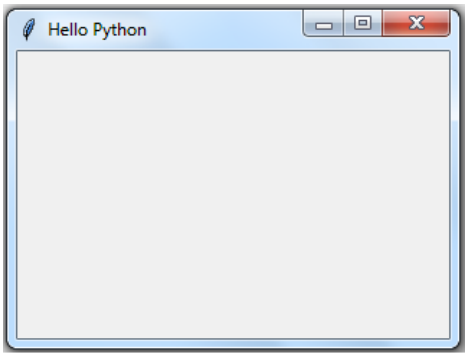
1. Import the Tkinter module using import Tkinter.
2. Create the GUI application main window using window =Tk( ).
3. Add one or more widgets to the GUI application.
4. Enter the main event loop to take action against each event triggered by the user.

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets. There are currently 15 types of widgets in Tkinter.

Example

```
import Tkinter

window = Tkinter.Tk()

window.title('Hello Python')

window.geometry("300x200+10+20")

window. mainloop()
```

Output:



1. import the TKinter module.
2. After importing, setup the application object by calling the Tk() function. This will create a top-level window (root) having a frame with a title bar, control box with the minimize and close buttons.
3. The geometry() method defines the width, height and coordinates of the top left corner of the frame as below:
   window.geometry("widthxheight+XPOS+YPOS")

Geometry Management

All Tkinter widgets have access to specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

The *pack()* Method − This geometry manager organizes widgets in blocks before placing them in the parent widget.

The *grid()* Method − This geometry manager organizes widgets in a table-like structure in the parent widget.

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

Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets. There are currently 15 types of widgets in Tkinter.

S. No.	Widget	Description about Widget
1.	Button	The Button widget is used to display buttons in your application
2.	Canvas	The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
3.	Checkbutton	The Check button widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
4.	Label	The Label widget is used to provide a single-line caption for other widgets. It can also contain images.
5.	Listbox	The List box widget is used to provide a list of options to a user.
6.	Menubutton	The Menu button widget is used to display menus in your application.
7.	Radiobutton	The Radio button widget is used to display a number of options as radio buttons. The user can select only one option at a time.
8.	Scrollbar	The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.
9.	Message	The Message widget is used to display multiline text fields for accepting values from a user.
10.	Text	The Text widget is used to display text in multiple lines.
11.	tkMessageBox	This module is used to display message boxes in your applications.

Button Widget

The button can be created using the Button class. The Button class constructor requires a reference to the main window and to the options.

Syntax:

Button(window, attributes)

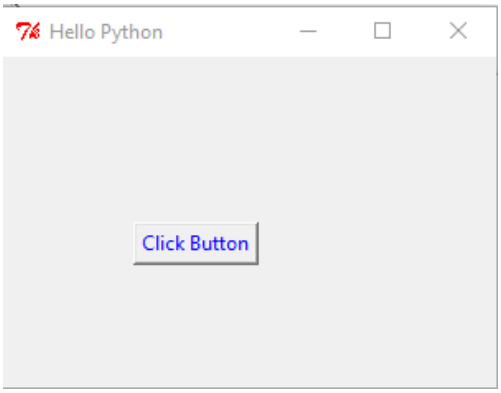
Optional Attributes:

- text : caption of the button
- bg : background colour
- fg : foreground colour
- font : font name and size
- image : to be displayed instead of text
- command : function to be called when clicked

Example:

```
from tkinter import *
window=Tk()
btn=Button(window, text="Click Button", fg='blue')
btn.place(x=80, y=100)
window.title('Hello Python')
window.geometry("300x200+10+10")
window.mainloop()
```

Output:



Label

A label can be created using the Label class. Option parameters are similar to the Button object.

