PYTHON PROGRAMMING

I B.TECH II SEM FOR

CSE

(JNTUK)

(R20)

HUMANITIES & BASIC SCIENCES DEPARTMENT

V S M COLLEGE OF ENGINEERING

RAMCHANDRAPURAM

E.G. Dt. - 533255

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY KAKINADA
KAKINADA – 533 003, Andhra Pradesh, India

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

I Year – II Semester		L	T	P	C
		3	0	0	3
PYTHON PROGRAMMING					

Course Objectives:
The Objectives of Python Programming are

- To learn about Python programming language syntax, semantics, and the runtime environment
- To be familiarized with universal computer programming concepts like data types, containers
- To be familiarized with general computer programming concepts like conditional execution, loops & functions
- To be familiarized with general coding techniques and object-oriented programming

Course Outcomes:

- Develop essential programming skills in computer programming concepts like data types, containers
- Apply the basics of programming in the Python language
- Solve coding tasks related conditional execution, loops
- Solve coding tasks related to the fundamental notions and techniques used in object-oriented programming

UNIT I
Introduction: Introduction to Python, Program Development Cycle, Input, Processing, and Output, Displaying Output with the Print Function, Comments, Variables, Reading Input from the Keyboard, Performing Calculations, Operators. Type conversions, Expressions, More about Data Output.
Data Types, and Expression: Strings Assignment, and Comment, Numeric Data Types and Character Sets, Using functions and Modules.
Decision Structures and Boolean Logic: if, if-else, if-elif-else Statements, Nested Decision Structures, Comparing Strings, Logical Operators, Boolean Variables. Repetition Structures: Introduction, while loop, for loop, Calculating a Running Total, Input Validation Loops, Nested Loops.

UNIT II
Control Statement: Definite iteration for Loop Formatting Text for output, Selection if and if else Statement Conditional Iteration The While Loop
Strings and Text Files: Accessing Character and Substring in Strings, Data Encryption, Strings and Number Systems, String Methods Text Files.

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

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY KAKINADA
KAKINADA – 533 003, Andhra Pradesh, India

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

UNIT IV

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

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

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

UNIT V

Errors and Exceptions: Syntax Errors, Exceptions, Handling Exceptions, Raising Exceptions, User-defined Exceptions, Defining Clean-up Actions, Redefined Clean-up Actions.

Graphical User Interfaces: The Behavior of Terminal Based Programs and GUI -Based, Programs, Coding Simple GUI-Based Programs, Other Useful GUI Resources.

Programming**:** Introduction to Programming Concepts with Scratch.

Text Books

1) Fundamentals of Python First Programs, Kenneth. A. Lambert, Cengage.
2) Python Programming: A Modern Approach, Vamsi Kurama, Pearson.

Reference Books:

1) Introduction to Python Programming, Gowrishankar.S, Veena A, CRC Press.
2) Introduction to Programming Using Python, Y. Daniel Liang, Pearson.

e-Resources:

https://www.tutorialspoint.com/python3/python_tutorial.pdf

VSM COLLEGE OF ENGINEERING

RAMACHANDRAPURAM

DEPARTMENT OF BASIC SCIENCES AND HUMANITIES

Course Title	Year/Sem	Branch	Periods per Week
PYTHON PROGRAMMING	I/II	CSE BRANCH	5

Course Outcomes:

- To learn about Python programming language syntax, semantics, and the runtime environment
- To be familiarized with universal computer programming concepts like data types, containers
- To be familiarized with general computer programming concepts like conditional execution, loops & functions
- To be familiarized with general coding techniques and object-oriented programming

Unit No	Outcomes	Name of the Topic	No. of Periods required	Total Periods	Reference Book	Methodology to be adopted
		Unit-1				
I	CO 1, CO3, CO4 Python Introduction, Data Types, and Expression, Decision Structures and Boolean Logic	Introduction to Python, Program Development Cycle	1	14	T1, T2 R20	Black Board
		Input, Processing, and Output, Displaying Output with the Print Function	2			Black Board
		Comments, Variables, Reading Input from the Keyboard	1			Black Board
		Performing Calculations, Operators. Type conversions	1			E- CLASS ROOM
		Expressions, More about Data Output	1			Black Board
		Strings Assignment, and Comment	1			E- CLASS ROOM
		Numeric Data Types and Character Sets, Using functions and Modules	2			Black Board
		if, if-else, if-elif-else Statements, Nested Decision Structure	1			
		Comparing Strings, Logical Operators, Boolean Variables	1			Black Board
		Repetition Structures: Introduction, while loop, for loop	1			Black Board
		Calculating a Running Total, Input Validation Loops	1			E- CLASS ROOM
		Nested Loops	1			Black Board

		Unit-2				
II	CO 1, CO3,CO4 Control Statement, Strings and Text Files	Definite iteration for Loop Formatting Text for output	2	12	T1, T2 R20	Black Board
		Selection if and if else Statement Conditional Iteration The While Loop	2			Black Board
		Accessing Character and Substring in Strings	2			E-CLASS ROOM
		Data Encryption	2			Black Board
		Strings and Number Systems	2			Black Board
		String Methods Text Files	2			Black Board
		Unit-3				
III	CO1, CO2, CO3 List and Dictionaries, Design with Function, Modules	Lists, Defining Simple Functions	2	10	T1, T2 R20	Black Board
		Dictionaries	1			Black Board
		Functions as Abstraction Mechanisms	1			E-CLASS ROOM
		Design with Recursive Functions	1			Black Board
		Case Study Gathering Information from a File System	1			
		Managing a Program's Namespace, Higher Order Function	2			Black Board
		Modules, Standard Modules	1			E- CLASS ROOM
		Packages	1			Black Board

IV	CO 1, CO3, CO4 File Operations, Object Oriented Programming, Design with Classes	**Unit-4**		16		
		Reading config files in python, Writing log files in python	2			Black Board
		Understanding read functions, read(), readline() and readlines()	2			Black Board
		Understanding write functions	1			E-CLASS ROOM
		write() and writelines(), Manipulating file pointer using seek	1			Black Board
		Programming using file operations	1			Black Board
		Concept of class, object and instances, Constructor	2			Black Board
		class attributes and destructors, Real time use of class in live projects	2			Black Board
		Inheritance , overlapping and overloading operators	1			Black Board
		Adding and retrieving dynamic attributes of classes, Programming using Oops support	2			E-CLASS ROOM
		Objects and Classes, Data modeling Examples	1			Black Board
		Case Study An ATM, Structuring Classes with Inheritance and Polymorphism	1			Black Board

V	CO 1, CO2, CO3, CO4 Errors and Exceptions, Graphical User Interfaces, Programming	**Unit-5**		12	T1, T2 R20	Black Board
		Syntax Errors, Exceptions, Handling Exceptions	2			Black Board
		Raising Exceptions, User-defined Exceptions	2			Black Board
		Defining Clean-up Actions, Redefined Clean-up Actions	2			E-CLASS ROOM
		The Behavior of Terminal Based Programs and GUI - Based, Programs	2			Black Board
		Coding Simple GUI-Based Programs, Other Useful GUI Resources	2			Black Board
		Introduction to Programming Concepts with Scratch	2			Black Board

Text Books

1) Fundamentals of Python First Programs, Kenneth. A. Lambert, Cengage.
2) Python Programming: A Modern Approach, Vamsi Kurama, Pearson.

Reference Books:

1) Introduction to Python Programming, Gowrishankar.S, Veena A, CRC Press.
2) Introduction to Programming Using Python, Y. Daniel Liang, Pearson.

e-Resources:

https://www.tutorialspoint.com/python3/python_tutorial.pdf

Faculty Member **Head of the Department** **Principal**

PYTHON

What is Python?

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

- web development (server-side),
- software development,
- mathematics,
- system scripting.
-

What can Python do?

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

Why Python?

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

Good to know

- In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax compared to other programming languages

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

Example

```
print("Hello, World!")
```

Python Install

Many PCs and Macs will have python already installed.

To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

```
C:\Users\Your Name>python --version
```

To check if you have python installed on a Linux or Mac, then on linux open the command line or on Mac open the Terminal and type:

```
python --version
```

If you find that you do not have Python installed on your computer, then you can download it for free from the following website: https://www.python.org/

Python Quickstart

Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.

The way to run a python file is like this on the command line:

```
C:\Users\Your Name>python helloworld.py
```

Where "helloworld.py" is the name of your python file.

Let's write our first Python file, called helloworld.py, which can be done in any text editor.

helloworld.py

```
print("Hello, World!")
```

Try it Yourself »

Simple as that. Save your file. Open your command line, navigate to the directory where you saved your file, and run:

```
C:\Users\Your Name>python helloworld.py
```

The output should read:

```
Hello, World!
```

The Python Command Line

To test a short amount of code in python sometimes it is quickest and easiest not to write the code in a file. This is made possible because Python can be run as a command line itself.

Type the following on the Windows, Mac or Linux command line:

```
C:\Users\Your Name>python
```

Or, if the "python" command did not work, you can try "py":

```
C:\Users\Your Name>py
```

From there you can write any python, including our hello world example from earlier in the tutorial:

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> print("Hello, World!")
```

Which will write "Hello, World!" in the command line:

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> print("Hello, World!")
Hello, World!
```

Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:

```
exit()
```

Python Syntax

Execute Python Syntax

As we learned in the previous page, Python syntax can be executed by writing directly in the Command Line:

```
>>> print("Hello, World!")
Hello, World!
```

Execute Python SyntaxPython IndentationPython VariablesPython CommentsExercises

Or by creating a python file on the server, using the .py file extension, and running it in the Command Line:

```
C:\Users\Your Name>python myfile.py
```

Python Indentation

Indentation refers to the spaces at the beginning of a code line.
Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:

  print("Five is greater than two!")
```

Python will give you an error if you skip the indentation:

Example
Syntax Error:

```
if 5 > 2:
print("Five is greater than two!")
```

The number of spaces is up to you as a programmer, the most common use is four, but it has to be at least one.

Example

```
if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
        print("Five is greater than two!")
```

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

Example
Syntax Error:

```
if 5 > 2:
 print("Five is greater than two!")
        print("Five is greater than two!")
```

Python Variables

In Python, variables are created when you assign a value to it:

Example

Variables in Python:

```
x = 5
y = "Hello, World!"
```

Try it Yourself »

Python has no command for declaring a variable.

You will learn more about variables in the Python Variables chapter.

Comments

Python has commenting capability for the purpose of in-code documentation.

Comments start with a #, and Python will render the rest of the line as a comment:

Example

Comments in Python:

```
#This is a comment.
print("Hello, World!")
```

Try it Yourself »

Python Comments

Comments can be used to explain Python code.

Comments can be used to make the code more readable.

Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a #, and Python will ignore them:

Example

```
#This is a comment
print("Hello, World!")
```

Comments can be placed at the end of a line, and Python will ignore the rest of the line:

Example

```
print("Hello, World!") #This is a comment
```

A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code:

Example

```
#print("Hello, World!")
print("Cheers, Mate!")
```

Multi Line Comments

Python does not really have a syntax for multi line comments.
To add a multiline comment you could insert a # for each line:

Example

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Try it Yourself »

Or, not quite as intended, you can use a multiline string.
Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

Example

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

Try it Yourself »

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

Python Variables

Variables

Variables are containers for storing data values.

Creating Variables

Python has no command for declaring a variable.
A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "John"
print(x)
print(y)
```

Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

Example

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

Casting

If you want to specify the data type of a variable, this can be done with casting.

