Unit -1: Introduction to Python

Introduction of Python

Python is a widely used general-purpose, high level programming language. It was initially **designed by Guido van Rossum in 1991** and developedby Python Software Foundation. It was mainly developed foremphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Python is a programming language that lets you work quickly and integrate systems more efficiently.

Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming on both small and large scales.

Python features adynamic typesystem and automatic memory management. It supportsmultipleprogramming paradigms, including object-oriented, imperative, functionalandprocedural, and has a large and comprehensive standard library.

Python interpreters are available for many operating systems. C-Python, the reference implementation of Python, is open source software and has a community-based development model, as do nearly all of its variant implementations. C-Python is managed by the non-profit Python Software Foundation.

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.
- Pythoncan 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-orientated way or a functional way.

Good to know

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

Python Syntax compared to other programming languages

- Python was designed to 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. Otherprogramminglanguages often use curly-brackets for this purpose.

History

When we talk about the history of Python, we cannot miss ABC programming language because it was ABCs influence that led to the design and development of programming language called Python.

In the early 1980s**, Van Rossum** used to work at CWI **(Centrum voor Wiskunde en Informatics)** as an implementer of the programming language called ABC. Later at CWI in the late 1980s, while working on a new distributed operating system called AMOEBA, Van Rossum started looking for a scripting language with syntax like ABC but with the access to the Amoeba system calls. So Van Rossum himself started designing a new simple scripting language that could overcome the flaws of ABC.

Van Rossum started developing the new script in the late 1980s and finally introduced the first version of that programming language in 1991. This initial release has module system of Modula-3. Later on, this programming language was named 'Python'.

History of different versions release

The first ever version of Python (i.e. Python 1.0) was introduced in 1991. Since its inception and introduction of Version 1, the evolution of Python has reached up to Version 3.x (till 2017).

Here is the brief chart depicting the timeline of the release of different versions of Python programming language.

A list of python versions with its released date is given below.

Python 1.0	January 1994	**Python 3.0**	December 3, 2008
Python 1.5	December 31, 1997	**Python 3.1**	June 27, 2009
Python 1.6	September 5, 2000	**Python 3.2**	February 20, 2011
Python 2.0	October 16, 2000	**Python 3.3**	September 29, 2012
Python 2.1	April 17, 2001	**Python 3.4**	March 16, 2014
Python 2.2	December 21, 2001	**Python 3.5**	September 13, 2015
Python 2.3	July 29, 2003	**Python 3.6**	December 23, 2016
Python 2.4	November 30, 2004	**Python 3.6.4**	December 19, 2017
Python 2.5	September 19, 2006	**Python 3.6.7**	October 27, 2018
Python 2.6	October 1, 2008	**Python 3.7.0**	June 27,2018
Python 2.7	July 3, 2010	**Python 3.7.1**	October 20, 2018

Applications Area

Python is known for its general purpose nature that makes it applicable in almost each domain of software development. Python as a whole can be used in any sphere of development.

Here, we are specifying applications areas where python can be applied.

1. **Web Applications**

   We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautiful Soup, Feed parser etc. It also provides Frameworks such as Django, Pyramid, Flask etc to design and develop web based applications.

   Some important developments are: PythonWikiEngines, Pocoo, PythonBlogSoftware etc.

2. **Desktop GUI Applications**

   Python provides Tk GUI library to develop user interface in python based application. Some other useful toolkits wxWidgets, Kivy, pyqt that are useable on several platforms. The Kivy is popular for writing multitouch applications.

3. **Software Development**

   Python is helpful for software development process. It works as a support language and can be used for build control and management, testing etc.

4. **Scientific and Numeric**

   Python is popular and widely used in scientific and numeric computing. Some usefullibrary and package are SciPy, Pandas, IPython etc. SciPy is group of packages of engineering, science and mathematics.

5. **Business Applications**

   Python is used to build Business applications like ERP and e-commerce systems.Tryton is a high level application platform.

6. **Console Based Application**

   We can use Python to develop console based applications.

   For example: IPython.

7. **Audio or video based Applications**

   Python is awesome to perform multiple tasks and can be used to develop multimedia applications. Some of real applications are: TimPlayer, cplay etc.

8. **3D CAD Applications**

   To create CAD application Fandango is a real application which provides full features of CAD.

9. Enterprise Applications

Python can be used to create applications which can be used within an Enterprise or an Organization. Some real time applications are: OpenErp, Tryton, Picalo etc.

10. Applications for Images

Using Python several applications can be developed for image. Applications developed are: VPython, Gogh, imgSeek etc.

There are several such applications which can be developed using Python

Differences between program and script

- **Program is executed**
  (I.e. the source is first compiled, and the result of that compilation is expected)
  A "program" in general, is a **sequence of instructions written so that a computer can perform certain task.**

- **Script is interpreted**
  A "script" is code written in a scripting language. A scripting language is nothing but **a type of programming language in which we can write code to control another** software application.

  It is a computer language with a series of commands within a file that is capable of being executed without being compiled.It has fewer syntactical constructions than other languages:

- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PHP
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Python Features

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read**: Python code is more clearly defined and visible to the eyes.
- **Easy-to-use:** Python can be easily used to run the programs.
- **Interpreted:** Python programs are compiled automatically to an intermediate form called byte-code, which the interpreter then reads. This gives Python the development speed of an interpreter without the performance loss inherent in purely interpreted languages
- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.

- **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode:** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code. In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms. As long as you have a compatible Python interpreter installed, Python programs will run in exactly the same manner, irrespective of platform
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable:** Python provides a better structure and support for large programs.
- Python supports functional and structured programming methods as well as oop concepts as polymorphism, operation overloading, and multiple inheritance.
- **Indentation:** Indentation is one of the greatest feature in Python makes programming effective.
- It's free (open source) as Downloading and installing Python is free and easy and Source code is easily accessible.
- **Powerful:** Python is powerful because of its dynamic typing, built in-types and tools, library utilities, automatic memory management, automatic garbage collection.

Python Weakness

- Python is slow.
- Python is un-typed, which means that a whole bunch of errors that are easily detected ahead of time aren't detected in Python until they happen.
- Syntactic whitespace can be nice, but it makes code generation harder than it has to be.
- You can't arbitrarily change the names of variables, because some variable occurrences can hide in strings, which makes the binding structure of Python un-decidable.
- It's near impossible to build a high-graphic 3D game using Python.
- Python is not good for multi-processor work.
- Python is not a very good language for mobile development.

Downloading

1. Click Python Download.
   The following page will appear in your browser.
2. Click the **Download Python 3.7.0** button.

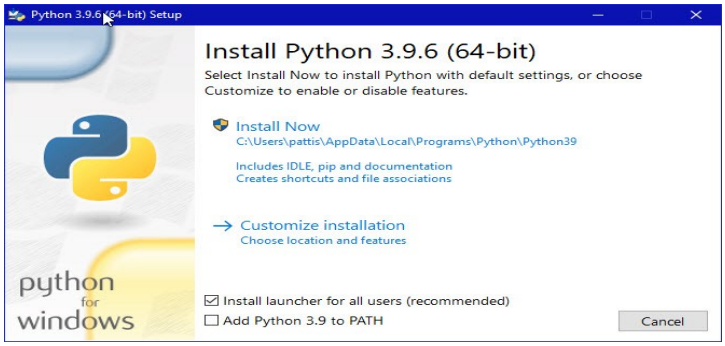
The file named **python-3.7.0.exe** should start downloading into your standard download folder. This file is about 30 Mb so it might take a while to download fully if you are on a slow internet connection (it took me about 10 seconds over a cable modem).
The file should appear as

3. Move this file to a more permanent location, so that you can install Python (and reinstall it easily later, if necessary).
4. Feel free to explore this webpage further; if you want to just continue the installation, you can terminate the tab browsing this webpage.
5. Start the **Installing** instructions directly below.

Installing

1. Double-click the icon labeling the file**python-3.7.0.exe**.
   An**Open File - Security Warning** pop-up windowwill appear.

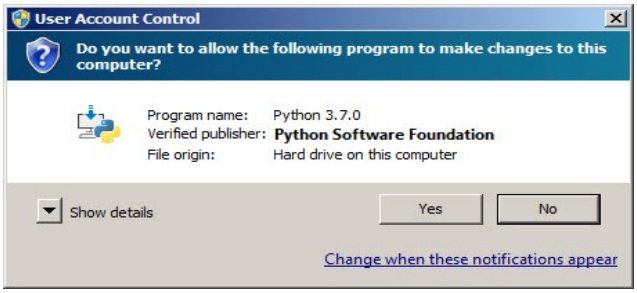
2. Click **Run**.
   A **Python 3.7.0 (32-bit) Setup** pop-upwindowwill appear.



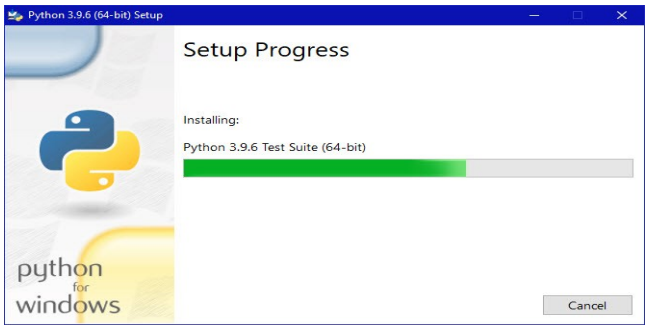
Ensure that the **Install launcher for all users (recommended)** and the **Add Python 3.7 to PATH** checkboxes at the bottom are checked.

If the Python Installer finds an earlier version of Python installed on your computer, the **Install Now** message will instead appear as **Upgrade Now** (and the checkboxes will not appear).

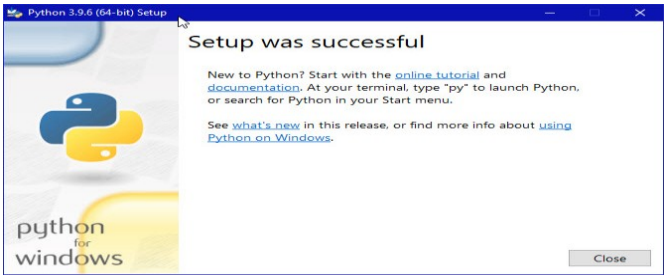
3. Highlight the **Install Now** (or **Upgrade Now**) message, and then click it.
   A **User Account Conrol** pop-up window will appear, posing the question **Do you want the allow the following program to make changes to this computer?**



4. Click the **Yes** button.
   A new **Python 3.7.0 (32-bit) Setup** pop-up window will appear with a **Setup Progress** message and a progress bar.



During installation, it will show the various components it is installing and move the progress bar towards completion. Soon, a new **Python 3.7.0 (32-bit) Setup**pop-up window will appear with a **Setup was successfully** message.

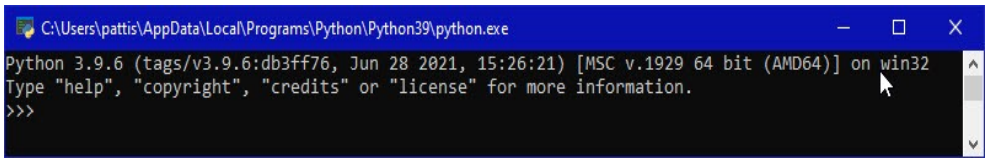


5. Click the **Close** button.
   Python should now be installed.

Verifying

To try to verify installation,

1. Navigate to the directory**C:\Users\Pattis\AppData\Local\Programs\Python\Python37-32**(or to whatever directory Python was installed: see the pop-up window for Installing step 3).
2. Double-click the icon/file **python.exe**.
   The following pop-up window will appear.



A pop-up window with the title **C:\Users\Pattis\AppData\Local\Programs\Python\Python37-32** appears, and inside the window; on the first line is the text **Python 3.7.0...**(notice that it should also say 32 bit). Inside the window, at the bottom left, is the **prompt >>>:** type **exit()** to this prompt and press **enter** to terminate Python.

You should keep the file **python-3.7.0.exe** somewhere on your computer in case you need to reinstall Python (not likely necessary).

You may now follow the instructions to download and install Java (if you have not already done so), and then the instruction to download and install the

Eclipse IDE (for Python, Java, or both). Note: you need to download/install Java even if you are using Eclipse only for Python)

Tokens

- Tokens can be defined as a punctuator mark, reserved words and each individual word in a statement.
- Token is the smallest unit inside the given program.
  There are following tokens in Python:
  - Keywords.
  - Identifiers.
  - Literals.
  - Operators.

Keywords

Keywords are the reserved words in Python. We cannot use a keyword as variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language. Keywords are case sensitive. There are 33 keywords in Python 3.3. This number can vary slightly in course of time. All the keywords except True, False and none are in lowercase and they must be written as it is. The lists of all the keywords are given below.

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

Identifiers

Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

Rules for identifiers

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myClass, var_1 and print_this_to_screen, all are valid example.
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
3. Keywords cannot be used as identifiers.
4. We cannot use special symbols like !, @, #, $, % etc. in our identifier.
5. Identifier can be of any length.

Variables

Variable is a name which is used to refer memory location. Variable also known as identifier and used to hold value. We don't need to specify the type of variable because Python is a type infer language and smart enough to get variable type.

Variable names can be a group of both letters and digits, but they have to begin with a letter or an underscore. It is recommended to use lowercase letters for variable name. **Jjkcc** and **jjkcc** both are two different variables.

Declaring Variable and Assigning Values

Python does not bind us to declare variable before using in the application. It allows us to create variable at required time. We don't need to declare explicitly variable in Python. When we assign any value to the variable that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

Eg:

Multiple Assignments

Python allows us to assign a value to multiple variables in a single statement which is also known as multiple assignments. We can apply multiple assignments in two ways either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Let's see given examples.