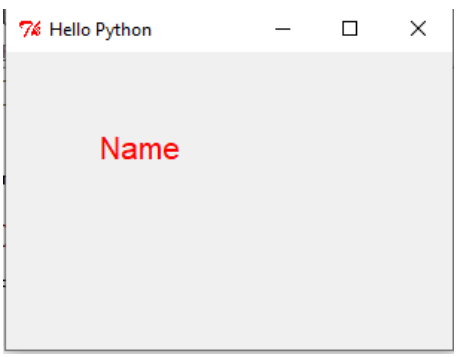
Syntax:

Label(window, attributes)

Example:

```
from tkinter import *
window=Tk()
lbl=Label(window, text="Name", fg='red', font=("Helvetica", 16))
lbl.place(x=60, y=50)
window.title('Hello Python')
window.geometry("300x200+10+10")
window.mainloop()
```

Output:



Entry

Entry widget used to accepting the user input in single-line text box. For multi-line text input use the Text widget.

- bd : border size of the text box; default is 2 pixels.
- show : to convert the text box into a password field, set show property to "*".

Syntax:

txtfld=Entry(window, text="This is Entry Widget", bg='black',fg='white', bd=5)

Selection Widgets

Radio button:

The Radio button widget is used to display a number of options as radio buttons. The user can select only one option at a time.

Check button:

The Check button widget is used to display a number of options as checkboxes. The user can select multiple options at a time.

Combo box:

This class is defined in the ttk module of tkinter package. It populates drop down data from a collection data type, such as a tuple or a list as values parameter.

List box:

Unlike Combobox, this widget displays the entire collection of string items. The user can select one or multiple items.

Canvas:

You can use the methods **create_rectangle**, **create_oval**, **create_arc**, **create_polygon**, or **create_line** to draw a rectangle, oval, arc, polygon, or line on a canvas.

Event Handling

In Tkinter, there are two ways to register an event with a widget.

First way is by using the bind() method and the second way is by using the command parameter in the widget constructor.

Bind() Method

The bind() method associates an event to a call back function so that, when the even occurs, the function is called.

Syntax:

Widget.bind(event, callback)

For example, to invoke the MyButtonClicked() function on left button click, use the following code:

Example: Even Binding

```
from tkinter import *
window=Tk()
btn = Button(window, text='OK')
btn.bind('<Button-1>', MyButtonClicked)
```

Command Parameter

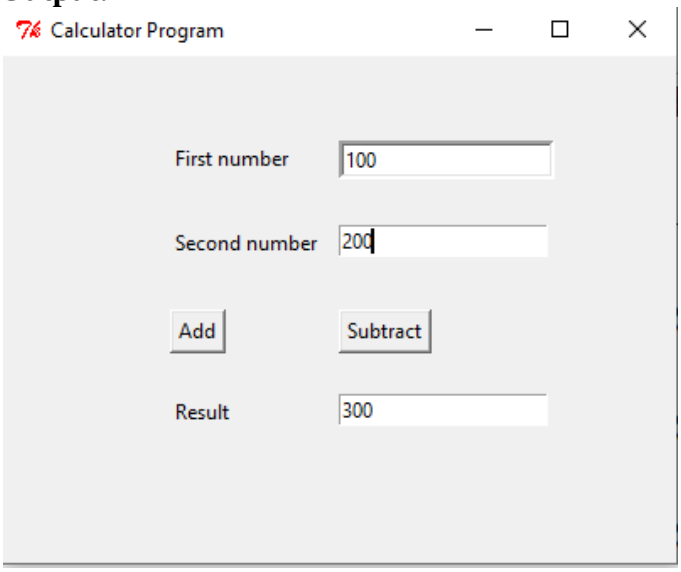
Constructor methods of many widget classes have an optional parameter called command. This command parameter is set to callback the function which will be invoked whenever its bound event occurs. This method is more convenient than the bind() method.

```
btn = Button(window, text='OK', command=myEventHandlerFunction)
```