Example

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

Get the Type

You can get the data type of a variable with the type() function.

Example

```
x = 5
y = "John"
print(type(x))
print(type(y))
```

Single or Double Quotes?

String variables can be declared either by using single or double quotes:

Example

```
x = "John"
# is the same as
x = 'John'
```

Try it Yourself »

Case-Sensitive

Variable names are case-sensitive.

Example
This will create two variables:

```
a = 4
A = "Sally"
#A will not overwrite a
```

Python - Variable Names

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Example
Legal variable names:

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

Example
Illegal variable names:

```
2myvar = "John"
my-var = "John"
my var = "John"
```

Multi Words Variable Names

Variable names with more than one word can be difficult to read.
There are several techniques you can use to make them more readable:
Camel Case
Each word, except the first, starts with a capital letter:

```
myVariableName = "John"
```

One Value to Multiple Variables

And you can assign the *same* value to multiple variables in one line:

Example

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Unpack a Collection

If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called *unpacking*.

Example

Unpack a list:

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

Python - Output Variables

Output Variables

The Python print() function is often used to output variables.

Example

```
x = "Python is awesome"
print(x)
```

In the print() function, you output multiple variables, separated by a comma:

Example

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

You can also use the + operator to output multiple variables:

Example

```
x = "Python "
y = "is "
```

```
z = "awesome"
print(x + y + z)
```

For numbers, the + character works as a mathematical operator:

Example

```
x = 5
y = 10
print(x + y)
```

In the print() function, when you try to combine a string and a number with the + operator, Python will give you an error:

Example

```
x = 5
y = "John"
print(x + y)
```

The best way to output multiple variables in the print() function is to separate them with commas, which even support different data types:

Example

```
x = 5
y = "John"
print(x, y)
```

Global Variables

Variables that are created outside of a function (as in all of the examples above) are known as global variables.

Global variables can be used by everyone, both inside of functions and outside.

Example

Create a variable outside of a function, and use it inside the function

```
x = "awesome"

def myfunc():
  print("Python is " + x)

myfunc()
```

Try it Yourself »

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example

Create a variable inside a function, with the same name as the global variable

```
x = "awesome"

def myfunc():
```

```
  x = "fantastic"
  print("Python is " + x)

myfunc()

print("Python is " + x)
```

The global Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

To create a global variable inside a function, you can use the global keyword.

Example

If you use the global keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

Try it Yourself »

Also, use the global keyword if you want to change a global variable inside a function.

Example

To change the value of a global variable inside a function, refer to the variable by using the global keyword:

```
x = "awesome"

def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

Python Data Types

Built-in Data Types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	Str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	Dict
Set Types:	set, frozenset
Boolean Type:	Bool
Binary Types:	bytes, bytearray, memoryview
None Type:	NoneType

Getting the Data Type

You can get the data type of any object by using the type() function:

Example

Print the data type of the variable x:

```
x = 5
print(type(x))
```

Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set

x = True	bool

Python Numbers

There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them:

Example

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

To verify the type of any object in Python, use the type() function:

Example

```
print(type(x))
print(type(y))
print(type(z))
```

Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Example
Integers:

```
x = 1
y = 35656222554887711
z = -3255522

print(type(x))
print(type(y))
print(type(z))
```

Float

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Example
Floats:

```
x = 1.10
y = 1.0
z = -35.59

print(type(x))
print(type(y))
print(type(z))
```

Float can also be scientific numbers with an "e" to indicate the power of 10.

Example
Floats:

```
x = 35e3
y = 12E4
z = -87.7e100

print(type(x))
print(type(y))
print(type(z))
```

Complex

Complex numbers are written with a "j" as the imaginary part:

Example
Complex:

```
x = 3+5j
y = 5j
z = -5j

print(type(x))
print(type(y))
print(type(z))
```

Type Conversion

You can convert from one type to another with the int(), float(), and complex() methods:

Example
Convert from one type to another:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

Random Number

Python does not have a random() function to make a random number, but Python has a built-in module called random that can be used to make random numbers:

Example

Import the random module, and display a random number between 1 and 9:

```
import random
print(random.randrange(1, 10))
```

Try it Yourself »

Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example

```
a = "Hello"
print(a)
```

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example
You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Or three single quotes:

Example

```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(a)
```

Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

Example
Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1])
```

Try it Yourself »

Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a for loop.

Example
Loop through the letters in the word "banana":

```
for x in "banana":
  print(x)
```

Try it Yourself »

String Length

of a string, use the len() function.

Example
The len() function returns the length of a string:

```
a = "Hello, World!"
print(len(a))
```

Try it Yourself »

Python - Slicing Strings

You can return a range of characters by using the slice syntax.
Specify the start index and the end index, separated by a colon, to return a part of the string.

Example
Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"
print(b[2:5])
```

Note: The first character has index 0.

Slice From the Start
By leaving out the start index, the range will start at the first character:

Example
Get the characters from the start to position 5 (not included):

```
b = "Hello, World!"
print(b[:5])
```

Slice To the End
By leaving out the *end* index, the range will go to the end:

Example
Get the characters from position 2, and all the way to the end:

```
b = "Hello, World!"
print(b[2:])
```

Negative Indexing
Use negative indexes to start the slice from the end of the string:

Example
Get the characters:
From: "o" in "World!" (position -5)
To, but not included: "d" in "World!" (position -2):

```
b = "Hello, World!"
print(b[-5:-2])
```

Python - Modify Strings

Python has a set of built-in methods that you can use on strings.
Upper Case

Example

The upper() method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

Lower Case

Example

The lower() method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

Replace String

Example

The replace() method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Try it Yourself »

Split String

The split() method returns a list where the text between the specified separator becomes the list items.

Example

The split() method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

Try it Yourself »

String Methods

String Concatenation

To concatenate, or combine, two strings you can use the + operator.

Example

Merge variable a with variable b into variable c:

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

Try it Yourself »

Example

To add a space between them, add a " ":

```
a = "Hello"
b = "World"
```

```
c = a + " " + b
print(c)
```

Python - Format - Strings

String Format
As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

Example
```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

But we can combine strings and numbers by using the format() method!
The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are:

Example
Use the format() method to insert numbers into strings:
```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```
The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

Example
```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```
You can use index numbers {0} to be sure the arguments are placed in the correct placeholders:

Example
```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

Python - Escape Characters

Escape Character
To insert characters that are illegal in a string, use an escape character.
An escape character is a backslash \ followed by the character you want to insert.
An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

Example
You will get an error if you use double quotes inside a string that is surrounded by double quotes:

```
txt = "We are the so-called "Vikings" from the north."
```

To fix this problem, use the escape character \":

Example
The escape character allows you to use double quotes when you normally would not be allowed:

```
txt = "We are the so-called \"Vikings\" from the north."
```

Escape Characters
Other escape characters used in Python:

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

Python - String Methods

String Methods
Python has a set of built-in methods that you can use on strings.
Note: All string methods returns new values. They do not change the original string.

Method	Description

capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
etc…	

Boolean Values

In programming you often need to know if an expression is True or False.
You can evaluate any expression in Python, and get one of two answers, True or False.
When you compare two values, the expression is evaluated and Python returns the Boolean answer:

Example

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

Try it Yourself »

When you run a condition in an if statement, Python returns True or False:

Example

Print a message based on whether the condition is True or False:

```
a = 200
b = 33

if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Try it Yourself »

Evaluate Values and Variables

The bool() function allows you to evaluate any value, and give you True or False in return,

Example
Evaluate a string and a number:

```
print(bool("Hello"))
print(bool(15))
```

Try it Yourself »

Example
Evaluate two variables:

```
x = "Hello"
y = 15

print(bool(x))
print(bool(y))
```

Python Operators

Python Operators
Operators are used to perform operations on variables and values.
In the example below, we use the + operator to add together two values:

Example

```
print(10 + 5)
```

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators
Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y

/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Python Assignment Operators
Assignment operators are used to assign values to variables:

Operator	**Example**	**Same As**
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Python Comparison Operators
Comparison operators are used to compare two values:

Operator	**Name**	**Example**

==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Python Logical Operators
Logical operators are used to combine conditional statements:

Operator	**Description**	**Example**
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Python Identity Operators
Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	**Description**	**Example**
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

---

Python Membership Operators
Membership operators are used to test if a sequence is presented in an object:

Operator	**Description**	**Example**
in	Returns True if a sequence with the specified value is present in the object	x in y

not in	Returns True if a sequence with the specified value is not present in the object	x not in y

---

Python Bitwise Operators
Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Python Lists

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

List

Lists are used to store multiple items in a single variable.
Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
Lists are created using square brackets:

Example
Create a List:

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

List Items
List items are ordered, changeable, and allow duplicate values.
List items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered
When we say that lists are ordered, it means that the items have a defined order, and that order will not change.
If you add new items to a list, the new items will be placed at the end of the list.

Changeable
The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates
Since lists are indexed, lists can have items with the same value:

Example
Lists allow duplicate values:

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

List Length
To determine how many items a list has, use the len() function:

Example
Print the number of items in the list:

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

List Items - Data Types
List items can be of any data type:

Example
String, int and boolean data types:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

A list can contain different data types:

Example
A list with strings, integers and boolean values:

```
list1 = ["abc", 34, True, 40, "male"]
```

Python - Access List Items

Access Items

List items are indexed and you can access them by referring to the index number:

Example
Print the second item of the list:

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

Negative Indexing
Negative indexing means start from the end
-1 refers to the last item, -2 refers to the second last item etc.

Example
Print the last item of the list:

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

Range of Indexes
You can specify a range of indexes by specifying where to start and where to end the range.
When specifying a range, the return value will be a new list with the specified items.