Assigning single value to multiple variables		**Assigning multiple values to multiple variables**	
Eg: x=y=z=50 print x print y print z	**Output:** >>> 50 50 50 >>>	**Eg** a,b,c=5,10,15 print a print b print c	**Output:** >>> 5 10 15 >>>

The values will be assigned in the order in which variables appears.

Literals

Literals can be defined as a data that is given in a variable or constant.

String literals

String literals can be formed by enclosing a text in the quotes. We can use both single aswellas double quotes for a String.

"Aman" , '12345'

Types of Strings:

There are two types of Strings supported in Python

Single line String	**Multi line String**	
Strings that are terminated within a single line are known as Single line Strings.	A piece of text that is spread along multiple lines is known as Multiple lines String. There are two ways to create Multiline Strings:	

Eg:
>>> text1='hello'
	1)**Adding black slash at the end of each line.**
	Eg:
	>>> text1='hello\
	user'
	>>> text1
	'hellouser'
	>>>
		2) **Using triple quotation marks:-**
		Eg:
		>>> str2='''welcome
		to
		SSSIT'''
		>>> print str2
		welcome
		to
		SSSIT
		>>>

Numeric literals

Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.

Int (signed integers)	Long (long integers)	Float (floating point)	Complex (complex)
Numbers (can be both positive and negative) with no fractional part.eg: 100	Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L	Real numbers with both integer and fractional part eg: -26.2	In the form of a+bj where a forms the real part and b forms the imaginary part of complex number. eg: 3.14j

Boolean literals:

A Boolean literal can have any of the two values: **True or False.**

Special literals.

Python contains one special literal i.e., None.None is used to specify to that field that is not created. It is also used for end of lists in Python.

Eg:

```
>>> val1=10
>>> val2=None
>>> val1
10
```

```
>>> val2
>>> print val2
None
>>>
```

Literal Collections.

The most basic data structure in Python is the sequence. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all the sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Lists

The list is the most versatile datatype available in Python, which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that the items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"];
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index.

Example	Output
#!/usr/bin/python3 **list1 = ['physics', 'chemistry', 1997, 2000]** **list2 = [1, 2, 3, 4, 5, 6, 7]** **print ("list1[0]: ", list1[0])** **print ("list2[1:5]: ", list2[1:5])**	list1[0]: physics list2[1:5]: [2, 3, 4, 5]

Updating Lists

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

Example	Output
#!/usr/bin/python3 **list = ['physics', 'chemistry', 1997, 2000]** **print ("Value available at index 2 : ", list[2])** **list[2] = 2001** **print ("New value available at index 2 : ", list[2])**	Value available at index 2 : 1997 New value available at index 2 : 2001

Delete List Elements

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

Example	Output
#!/usr/bin/python3 **list = ['physics', 'chemistry', 1997, 2000]** **print (list)** **del list[2]** **print ("After deleting value at index 2 : ", list)**	['physics', 'chemistry', 1997, 2000] After deleting value at index 2 : ['physics', 'chemistry', 2000]

Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1,2,3] : print (x,end = ' ')	1 2 3	Iteration

Indexing, Slicing and Matrixes

Since lists are sequences, indexing and slicing work the same way for lists as they do for strings.Assuming the following input

L = ['C++'', 'Java', 'Python']

Expression	Results	Description
L[2]	'Python'	Offsets start at zero
L[-2]	'Java'	Negative: count from the right
L[1:]	['Java', 'Python']	Slicing fetches sections

Built-in List Functions

No.	Function	Description
1	len(list)	Gives the total length of the list.
2	max(list)	Returns item from the list with max value.
3	min(list)	Returns item from the list with min value.
4	list(seq)	Converts a tuple into list.

Built-in List Methods

No.	Methods	Description
1	list.append(obj)	Appends object obj to list
2	list.count(obj)	Returns count of how many times obj occurs in list
3	list.extend(seq)	Appends the contents of seq to list
4	list.index(obj)	Returns the lowest index in list that obj appears
5	list.insert(index, obj)	Inserts object obj into list at offset index
6	list.pop(obj = list[-1])	Removes and returns last object or obj from list
7	list.remove(obj)	Removes object obj from list
8	list.reverse()	Reverses objects of list in place
9	list.sort([func])	Sorts objects of list, use compare func if given

Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The main difference between the tuples and the lists is that the tuples cannot be changed unlike lists. Tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally, you can put these comma-separated values between parentheses also.

tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5)
tup3 = "a", "b", "c", "d"

The empty tuple is written as two parentheses containing nothing

tup1 = ();

To write a tuple containing a single value you have to include a comma, even though there is only one value

tup1 = (50,)

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain the value available at that index.

Example	Output
tup1 = ('physics', 'chemistry', 1997, 2000) tup2 = (1, 2, 3, 4, 5, 6, 7) print ("tup1[0]: ", tup1[0]) print ("tup2[1:5]: ", tup2[1:5])	tup1[0]: physics tup2[1:5]: (2, 3, 4, 5)

Updating Tuples

Tuples are immutable, which means you cannot update or change the values of tuple elements. You are able to take portions of the existing tuples to create new tuples as the following example demonstrates

Example	Output

tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')
# Following action is not valid for tuples 	
# tup1[0] = 100; 	
# So let's create a new tuple as follows 	
tup3 = tup1 + tup2
print (tup3)
	(12, 34.56, 'abc', 'xyz')

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.
To explicitly remove an entire tuple, just use the del statement.

Example	Output
#!/usr/bin/python3 **tup = ('physics', 'chemistry', 1997, 2000);** **print (tup)** **del tup;** **print ("After deleting tup : ")** **print (tup)**	('physics', 'chemistry', 1997, 2000) After deleting tup : Traceback (most recent call last): File "test.py", line 9, in <module> print tup; NameError: name 'tup' is not defined

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1,2,3) : print (x, end = ' ')	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Since tuples are sequences, indexing and slicing work the same way for tuples as they do for strings, assuming the following input –
T=('C++', 'Java', 'Python')

Expression	Results	Description
T[2]	'Python'	Offsets start at zero
T[-2]	'Java'	Negative: count from the right
T[1:]	('Java', 'Python')	Slicing fetches sections

No Enclosing Delimiters

No enclosing Delimiters is any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples.

Built-in Tuple Functions

No.	Function	Description
1	len(tuple)	Gives the total length of the tuple.
2	max(tuple)	Returns item from the tuple with max value.
3	min(tuple)	Returns item from the tuple with min value.
4	tuple(seq)	Converts a list into tuple.

Dictionary

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

Example	Output
#!/usr/bin/python3 dict = {'Name': 'ABC', 'Age': 7, 'Class': 'First'} print ("dict['Name']: ", dict['Name']) print ("dict['Age']: ", dict['Age'])	dict['Name']: ABC dict['Age']: 7

If we attempt to access a data item with a key, which is not a part of the dictionary, we get an error as follows –

Example	Output
dict = {'Name': 'ABC', 'Age': 7, 'Class': 'First'}; print ("dict['Alice']: ", dict['Alice'])	dict['ABC']: Traceback (most recent call last): File "test.py", line 4, in <module> print "dict['Alice']: ", dict['Alice']; KeyError: 'Alice'

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown in a simple example given below.

Example	Output
dict = {'Name': 'ABC', 'Age': 7, 'Class': 'First'} dict['Age'] = 8; # update existing entry dict['School'] = "DPS School" # Add new entry print ("dict['Age']: ", dict['Age']) print ("dict['School']: ", dict['School'])	dict['Age']: 8 dict['School']: DPS School

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation. To explicitly remove an entire dictionary, just use the del statement.

Example	Output
dict = {'Name': 'ABC', 'Age': 7, 'Class': 'First'} del dict['Name'] # remove entry with key 'Name' dict.clear() # remove all entries in dict del dict # delete entire dictionary print ("dict['Age']: ", dict['Age']) print ("dict['School']: ", dict['School'])	dict['Age']: Traceback (most recent call last): File "test.py", line 8, in <module> print "dict['Age']: ", dict['Age']; TypeError: 'type' object is unsubscriptable

An exception is raised because after del dict, the dictionary does not exist anymore

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.There are two important points to remember about dictionary keys.

1. More than one entry per key is not allowed. This means no duplicate key is allowed. When duplicate keys are encountered during assignment, the last assignment wins.

Example	Output
dict = {'Name': 'ABC', 'Age': 7, 'Name': 'DEF'} print ("dict['Name']: ", dict['Name'])	dict['Name']: DEF

2. Keys must be immutable. This means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

Example	Output
dict = {['Name']: 'abc', 'Age': 7} print ("dict['Name']: ", dict['Name'])	Traceback (most recent call last): File "test.py", line 3, in <module> dict = {['Name']: 'abc', 'Age': 7} TypeError: list objects are unhashable

Built-in Dictionary Functions

No.	Function	Description
1	len(dict)	Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
2	str(dict)	Produces a printable string representation of a dictionary
3	type(variable)	Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Dictionary methods

No.	Method	Description
1	dict.clear()	Removes all elements of dictionary dict
2	dict.copy()	Returns a shallow copy of dictionary dict
3	dict.fromkeys()	Create a new dictionary with keys from seq and values set to value.
4	dict.get(key, default=None)	For key key, returns value or default if key not in dictionary
5	dict.items()	Returns a list of dict's (key, value) tuple pairs
6	dict.keys()	Returns list of dictionary dict's keys
7	dict.setdefault(key, default = None)	Similar to get(), but will set dict[key] = default if key is not already in dict
8	dict.update(dict2)	Adds dictionary dict2's key-values pairs to dict
9	dict.values()	Returns list of dictionary dict's values

Set Function

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed).

However, the set itself is mutable. We can add or remove items from it.

Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

Create a set

A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function set().

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary, as its element.

Example	Output
my_set = {1, 2, 3} print(my_set)	{1, 2, 3}

# set of mixed datatypes 	
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
	{1.0, 'Hello', (1, 2, 3)}

my_set = {1,2,3,4,3,2} print(my_set)	{1, 2, 3, 4}
my_set = set([1,2,3,2]) print(my_set)	{1, 2, 3}

Creating an empty set is a bit tricky.

Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements we use the set() function without any argument

Example	Output
my_set = {1,3}	{1, 3}

Change a set

Sets are mutable. But since they are unordered, indexing have no meaning.We cannot access or change an element of set using indexing or slicing. Set does not support it.

We can add single element using the add() method and multiple elements using the update() method. Theupdate()methodcan take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

print(my_set)	
my_set.add(2) print(my_set)	{1, 2, 3}

# add multiple elements 	
my_set.update([2,3,4])
print(my_set)
	{1, 2, 3, 4}

# add list and set 	
my_set.update([4,5], {1,6,8})
print(my_set)
	{1, 2, 3, 4, 5, 6, 8}

Remove elements from a set

A particular item can be removed from set using methods, discard() and remove().

The only difference between the two is that, while using discard() if the item does not exist in the set, it remains unchanged. But remove()will raise an error in such condition.

Example	Output
my_set = {1, 3, 4, 5, 6} print(my_set)	{1, 3, 4, 5, 6}

# discard an element 	
my_set.discard(4)
print(my_set)
	{1, 3, 5, 6}

# remove an element 	
my_set.remove(6)
print(my_set)
	{1, 3, 5}

# discard an element 	
my_set.discard(2)
print(my_set)
	{1, 3, 5}

Similarly, we can remove and return an item using the pop() method. Set being unordered, there is no way of determining which item will be popped. It is completely arbitrary.We can also remove all items from a set using clear().

Example	Output
my_set = set("HelloWorld") print(my_set)	{'r', 'W', 'H', 'o', 'l', 'd', 'e'}

# pop an element 	
print(my_set.pop())
	r

# pop another element 	
my_set.pop()
print(my_set)
	{'H', 'o', 'l', 'd', 'e'}

# clear my_set 	
my_set.clear()
print(my_set)
	set()

Set Methods

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with set objects.

Built in Methods

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set

discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns True if two sets have a null intersection
issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
pop()	Removes and returns an arbitary set element. Raise KeyError if the set is empty
remove()	Removes an element from the set. If the element is not a member, raise a KeyError
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

Built-in Functions with Set

Built-in functions like all(), any(), enumerate(), len(), max(), min(), sorted(), sum() etc. are commonly used with set to perform different tasks.

Function	Description
all()	Return True if all elements of the set are true (or if the set is empty).
any()	Return True if any element of the set is true. If the set is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of set as a pair.
len()	Return the length (the number of items) in the set.
max()	Return the largest item in the set.
min()	Return the smallest item in the set.
sorted()	Return a new sorted list from elements in the set(does not sort the set itself).
sum()	Return the sum of all elements in the set.

Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Set Operation & Description	Example	Output
UNION Union of A and B is a set of all elements from both sets. Union is performed using | operator. Same can be accomplished using the method union().	A = {1, 2, 3, 4, 5} B = {4, 5, 6, 7, 8} print(A | B) print(A.union(B)) print(B.union(A))	{1, 2, 3, 4, 5, 6, 7, 8} {1, 2, 3, 4, 5, 6, 7, 8} {1, 2, 3, 4, 5, 6, 7, 8}
Intersection Intersection of A and B is a	A = {1, 2, 3, 4, 5} B = {4, 5, 6, 7, 8}	{4, 5} {4, 5}

set of elements that are common in both sets. Intersection is performed using &operator. Same can be accomplished using the method intersection()	print(A & B) print(A.intersection(B)) print(B.intersection(A))	{4, 5}
Difference Difference of A and B (A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of element in B but not in A. Difference is performed using - operator. Same can be accomplished using the method difference().	A = {1, 2, 3, 4, 5} B = {4, 5, 6, 7, 8} print(A - B) A.difference(B) print(B – A) B.difference(A)	{1, 2, 3} {1, 2, 3} {8, 6, 7} {8, 6, 7}
Symmetric Difference Symmetric Difference of A and B is a set of elements in both A and B except those that are common in both. Symmetric difference is performed using ^ operator. Same can be accomplished using the method symmetric_difference()	A = {1, 2, 3, 4, 5} B = {4, 5, 6, 7, 8} print(A ^ B) A.symmetric_difference (B) B.symmetric_difference (A)	{1, 2, 3, 6, 7, 8} {1, 2, 3, 6, 7, 8} {1, 2, 3, 6, 7, 8} {1, 2, 3, 6, 7, 8}

Frozenset

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets.

Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

Frozensets can be created using the function frozenset (). This datatype supportsmethods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable it does not have methods that add or remove elements.

Example	Output
A= frozenset([1, 2, 3, 4]) B = frozenset([3, 4, 5, 6]) print(A.isdisjoint(B)) print(A.difference(B)) print(A | B) A.add(3)	False frozenset({1, 2}) frozenset({1, 2, 3, 4, 5, 6}) AttributeError: 'frozenset' object has no attribute 'add'

Basic Operators

Operators are the constructs which can manipulate the value of operands. Considerthe expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Types of Operator

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Assignment Operators
4. Logical Operators
5. Bitwise Operators
6. Membership Operators
7. Identity Operators

Arithmetic Operators

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

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

Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators. Assume variable a holds 10 and variable b holds 20, then

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

<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Assignment Operators

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

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
***= Multiply AND**	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
****= Exponent AND**	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //=a is equivalent to c = c // a

Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows –

a = 0011 1100 **b = 0000 1101**	**a&b = 0000 1100** **a|b = 0011 1101** **a^b = 0011 0001** **~a = 1100 0011**

There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b)(means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed

		binary number.
<< **Binary Left Shift**	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> **Binary Right Shift**	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then Used to reverse the logical state of its operand.

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then −

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

Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Identity Operators

Identity operators compare the memory locations of two objects.

Operator	Description	Example
Is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Operators Precedence(Hierarchy)

The following table lists all operators from highest precedence to lowest.

No	Operator	Description
1	**	Exponentiation (raise to the power)
2	~, +, -	Complement, unary plus and minus (method names for the last two are +@ and -@)
3	* ,/, %, //	Multiply, divide, modulo and floor division
4	+, -	Addition and subtraction
5	>>, <<	Right and left bitwise shift
6	&	Bitwise 'AND'
7	^, |	Bitwise exclusive `OR' and regular `OR'
8	<=, <, >, >=	Comparison operators
9	<> ,== ,!=	Equality operators
10	=, %=, /=, //=, -=, +=, *=, **=	Assignment operators
11	**Is , is not**	Identity operators
12	**In , not in**	Membership operators
13	**not , or , and**	Logical operators

Comment

Python Comment is a programmer's tool. We use them to explain the code, and the interpreter ignores them. You never know when a programmer may have to understand code written by another and make changes to it. Other times, give your brain a month's time, and it might forget what it once had conjured up in your code. For these purposes, good code will hold comments in the right places.

In C++, we have // for single-lined comments, and /* … */ for multiple-lined comments.

Single lined comment	Multi lined Comment
single line comment must start with **# Symbol**	Multi lined comment can be given inside **triple quotes.**
Eg: # This is single line comment.	Eg: ''''' This Is Multipline comment'''

Decision Control Structure

Decision making is anticipation of conditions occurring which is used to test specified condition. We can use if statement to perform conditional operations in our Python application.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

The if statement executes only when specified condition is true. We can pass any valid expression into the if parentheses.

There are various types of if statements in Python.

- if statement
- if-else statement
- nested if statement

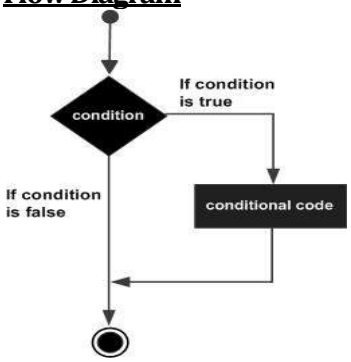
- if- elif ladder

if statement

if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax

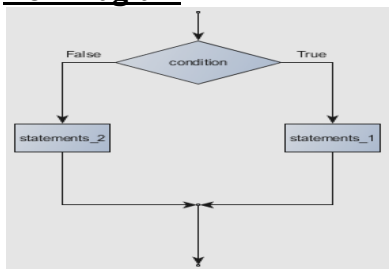
If the Boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If Boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

if expression: **Statement**	**Flow Diagram**  	Example if a==10: print "Value is 10" Output: Value is 10

As the condition present in the if statement is false. So, the block below the if statement is not executed.

if- else

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:
if (condition):
# Executes this block if 	
# condition is true 	
else:
# Executes this block else 	
# condition is false 	
	Example
	year=2000
	if year%4==0:
	print "Year is Leap"
	else:
	print "Year is not Leap"
	
	Output:
	Year is Leap
		Flow Diagram
		 

The block of code following the else statement is executed as the condition present in the if statement is false after call the statement which is not in block (without spaces).

Nested-if

A nested if is an if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:	**Example**

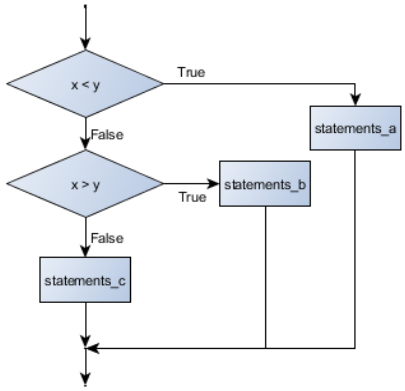
if (test condition 1):
# Executes when condition1 is true 	
if (test condition 2):
# Executes when condition1 and condition 2 is true 	
else:
# Executes when condition1 True and condition 2 is false 	
else:
# Executes when condition1 is false 	
	i =10
	if(i !=0):
	# First if statement
	if(i < 0):
	print("Positive Number")
	# Will only be executed if statement
	else:
	print("negative Number")
	else:
	print("ZERO Number")

	Output: Positive Number

if-elif-else ladder

A user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:	**Example**	

if (condition):
statement
elif (condition):
statement
.
.
else:
statement
	i =0
	if(i ==0):
	print("ZERO Number")
	elif(i>0)
	print("Positive Number")
	else:
	print("negative Number")
		

	Output: ZERO Number	

Iteration

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

Repeated execution of a set of statements is called **iteration**. Because iteration is so common, Python provides several language features to make it easier. We've already seen the for statement in chapter 3. This is the form of iteration you'll likely be using most often. But in this chapter we've going to look at the while statement — another way to have your program do iteration, useful in slightly different circumstances.

Before we do that, let's just review a few ideas

For Statement

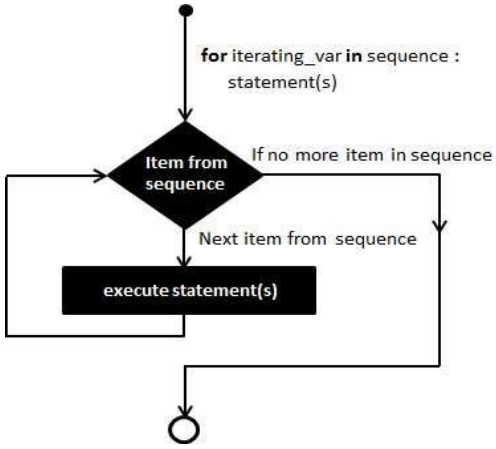
The simplest for statement looks like this:for variable in iterable. The suite is an indented block of statements. Any statement is allowed in the block, including indentedforstatements.

The variable is a variable name. The suite will be executed iteratively with the variable set to each of the values in the given iterable. Typically, the suitewill use the variable, expecting it to have a distinct value on each pass.

There are a number of ways of creating the necessary iterable collection of values. The most common is to use the range() function to generate a suitable list. We can also create the list manually, using a sequence display

The range() function has 3 forms:

- range(x) generates x distinct values, from 0 to x-1, incrementing by 1.
- Mathematicians describe this as a "half-open interval" and write it. (0, x)
- range(x, y) generates y-x distinct values from x to y-1, incrementing by 1. .(x,y)
- range(x, y, z) generates values from x to y-1, incrementing by z: ,[x,x + z,x+ 2z,… x+ kz < y] for some integer k.



A sequence display looks like this:

[expression ⟨ , … ⟩]

It's a list of expressions, usually simply numbers, separated by commas. The square brackets are essential for marking a sequence. examples.

```
for i in range(6):
    print i+1
```

This first example uses range() to create a sequence of 6 values from 0 to just before 6. The for statement iterates through the sequence, assigning each value to the local variable i. The print statement has an expression that adds one to i and prints the resulting value.

```
for i in range(1,7):
    print i
```

This second example uses the range() to create a sequence of 6 values from 1 to just before 7. The for statement iterates through the sequence, assigning each value to the local variable j . The print statement prints the value.

```
for i in range(1,36,2):
    print i
```

This example uses range() to create a sequence of 36/2=18 values from 1 to just before 36 stepping by 2. This will be a list of odd values from 1 to 35. The for statement iterates through the sequence, assigning each value to the local variable o. The print statement prints all 18 values.

```
for I in [1,3,5,7,9,12,14,16,18,19,21,23,25,27,30,32,34,36]:
    print i, "red"
```

This example uses an explicit sequence of values. These are all of the red numbers on a standard roulette wheel. It then iterates through the sequence,assigning each value to the local variable r. The print statement prints all 18 values followed by the word "red".

While Loop Statements

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

while expression:
statement(s)

Here, statement(s) may be a single statement or a block of statements. Theconditionmay be any expression, and true is any non-zero value. The loop iterates while the condition is true.

Whenthe condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

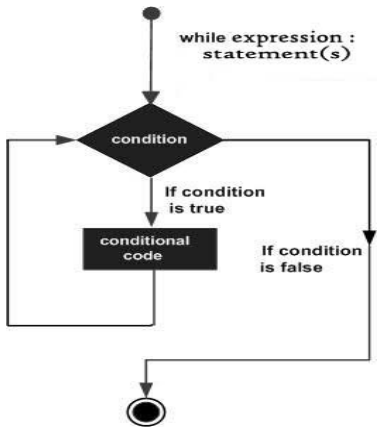
Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

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

output

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```



Else Statement with Loops

Python supports to have an else statement associated with a loop statement. If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.

If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

Example	**Output**
count = 0 while count < 5: print count, " is less than 5"	0 is less than 5 1 is less than 5 2 is less than 5

count = count + 1 else: print count, " is not less than 5"	3 is less than 5 4 is less than 5 5 is not less than 5

Nested loops

Python programming language allows to using one loop inside another loop. Followingsection shows few examples to illustrate the concept.

Syntax	
for iterating_var in sequence: **for iterating_var in sequence:** **statements(s)** **statements(s)**	**while expression:** **while expression:** **statement(s)** **statement(s)**

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

```
i = 2
while(i < 100):
  j = 2
  while(j <= (i/j)):
    if not(i%j): break
    j = j + 1
  if (j > i/j) : print i, " is prime"
  i = i + 1
print "Good bye!"
```

Loop Control Statements

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

S.N.	Control Statement and Description
1	**Break Statement** Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	**Continue Statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	**Pass Statement** The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Break Statement

It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.

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

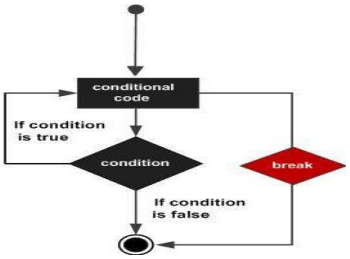
If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Example

```
for letter in 'Python':
  if letter == 'h':
break
  print 'Current Letter :', letter
```

Output

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



Continue statement

It returns the control to the beginning of the while loop. The continuestatement rejects all the remainingstatements in the current iteration of the loop andmoves the control back to the top of the loop.Thecontinuestatement can be used in bothwhileandforloops.

Syntax:

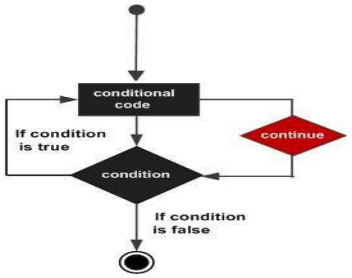
Continue

Example

```
for letter in 'Python':
  if letter == 'h':
    continue
  print 'Current Letter :', letter
```

Output

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



Pass Statement

It is used when a statement is required syntactically but you do not want any command or code to execute.