Example:

```
from Tkinter import *
class MyWindow:
    def __init__(self, win):
        self.lbl1=Label(win, text='First number')
        self.lbl2=Label(win, text='Second number')
        self.lbl3=Label(win, text='Result')
        self.t1=Entry(bd=3)
        self.t2=Entry()
        self.t3=Entry()
        self.btn1 = Button(win, text='Add')
        self.btn2=Button(win, text='Subtract')
        self.lbl1.place(x=100, y=50)
        self.t1.place(x=200, y=50)
        self.lbl2.place(x=100, y=100)
        self.t2.place(x=200, y=100)
        self.b1=Button(win, text='Add', command=self.add)
        self.b2=Button(win, text='Subtract')
        self.b2.bind('<Button-1>', self.sub)
        self.b1.place(x=100, y=150)
        self.b2.place(x=200, y=150)
        self.lbl3.place(x=100, y=200)
        self.t3.place(x=200, y=200)
    def add(self):
        self.t3.delete(0, 'end')
        num1=int(self.t1.get())
        num2=int(self.t2.get())
        result=num1+num2
        self.t3.insert(END, str(result))
    def sub(self, event):
        self.t3.delete(0, 'end')
        num1=int(self.t1.get())
        num2=int(self.t2.get())
        result=num1-num2
        self.t3.insert(END, str(result))

window=Tk()
mywin=MyWindow(window)
window.title('Hello Python')
window.geometry("400x300+10+10")
window.mainloop()
```

Output:



Displaying Images

You can add an image to a label, button, check button, or radio button.

To create an image, use the PhotoImage class as follows:

photo = PhotoImage(file = imagefilename)

The image file must be in GIF format. You can use a conversion utility to convert image files in other formats into GIF format.

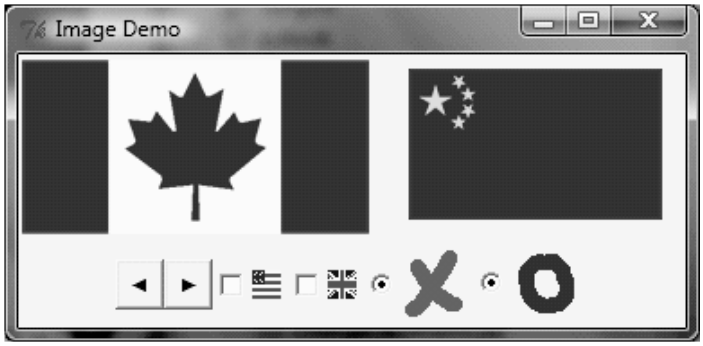
Listing 9.12 shows you how to add images to labels, buttons, check buttons, and radio buttons.

You can also use the create_image method to display an image in a canvas, as shown in Figure 9.13.

```
from tkinter import * # Import all definitions from tkinter
class ImageDemo:
    def _ _init_ _(self):
        window = Tk() # Create a window
        window.title("Image Demo") # Set title
        chinaImage = PhotoImage(file = "image/china.gif")
        leftImage = PhotoImage(file = "image/left.gif")
        rightImage = PhotoImage(file = "image/right.gif")
        usImage = PhotoImage(file = "image/usIcon.gif")
        ukImage = PhotoImage(file = "image/ukIcon.gif"
        crossImage = PhotoImage(file = "image/x.gif")
        circleImage = PhotoImage(file = "image/o.gif")
        frame1 = Frame(window)
        frame1.pack()
        Label(frame1, ).pack(side = LEFT)
        canvas = Canvas(frame1)
        canvas.create_image(90, 50, image = chinaImage)

        canvas["width"] = 200
        canvas["height"] = 100
```

```
        canvas.pack(side = LEFT)
        frame2 = Frame(window)
        frame2.pack()
        Button(frame2, image = leftImage).pack(side = LEFT)
        Button(frame2, image = rightImage).pack(side = LEFT)
        Checkbutton(frame2, image = ukImage).pack(side = LEFT)
       Checkbutton(frame2, image = usImage).pack(side = LEFT)
        Radiobutton(frame2, image = circleImage).pack(side = LEFT)
        Radiobutton(frame2, image = crossImage).pack(side = LEFT)

    window.mainloop() # Create an event loop
ImageDemo() # Create GUI
```

Output:



The program places image files in the image folder in the current program directory, then creates **PhotoImage** objects for several images in lines 9–16. These objects are used in widgets.