Example
Return the third, fourth, and fifth item:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

Example
This example returns the items from the beginning to, but NOT including, "kiwi":

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
```

By leaving out the end value, the range will go on to the end of the list:

Example
This example returns the items from "cherry" to the end:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
```

Insert Items
To insert a new list item, without replacing any of the existing values, we can use
the insert() method.
The insert() method inserts an item at the specified index:

Example
Insert "watermelon" as the third item:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

Python - Add List Items

Append Items

To add an item to the end of the list, use the append() method:

Example

Using the append() method to append an item:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

Insert Items

To insert a list item at a specified index, use the insert() method.

The insert() method inserts an item at the specified index:

Example

Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Extend List

To append elements from *another list* to the current list, use the extend() method.

Example

Add the elements of tropical to thislist:

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

Remove Specified Item

The remove() method removes the specified item.

Example

Remove "banana":

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

Remove Specified Index

The pop() method removes the specified index.

Example
Remove the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

Try it Yourself »

If you do not specify the index, the pop() method removes the last item.

Example
Remove the last item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

Try it Yourself »

The del keyword also removes the specified index:

Example
Remove the first item:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

The del keyword can also delete the list completely.

Example
Delete the entire list:

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

Sort List Alphanumerically

List objects have a sort() method that will sort the list alphanumerically, ascending, by default:

Example

Sort the list alphabetically:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

```
['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

Try it Yourself »

Example
Sort the list numerically:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

```
[23, 50, 65, 82, 100]
```

Python - List Methods

List Methods

Python has a set of built-in methods that you can use on lists.

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

Tuple

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

Example
Create a Tuple:

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

Try it Yourself »

Tuple Items

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

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

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

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

Example
Tuples allow duplicate values:

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

Tuple Length
To determine how many items a tuple has, use the len() function:

Example
Print the number of items in the tuple:

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

Try it Yourself »

Create Tuple With One Item
To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Example
One item tuple, remember the comma:

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Try it Yourself »

Tuple Items - Data Types

Tuple items can be of any data type:

Example
String, int and boolean data types:

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

Try it Yourself »

A tuple can contain different data types:

Example
A tuple with strings, integers and boolean values:

```
tuple1 = ("abc", 34, True, 40, "male")
```

Python - Access Tuple Items

Access Tuple Items

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

Example
Print the second item in the tuple:

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

Note: The first item has index 0.

Negative Indexing

Negative indexing means start from the end.

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

Example
Print the last item of the tuple:

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

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

Example
Return the third, fourth, and fifth item:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

Example
This example returns the items from index -4 (included) to index -1 (excluded)

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

```
('orange', 'kiwi', 'melon')
```

Check if Item Exists

To determine if a specified item is present in a tuple use the in keyword:

Example

Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Python - Update Tuples

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.

But there are some workarounds.

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example

Convert the tuple into a list to be able to change it:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

```
("apple", "kiwi", "cherry")
```

Python Sets

```
myset = {"apple", "banana", "cherry"}
```

Set

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.

*** Note:** Set *items* are unchangeable, but you can remove items and add new items.

Sets are written with curly brackets.

Example
Create a Set:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

Set Items
Set items are unordered, unchangeable, and do not allow duplicate values.

Unordered
Unordered means that the items in a set do not have a defined order.
Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Unchangeable
Set items are unchangeable, meaning that we cannot change the items after the set has been created.
Once a set is created, you cannot change its items, but you can remove items and add new items.

Duplicates Not Allowed
Sets cannot have two items with the same value.
Example
Duplicate values will be ignored:

```
thisset = {"apple", "banana", "cherry", "apple"}

print(thisset)
```

Get the Length of a Set
To determine how many items a set has, use the len() function.
Example
Get the number of items in a set:

```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))
```

Set Items - Data Types
Set items can be of any data type:
Example
String, int and boolean data types:

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

A set can contain different data types:

Example
A set with strings, integers and boolean values:

```
set1 = {"abc", 34, True, 40, "male"}
```

Access Items

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Example
Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

Try it Yourself »

Example
Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

Try it Yourself »

Python - Add Set Items

Add Items

Once a set is created, you cannot change its items, but you can add new items.

To add one item to a set use the add() method.

Example
Add an item to a set, using the add() method:

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

Add Sets

To add items from another set into the current set, use the update() method.

Example

Add elements from tropical into thisset:

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)
```

```
{'apple', 'mango', 'cherry', 'pineapple', 'banana', 'papaya'}
```

Python - Remove Set Items

Remove Item

To remove an item in a set, use the remove(), or the discard() method.

Example

Remove "banana" by using the remove() method:

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

Note: If the item to remove does not exist, remove() will raise an error.

Example

Remove "banana" by using the discard() method:

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

Note: If the item to remove does not exist, discard() will **NOT** raise an error.

You can also use the pop() method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the pop() method is the removed item.

Example

Remove the last item by using the pop() method:

```
thisset = {"apple", "banana", "cherry"}

x = thisset.pop()
```

```
print(x)

print(thisset)
```
apple
{'cherry', 'banana'}

Note: Sets are *unordered*, so when using the pop() method, you do not know which item that gets removed.

Example
The clear() method empties the set:
```
thisset = {"apple", "banana", "cherry"}

thisset.clear()

print(thisset)
```
set()

Set Methods

Python has a set of built-in methods that you can use on sets.

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
	Removes the items in this set that are also included in another, specified set etc..

Python Dictionaries

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

Dictionary
Dictionaries are used to store data values in key:value pairs.
A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

Dictionaries are written with curly brackets, and have keys and values:

Example

Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Ordered or Unordered?

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

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

Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

Changeable

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

Example

Duplicate values will overwrite existing values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Python - Access Dictionary Items

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example
Get the value of the "model" key:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
```

There is also a method called get() that will give you the same result:

Example
Get the value of the "model" key:

```
x = thisdict.get("model")
```

Get Keys
The keys() method will return a list of all the keys in the dictionary.

Example
Get a list of the keys:

```
x = thisdict.keys()
```

The list of the keys is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

Example
Add a new item to the original dictionary, and see that the keys list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change
```

Get Values
The values() method will return a list of all the values in the dictionary.

Example
Get a list of the values:

```
x = thisdict.values()
```

The list of the values is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

Example
Make a change in the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

Example
Add a new item to the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

Get Items
The items() method will return each item in a dictionary, as tuples in a list.

Example
Get a list of the key:value pairs

```
x = thisdict.items()
```

The returned list is a *view* of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.

Example
Make a change in the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.items()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

Example
Add a new item to the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.items()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

Check if Key Exists
To determine if a specified key is present in a dictionary use the in keyword:

Example

Check if "model" is present in the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Python - Change Dictionary Items

Change Values

You can change the value of a specific item by referring to its key name:

Example

Change the "year" to 2018:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Update Dictionary

The update() method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Update the "year" of the car by using the update() method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Removing Items

There are several methods to remove items from a dictionary:

Example

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

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
```

```
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

Try it Yourself »

Example

The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

Try it Yourself »

Example

The del keyword removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

Try it Yourself »

Example

The del keyword can also delete the dictionary completely:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

Try it Yourself »

Example

The clear() method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Dictionary Methods
Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key

etc.

Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops.
An "if statement" is written by using the if keyword.

Example
If statement:

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".
Indentation
Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example
If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Elif

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

Example

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

Try it Yourself »

In this example a is equal to b, so the first condition is not true, but the elif condition is true, so we print to screen that "a and b are equal".

Else

The else keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

Try it Yourself »

In this example a is greater than b, so the first condition is not true, also the elif condition is not true, so we go to the else condition and print to screen that "a is greater than b".

You can also have an else without the elif:

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Try it Yourself »

Nested If

You can have if statements inside if statements, this is called *nested* if statements.

Example

```
x = 41

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

Try it Yourself »

The pass Statement

if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

Example

```
a = 33
b = 200

if b > a:
  pass
```

Python While Loops

Python Loops

Python has two primitive loop commands:

- while loops
- for loops

The while Loop

With the while loop we can execute a set of statements as long as a condition is true.

Example

Print i as long as i is less than 6:

```
i = 1
while i < 6:
  print(i)
  i += 1
```

Note: remember to increment i, or else the loop will continue forever.

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

The break Statement

With the break statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

The else Statement

With the else statement we can run a block of code once when the condition no longer is true:

Example

Print a message once the condition is false:

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

Python For Loops

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

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

Example

Print each fruit in a fruit list:

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

Try it Yourself »

The for loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word "banana":

```
for x in "banana":
  print(x)
```

Try it Yourself »

The break Statement

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

Example

Exit the loop when x is "banana":

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

Try it Yourself »

Example

Exit the loop when x is "banana", but this time the break comes before the print:

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

Try it Yourself »

The continue Statement

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

Example

Do not print banana:

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

Try it Yourself »

The range() Function

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

Example
Using the range() function:

```
for x in range(6):
  print(x)
```

Try it Yourself »

Note that range(6) is not the values of 0 to 6, but the values 0 to 5.
The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

Example
Using the start parameter:

```
for x in range(2, 6):
  print(x)
```

Try it Yourself »

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, **3**):

Example
Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):
  print(x)
```

Try it Yourself »

Else in For Loop
The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

Example
Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Try it Yourself »

Note: The else block will NOT be executed if the loop is stopped by a break statement.

Example
Break the loop when x is 3, and see what happens with the else block:

```
for x in range(6):
  if x == 3: break
  print(x)
```

```
else:
  print("Finally finished!")
```
Try it Yourself »

Nested Loops
A nested loop is a loop inside a loop.
The "inner loop" will be executed one time for each iteration of the "outer loop":

Example
Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```
Try it Yourself »

The pass Statement
for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

Example

```
for x in [0, 1, 2]:
  pass
```

Python Functions

A function is a block of code which only runs when it is called.
You can pass data, known as parameters, into a function.
A function can return data as a result.