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

Syntax

Pass

Example

```
for letter in 'Python':
  if letter == 'h':
    pass
    print 'This is pass block'
```

Output

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
```

print 'Current Letter :', letter print "Good bye!"	Current Letter : o Current Letter : n Good bye!

Iterator

Iterator in python is any python type that can be used with a 'for in loop'. Python lists, tuples, dicts and sets are all examples of inbuilt iterators. These types are iteratorsbecause they implement following methods. In fact, any object that wants to be an iterator must implement following methods.

1. **__iter__** method that is called on initialization of an iterator. This should return an object that has a next or __next__ method.
2. **next (__next__)** The iterator next method should return the next value for the iterable. When an iterator is used with a 'for in' loop, the for loop implicitly calls next() on the iterator object. This method should raise a StopIteration to signal the end of the iteration.

The iter() method creates an object which can be iterated one element at a time.These objects are useful when coupled with loops like for loop, while loop.

syntax : - iter(object[, sentinel])

Parameters

object - object whose iterator has to be created (can be sets, tuples, etc.)

sentinel (Optional) - special value that is used to represent the end of a sequence

Depending upon the arguments passed, iter() must have the following attributes.

Different cases for iter() parameters		
Object	**Sentinel**	**Description**
Collection object (set, tuple)	None	Creates iterator for the collection object
User defined object (Custom object)	None	Either: implement __iter__() and __next__() methods implement __getitem__() method with integer arguments starting at 0
User defined object (Custom object)which doesn't implement __iter__() and __next() OR __getitem__()	None	Raises TypeError exception
Callable object	Provided	Returns iterator that calls object with no arguments for each call to its __next__() method If sentinel found, raises StopIteration exception

Return value from iter()

The iter() method returns iterator object for the given object that loops through each element in the object. In case of sentinel provided, it returns the iterator object that calls the callable object until the sentinel character isn't found.

Example	Output
my_list = [1, 2, 3] a=iter(my_list) print(next(a)) print(next(a)) print(next(a)) print(next(a))	1 2 3 Traceback (most recent call last): File "C:/Python37/Iter.py", line 6, in <module> print(next(a)) StopIteration

String and Input

String

A string is a sequence of characters. A character is simply a symbol. For example, the English language has 26 characters. Computers do not deal with characters; they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's. This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used.

In Python, string is a sequence of Unicode character. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

Example	Output
var1 = 'Hello World!' var2 = "Python Programming" print ("var1[0]: ", var1[0]) print ("var2[1:5]: ", var2[1:5])	var1[0]: H var2[1:5]: ytho

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether

Example	Output
var1 = 'Hello World!' print ("Updated String :- ", var1[:6] + 'Python')	Updated String :- Hello Python

Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range

		0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String Special Operators

Assume string variable a holds 'Hello' and variable b holds 'Python', then –

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give - HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family.

print (" My College Name is %s" %('JJKCC'))

When the above code is executed, it produces the following result

My College Name is JJKCC

Built-in String Methods

No.	Methods	Description
1	**capitalize()**	Capitalizes first letter of string
2	**center(width, fillchar)**	Returns a string padded with *fillchar* with the original

		string centered to a total of *width* columns.
3	**count(str, beg = 0,end = len(string))**	Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	**decode(encoding = 'UTF-8',errors = 'strict')**	Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	**encode(encoding = 'UTF-8',errors = 'strict')**	Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	**endswith(suffix, beg = 0, end = len(string))**	Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	**expandtabs(tabsize = 8)**	Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	**find(str, beg = 0 end = len(string))**	Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	**index(str, beg = 0, end = len(string))**	Same as find(), but raises an exception if str not found.
10	**isalnum()**	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	**isalpha()**	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	**isdigit()**	Returns true if string contains only digits and false otherwise.
13	**islower()**	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	**isnumeric()**	Returns true if a unicode string contains only numeric characters and false otherwise.
15	**isspace()**	Returns true if string contains only whitespace characters and false otherwise.
16	**istitle()**	Returns true if string is properly "titlecased" and false otherwise.
17	**isupper()**	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	**join(seq)**	Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	**len(string)**	Returns the length of the string
20	**ljust(width[, fillchar])**	Returns a space-padded string with the original string left-justified to a total of width columns.
21	**lower()**	Converts all uppercase letters in string to lowercase.
22	**lstrip()**	Removes all leading whitespace in string.

23	**maketrans()**	Returns a translation table to be used in translate function.
24	**max(str)**	Returns the max alphabetical character from the string str.
25	**min(str)**	Returns min alphabetical character from the string str.
26	**replace(old, new [, max])**	Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	**rfind(str, beg = 0,end = len(string))**	Same as find(), but search backwards in string.
28	**rindex(str, beg = 0, end = len(string))**	Same as index(), but search backwards in string.
29	**rjust(width,[, fillchar])**	Returns a space-padded string with the original string right-justified to a total of width columns.
30	**rstrip()**	Removes all trailing whitespace of string.
31	**split(str="", num=string.count(str)) .**	Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given
32	**splitlines(num=string.count('\n'))**	Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	**startswith(str, beg=0,end=len(string))**	Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	**strip([chars])**	Performs both lstrip() and rstrip() on string
35	**swapcase()**	Inverts case for all letters in string.
36	**title()**	Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	**translate(table, deletechars="")**	Translates string according to translation table str(256 chars), removing those in the del string.
38	**upper()**	Converts lowercase letters in string to uppercase.
39	**zfill (width)**	Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	**isdecimal()**	Returns true if a unicode string contains only decimal characters and false otherwise.

Input Function

There are hardly any programs without any input. Input can come in various ways, for example, from a database, another computer, mouse clicks and movements or from the internet. Yet, in most cases the input stems from the keyboard. For this purpose, Python provides the function input (). Input () has an optional parameter, which is the prompt string.

If the input function is called, the program flow will be stopped until the user has given an input and has ended the input with the return key. The text of the optional parameter, i.e. the prompt, will be printed on the screen.

The input of the user will be returned as a string without any changes. If this raw input has to be transformed into another data type needed by the algorithm, we can use either a casting function or the eval function.

Let's have a look at the following example:

```
person = input('Enter your name: ')
print('Hello', person)
```

Run the program. In the Shell you should see

Enter your name:

Follow the instruction (and press Enter). Make sure the typing cursor is in the Shell window, at the end of this line. After you type your response, you can see that the program has taken in the line you typed. That is what the built-in function input does: First it prints the string you give as a parameter (in this case 'Enter your name: '), and then it waits for a line to be typed in, and returns the string of characters you typed. In the **hello_you.py**program this value is assigned to the variable person, for use later.

Print with Keyword Parameter

One way to put punctuation but no space after the person in hello_you.py is to use the plus operator, +. Another approach is to change the default separator between fields in the print function. This will introduce a new syntax feature, keyword parameters. The print function has a keyword parameter named sep. If you leave it out of a call to print, as we have so far, it is set equal to a space by default. If you add a final field, sep='', in the print function in hello_you.py, you get the following example file, hello_you2.py:

```
'''Hello to you!  Illustrates sep with empty string in print.'''
person = input('Enter your name: ')
print('Hello ', person, '!')
```

Try the program.

Keyword parameters must be listed at the end of the parameter list.

Numbers and Strings of Digits

Consider the following problem: Prompt the user for two numbers, and then print out a sentence stating the sum. For instance if the user entered 2 and 3, you would print 'The sum of 2 and 3 is 5.'

You might imagine a solution like the example file addition1.py, shown below. There is a problem. Can you figure it out before you try it? Hint: [2]

```
'''Error in addition from input.'''
x = input("Enter a number: ")
y = input("Enter a second number: ")
print('The sum of ', x, ' and ', y, ' is ', x+y, '.', sep='') #error
```

End up running it in any case.

We do not want string concatenation, but integer addition. We need integer operands. Briefly mentioned in Whirlwind Introduction To Types and Functions was the fact that we can use type names as functions to convert types. One approach would be to

do that. Further variable names are also introduced in the example addition2.py file below to emphasize the distinctions in types. Read and run:

```
'''Conversion of strings to int before addition'''
xString = input("Enter a number: ")
x = int(xString)
yString = input("Enter a second number: ")
y = int(yString)
print('The sum of ', x, ' and ', y, ' is ', x+y, '.', sep='')
```

Needing to convert string input to numbers is a common situation, both with keyboard input and later in web pages. While the extra variables above emphasized the steps, it is more concise to write as in the variation in example file, addition3.py, doing the conversions to type int immediately:

```
'''Two numeric inputs, with immediate conversion'''
x = int(input("Enter a number: "))
y = int(input("Enter a second number: "))
print('The sum of ', x, ' and ', y, ' is ', x+y, '.', sep='')
```

The simple programs so far have followed a basic *programming pattern*: input-calculate-output. Get all the data first, calculate with it second, and output the results last. The pattern sequence would be even clearer if we explicitly create a named result variable in the middle, as in addition4.py

```
'''Two numeric inputs, explicit sum'''
x = int(input("Enter an integer: "))
y = int(input("Enter another integer: "))
sum = x+y
print('The sum of ', x, ' and ', y, ' is ', sum, '.', sep='')
```

We will see more complicated patterns, which involve repetition, in the future. The input function produces values of string type.

Exercise for Addition

Write a version, add3.py that asks for three numbers, and lists all three, and their sum, in similar format to addition4.py displayed above.

Exercise for Quotients

Write a program, quotient.py that prompts the user for two integers, and then prints them out in a sentence with an integer division problem like

The quotient of 14 and **3** is **4** with **a remainder of 2**

Review Division and Remainders if you forget the integer division or remainder operator.

String Format Operation

In grade school quizzes a common convention is to use fill-in-the blanks. For instance,

Hello ____!

and you can fill in the name of the person greeted, and combine given text with a chosen insertion. *We use this as an analogy:* Python has a similar construction, better called fill-in-the-braces. There is a particular operation on strings called format, that

makes substitutions into places enclosed in braces. For instance the example file, hello_you3.py, creates and prints the same string as in hello_you2.py from the previous section:

```
'''Hello to you!  Illustrates format with {} in print.'''
person = input('Enter your name: ')
greeting = 'Hello, {}!'.format(person)
print(greeting)
```

There are several new ideas here!

First *method* calling syntax for *objects* is used. You will see this very important modern syntax in more detail at the beginning of the next chapter in Object Orientation. All data in Python are objects, including strings. Objects have a special syntax for functions, called *methods*, associated with the *particular type of object*. In particular str objects have a method called format. The syntax for methods has the object followed by a period followed by the method name, and further parameters in **parentheses.**

***object.methodname*(*parameters*)**

In the example above, the object is the string 'Hello {}!'. The method is named format. There is one further parameter, person.

The string for the format method has a special form, with braces embedded. Places where braces are embedded are replaced by the value of an expression taken from the parameter list for the format method. There are many variations on the syntax between the braces. In this case we use the syntax where the first (and only) location in the string with braces has a substitution made from the first (and only) parameter.

In the code above, this new string is assigned to the identifier greeting, and then the string is printed.

The identifier greeting was introduced to break the operations into a clearer sequence of steps. However, since the value of greeting is only referenced once, it can be eliminated with the more concise version:

```
person = input('Enter your name: ')
print('Hello {}!'.format(person))
```

Consider the interview program. Suppose we want to add a period at the end of the sentence (with no space before it). One approach would be to combine everything with plus signs. Another way is printing with keyword sep=''. Another approach is with string formatting. Using our grade school analogy, the idea is to fill in the blanks in
_____ will interview _____ at _____.

There are multiple places to substitute, and the format approach can be extended to multiple substitutions: Each place in the format string where there is '{}', the format operation will substitute the value of the next parameter in the format parameter list.

Run the example file interview2.py, and check that the results from all three methods match.

```
'''Compare print with concatenation and with format string.'''
applicant = input("Enter the applicant's name: ")
interviewer = input("Enter the interviewer's name: ")
time = input("Enter the appointment time: ")
print(interviewer + ' will interview ' + applicant + ' at ' + time +'.')
print(interviewer, ' will interview ', applicant, ' at ', time, '.', sep='')
print('{} will interview {} at {}.'.format(interviewer, applicant, time))
```

Sometimes you want a single string, but not just for printing. You can combine pieces with the + operator, but then all pieces must be strings or *explicitly converted to strings*. An advantage of the format method is that it will convert types to string automatically, like the print function. Here is another variant of our addition sentence example, addition4a.py, using the format method.

```
'''Two numeric inputs, explicit sum'''
x = int(input("Enter an integer: "))
y = int(input("Enter another integer: "))
sum = x+y
sentence = 'The sum of {} and {} is {}.'.format(x, y, sum)
print(sentence)
```

Conversion to strings was not needed in interview2.py. (Everything started out as a string.) In addition4a.py, however, the automatic conversion of the integers to strings is useful.

So far there is no situation that requires a format string instead of using other approaches. Sometimes a format string provides a shorter and simpler expression. Except where specifically instructed in an exercise for practice, use whatever approach to combining strings and data that you like. There are many elaborations to the fields in braces to control formatting. We will look at one later, String Formats for Float Precision, where format strings are particularly useful.

A technical point: Since braces have special meaning in a format string, there must be a special rule if you want braces to actually be included in the final*formatted*string. The rule is to double the braces: '{{' and '}}'. The example code formatBraces.py, shown below, makes setStr refer to the string 'The set is {5,9}.'. The initial and final doubled braces in the format string generate literal braces in the formatted string:

```
'''Illustrate braces in a formatted string.'''
a = 5
b = 9
setStr = 'The set is {{{}, {}}}.'.format(a, b)
print(setStr)
```

This kind of format string depends directly on the order of the parameters to the format method. There is another approach with a dictionary, that was used in the first sample program, madlib.py, and will be discussed more in Dictionaries and String Formatting. The dictionary approach is probably the best in many cases, but the count-based approach is an easier start, particularly if the parameters are just used once, in order.

Optional elaboration with explicitly numbered entries

Imagine the format parameters numbered in order, starting from 0. In this case 0, 1, and 2. The number of the parameter position may be included inside the braces, so an alternative to the last line of interview2.py is (added in example file interview3.py):

```
print('{0} will interview {1} at {2}.'.format(interviewer, applicant, time))
```

This is more verbose than the previous version, with no obvious advantage. However, if you desire to use some of the parameters more than once, then the approach with the numerical identification with the parameters is useful. Every place the string includes '{0}', the format operation will substitute the value of the initial parameter in the list. Wherever '{1}' appears, the next format parameter will be substituted....

Predict the results of the example file arith.py shown below, if you enter 5 and 6. Then check yourself by running it. In this case the numbers referring to the parameter positions are necessary. They are both repeated and used out of order:

```
'''Fancier format string example with
parameter identification numbers
-- useful when some parameters are used several times.'''

x = int(input('Enter an integer: '))
y = int(input('Enter another integer: '))
formatStr = '{0} + {1} = {2}; {0} * {1} = {3}.'
equations = formatStr.format(x, y, x+y, x*y)
print(equations)
```

Try the program with other data.

Now that you have a few building blocks, you will see more exercises where you need to start to do creative things. You are encouraged to go back and reread Learning to Problem-Solve.

What is a function in Python?

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Python gives you many built-in functions like print (), etc. but you can also create your own functions. These functions are called ***user-defined functions.***

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

1. Keyword **def** marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of identifiers
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (**docstring**) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

Syntax

```
def functionname( parameters ):
        "function_docstring"
        function_suite
        return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print (str)
    return
```

Docstring

The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does.

Although optional, documentation is a good programming practice. Unless you can remember what you had for dinner last week, always document your code.