The image is a property in **Label**, **Button**, **Checkbutton**, and **RadioButton**.

Image is not a property for **Canvas**, but you can use the **create_image** method to display an image on the canvas (line 23). In fact, you can display multiple images in one canvas.

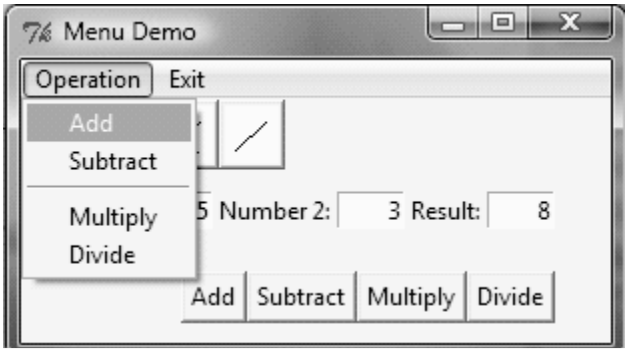
Menus

Tkinter provides a comprehensive solution for building graphical user interfaces. This section introduces menus, popup menus, and toolbars.

Menus make selection easier and are widely used in windows. You can use the Menu class to create a menu bar and a menu, and use the add_command method to add items to the menu.

```
from tkinter import *
class MenuDemo:
    def _ _init_ _(self):
    window = Tk()
    window.title("Menu Demo")
    menubar = Menu(window)
    window.config(menu = menubar)
    operationMenu = Menu(menubar, tearoff = 0)
    menubar.add_cascade(label = "Operation", menu = operationMenu)
    operationMenu.add_command(label = "Add", command = self.add)
    operationMenu.add_command(label = "Subtract", command = self.subtract)
    operationMenu.add_separator()
    operationMenu.add_command(label = "Multiply", command = self.multiply)
    operationMenu.add_command(label = "Divide", command = self.divide)
```

```
        exitmenu = Menu(menubar, tearoff = 0)
        menubar.add_cascade(label = "Exit", menu = exitmenu)
        exitmenu.add_command(label = "Quit", command = window.quit)
        frame0 = Frame(window) # Create and add a frame to window
        plusImage = PhotoImage(file = "image/plus.gif") create an image
        minusImage = PhotoImage(file = "image/minus.gif")
        timesImage = PhotoImage(file = "image/times.gif")
        divideImage = PhotoImage(file = "image/divide.gif")
        Button(frame0, image = plusImage, command = self.add).grid(row = 1, column     = 1,
sticky = W)
        Button(frame0, image = minusImage, command = self.subtract).grid(row = 1, column =
2)
        Button(frame0, image = timesImage, command = self.multiply).grid(row = 1, column =
3)
        Button(frame0, image = divideImage, command = self.divide).grid(row = 1, column =
4)
         frame1 = Frame(window)
         frame1.grid(row = 2, column = 1, pady = 10)
         Label(frame1, text = "Number 1:").pack(side = LEFT)
         self.v1 = StringVar()
         Entry(frame1, width = 5, textvariable = self.v1, justify = RIGHT).pack(side = LEFT)
         Label(frame1, text = "Number 2:").pack(side = LEFT)
        self.v2 = StringVar()
        Entry(frame1, width = 5, textvariable = self.v2, justify = RIGHT).pack(side = LEFT)
        Label(frame1, text = "Result:").pack(side = LEFT)
        self.v3 = StringVar()
        Entry(frame1, width = 5, textvariable = self.v3, justify = RIGHT).pack(side = LEFT)
        frame2 = Frame(window) # Create and add a frame to window
        frame2.grid(row = 3, column = 1, pady = 10, sticky = E)
        Button(frame2, text = "Add", command = self.add).pack(side=LEFT)
        Button(frame2, text = "Subtract", command = self.subtract).pack(side = LEFT)
        Button(frame2, text = "Multiply", command = self.multiply).pack(side = LEFT)
        Button(frame2, text = "Divide", command = self.divide).pack(side = LEFT)
mainloop()
def add(self):
      self.v3.set(eval(self.v1.get()) + eval(self.v2.get()))
def subtract(self):
        self.v3.set(eval(self.v1.get()) - eval(self.v2.get()))
def  multiply(self):
        self.v3.set(eval(self.v1.get()) * eval(self.v2.get()))
def divide(self):
        self.v3.set(eval(self.v1.get()) / eval(self.v2.get()))
MenuDemo()
```