Creating a Function
In Python a function is defined using the def keyword:

Example

```
def my_function():
  print("Hello from a function")
```

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

Example

```
def my_function():
  print("Hello from a function")
```

my_function()

Arguments
Information can be passed into functions as arguments.
Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.
The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example
```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```
Parameters or Arguments?

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.
From a function's perspective:
A parameter is the variable listed inside the parentheses in the function definition.
An argument is the value that is sent to the function when it is called.

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

Example
This function expects 2 arguments, and gets 2 arguments:
```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```
Try it Yourself »
If you try to call the function with 1 or 3 arguments, you will get an error:
Example
This function expects 2 arguments, but gets only 1:
```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

Keyword Arguments
You can also send arguments with the *key* = *value* syntax.
This way the order of the arguments does not matter.

Example

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

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Try it Yourself »

```
The youngest child is Linus
```

efault Parameter Value
The following example shows how to use a default parameter value.
If we call the function without argument, it uses the default value:

Example

```
def my_function(country = "Norway"):
  print("I am from " + country)

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

Try it Yourself »

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.
E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example

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

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

my_function(fruits)
```

Try it Yourself »

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

Example

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

print(my_function(3))
print(my_function(5))
print(my_function(9))
```
Try it Yourself »

The pass Statement
function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example
```
def myfunction():
  pass
```
Try it Yourself »

Recursion

Python also accepts function recursion, which means a defined function can call itself.
Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.
The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.
In this example, tri_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).
To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Example
Recursion Example
```
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

Tryt Yourself »

Recursion Example Results

1
3
6
10
15
21

UNIT -2

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

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

CONTROL STATEMENTS :Looping Statements

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

1 Definite iteration for Loop

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

    for <variable> in range(<an integer expression>):

    <statement-1>

    .

    .

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

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

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

Count-Controlled Loops:

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

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

  <loop body>

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

Loop Errors: Off-by-One Error:

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

Traversing the Contents of a Data Sequence:

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

for <variable> in <sequence>:

<do something with variable>

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

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

Specifying the Steps in the Range :

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

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

[1, 3, 5]

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

[1, 4]

Loops That Count Down

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

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

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

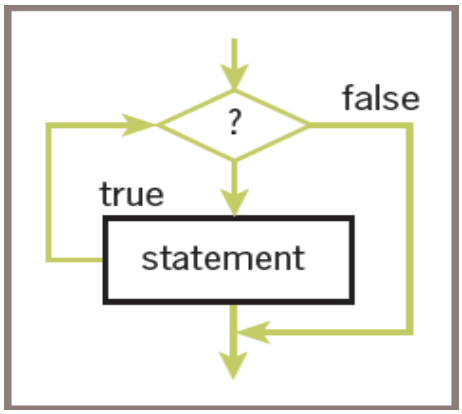
2. Conditional Iteration: The while Loop

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

  while <condition>:

  <sequence of statements>

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




```
# Summation with a while loop

theSum = 0

count = 1

while  count <= 10:

theSum += count

count += 1

print(theSum)
```

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

The while True Loop and the break Statement

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

```
while True:

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

    if number >= 0 and number <= 100:

            print(number) # Just echo the valid input

            break

    else:

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

OUTPUT

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

Enter the numeric grade: 101

Error: grade must be between 100 and 0

Enter the numeric grade: –1

Error: grade must be between 100 and 0

Enter the numeric grade: 45

45

Random Numbers

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

```
>>> import random
>>> for roll in range(10):
print(random.randint(1, 6), end = " ")
2 4 6 4 3 2 3 6 2 2
```

3. Formatting Text for output

- Many data-processing applications require output that has a **tabular format,** like that used in spreadsheets or tables of numeric data.
- In this format, numbers and other information are aligned in columns that can be either left-justified or right-justified.
  - A column of data is left-justified if its values are vertically aligned beginning with their leftmost characters.
  - A column of data is right-justified if its values are vertically aligned beginning with their rightmost characters.
- The total number of data characters and additional spaces for a given datum in a formatted string is called its **field width.**

FORMATTING STRING:

- The simplest form of this operation is the following:

  <format string> % <datum>

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

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

' four'

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

'four '
```

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

FORMATTING INTEGERS:

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

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

WITHOUT FORMATTING (integers):

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


7 10000000

8 100000000

9 1000000000

10 10000000000
```

WITH FORMATTING (integers):

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

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

FORMATTING FLOAT:

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

  %<field width>.<precision>f

  where .<precision> is optional.

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

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

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

4. Strings

- E-mail, text messaging, Web pages, and word processing all rely on and manipulate data consisting of strings of characters.
- A string is a sequence of characters enclosed in single or double quotation marks.
- The following session with the Python shell shows some example strings:

```
>>> 'Hello there!'
'Hello there!'
>>> "Hello there!"
'Hello there!'
>>> "
"
>>> ""
"
```

Accessing Character and Substring in Strings

The Structure of Strings:

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

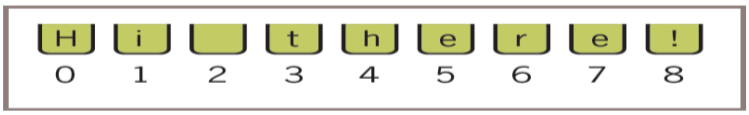
  >>> **len("Hi there!")**

  9

  >>> len("")

  0

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



The Subscript Operator:

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

  <a string>[<integer or index>]

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

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

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

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

2

3 t

4 h

5 e

6 r

7 e

8 !

Slicing for Substrings :

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

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

'shm'
```

Testing for a Substring with the in Operator:

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

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

5. Strings and Number Systems

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

The Positional System for Representing Numbers

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

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

Converting Binary to Decimal

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

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

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

```
bitString = input("Enter a string of bits: ")
decimal = 0
exponent = len(bitString) - 1
for digit in bitString:
    decimal = decimal + int(digit) * 2 ** exponent
```

```
        exponent = exponent - 1
    print("The integer value is", decimal)
```

OUTPUT :

Enter a string of bits: 1111

The integer value is 15

Enter a string of bits: 101

The integer value is 5

Converting Decimal to Binary:

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

Python script to convert decimal number to binary number

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

Enter a decimal integer: 34

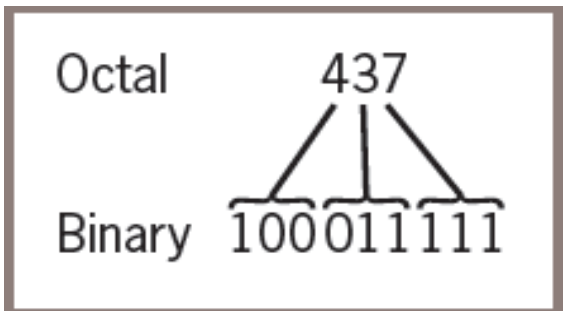
Quotient	Remainder	Binary
17	0	0
8	1	10
4	0	010

2	0	0010
1	0	00010
0	1	100010

The binary representation is 100010

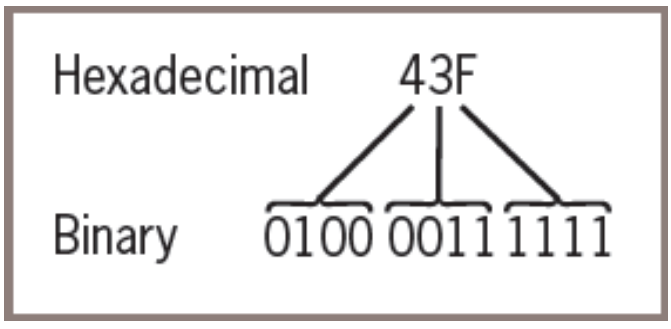
Octal Numbers

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



Hexadecimal Numbers:

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



5 String Methods :

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

Examples :

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

```
>>> s.upper()

'HI THERE!'

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

'Ho there!'