```
def docstri(name):
        """This function greets to  the person passed in asparameter"""
        print("Hello, " + name + ". Good morning!")

print (docstri.__doc__)
This functiondocstri to
  the person passed into the
  name parameter
```

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as __doc__ attribute of the function.

Calling a Function

Defining a function gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is an example to call the printme() function –

```
# Function definition
def printme( str ):
        "This prints a passed string into this function"
        print (str)
        return
# call function
        printme("This is first call user defined function!")
        printme("Now Again second Time call to the same function")
```

Output:

```
This is first call user defined function!
Now Again second Time call to the same function
```

Pass by Reference vs Value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example –

```
def changeme( mylist ):
```

```
   "This changes a passed list into this function"
   print ("Values inside the function before change: ", mylist)
  mylist[2]=50
   print ("Values inside the function after change: ", mylist)
   return

# call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

Here, we are maintaining reference of the passed object and appending values in the same object. Therefore, this would produce the following result −

```
Values inside the function before change:  [10, 20, 30]
Values inside the function after change:  [10, 20, 50]
Values outside the function:  [10, 20, 50]
```

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
# Function definition is here
def changeme( mylist ):
"This changes a passed list into this function"
mylist = [1,2,3,4] # This would assi new reference in mylist
print ("Values inside the function: ", mylist)
return

# Now you can call changeme function
        mylist = [10,20,30]
        changeme( mylist )
        print ("Values outside the function: ", mylist)
```

The parameter **mylist** is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result −

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

Function Arguments

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

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function **printme()**, you definitely need to pass one argument, otherwise it gives a syntax error as follows –

Example	Output
Function definition is here **def printme(str):** **"This prints a passed string into this function"** **print (str)** **return** **# call printme function** **printme()**	**Traceback (most recent call last):** **File "test.py", line 11, in <module>** **printme();** **TypeError: printme() takes exactly 1 argument (0 given)**

Keyword Arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the **printme()** function in the following ways

Example	Output
# Function definition is here **def printme(str):** **"This prints a passed string into this function"** **print (str)** **return** **# call function** **printme(str = "My string")**	**My string**

The following example gives a clearer picture. Note that the order of parameters does not matter.

Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed –

Example	Output
# Function definition is here **def printinfo(name, age = 35):** **"This prints a passed info into this function"** **print ("Name: ", name)** **print ("Age ", age)** **return** **# all Functin** **printinfo(age = 50, name = "JJKCC")** **printinfo(name = "JJKCC")**	**Name: JJKCC** **Age 50** **Name: JJKCC** **Age 35**

Variable-length Arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is given below –

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all non keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example

Example	Output
def printinfo(arg1,*vartuple): **"This prints a variable passed arguments"** **print("Output is: ")** **print(arg1)** **forvarin vartuple:** **print(var)** **return** **printinfo(10)** **printinfo(70,60,50)**	**Output is** **10** **Output is** **70** **60** **50**

The Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the **def** keyword. You can use the **lambda** keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambdas are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is to stack allocation by passing function, during invocation for performance reasons.

Syntax

The syntax of **lambda** functions contains only a single statement, which is as follows –

lambda [arg1 [,arg2,.....argn]]:expression

Following is an example to show how **lambda** form of function works –

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2

# Now you can call sum as a function
print ("Value of total : ", sum( 10, 20 ))
print ("Value of total : ", sum( 20, 20 ))
```

When the above code is executed, it produces the following result –

```
Value of total :  30
Value of total :  40
```

The return Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return

Example	Output

# Function definition is here 	
def sum(arg1, arg2):
  # Add both the parameters and return them."
  total = arg1 + arg2
  print ("Inside the function : ", total)
  return total
 	
# Now call function 	
total = sum(10, 20)
print ("Outside the function : ", total)
	Inside the function : 30
	Outside the function : 30

Recursion

What is recursion?

Recursion is the process of defining something in terms of itself. A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Recursive Function

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions. Following is an example of recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720

Example

```
def calc_factorial(x):
   """This is a recursive function   to find the factorial of an integer"""
   if x == 1:
      return 1
   else:
      return (x * calc_factorial(x-1))
num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

In the above example, calc_factorial() is a recursive functions as it calls itself. When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function call multiples the number with the factorial of number 1 until the number is equal to one. This recursive call can be explained in the following steps.

```
alc_factorial(4)          # 1st call with 4
4 * calc_factorial(3)       # 2nd call with 3
4 * 3 * calc_factorial(2)     # 3rd call with 2
4 * 3 * 2 * calc_factorial(1)  # 4th call with 1
4 * 3 * 2 * 1             # return from 4th call as number=1
4 * 3 * 2               # return from 3rd call
4 * 6                 # return from 2nd call
24                  # return from 1st call
```

Our recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

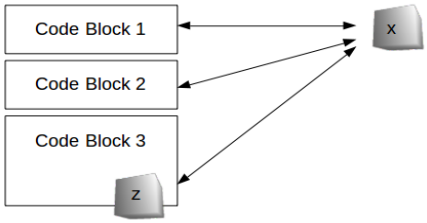
Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

Global Variables



There are two types of variables: **global variables** and **local variables**.A global variable can be reached anywhere in the code, a local only in the scope.

A **global variable** (x) can be reached and modified anywhere in the code; **local variable** (z) exists only in block 3.

Local variables	Global variables
Local variables can only be reached in their scope. The example below has two local variables: x and y. **def sum(x,y):** **sum = x + y** **return sum** **print(sum(8,6))** The variables x and y can only be used inside the function sum, they don't exist outside of the function. Local variables cannot be used outside of their scope, this line will not work: print(x)	A global variable can be used anywhere in the code. In the example below we define a global variable z **z = 10** **def afunction():** **global z** **print(z)** **afunction()** **print(z)** The global variable z can be used all throughout the program, inside functions or outside. A global variable can modified inside a function and change for the entire program: **z = 10** **def afunction():** **global z** **z = 9** **afunction()** **print(z)** After calling afunction(), the global variable is changed for the entire program

Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable __name__. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents.

Example

The Python code for a module named aname normally resides in a file named aname.py. Here's an example of a simple module, support.py

```
def print_func( par ):
  print "Hello : ", par
  return
```

The import Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax −

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module support.py, you need to put the following command at the top of the script −

```
# Import module support
import support
# Now you can call defined function that module as follows
support.print_func("JJKCC")
```

When the above code is executed, it produces the following result −

```
Hello : JJKCC
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

The from...import Statement

Python's from statement lets you import specific attributes from a module into the current namespace. The from...import has the following syntax –

from modname import name1[, name2[, ... nameN]]

For example, to import the function fibonacci from the module fib, use the following statement –

from fib import fibonacci

This statement does not import the entire module fib into the currentnamespace;it just introduces the item fibonacci from the module fib into the global symboltable of the importing module.

The from...import * Statement

It is also possible to import all names from a module into the current namespace by using the following import statement –

from modname import *

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the**sys.path**variable. The **sys.path** variable contains the current directory, PYTHONPATH, and theinstallation-dependentdefault.

The PYTHONPATH Variable

The PYTHONPATH is an environment variable, consisting of a list of directories. Thesyntax of PYTHONPATH is the same as that of the shell variable PATH.Here is a typical PYTHONPATH from a Windows system –

set PYTHONPATH = c:\python37\lib;

And here is a typical PYTHONPATH from a UNIX system

set PYTHONPATH = /usr/local/lib/python

Namespaces and Scoping

Variables are names (identifiers) that map to objects. A ***namespace*** is a dictionary of variable names (keys) and their corresponding objects (values).

A Python statement can access variables in **a *local namespace*** and in the ***global namespace*.** If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

The statement ***global VarName*** tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable *Money* in the global namespace. Within the function ***Money*****,** we assign *Money* a value, therefore Python assumes ***Money*****as** a local variable. However, we accessed the value of the local variable *Money* before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

```
#!/usr/bin/python
Money = 2000
def AddMoney():
  # Uncomment the following line to fix the code:
  # global Money
    Money = Money + 1
print Money
AddMoney()
        print Money
```

The dir( ) Function

The dir() built-in function returns a sorted list of strings containing the names defined by a module. The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example –

```
#!/usr/bin/python
# Import built-in module math
import math
content = dir(math)
print content
```

When the above code is executed, it produces the following result –

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

Here, the special string variable *__name__* is the module's name, and *__file__* is the filename from which the module was loaded.

The globals() andlocals() Functions

The ***globals()*** and ***locals()*** functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If **locals()** is called from within a function, it will return all the names that can be accessed locally from that function.

If **globals()** is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

The reload() Function

When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to re-execute the top-level code in a module, you can use the ***reload()*** function. The **reload()** function imports a previously imported module again. The syntax of the **reload()** function is this −

reload(module_name)

Here, ***module_name*** is the name of the module you want to reload and not the string containing the module name. For example, to reload *hello* module, do the following −

reload(hello)

Packages in Python

A package is a hierarchical file directory structure that defines a single Pythonapplication environment that consists of modules and sub-packages and sub-sub-packages, and so on.Consider a file *Pots.py* available in *Phone* directory. This file has following line of source code −

```
#!/usr/bin/python
def Pots():
  print "I'm Pots Phone"
```

Similar way, we have another two files having different functions with the same name as above −

- *Phone/Isdn.py* file having function Isdn()
- *Phone/G3.py* file having function G3()

Now, create one more file __init__.py in *Phone* directory −

- Phone/__init__.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in __init__.py as follows −

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

After you add these lines to __init__.py, you have all of these classes available when you import the Phone package.

```
#!/usr/bin/python
# Now import your Phone Package.
import Phone
Phone.Pots()
Phone.Isdn()
Phone.G3()
```

When the above code is executed, it produces the following result −

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

In the above example, we have taken example of single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

Python File Handling

You have been reading and writing to the standard input and output. Now, we will see how to use actual data files. Python provides basic functionsand methods necessary to working on Files by default. A File is an external storage on hard disk from where data can be stored and retrieved. You can do most of the file manipulation using a **file** object

OPEN Function

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(file_name [, access_mode][, buffering])

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 an optional parameter and the default file access mode is read (r).
- **buffering** − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

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

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
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.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
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.
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.
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.
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
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.
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.
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.
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.

File Object Attributes

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

list of all the attributes related to a file object

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

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.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

fileObject.close();

Reading and Writing Files

The file object provides a set of access methods to make our lives easier. We would see how to use **read()** and **write()** methods to read and write files.

write () Method

The **write()** method writes any string to an open file. It is important to note that Python strings can have binary data and not just text. The write() method does not add a newline character ('\n') to the end of the string –

Syntax

fileObject.write(string);

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.

File and Directory Related Methods

There are three important sources, which provide a wide range of utility methods to handle and manipulate files & directories on Windows and Unix operating systems.

File Methods

A **file** object is created using *open* function and here is a list of functions which can be called on this object

Methods	Description
file.close()	Close the file. A closed file cannot be read or written any more.
file.flush()	Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects.
file.fileno()	Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.
file.isatty()	Returns True if the file is connected to a tty(-like) device, else False.
next(file)	Returns the next line from the file each time it is being called.
file.read([size])	Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes).
file.readline([size])	Reads one entire line from the file. A trailing newline character is kept in the string.
file.readlines([sizehint])	Reads until EOF using readline() and return 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.
file.seek(offset[, whence])	Sets the file's current position
file.tell()	Returns the file's current position
file.truncate([size])	Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.
file.write(str)	Writes a string to the file. There is no return value.
file.writelines(sequence)	Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings.

Function as Object

In Python, functions are **first-class objects**. That means that they can be treated like objects of any other type, e.g., int or list. They have types, e.g., the expression type(fact) has the value ; they can appear in expressions, e.g., as the right-hand side of an assignment statement or as an argument to a function; they can be elements of lists; etc.

Using functions as arguments can be particularly convenient in conjunction with lists. It allows a style of coding called higher-order programming.

Example	OUTPUT
def fact(n): """Assumes that n is an int > 0 Returns n!"""	L = [1, -4, 6.66] Apply abs to each element

```
    if n == 1:
        return n
    else:
        return n*fact(n - 1)

def apply(L, f):
    """Assumes L is a list, f a function Mutates L by replacing each element, e, of L by f(e)"""
    # print(type(f))
    for i in range(len(L)):
            L[i] = f(L[i])

L = [1, -4, 6.66]
print('L =', L)
print ('Apply abs to each element of L.')
apply(L, abs)
print('L =', L)
print ('Apply int to each element of', L)
apply(L, int)
print ('L =', L)
print ('Apply factorial to each element of', L)
apply(L, fact)
print ('L =', L)
```

```
of L.
L = [1, 4, 6.66]
Apply int to each element of [1, 4, 6.66]
L = [1, 4, 6]
Apply factorial to each element of [1, 4, 6]
L = [1, 24, 720]
```

The function apply() is called higher-order because it has an argument that is itself a function. The first time it is called, it mutates L by applying the unary built-in function abs to each element.

The second time it is called, it applies a type conversion to each element. The third time it is called, it replaces each element by the result of applying the function fact to each element. It prints

Python has a built-in higher-order function, map, that is similar to, but more general than, the apply() function. In its simplest form the first argument to map is a unary function (i.e., a function that has only one parameter) and the second argument is any ordered collection of values suitable as arguments to the first argument. It returns a list generated by applying the first argument to each element of the second argument. For example, the expression map(fact, [1, -4, 6.66]) has the value [1, 4, 6.66].