Popup Menus

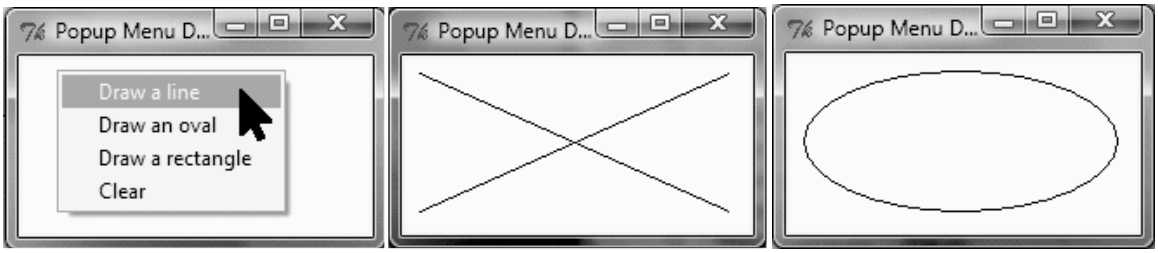
A popup menu, also known as a context menu, is like a regular menu, but it does not have a menu bar and it can float anywhere on the screen.

Creating a popup menu is similar to creating a regular menu. First, you create an instance of

Menu, and then you can add items to it. Finally, you bind a widget with an event to pop up the menu.

```
from tkinter import * # Import all definitions from tkinter
class PopupMenuDemo:
def _ _init_ _(self):
    window = Tk() # Create a window
    window.title("Popup Menu Demo") # Set title
    self.menu = Menu(window, tearoff = 0)
    self.menu.add_command(label = "Draw a line",
   command = self.displayLine)
   self.menu.add_command(label = "Draw an oval",
   command = self.displayOval)
  self.menu.add_command(label = "Draw a rectangle",
  command = self.displayRect)
  self.menu.add_command(label = "Clear",
  command = self.clearCanvas)
  self.canvas = Canvas(window, width = 200, height = 100, bg = "white")
  self.canvas.pack()
  self.canvas.bind("<Button-3>", self.popup)
  window.mainloop() # Create an event loop
```

```
def displayRect(self):
    self.canvas.create_rectangle(10, 10, 190, 90, tags = "rect")
def displayOval(self):
    self.canvas.create_oval(10, 10, 190, 90, tags = "oval")
def displayLine(self):
    self.canvas.create_line(10, 10, 190, 90, tags = "line")
    self.canvas.create_line(10, 90, 190, 10, tags = "line")
def clearCanvas(self):
    self.canvas.delete("rect", "oval", "line")
def popup(self, event):
    self.menu.post(event.x_root, event.y_root)
PopupMenuDemo() # Create GUI
```



A canvas is created to display the shapes. The menu items use callback functions to instruct the canvas to draw shapes.

Mouse, Key Events, and Bindings

You can use the **bind** *method to bind mouse and key events to a widget.*

The preceding example used the widget's **bind** method to bind a mouse event with a callback handler by using the syntax:

widget.bind(event, handler)

If a matching event occurs, the handler is invoked. In the preceding example, the event is

<Button-3> and the handler function is **popup**. The event is a standard Tkinter object,

which is automatically created when an event occurs. Every handler has an event as its argument.