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

'Hi there!'
```

6) Data Encryption

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

Caesar cipher:

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

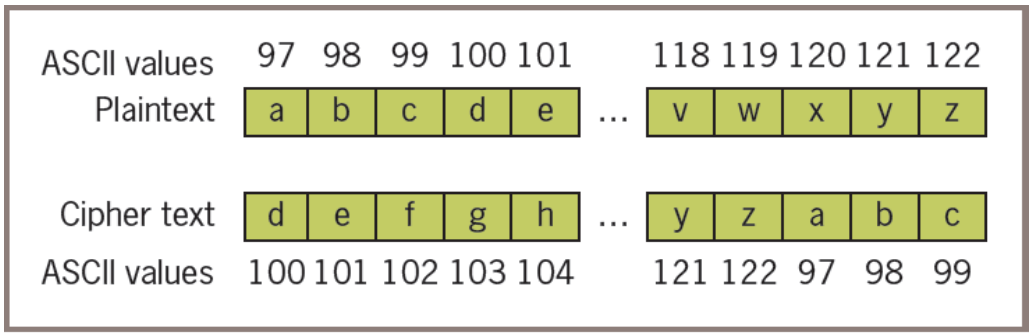


Figure 4-2 A Caesar cipher with distance +3 for the lowercase alphabet

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

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

OUTPUT :

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

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

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

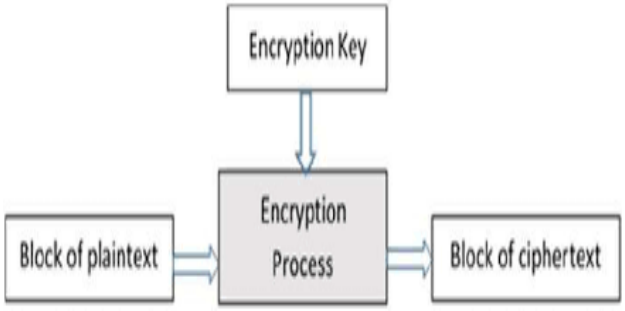
OUTPUT :

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

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

Block cipher:

- A block cipher uses plaintext characters to compute two or more encrypted characters.
- This is accomplished by using a mathematical structure known as an **invertible matrix to determine the values of** the encrypted characters.
- The matrix provides the key in this method. The receiver uses the same matrix to decrypt the cipher text.
- The fact that information used to determine each character comes from a block of data makes it more difficult to determine the key


Encryption Key
Block of plaintext
Encryption Process
Block of ciphertext

UNIT- 3

LISTS , DICTIONARIES, FUNCTIONS AND MODULES

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

Lists :

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

List Literals and Basic Operators :

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

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

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

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

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

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

List Methods :

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

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

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

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

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

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

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

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

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

NOTE:

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

Dictionaries:

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

Dictionary Literals:

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

Adding Keys and Replacing Values :

- You add a new key/value pair to a dictionary by using the subscript operator []. The form of this operation is the following:

  » <a dictionary>[<a key>] = <a value>

- The next code segment creates an empty dictionary and adds two new entries:

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

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

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

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

Accessing Values :

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

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

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

Removing Keys :

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

Traversing a Dictionary :

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

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

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

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

Gives same output as the previous one

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

Dictionary Operations :

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

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

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

Conversion of hexadecimal to Binary:

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

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

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

FUNCTIONS:

1. Design with Function

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

Functions as Abstraction Mechanisms

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

Functions Eliminate Redundancy

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

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

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

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

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

Functions Hide Complexity

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

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

Functions Support General Methods with Systematic Variations

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

Functions Support the Division of Labor

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

2 . Problem Solving with Top-Down Design

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

The Design of the Text-Analysis Program

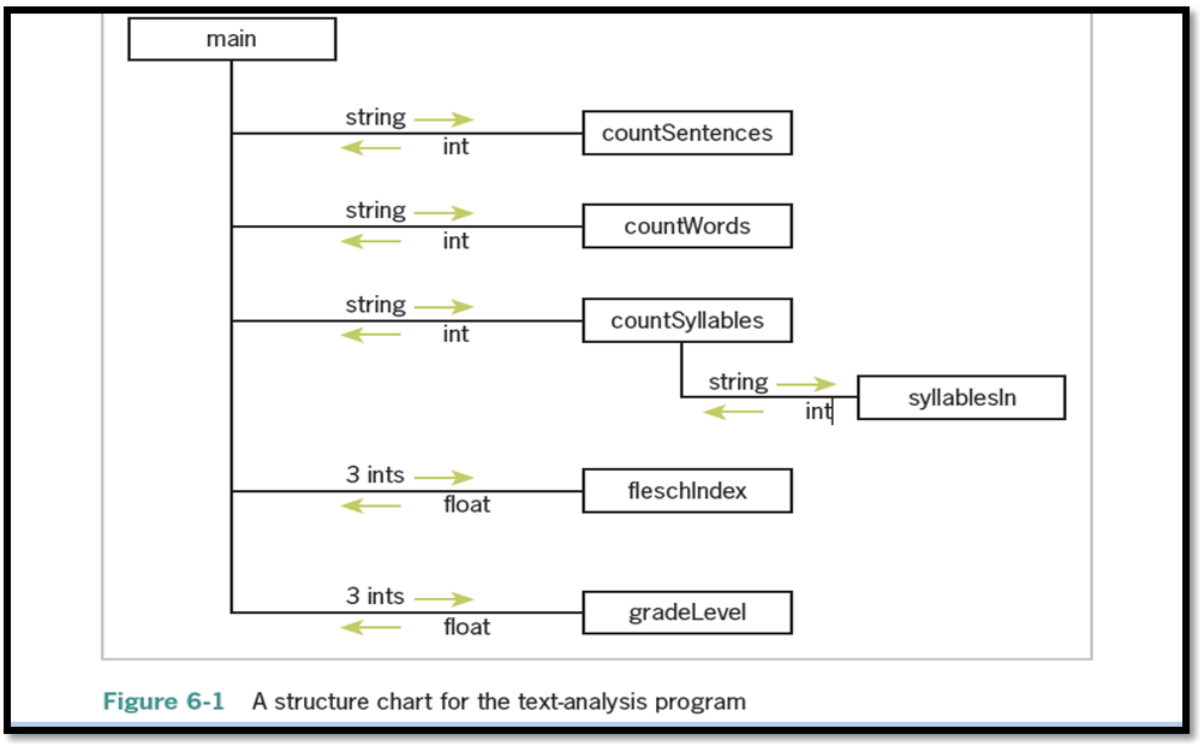


Figure 6-1 A structure chart for the text-analysis program

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

Structure chart

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

3 . Design with Recursive Functions

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

Defining a Recursive Function :

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

#Python **recursive** function for summation

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

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

Using Recursive Definitions to Construct Recursive Functions

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

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

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

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

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

Infinite Recursion:

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

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

RecursionError: maximum recursion depth exceeded in comparison

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

The Costs and Benefits of Recursion :

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

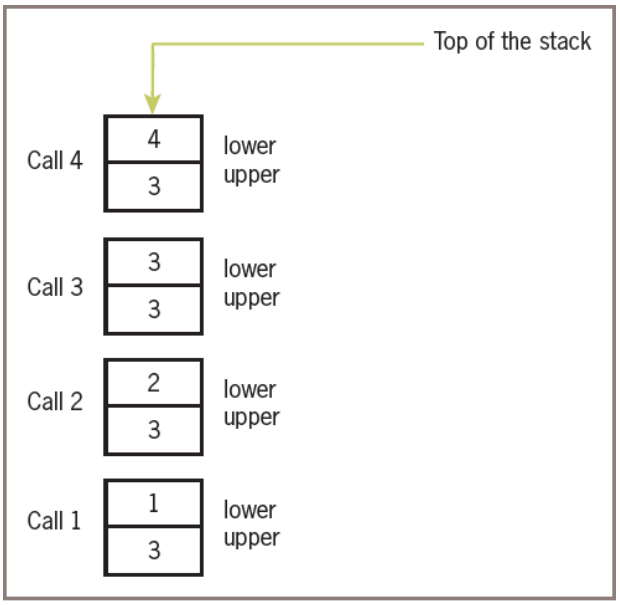


Figure 6-4 The stack frames for `displayRange(1, 3)`

4. Case Study Gathering Information from a File System

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

Request:

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

Analysis:

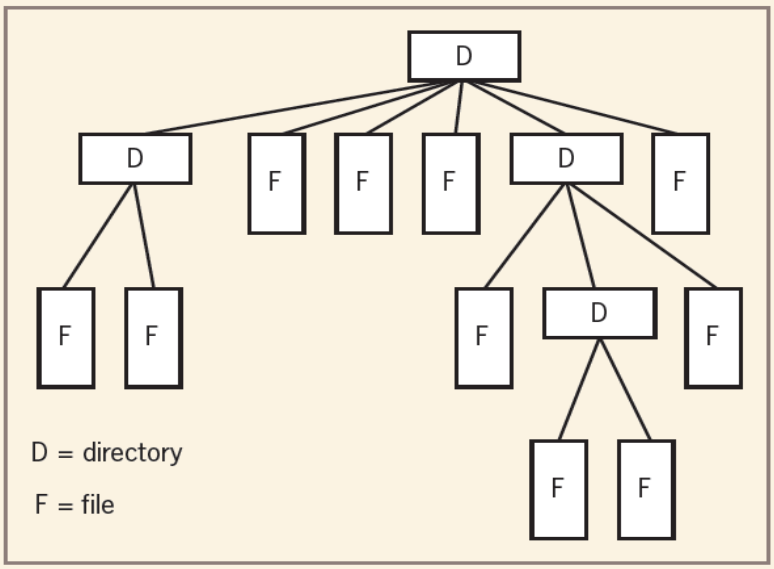


Figure 6-5 The structure of a file system

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

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

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

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

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

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

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

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

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

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

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

if__name__== "_main_":
    main()
```

5. Managing a Program's Namespace

Namespaces in Python

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

i) The Built-In Namespace

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

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

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

ii) The Global Namespace

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

iii) The Local and Enclosing Namespaces

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

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

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

Output :

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

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

Scope:

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

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

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

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

Lifetime:

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

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

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

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

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

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

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

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

6. Anonomyous Function or Lambda function:

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

lambda arguments: expression

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

EXAMPLE 1:

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

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

EXAMPLE 2 :Lambda with filter():

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

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

EXAMPLE 3: Lambda with map():

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

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

7. Higher Order Functions

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

map() :

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

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

 # Python program to demonstrate working of map.

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

Output:

[2, 4, 6, 8]

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

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

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

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

OUTPUT :

The filtered letters are: e e

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

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

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

8. Modules in Python:

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

User defined module :

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

```
# Python Module example

def add(a, b):

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

    result = a + b

     return result
```

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

```
>>> import example
```

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

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

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

9.5
```

- Modules are imported by using import statement

Syntax:

i) import module_name

Example:

```
>>>import math

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

5.0
```

ii) from….import statement:

A module may contain definition of many functions and variables.

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

Syntax:

from module_name import function_name/variable_name

e.g.1

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

Thu Aug 26 15:08:52 2021

e.g.2

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

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

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

iii) "as keyword":

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

e.g.

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

Creating a module: num.py

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

```
    return(x**y)
```

Example program:

```
import  num

print("Square of 10",num.square(10))

print("Cube of 10",num.cube(10))

print("Power of 10, 2 is ",num.power(10,5))
```

9. Packages in Python:

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

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

Creating a Package:

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

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

greet.py

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

functions.py

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

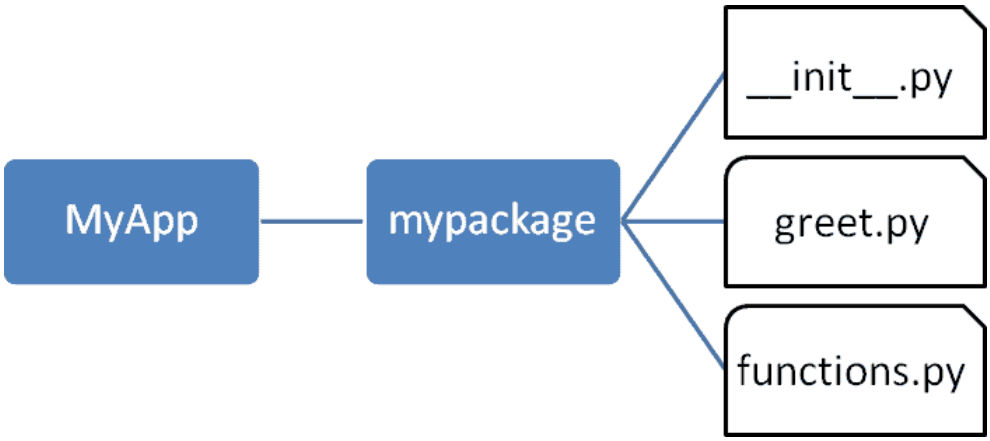
```
    return (x+y)/2

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

__init__.py :

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

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

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



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

```
>>> from mypackage import functions
>>> functions.power(3,2)
9
```

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

```
>>> from mypackage.functions import sum
>>> sum(10,20)
```

30

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

UNIT-4

Syllabus:

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

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

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

Files in Python:

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

Opening and Closing Files

The open () Method

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

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

Here are parameter details –

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

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

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

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

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

The file Object Attributes:

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

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

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

Example:

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

#File object attributes

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

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

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

f.close()
```

The close () Method

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

Syntax: fileObject.close()

Example:

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

Reading and Writing Files

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

Understanding write () and writelines ()

The write () Method

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

Syntax: fileObject.write(string)

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

Example:

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

The writelines () method:

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

Syntax: fileObject.writelines(sequence)

Parameters

Sequence – This is the Sequence of the strings.

Return Value-This method does not return any value.

Example

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

#writing content into file using write method

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

)

 f.close()
```

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

The read () Method

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

Syntax: fileObject.read([count])

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

Example

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

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


 f.close()

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

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

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

The readline () Method

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

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

Syntax: fileObject.readline(size)

Parameters

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

Return Value

- This method returns the line read from the file.

Example

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

#writing content into file using write method

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

f.close()
```

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

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

The readlines () Method

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

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

Syntax: fileObject.readlines(sizehint)

Parameters

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

Return Value

This method returns a list containing the lines.

Example

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

#writing content into file using write method

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

 f.close()

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

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

print(f.readlines())
```

Manipulating file pointer using seek():

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

Syntax: file_object.tell()

Example:

```
# Open a file

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

str = fo.read(10)

print("Read String is : ", str)

# Check current position

position = fo.tell()

print("Current file position : ", position)

fo.close()
```

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

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

Parameters:

Offset: Number of postions to move forward

from_what: It defines point of reference.

Returns: Does not return any value

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

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

1: sets the reference point at the current file position

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

By default from_what argument is set to 0.

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

Example:

```
# Open a file
```

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

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

# Check current position

position = fo.tell()

print("Current file position : ", position)

# Reposition pointer at the beginning once again

position = fo.seek(0, 0);

str = fo.read(10)

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

# Close opend file

fo.close()
```

File processing operations:

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

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

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

Syntax: os.rename(current_file_name, new_file_name)

Example:

```
import os

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

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

create.(This method is used to create directory)

Syntax: os.mkdir(directory name)

Example:

```
import os

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

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

Syntax: os.rmdir(directory name)

Example:

```
import os

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

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

Syntax: os.chdir(newdir)

Example:

```
import os

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

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

Syntax: os.remove(filename)

Example:

```
import os

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

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

Syntax: os.getcwd()

Example:

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

WRITING AND READING CONFIG FILES IN PYTHON

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

Creating and writing config file in Python

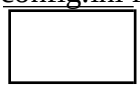
In Python we have configparser module which can help us with creation of config files (.ini format).

Program:

```
from configparser import ConfigParser

#Get the configparser object

config_object = ConfigParser()

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

config_object["USERINFO"] = {

 "admin": "Chankey Pathak",

 "loginid": "chankeypathak",

 "password": "tutswiki"

 }

config_object["SERVERCONFIG"] = {
```

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

 "port": "8080",

 "ipaddr": "8.8.8.8"
}

#Write the above sections to config.ini file

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

 config_object.write(conf)
```

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



is its content.

```
[USERINFO]

admin = Chankey Pathak

password = tutswiki

loginid = chankeypathak


[SERVERCONFIG]

host = tutswiki.com

ipaddr = 8.8.8.8

port = 8080
```

Reading a key from config file:

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

Program:

```
from configparser import ConfigParser
```

```
#Read config.ini file

config_object = ConfigParser()

config_object.read("config.ini")

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

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

output:

Password is tutswiki

UNIT-4
PART-2

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

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

Introduction

We have two programming techniques namely

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

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

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

Overview of OOP Terminology

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

current instance of a class.

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

Benefits of OOP

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

Classes:

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

   Example :

   All integer variables that we define in our program are instances of class int. >>> a=10

   >>> type(a)

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

Defining a class:

Python has a very simple syntax of defining a class.

Syntax :

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

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

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

Note : the class members are accessed through class object

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

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

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

Syntax of object creation:

Object-name=class-name()

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

Example :

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

self variable and class methods:

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

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

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

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

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

Example 1 :

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


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

Output:

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

Constructor method:

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

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

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

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

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

constructor. Example:

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

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

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

Output:

>>>

Constructor initialised

XXXX YYYY

Person number : 0

After setting Name:

vamsi kurama

Person number : 7

Destructor:

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

Syntax of destructor declaration:

```
def__del_(self):
 # body of destructor
```

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

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

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

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

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

obj = Employee()
del obj
```

Output:

Employee created

Destructor called, Employee deleted

Inheritance:

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

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

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

Syntax to inherit a class:

Class MySubClass(object):

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

Example :

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

Example 2:

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

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

Types of inheritance:

Python supports the following types of inheritan:

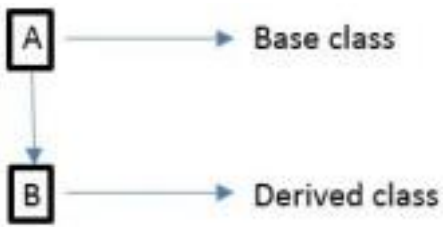
i) Single inheritance
ii) Multiple Inheritance
iii) Multi-level Inheritance
iv) Multi path Inheritance

Single Inheritance:

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

Syntax:

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



Example:

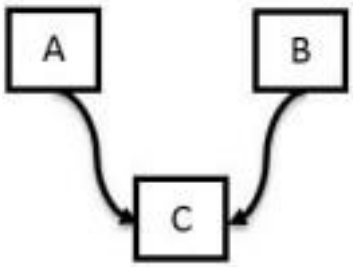
```
class A:
    i=10
class B(A):
    j=20
```

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

Multiple Inheritance:

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

Syntax:




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

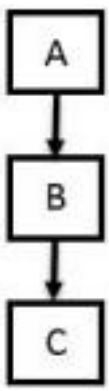
e.g.

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

Multi-Level Inheritance:

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

Syntax:



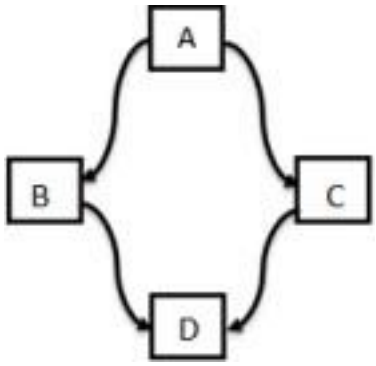

```
class Baseclass:
        <body of base class>
class Derivedclass1(Baseclass):
        <body of derived class 1>
class Derivedclass2(Derivedclass1):
        <body of the derived class2>
```

e.g.

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

Multi Path Inheritance:

Syntax:




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

e.g.

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

Polymorphism:

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

Polymorphism in operators

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

Example:

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

Polymorphism in built-in functions

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

Example:

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

Polymorphism in inheritance:

Method Overriding:

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

e.g.

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

OUTPUT :

I am class B and I have data 20

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

Super():

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

e.g

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

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

OUTPUT:

I am class A and I have data 10

I am class B and I have data 20

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

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

overloading operators

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

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

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

Output:

3

LearnFor

12

LearnLearnLearnLearn

Example 2:

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

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

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

OUTPUT:

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

Case Study An ATM:

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

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

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

OUTPUT:

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

Adding and retrieving dynamic attributes of classes:

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

Example:

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

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

UNIT 5 PART -1

EXCEPTION HANDLING

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

Errors:

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

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

  Example :

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

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

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

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

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

Exceptions:

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

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

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

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

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

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

Example:

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

output:

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

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

Handling Exceptions:

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

The syntax for try–except block can be given as

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

Example:

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

Output:

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

Denominator can’t be zero

Multiple except blocks:

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

syntax:

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

e.g.

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

(OR)

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

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

OUTPUT :

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

Raising Exceptions:

The raise keyword is used to raise an exception.You can define what kind of error to raise, and the text to print to the user**.**

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

You can Explicitly raise an exception using the raise keyword.

The general syntax for the raise statement is

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

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

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

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

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

OUTPUT:

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

Defining clean-up actions(The finally Block)

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

Syntax:

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

Due to any exception, operations written here will be skipped

finally:

This would always be executed.

.....

e.g.

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

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

try:

    quo=num/den

    print(quo)



except ZeroDivisionError:

    print("Denominator cant be zero")

else:

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



finally:

     print("TASK DONE")
```

OUTPUT:

```
>>>

enter numerator4

enter denominator2

2.0

This line is executed when there is no exception

TASK DONE
```

>>>

enter numerator4

enter denominator0

Denominator cant be zero

TASK DONE

Built-in and User-defined Exceptions:

Built-in Exceptions:

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

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

IOError Raised when an input/ output operation fails

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

User –defined exception:

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

Example:

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

OUTPUT:

user defined exception generated with value 10

UNIT 5 PART-II

Graphical User Interface:

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

- Tkinter
- wxPython
- JPython

Tkinter Programming

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

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

Example:

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

This would create a following window –

Tkinter Widgets:

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

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

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

Standard attributes for widgets

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

Geometry Management:

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

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

structure in the parent widget.

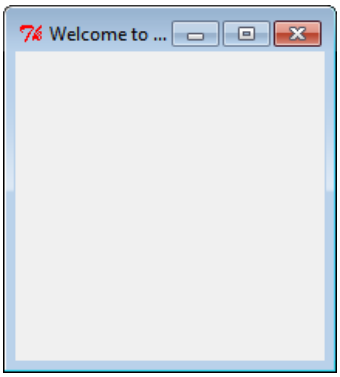
- The place() Method -This geometry manager organizes widgets by placing them in a specific position in the parent widget.

1) Creation of a window/widget

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

Program:

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



2) Creating a window with specific dimensions and a label

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

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

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

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

So the complete code will be like this:

Program

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

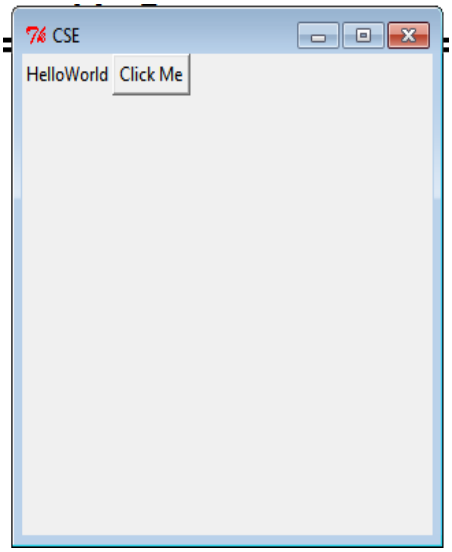
3)Adding a Button to window/widget

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

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

Program:

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



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

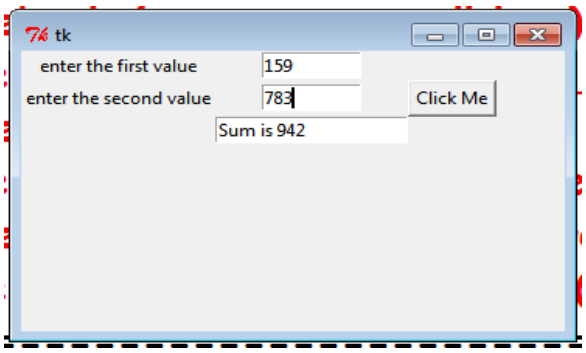
You can create a textbox using Tkinter Entry class like this:

= Entry(window,width=10)

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

Program:

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


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

5) Creating 2 checkboxes

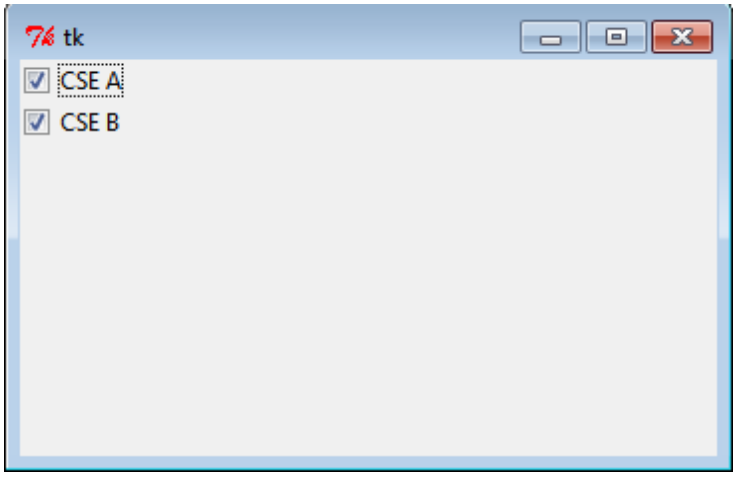
To create a checkbutton, you can use Checkbutton class like this:

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

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

the following program creates a check box.