The following example defines the handler using the event as the argument:

menu.post(event.x_root, event.y_root)

The **event** object has a number of properties describing the event pertaining to the event.

For example, for a mouse event, the **event** object uses the **x**, **y** properties to capture the current mouse location in pixels.

The mouse and key events are processed and the processing information is displayed in the command window.

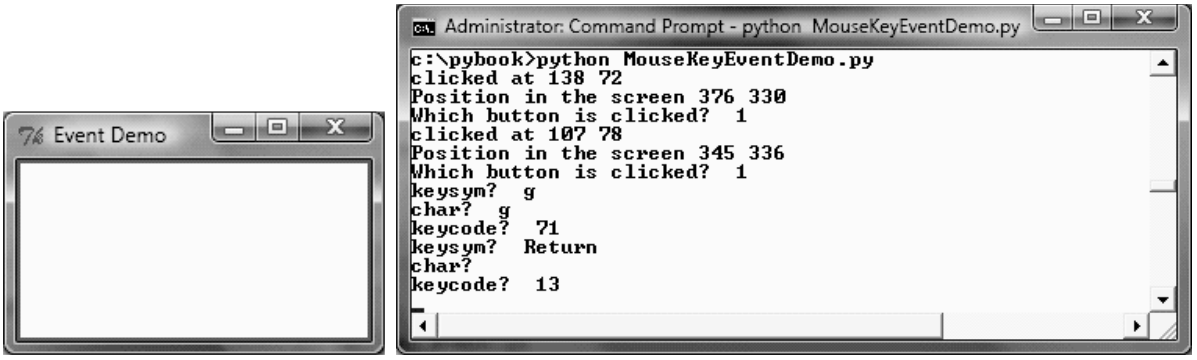
Event Description

Event	Descriptions
<Bi-Motion>	An event occurs when a mouse button is moved while being held down on the widget.
<Button-i>	Button-1, Button-2, and Button-3 identify the left, middle, and right buttons. When a mouse button is pressed over the widget, Tkinter automatically grabs the mouse pointer's location. Button Pressed-*i* is synonymous with Button-*i*.
<ButtonReleased-i>	An event occurs when a mouse button is released.
<Double-Button-i>	An event occurs when a mouse button is double-clicked.
<Enter>	An event occurs when a mouse pointer enters the widget.
<Key>	An event occurs when a key is pressed.
<Leave>	An event occurs when a mouse pointer leaves the widget.
<Return>	An event occurs when the *Enter* key is pressed. You can bind any key such as *A*, *B*, *Up*, *Down*, *Left*, *Right* in the keyboard with an event.
<Shift+A>	An event occurs when the *Shift+A* keys are pressed. You can combine *Alt*, *Shift*, and *Control* with other keys.
<Triple-Button-i>	An event occurs when a mouse button is triple-clicked.

```
from tkinter import * # Import all definitions from tkinter

class MouseKeyEventDemo:

    def _ _init_ _(self):

    window = Tk() # Create a window

    window.title("Event Demo") # Set a title

    canvas = Canvas(window, bg = "white", width = 200, height = 100)

    canvas.pack()
```

```
        canvas.bind("<Button-1>", self.processMouseEvent)
        canvas.bind("<Key>", self.processKeyEvent)
        canvas.focus_set()
        window.mainloop() # Create an event loop
    def processMouseEvent(self, event):
        print("clicked at", event.x, event.y)
        print("Position in the screen", event.x_root, event.y_root)
        print("Which button is clicked? ", event.num)
    def processKeyEvent(self, event):
        print("keysym? ", event.keysym)
        print("char? ", event.char)
        print("keycode? ", event.keycode)
MouseKeyEventDemo() # Create GUI
```

Output:



Animations

Animations can be created by displaying a sequence of drawings.

The **Canvas** class can be used to develop animations. You can display graphics and text on the canvas and use the **move(tags, dx, dy)** method to move the graphic with the specified tags **dx** pixels to the right if **dx** is positive and **dy** pixels down if **dy** is positive. If **dx** or **dy** is negative, the graphic is moved left or up.