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



6)Creating 3 radio buttons

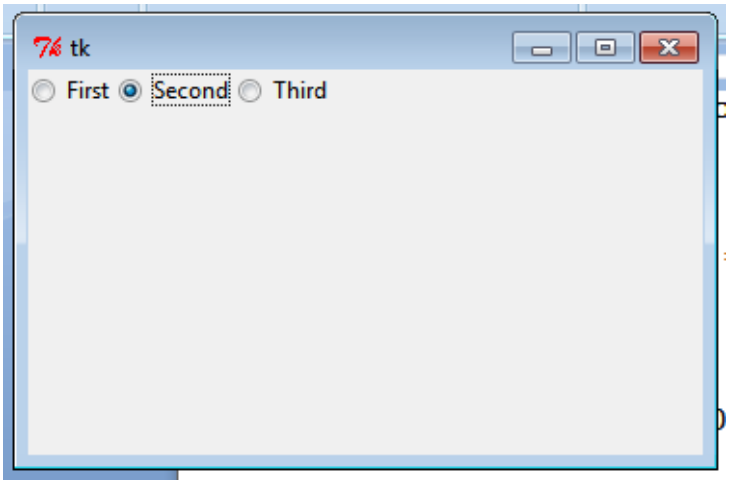
To add radio buttons, simply you can use RadioButton class like this:

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

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

the following program creates a check box

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



7) Creating a message box on clicking a button

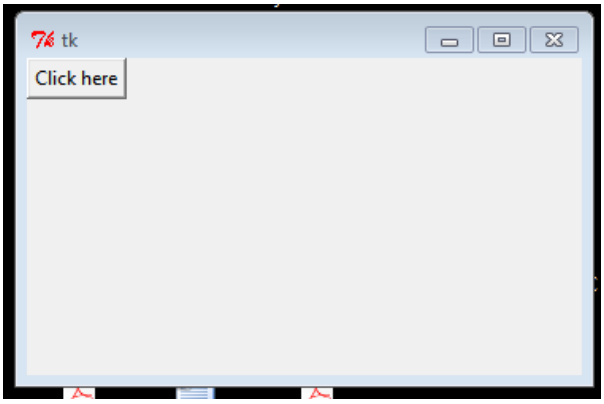
You can show a warning message or error message the same way. The only thing that needs to be changed is the message function.

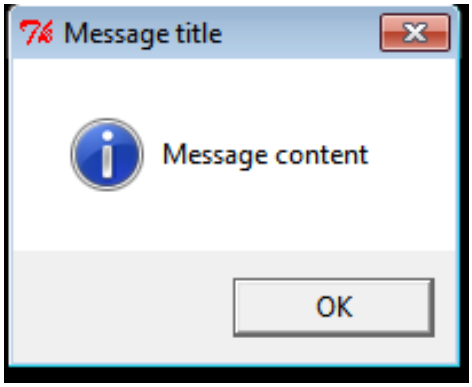
You can show a warning message or error message the same way. The only thing that needs to be changed is the message function

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

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

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



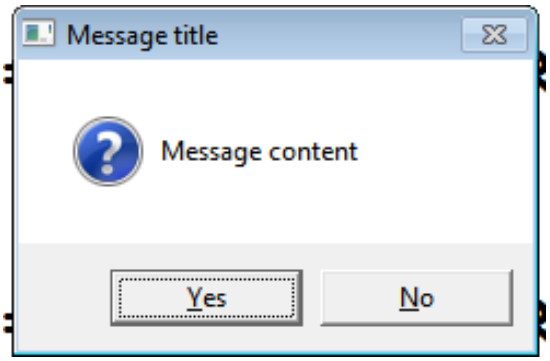

Message title
Message content
OK

8) Creating various message boxes

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

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

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



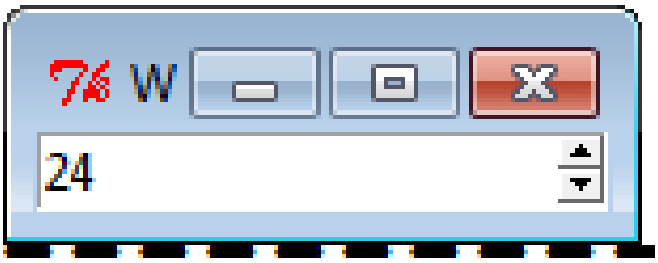
9) Creating a Spinbox

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

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

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

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



Introduction to programming concepts of scratch

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

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

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

Sprites

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

Sequences

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

Iteration (looping)

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

Conditional statements

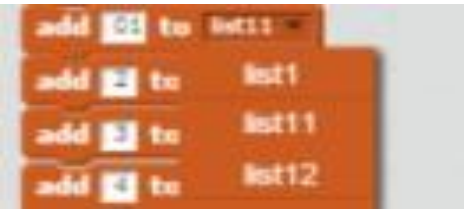
A conditional statement is a set of rules performed if a certain condition is met. In Scratch, the if and if-else blocks check for a condition.

Variables

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

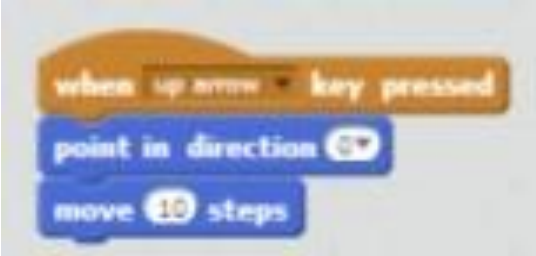
Lists (arrays)

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



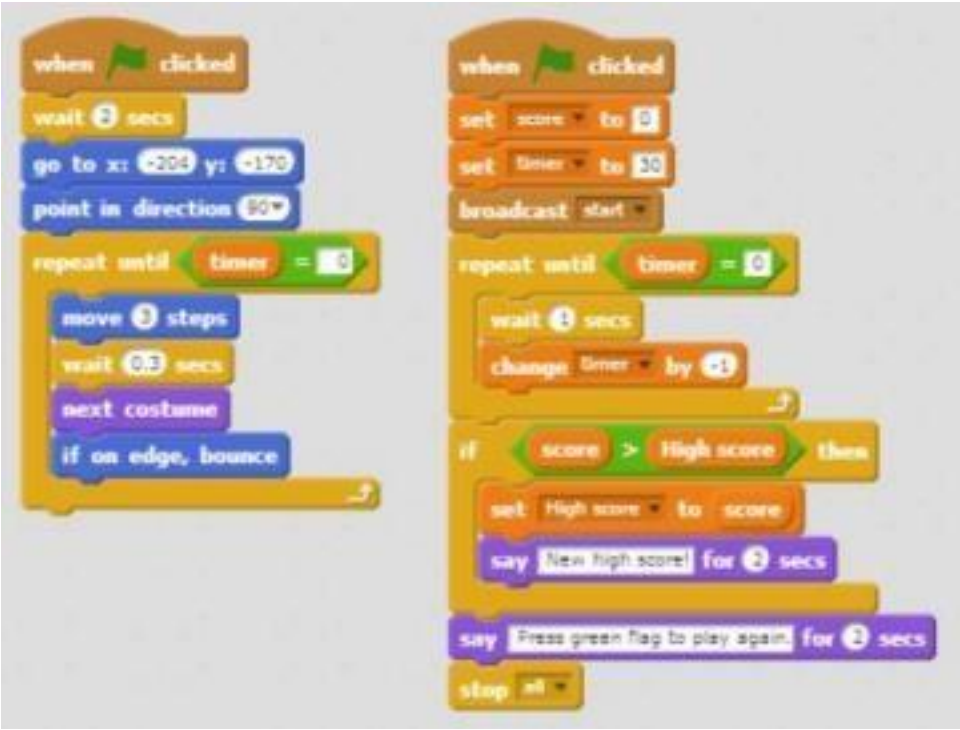
Event Handling

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



Threads

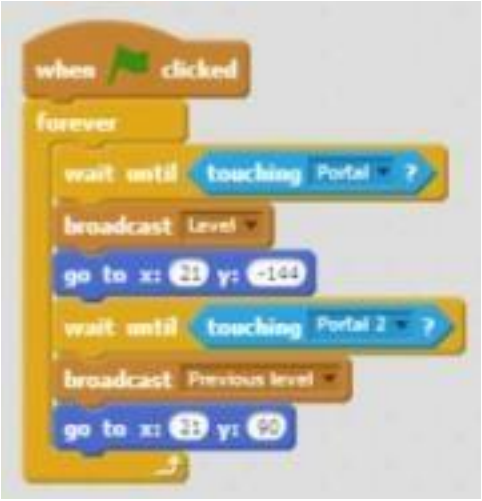
A thread just refers to the flow of a particular sequence of code within a program. A thread cannot run on its own, but runs within a program. When two threads launch at the same time it is called parallel execution.



Coordination & Synchronisation

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

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



Keyboard input

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

Boolean logic

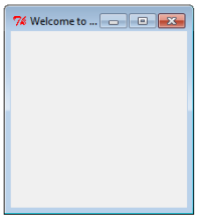
Boolean logic is a form of algebra in which all values are reduced to either true or false. The and, or, not statements are examples of Boolean logic.

User interface design

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

1) Creation of a window/widget

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

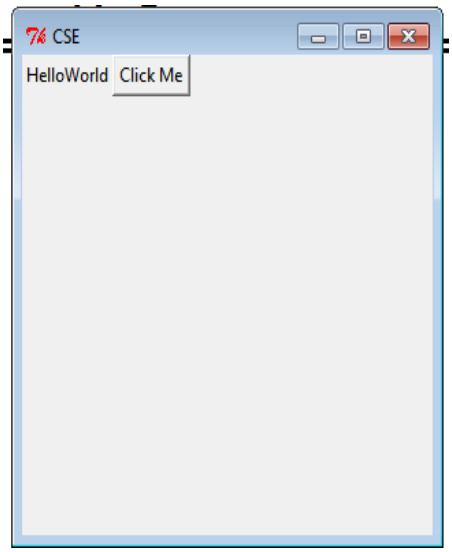


2) Creating a window with specific dimensions and a label

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

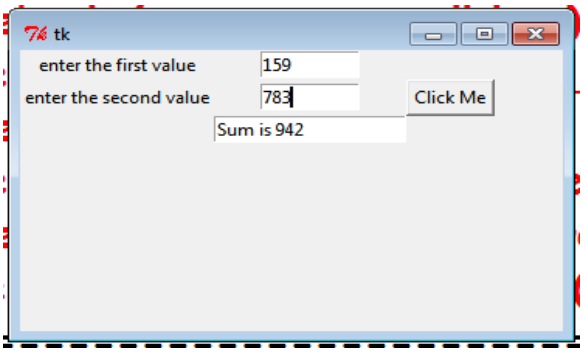
3)Adding a Button to window/widget

```
from tkinter import *

window = Tk()

window.geometry("300x300")

window.title("CSE")

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

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

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

window.mainloop()
```



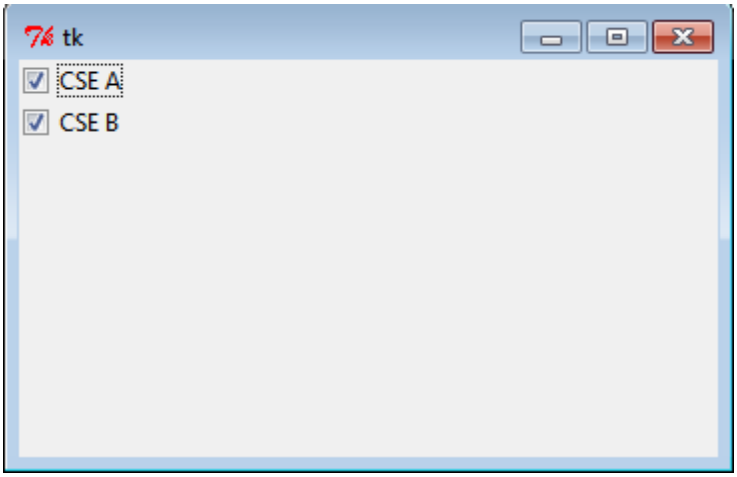
4)Creating 2 text fields to enter 2 numbers, and a button when clicked gives sum of the 2 numbers and displays it in 3rd text field

```
from tkinter import *

window = Tk()

window.geometry('350x200')

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

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

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

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

txt1 = Entry(window,width=10)

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

txt2 = Entry(window,width=10)

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

txt3 = Entry(window,width=20)

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

def clicked():

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

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

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

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

window.mainloop()
```



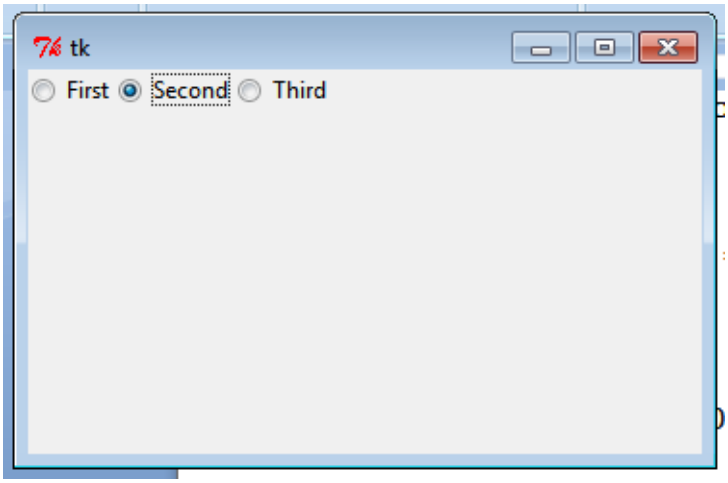
5) Creating 2 checkboxes

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



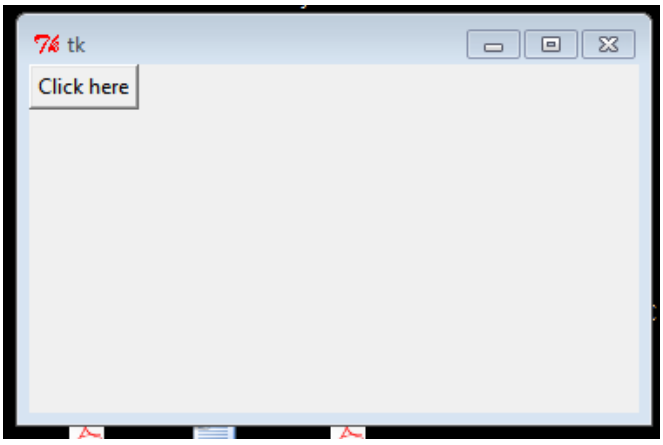
6)Creating 3 radio buttons

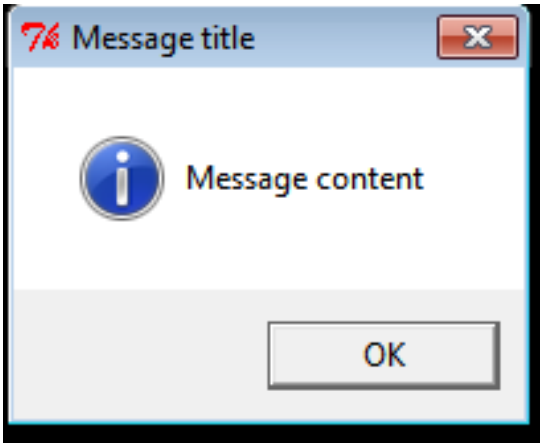
```
from tkinter import *
from tkinter.ttk import *
window = Tk()
window.geometry('350x200')
rad1 = Radiobutton(window,text='First', value=1)
rad2 = Radiobutton(window,text='Second', value=2)
rad3 = Radiobutton(window,text='Third', value=3)
rad1.grid(column=0, row=0)
rad2.grid(column=1, row=0)
rad3.grid(column=2,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



7) Creating a message box on clicking a button

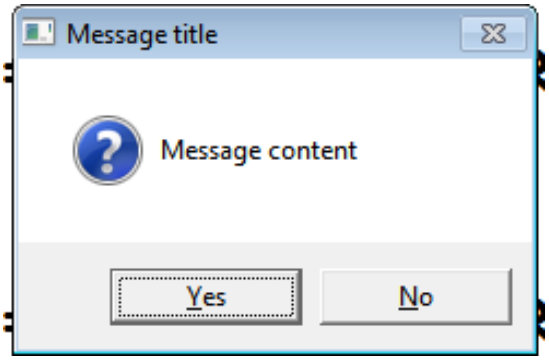
```
from tkinter import *
window = Tk()
window.geometry('350x200')
def clicked():
    messagebox.showinfo('Message title ', 'Message content')
    # messagebox.showerror('Message title ', 'Message content')
    # messagebox.show('Message title ', 'Message content')
btn = Button(window,text='Click here',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





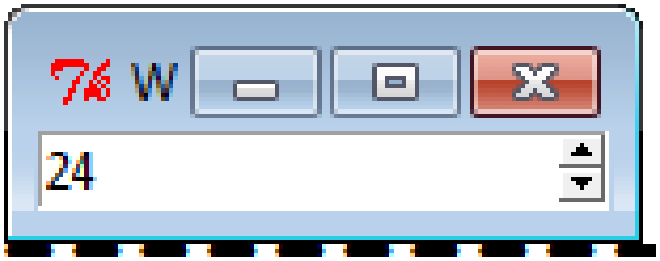
8) Creating various message boxes

```
from tkinter import messagebox

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

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

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

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

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



9) Creating a Spinbox

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



Code No: R201225 **R20** SET - 1

I B. Tech II Semester Regular Examinations, September- 2021

PYTHON PROGRAMMING

(Com. To CSE, IT, CSE-AI&ML,CSE-AI, CSE-DS, CSE-AI&DS, AI&DS)

Time: 3 hours Max. Marks: 70

Answer any five Questions one Question from Each Unit
All Questions Carry Equal Marks

~~~~~~

UNIT-I

1	a)	Summarize the precedence of mathematical operators in Python.	(7M)
	b)	Illustrate various conditional statements used in Python programming.	(7M)

Or

2	a)	Write a Python program to demonstrate explicit type conversion.	(7M)
	b)	Demonstrate the use of break and continue keywords in looping structure using a code snippet.	(7M)

UNIT-II

3	a)	Write a program to compute only even numbers sum within the given natural number using a continue statement.	(7M)
	b)	Is String a mutable data type? Also explain the string operations length and slicing in detail with an appropriate example	(7M)

Or

4	a)	Compare and contrast for loop and while loop.	(7M)
	b)	Write a Python program to check whether a given number is Armstrong number or not.	(7M)

UNIT-III

5	a)	Does mutability support for list, if yes explain any two methods with example?	(7M)
	b)	Write a program to read one subject mark and print pass or fail Use single return values function with argument.	(7M)

Or

6	a)	Write a brief note on PIP. Explain installing packages via PIP.	(7M)
	b)	Write a Python program to read a word and print the number of letters, vowels and percentage of vowels in the word using a dictionary.	(7M)

UNIT-IV

7	a)	Create a class Employee with data members name, department and salary. Create suitable methods for reading and printing employee information.	(7M)
	b)	How to implement method overriding in Python? Explain.	(7M)

Or

8	a)	Write a Python program that reads a text file and changes the file by capitalizing each character of file.	(7M)
	b)	Illustrate the concept of pure function with Python code.	(7M)

UNIT-V

9	a)	What s the difference between else block and finally block in exception handling? Explain with an example program.	(7M)
	b)	How to create two radio button sets (one for gender and another for Indian or not) on the same canvas? Illustrate.	(7M)

Or

10		Illustrate the use of the four main elements of scratch- Programming palette, storage area, Sprites and Script.	(14M)

|''|'||||''|''|||'|
~~~~~~