AnimationDemo.py

```
from tkinter import * # Import all definitions from tkinter
```

```
class AnimationDemo:
    def _ _init_ _(self):
        window = Tk() # Create a window
        window.title("Animation Demo") # Set a title
        width = 250 # Width of the canvas
        canvas = Canvas(window, bg = "white", width = 250, height = 50)
        canvas.pack()
        x = 0 # Starting x position
        canvas.create_text(x, 30,text = "Message moving?", tags = "text")
        dx = 3
        while True:
            canvas.move("text", dx, 0)
            canvas.after(100)
            canvas.update()
            if x < width:
                23 x += dx # Get the current position for string
            else:
                25 x = 0 # Reset string position to the beginning
                canvas.delete("text")
                canvas.create_text(x, 30, text = "Message moving?", tags = "text")
        window.mainloop() # Create an event loop
AnimationDemo() # Create GUI
```

Output:

The program creates a canvas (line 9) and displays text on the canvas at the specified initial location (lines 13–15). The animation is done essentially in the following three statements in a loop (lines 19–21):

canvas.move(**"text"**, dx, **0**) # Move text dx unit

canvas.after(**100**) # Sleep for 100 milliseconds

canvas.update() # Update canvas

The *x*-coordinate of the location is moved to the right **dx** units by invoking **canvas.move** (line 19). Invoking **canvas.after(100)** puts the program to sleep for **100** milliseconds (line 20). Invoking **canvas.update()** redisplays the canvas (line 21).

You can add tools to control the animation's speed, stop the animation, and resume the animation.

Scrollbars

A **Scrollbar** *widget can be used to scroll the contents in a* **Text**, **Canvas**, *or* **Listbox** *widget vertically or horizontally.*

ScrollText.py

from tkinter **import** * # Import all definitions from tkinter

class ScrollText:

    def _ _init_ _(self):

        window = Tk() # Create a window

        window.title(**"Scroll Text Demo"**) # Set title

        frame1 = Frame(window)

        frame1.pack()

        scrollbar = Scrollbar(frame1)

        scrollbar.pack(side = RIGHT, fill = Y)

        text = Text(frame1, width = **40**, height = **10**, wrap = WORD, yscrollcommand = scrollbar.set)

        text.pack()

        scrollbar.config(command = text.yview)

        window.mainloop() # Create an event loop

ScrollText() # Create GUI

The program creates a **Scrollbar** (line 10) and places it to the right of the text (line 11).

The scrollbar is tied to the **Text** widget (line 15) so that the contents in the **Text** widget can be scrolled through.

Standard Dialog Boxes

You can use standard dialog boxes to display message boxes or to prompt the user to enter numbers and strings.

Finally, let's look at Tkinter's standard dialog boxes (often referred to simply as *dialogs*).

```
import tkinter.messagebox

import tkinter.simpledialog

import tkinter.colorchooser

tkinter.messagebox.showwarning("showwarning", "This is a warning")

tkinter.messagebox.showerror("showerror", "This is an error")

isYes = tkinter.messagebox.askyesno("askyesno", "Continue?")

print(isYes)

isOK = tkinter.messagebox.askokcancel("askokcancel", "OK?")

print(isOK)

isYesNoCancel = tkinter.messagebox.askyesnocancel("askyesnocancel", "Yes, No, Cancel?")

print(isYesNoCancel)

name = tkinter.simpledialog.askstring( "askstring", "Enter your name")

print(name)

age = tkinter.simpledialog.askinteger("askinteger", "Enter your age")

print(age)

weight = tkinter.simpledialog.askfloat("askfloat", "Enter your weight")

print(weight)
```

These functions are defined in the **tkinter.messagebox** module. The **askyesno** function displays the *Yes* and *No* buttons in the dialog box. The function returns **True** if the *Yes* button is clicked or **False** if the *No* button is clicked.