Generally, the first argument to map can be of function of n arguments, in which case it must be followed by n subsequent ordered collections. For example, the code

```
L1 = [1, 28, 36]
L2 = [2, 57, 9]
print map(min, L1, L2)
```

prints the list

```
[1, 28, 9]
```

Specifications

Figure 1 defines a function, findRoot that generalizes the bisection search we used to find square roots in **Figure 2**. It also contains a function, testFindRoot that can be used to test whether or not findRoot works as intended.

The function testFindRoot is almost as long as findRoot itself. To inexperienced programmers, writing test functions such as this often seems to be a waste of effort. Experienced programmers know, however, that an investment in writing testing code often pays big dividends. It certainly beats typing test cases into the shell over and over again during debugging (the process of finding out why a program does not work, and then fixing it). It also forces us to think about which tests are likely to be most illuminating.

The text between the triple quotation marks is called a docstring in Python. By convention, Python programmers use docstrings to provide specifications of functions. These docstrings can be accessed using the built-in function help.

If we enter the shell and type help(abs), the system will display Help on built-in function abs in module __builtin__:

abs(...)

abs(number) -> number

Return the absolute value of the argument. If the code in **Figure 1** (below) has been loaded into IDLE, typing help(findRoot) in the shell will display

Help on function findRoot in module __main__:

findRoot(x, power, epsilon)

Assumes x and epsilon int or float, power an int,

epsilon > 0 & power >= 1

Returns float y such that y**power is within epsilon of x.

If such a float does not exist, it returns None

If we type findRoot (in either the shell or the editor, the list of formal parameters and the first line of the docstring will be displayed.

Figure 1

```
def findRoot(x, power, epsilon):
"""Assumes x and epsilon int or float, power an int,
   epsilon > 0 & power >= 1
   Returns float y such that y**power is within epsilon of x.
   If such a float does not exist, it returns None"""
        if x < 0 and power%2 == 0:
        return None
        low = min(-1.0, x)
        high = max(1.0, x)
        ans = (high + low)/2.0
        while abs(ans**power - x) >= epsilon:
                if ans**power < x:
                        low = ans
                else:
                        high = ans
                ans = (high + low)/2.0
        return ans
def testFindRoot():
 epsilon = 0.0001
    for x in (0.25, -0.25, 2, -2, 8, -8):
        for power in range(1, 4):
            print( 'Testing x = ' + str(x) +\ ' and
```

Figure 2

```
def fib(x):
 """Assumes x an int >= 0 Returns Fibonacci of x"""
global numFibCalls
numFibCalls += 1
if x == 0 or x == 1:
   return 1
else:
    return fib(x-1) + fib(x-2)

def testFib(n):
    for i in range(n+1):
     global                numFibCalls
numFibCalls = 0
       print ('fib of', i, '=', fib(i))
       print ('fib called', numFibCalls,
'times.')
```

power = ' + str(power)) result = findRoot(x, power, epsilon) if result == None: print('No root' else: print' ', result**power, '~=', x)	

A specification of a function defines a contract between the implementer of a function and those who will be writing programs that use the function. We will refer to the users of a function as its clients. This contract can be thought of as containing two parts:

1. **Assumptions:** These describe conditions that must be met by clients of the function. Typically, they describe constraints on the actual parameters. Almost always, they specify the acceptable set of types for each parameter, and not infrequently some constraints on the value of one or more of the parameters. For example, the first two lines of the docstring of findRoot describe the assumptions that must be satisfied by clients of findRoot.
2. **Guarantees:** These describe conditions that must be met by the function, provided that it has been called in a way that satisfies the assumptions. The last two lines of the docstring of findRoot describe the guarantees that the implementation of the function must meet.

Functions are a way of creating computational elements that we can think of as primitives. Just as we have the built-in functions max and abs, we would like to have the equivalent of a built-in function for finding roots and for many other complex operations. Functions facilitate this by providing decomposition and abstraction.

Decomposition creates structure. It allows us to break a problem into modules that are reasonably self-contained, and that may be reused in different settings.

Abstraction hides detail. It allows us to use a piece of code as if it were a black box—that is, something whose interior details we cannot see, don't need to see, and shouldn't even want to see.21 The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context. The key to using abstraction effectively in programming is finding a notion of relevance that is appropriate for both the builder of an abstraction and the potential clients of the abstraction. That is the true art of programming.

Abstraction is all about forgetting. There are lots of ways to model this, for example, the auditory apparatus of most teenagers.

Teenager says: May I borrow the car tonight?

Parent says: Yes, but be back before midnight, and make sure that the gas tank is full.

Teenager hears: Yes.

The teenager has ignored all of those pesky details that he or she considers irrelevant. Abstraction is a many-to-one process. Had the parent said Yes, but be back before 2:00 a.m., and make sure that the car is clean, it would also have been abstracted to Yes.

By way of analogy, imagine that you were asked to produce an introductory computer science course containing twenty-five lectures. One way to do this would be to recruit twenty-five professors, and ask each of them to prepare a fifty-minute lecture on their favorite topic. Though you might get twenty-five wonderful hours, the whole thing is likely to feel like a dramatization of Pirandello's "Six Characters in Search of an Author" (or that political science course you took with fifteen guest lecturers). If each

professor worked in isolation, they would have no idea how to relate the material in their lecture to the material covered in other lectures.

Somehow, one needs to let everyone know what everyone else is doing, without generating so much work that nobody is willing to participate. This is where abstraction comes in. You could write twenty-five specifications, each saying what material the students should learn in each lecture, but not giving any detail about how that material should be taught. What you got might not be pedagogically wonderful, but at least it might make sense.

This is the way organizations go about using teams of programmers to get things done. Given a specification of a module, a programmer can work on implementing that module without worrying unduly about what the other programmers on the team are doing. Moreover, the other programmers can use the specification to start writing code that uses that module without worrying unduly about how that module is to be implemented.

The specification of findRoot is an abstraction of all the possible implementations that meet the specification. Clients of findRoot can assume that the implementation meets the specification, but they should assume nothing more. For example, clients can assume that the call findRoot(4.0, 2, 0.01) returns some value whose square is between 3.99 and 4.01. The value returned could be positive or negative, and even though 4.0, is a perfect square the value returned might not be 2.0 or -2.0

ARRAY

An array is a collection of elements of the same type. Arrays are popular in most programming languages like: Java, C/C++, and JavaScript.

In Python, this is the main difference between arrays and lists. While python lists can contain values corresponding to different data types, arrays in python can only contain values corresponding to same data type.

Create arrays

As you might have guessed from the above example, we need to import array module to create arrays. For example:

Syntax	Example
array(data type, value list)	**import array as arr** **a = arr.array('d', [1.1, 3.5, 4.5])** **print(a)**

Here, we created an array of float type. The letter 'd' is a data type code. This determines the type of the array during creation. Some of the data types are mentioned in the table below.

Type Code	C Type	Python Type	Minimum size in Bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8

'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Access Array Elements

We use index to access elements of an array: Remember, index starts from 0 (not 1) similar like lists

Example	**Output**
import array as arr **a = arr.array('i', [2, 5, 62, 5, 42, 52, 48, 5])** **print("First element:", a[0])** **print("Second element:", a[1])** **print("Second last element:", a[-1])**	**First element: 2** **Second element: 5** **Second last element: 15**

We can access a range of items in an array by using the slicing operator **(:)**

Example	**Output**
import array as arr **numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]** **array = arr.array('i', numbers_list)** **print(array [2:5])** **print(array[:-5])** **print(array[5:])** **print(array[:])**	**array('i', [62, 5, 42])** **array('i', [2, 5, 62])** **array('i', [52, 48, 5])** **array('i', [2, 5, 62, 5, 42, 52, 48, 5])**

append()

This function is used to **add the value** mentioned in its arguments at the **end** of the array.

Syntax Array_variable.append(Value);	**Output**
Example **import array as arr** **a = arr.array('i', [2, 5, 62, 5, 42, 52, 48, 5])** **a.append(100)**	**array** **('i', [2, 5, 62, 5, 42, 52, 48, 15, 100])**

insert (index, value)

This function is used to **add the value at the position (index)** specified in its argument.

Example	**Output**
import array as arr **a = arr.array('i', [2, 5, 62, 5, 42, 52, 48, 5])** **a.insert(3,100)**	

Unit -2: OOP Using Python

What is an Exception?

An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids your program to crash

Why use Exceptions?

Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error then you can use exception handling.

Raising an Exception

You can raise an exception in your own program by using the raise exception statement. Raising an exception breaks current code execution and returns the exception back until it is handled.

Exceptions should be class objects. The exceptions are defined in the module **exceptions**. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the **exceptions** module.

For class exceptions, in a **try** statement with an **except** clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via sub-classing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an "**associated value**" indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the **raise** statement. If the exception class is derived from the standard root class **Base-Exception**, the associated value is present as the exception instance's **args** attribute.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition "just like" the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

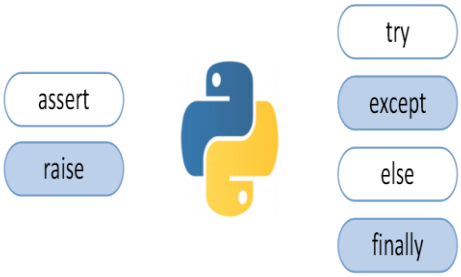
The built-in exception classes can be sub-classed to define new exceptions; programmers are encouraged to derive new exceptions from the **Exception**class or one of its sub-classes, and not from **Base-Exception**.

The following exceptions are only used as base classes for other exceptions.

No	Exception Name	Description
1	**Exception**	Base class for all exceptions
2	**StopIteration**	Raised when the next() method of an iterator does not point to any object.
3	**SystemExit**	Raised by the sys.exit() function.
4	**StandardError**	Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError**	Base class for all errors that occur for numeric calculation.
6	**OverflowError**	Raised when a calculation exceeds maximum limit for a numeric type.

7	**FloatingPointError**	Raised when a floating point calculation fails.
8	**ZeroDivisionError**	Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError**	Raised in case of failure of the Assert statement.
10	**AttributeError**	Raised in case of failure of attribute reference or assignment.
11	**EOFError**	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError**	Raised when an import statement fails.
13	**KeyboardInterrupt**	Raised when the user interrupts program execution, usually by pressing Ctrl+C.
14	**LookupError**	Base class for all lookup errors.
15	**IndexError**	Raised when an index is not found in a sequence.
16	**KeyError**	Raised when the specified key is not found in the dictionary.
17	**NameError**	Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError**	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError**	Base class for all exceptions that occur outside the Python environment.
20	**IOError**	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
21	**IOError**	Raised for operating system-related errors.
22	**SyntaxError**	Raised when there is an error in Python syntax.
23	**IndentationError**	Raised when indentation is not specified properly.
24	**SystemError**	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	**SystemExit**	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	**TypeError**	Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError**	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError**	Raised when a generated error does not fall into any category.
29	**NotImplementedError**	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Handling Exception



A Python program terminates as soon as itencountersan error. An error can be a syntax error or an exception. You will see what an exception is and how it differs from a syntax error. After that, you will learn about raising exceptions and making assertions. Then, you'll finish with a demonstration of the try and except block.

Exceptions versus Syntax Errors

Syntax errors occur when the parser detects an incorrect statement. Observe the following example:

```
>>> print( 0 / 0 ))
File "<stdin>", line 1
print(0/0))
^
```

SyntaxError: invalid syntax

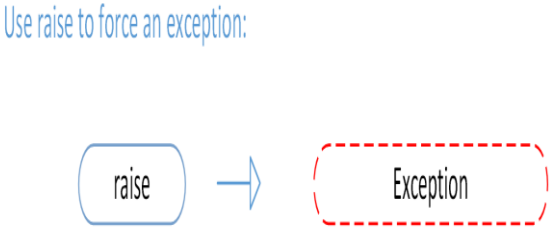
The arrow indicates where the parser ran into the **syntax error**. In this example, there was one bracket too many. Remove it and run your code again:

```
>>> print( 0 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

This time, you ran into an **exception error**. This type of error occurs whenever syntactically correct Python code results in an error. The last line of the message indicated what type of exception error you ran into.

Instead of showing the message exception error, Python details what type of exception error was encountered. In this case, it was a **ZeroDivisionError.** Python comes with **various built-in exceptions** as well as the possibility to create self-defined exceptions.

Raising an Exception



We can use raise to throw an exception if acondition occurs. The statements can be complemented with a custom exception.

If you want to throw an error when a certain condition occurs using raise, you could go about it like this:

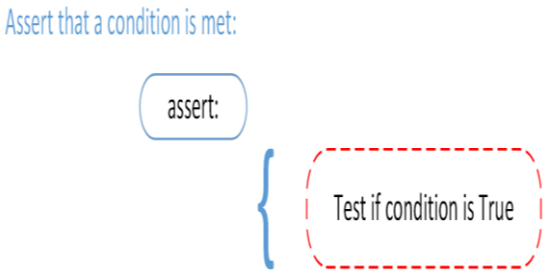
```
x=10
ifx>5:
raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
```

When you run this code, the output will be the following:

```
Traceback (most recent call last):
  File "<input>", line 4, in <module>
Exception: x should not exceed 5. The value of x was: 10
```

The program comes to a halt and displays our exception to screen; offering clues about what went wrong.

The AssertionError Exception



Instead of waiting for a program to crash midway, you can also start by making an assertion in Python. We assert that a certain condition is met. If this condition turns out to be True, then that is excellent! The program can continue. If the condition turns out to be False, you can have the program throw an AssertionError exception.

Have a look at the following example, where it is asserted that the code will be executed on a Linux system:.

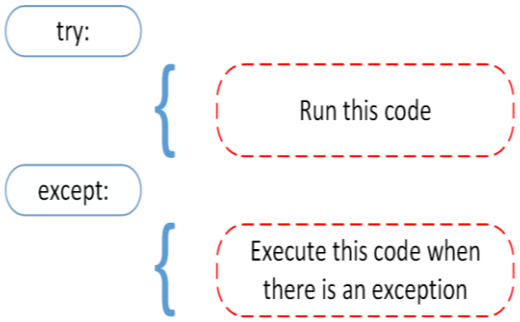
```
import sys
assert ('linux' in sys.platform), "This code runs on Linux only."
```

If you run this code on a Linux machine, the assertion passes. If you were to run this code on a Windows machine, the outcome of the assertion would be False and the result would be the following:

```
Traceback (most recent call last):
  File "<input>", line 2, in <module>
AssertionError: This code runs on Linux only.
```

In this example, throwing an AssertionError exception is the last thing that the program will do. The program will come to halt and will not continue. What if that is not what you want?

The try and except Block: Handling Exceptions



The try and except block in Python is used to catch and handle exceptions. Python executes code following the try statement as a **"normal"** part of the program. The code that follows the except statement is the program's response to any exceptions in the preceding try clause.

As you saw earlier, when syntactically correct code runs into an error, Python will throw an exception error.

This exception error will crash the program if it is unhandled. The exceptclause determines how your program responds to exceptions.

The following function can help you understand the try and except block:

```
deflinux_interaction():
assert('linux'insys.platform),"Function can only run on Linux
systems."
print('Doing something.')
```

The linux_interaction() can only run on a Linux system. The assert in this function will throw an AssertionError exception if you call it on an operating system other then Linux.

You can give the function a try using the following code:

```
try:
  linux_interaction()
except:
  pass
```

The way you handled the error here is by handing out a pass. If you were to run this code on a Windows machine, you would get the following output:

OUTPUT

(NONE OUTPUT like CLEAR SCREEN)

You got nothing. The good thing here is that the program did not crash. But it would be nice to see if some type of exception occurred whenever you ran your code. To this end, you can change the pass into something that would generate an informative message, like so:

```
try:
linux_interaction()
except:
print('Linux function was not executed')
```

Execute this code on a Windows machine:

OUTPUT

Linux function was not executed

When an exception occurs in a program running this function, the program will continue as well as inform you about the fact that the function call was not successful.

What you did not get to see was the type of error that was thrown as a result of the function call. In order to see exactly what went wrong, you would need to catch the error that the function threw.

The following code is an example where you capture the AssertionError and output that message to screen:

```
try:
  linux_interaction()
except AssertionError as error:
  print(error)
  print('The linux_interaction() function was not executed')
```

Running this function on a Windows machine outputs the following:

Output

```
Function can only run on Linux systems.
The linux_interaction() function was not executed
```

The first message is the AssertionError, informing you that the function can only be executed on a Linux machine. The second message tells you which function was not executed.

In the previous example, you called a function that you wrote yourself. When you executed the function, you caught the AssertionError exception and printed it to screen. Here's another example where you open a file and use a built-in exception:

```
try:
        withopen('file.log')asfile:
```

```
            read_data=file.read()
except:
  print('Could not open file.log')
```

If *file.log* does not exist, this block of code will output the following:

OUTPUT

```
Could not open file.log
```

This is an informative message, and our program will still continue to run. In the Python docs, you can see that there are a lot of built-in exceptions that you can use here. One exception described on that page is the following:

Exception FileNotFoundError

Raised when a file or directory is requested but doesn't exist. Corresponds to errno ENOENT. To catch this type of exception and print it to screen, you could use the following

```
try:
withopen('file.log')asfile:
read_data=file.read()
exceptFileNotFoundErrorasfnf_error:
print(fnf_error)
```

In this case, if *file.log* does not exist, the output will be the following:

OUTPUT

```
[Errno 2] No such file or directory: 'file.log'
```

You can have more than one function call in your try clause and anticipate catching various exceptions. A thing to note here is that the code in the try clause will stop as soon as an exception is encountered. Look at the following code. Here, you first call the linux_interaction() function and then try to open a file:

```
try:
linux_interaction()
withopen('file.log')asfile:
read_data=file.read()
exceptFileNotFoundErrorasfnf_error:
print(fnf_error)
exceptAssertionErroraserror:
print(error)
print('Linux linux_interaction() function was not executed')
```

If the file does not exist, running this code on a Windows machine will output the following:

OUTPUT

```
Function can only run on Linux systems.
Linux linux_interaction() function was not executed
```

Inside the try clause, you ran into an exception immediately and did not get to the part where you attempt to open *file.log*. Now look at what happens when you run the code on a Linux machine:

OUTPUT

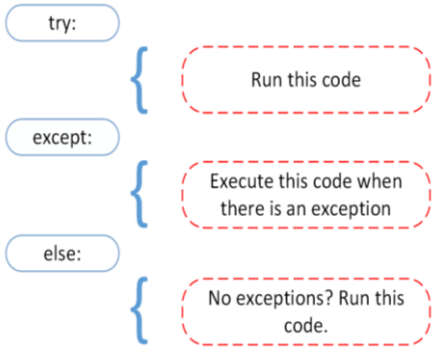
```
[Errno 2] No such file or directory: 'file.log'
```

Here are the key takeaways:

- A try clause is executed up until the point where the first exception is encountered.
- Inside the except clause, or the exception handler, you determine how the program responds to the exception.
- You can anticipate multiple exceptions and differentiate how the program should respond to them.
- Avoid using bare except clauses.

The else Clause

Using the else statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.



Look at the following example:

```
try:
linux_interaction()
exceptAssertionErroraserror:
print(error)
else:
print('Executing the else clause.')
```

If you were to run this code on a Linux system, the output would be the following:

OUTPUT

```
Doing something.
Executing the else clause.
```

Because the program did not run into any exceptions, the else clause was executed. You can also try to run code inside the else clause and catch possible exceptions there as well:

```
try:
        linux_interaction()
        exceptAssertionErroraserror:
        print(error)
else:
  try:
  withopen('file.log')asfile:
            read_data=file.read()
```

```
exceptFileNotFoundErrorasfnf_error:
print(fnf_error)
```

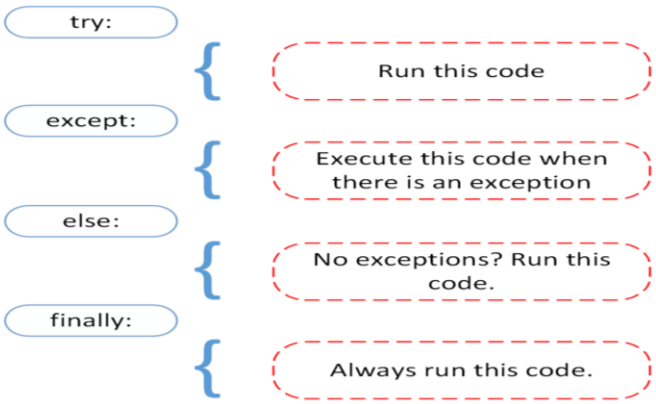
If you were to execute this code on a Linux machine, you would get the following result:

OUTPUT

```
Doing something.
[Errno 2] No such file or directory: 'file.log'
```

From the output, you can see that the linux_interaction() function ran. Because no exceptions were encountered, an attempt to open *file.log* was made. That file did not exist, and instead of opening the file, you caught the FileNotFoundError exception.

Cleaning Up After Using finally



Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using thefinallyclause.

Have a look at the following example:

```
try:
linux_interaction()
exceptAssertionErroraserror:
print(error)
else:
try:
withopen('file.log')asfile:
read_data=file.read()
exceptFileNotFoundErrorasfnf_error:
print(fnf_error)
finally:
print('Cleaning up, irrespective of any exceptions.')
```

In the previous code, everything in the finally clause will be executed. It does not matter if you encounter an exception somewhere in the try or else clauses. Running the previous code on a Windows machine would output the following:

OUTPUT

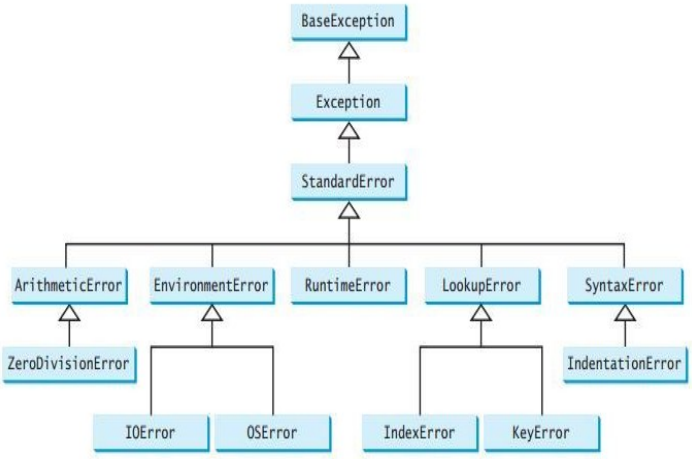
```
Function can only run on Linux systems.
Cleaning up, irrespective of any exceptions.
```

USER DEFINED EXCEPTION

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.

However, sometimes you may need to create a custom exception that serves your purpose.

In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from Exception class. Most of the built-in exceptions are also derived from this class.



As you can see from most of the exception classes in python extends from the BaseException class. You can derive you own exception class from BaseException class or from sublcass of BaseException like RuntimeError .

Create a new file called **NegativeAgeException.py** and write the following code.

```
class NegativeAgeException(RuntimeError):
    def __init__(self, age):
        super().__init__()
        self.age = age
```

Above code creates a new exception class named NegativeAgeException , which consists of only constructor which call parent class constructor using super().__init__() and sets the age .

```
def enterage(age):
    if age < 0:
        raise NegativeAgeException("Only positive integers are allowed")
    if age % 2 == 0:
        print("age is even")
    else:
        print("age is odd")
try:
    num = int(input("Enter your age: "))
    enterage(num)
except NegativeAgeException:
    print("Only positive integers are allowed")
except:
    print("something is wrong")
```

Assert Statement

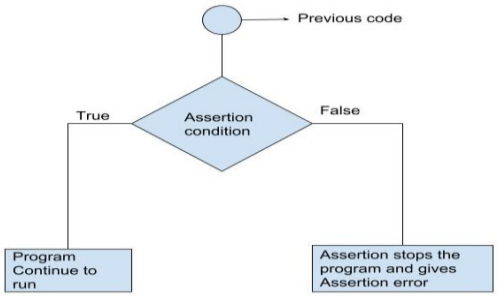
Assertions are statements that assert or state a fact confidently in your program. For example, while writing a division function, you're confident the divisor shouldn't be zero; you assert divisor is not equal to zero.

Assertions are simply Boolean expressions that checks if the conditions return true or not. If it is true, the program does nothing and moves to the next line of code. However, if it's false, the program stops and throws an error.

It is also a debugging tool as it brings the program on halt as soon as any error is occurred and shows on which point of the program error has occurred.

We can be clear by looking at the flowchart below:

Python has built-inassertstatementto use assertion condition in theprogram. Assert statement has a conditionor expression which is supposedto be always true. If the conditionisfalse assert halts the program and gives an AssertionError.



Syntax

assert <condition>

assert <condition>,<error message>

We can use assert statement in two ways as mentioned above. Assert statement has a condition and if the condition is not satisfied the program will stop and give **AssertionError**.

Assert statement can also have a condition and a optional error message.Ifthecondition is not satisfied assert stops the program and gives **AssertionError** along with the error message.

Let's take an example, where we have a function which will calculate the averageof the values passed by the user and the value should not be an emptylist. We will use **assertstatement** to check the parameter and if the length is ofthepassed list is zero, program halts.

Using assert with error message

```
def avg(marks):
  assert len(marks) != 0,"List is empty."
  return sum(marks)/len(marks)

mark2 = [55,88,78,90,79]
print("Average of mark2:",avg(mark2))
mark1 = []
print("Average of mark1:",avg(mark1))
```

When we run the above program, the output will be:

```
Average of mark2: 78.0
AssertionError: List is empty.
```

We passed a non-empty list mark2 and also an empty list mark1 to the avg() function and we got output for mark2 list but after that we got an error

AssertionError: List is empty. The assert condition was satisfied by the mark2 list and program to continue to run. However, mark1 doesn't satisfy the condition and gives an **AssertionError.**

Key Points

Assertions are the condition or Boolean expression which is always supposed to be true in the code.

Assert statement takes an expression and optional message. Assertstatementis used to check types, values of argument and the output of the function. Assert statement is used as debugging tool as it halts the program at the point where an error occurs.

Abstract data types

The data types you have seen so far are all concrete, in the sense that we have completely specified how they are implemented. For example, the Card class represents a card using two integers. As we discussed at the time, that is not the only way to represent a card; there are many alternative implementations.

An abstract data type, or ADT, specifies a set of operations (or methods) and the semantics of the operations (what they do), but it does not not specify the implementation of the operations. That's what makes it abstract.

It simplifies the task of specifying an algorithm if you can denote the operations you need without having to think at the same time about how the operations are performed.

Since there are usually many ways to implement an ADT, it might be useful to write an algorithm that can be used with any of the possible implementations.

Well-known ADTs, such as the Stack ADT in this chapter, are often implemented in standard libraries so they can be written once and used by many programmers.

The operations on ADTs provide a common high-level language for specifying and talking about algorithms.

When we talk about ADTs, we often distinguish the code that uses the ADT, called the client code, from the code that implements the ADT, called the provider code.

The Stack ADT

we will look at one common ADT, the stack. A stack is a collection, meaning that it is a data structure that contains multiple elements. Other collections we have seen include dictionaries and lists.

An ADT is defined by the operations that can be performed on it, which is called an interface. The interface for a stack consists of these operations:

Name	Description
__init__	Initialize a new empty stack.
push	Add a new item to the stack.
pop	Remove and return an item from the stack. The item that is returned is always the last one that was added.
is_empty	Check whether the stack is empty.

A stack is sometimes called a last in, first out or LIFO data structure, because the last item added is the first to be removed.

Implementing stacks with Python lists

The list operations that Python provides are similar to the operations that define a stack. The interface isn't exactly what it is supposed to be, but we can write code to translate from the Stack ADT to the built-in operations.

This code is called an implementation of the Stack ADT. In general, an implementation is a set of methods that satisfy the syntactic and semantic requirements of an interface.

Here is an implementation of the Stack ADT that uses a Python list:

```
class Stack :
  def __init__(self):
    self.items = []

  def push(self, item):
    self.items.append(item)

  def pop(self):
    return self.items.pop()

  def is_empty(self):
    return (self.items == [])
```

A Stack object contains an attribute named items that is a list of items in the stack. The initialization method sets items to the empty list.

To push a new item onto the stack, push appends it onto items. To pop an item off the stack, pop uses the homonymous (same-named) list method to remove and return the last item on the list.

Finally, to check if the stack is empty, is_empty compares items to the empty list. An implementation like this, in which the methods consist of simple invocations of existing methods, is called a veneer. In real life, veneer is a thin coating of good quality wood used in furniture-making to hide lower quality wood underneath. Computer scientists use this metaphor to describe a small piece of code that hides the details of an implementation and provides a simpler, or more standard, interface.

Pushing and popping

A stack is a generic data structure, which means that we can add any type of item to it. The following example pushes two integers and a string onto the stack:

```
>>> s = Stack()
>>> s.push(54)
>>> s.push(45)
>>> s.push("+")
```

We can use is_empty and pop to remove and print all of the items on the stack:

```
while not s.is_empty():
  print s.pop(),
```

The output is + 45 54. In other words, we just used a stack to print the items backward! Granted, it's not the standard format for printing a list, but by using a stack, it was remarkably easy to do.

You should compare this bit of code to the implementation of print_backward in the last chapter. There is a natural parallel between the recursive version of print_backward and the stack algorithm here. The difference is that print_backward uses the runtime stack to keep track of the nodes while it traverses the list, and then

prints them on the way back from the recursion. The stack algorithm does the same thing, except that is use a Stack object instead of the runtime stack.

Using a stack to evaluate postfix

In most programming languages, mathematical expressions are written with the operator between the two operands, as in 1 + 2. This format is called infix. An alternative used by some calculators is called postfix. In postfix, the operator follows the operands, as in 1 2 +.

The reason postfix is sometimes useful is that there is a natural way to evaluate a postfix expression using a stack:

Starting at the beginning of the expression, get one term (operator or operand) at a time. If the term is an operand, push it on the stack. If the term is an operator, pop two operands off the stack, perform the operation on them, and push the result back on the stack.

When you get to the end of the expression, there should be exactly one operand left on the stack. That operand is the result.

Parsing

To implement the previous algorithm, we need to be able to traverse a string and break it into operands and operators. This process is an example of parsing, and the results—the individual chunks of the string – are called tokens.

Python provides a split method in both the string and re (regular expression) modules. The function string.split splits a string into a list using a single character as a delimiter. For example:

```
>>> import string
>>> string.split("Now is the time"," ")
['Now', 'is', 'the', 'time']
```

In this case, the delimiter is the space character, so the string is split at each space. The function re.split is more powerful, allowing us to provide a regular expression instead of a delimiter. A regular expression is a way of specifying a set of strings. For example, [A-z] is the set of all letters and [0-9] is the set of all numbers. The ^ operator negates a set, so [^0-9] is the set of everything that is not a number, which is exactly the set we want to use to split up postfix expressions:

```
>>> import re
>>> re.split("([^0-9])", "123+456*/")
['123', '+', '456', '*', '', '/', '']
```

Notice that the order of the arguments is different from string.split; the delimiter comes before the string.

The resulting list includes the operands 123 and 456 and the operators * and /. It also includes two empty strings that are inserted after the operands.

Evaluating postfix

To evaluate a postfix expression, we will use the parser from the previous section and the algorithm from the section before that. To keep things simple, we'll start with an evaluator that only implements the operators + and *:

```
def eval_postfix(expr):
    import re
    token_list = re.split("([^0-9])", expr)
    stack = Stack()
    for token in token_list:
        if  token == '' or token == ' ':
```

```
            continue
        if  token == '+':
            sum = stack.pop() + stack.pop()
            stack.push(sum)
        elif token == '*':
            product = stack.pop() * stack.pop()
            stack.push(product)
        else:
            stack.push(int(token))
    return stack.pop()
```

The first condition takes care of spaces and empty strings. The next two conditions handle operators. We assume, for now, that anything else must be an operand. Of course, it would be better to check for erroneous input and report an error message, but

Let's test it by evaluating the postfix form of (56+47)*2:

```
>>> print eval_postfix ("56 47 + 2 \*")
206
```

That's close enough.

CLASS

Introduction

Python is an "object-oriented programming language." This means that almost all the code is implemented using a special construct called classes. Programmers use classes to keep related things together. This is done using the keyword "class," which is a grouping of object-oriented constructs.

- Almost everything in an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

OOP Terminology

- **Class –**A blueprint created by a programmer for an object. This defines a set of attributes that will characterize any object that is instantiated from this class.
- **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 is 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.

What is a class?

A class is a code template for creating objects. Objects have member variables and have behavior associated with them. In python a class is created by the keyword class.

An object is created using the constructor of the class. This object will then be called the instance of the class. In Python we create instances in the following manner

Instance = class_Name()

Creating Class

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

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

Example

```
class Student:
  'Common base class for all employees'
  count=0
  def __init__(self, name, age):
    self.name = name
    self.age = age
    Student.count+=1

  def displayCount(self):
   print ("Total Student %d" % Student.count)

  def displayStudent(self):
    print ("Name : ", self.name,  ", age: ", self.age)

st1 = Student("ABC",20)
st1.displayStudent()
st1.displayCount()
st2 = Student("DEF",30 )
st2.displayStudent()
print ("Total Student %d" %Student.count)
```

The variable Student.count is a class variable whose value is shared among all instances of this class. This can be accessed as Student.count from inside the class or outside the class.

The first method __init__() is a special method, which is called **class constructor or initialization method.**When you create a new instance of this class.

You declare other class methods like normal functions with the exception that the first argument to each method is self. Python adds the self argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

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

"This would create first object of Studnet class"

st1 = Student("ABC",20)

"This would create second object of Student class"

st2 = Student("DEF",30)

Accessing Attributes

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

```
st1.displayStudent()
st2.displayStudent()
print ("Total Student %d" %Student.count)
```

You can **add, remove, or modify** attributes of classes and objects at any time,

```
Student.semester = 5  # Add an ' semester ' attribute.
Student.semester = 6 # Modify ' semester ' attribute.
del Student.semester # Delete ' semester ' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions

Attribute	Meaning
getattr(obj, name[, default])	Access the attribute of object.
hasattr(obj,name)	Check if an attribute exists or not.
setattr(obj,name,value)	Set an attribute. If attribute does not exist, then it would be created.
delattr(obj, name)	Delete an attribute

Example

hasattr(Student, 'semester')	Returns true if 'semester' attribute exists
getattr(Student, 'semester')	Returns value of 'semester' attribute
setattr(Student, 'semester', 6)	Set attribute 'semester' at 6
delattr Student, 'semester')	Delete attribute 'semester'

Attributes of Class

Attributes in Python defines a property of an object, element or a file. There are two types of attributes:

Built-in Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute

Attribute	Meaning
__dict__	Dictionary containing the class's namespace.
__doc__	Class documentation string or none, if undefined.
__name__	Class name.
__module__	Module name in which the class is defined. This attribute is "__main__" in interactive mode.
__bases__	A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list

Example.

Example	Output

class Student:
'Common base class for all Student'
Count= 0
def __init__(self, name,age):
self.name = name
self.age = age
Student.Count += 1
def displayCount(self):
print ("Total Student %d" %Student.Count)
def displayStudent(self):
print("Name : ", self.name, ", Age: ", self.age)
print("Student.__doc__:",Student.__doc__)
print ("Student.__name__:", Student.__name__)
print("Student.__module__:",Student.__module__)
print ("Student.__bases__:", Student.__bases__)
print ("Student.__dict__:", Student.__dict__)
	Student.__doc__: Common base class for all Student
	Student.__name__: Student
	Student.__module__: __main__
	Student.__bases__: (<class 'object'>,)
	Student.__dict__: {'__module__': '__main__', '__doc__': 'Common base class for all Student', 'Count': 0, '__init__': <function Student.__init__ at 0x00799660>, 'displayCount': <function Student.displayCount at 0x0221C9C0>, 'displayStudent': <function Student.displayStudent at 0x02261150>, '__dict__': <attribute '__dict__' of 'Student' objects>, '__weakref__': <attribute '__weakref__' of 'Student' objects>}

Destroying Objects

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

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

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

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method **__del__()**, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.

Example	Output
classPoint: def __init__(self, x=0, y=0):	3083401324 3083401324 3083401324 Point destroyed

```
self.x = x
self.y = y
def __del__(self):
    class_name =self.__class__.__name__
print class_name,"destroyed"

pt1 =Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3)# prints the ids of
the obejcts
del pt1
del pt2
del pt3
```

Users defined Attributes:

Attributes are created inside the class definition. We can dynamically create new attributes for existing instances of a class. Attributes can be bound to class names as well.

Next, we have public, protected and private attributes.

Naming	Type	Meaning
Name	**Public**	These attributes can be freely used inside or outside of a class definition
_name	**Protected**	Protected attributes should not be used outside of the class definition, unless inside of a subclass definition
__name	**Private**	This kind of attribute is inaccessible and invisible. It's neither possible to read nor to write those attributes, except inside of the class definition itself

Next, let's understand the most important component in a python class objects.

Inheritance

Inheritance enables us to define a class that takes all the functionality from parent class and allows us to add more. In this article, you will learn to use inheritance in Python.

Inheritance refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

Syntax

class BaseClass:
Body of base class
class DerivedClass(BaseClass):
Body of derived class

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

Inheritance in Python

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon:
  def __init__(self, no_of_sides):
    self.n = no_of_sides
    self.sides = [0 for i in range(no_of_sides)]

  def inputSides(self):
    self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

  def dispSides(self):
    for i in range(self.n):
      print("Side",i+1,"is",self.sides[i])
```

This class has data attributes to store the number of sides, n and magnitude of each side as a list, sides.

Method inputSides() takes in magnitude of each side and similarly,dispSides()will display these properly.

A triangle is a polygon with 3 sides. So, we can created a class called Triangle which inherits from Polygon. This makes all the attributes available in class Polygon readily available in Triangle. We don't need to define them again (code re-usability).

Triangle is defined as follows.

```
class Triangle(Polygon):
  def __init__(self):
    Polygon.__init__ (self, 3)
def findArea(self):
    a, b, c = self.sides
    # calculate the semi-perimeter
    s = (a + b + c) / 2
    area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
    print('The area of the triangle is %0.2f' %area)
```

However, class Triangle has a new method findArea() to find and print the area of the triangle. Here is a sample run.

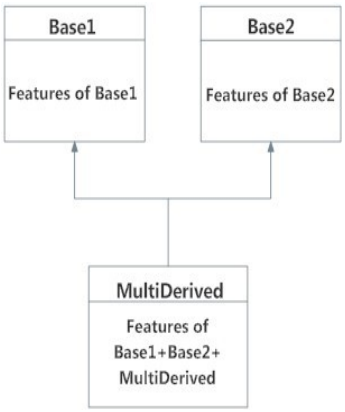
```
>>> t = Triangle()
>>> t.inputSides()

Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4

>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0

>>> t.findArea()
```

The area of the triangle is 6.00

We can see that, even though we did not define methods like inputSides() or dispSides() for class Triangle, we were able to use them.

If an attribute is not found in the class, search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Multiple Inheritance



Like C++, a class can be derived from more than one base classes in Python. This is called multiple inheritance. In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

Example

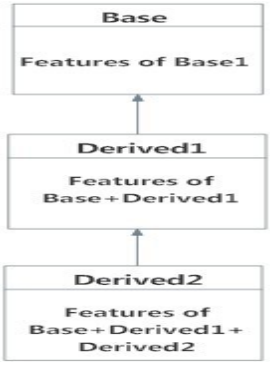
```
lass Base1:
   pass

class Base2:
   pass

class MultiDerived(Base1, Base2):
   pass
```

Here, MultiDerived is derived from classes Base1 and Base2.

Multilevel Inheritance

On the other hand, we can also inherit form a derived class. This is called multilevel inheritance. It can be of any depth in Python. In multilevel inheritance, a feature of the base class and the derived class is inherited into the new derived class.

An example with corresponding visualization is given below.




```
class Base:
        pass
class Derived1(Base):
        pass
class Derived2(Derived1):
        pass
```

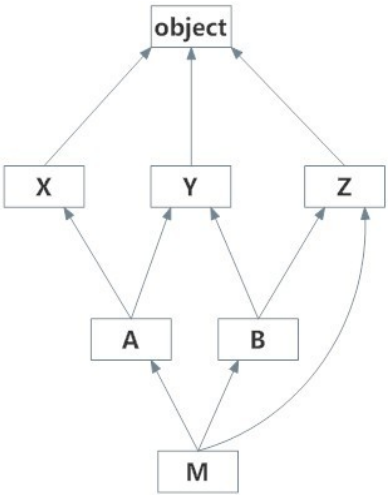
Here, Derived1 is derived from Base, and Derived2 is derived from Derived1

Method Resolution Order

Every class in Python is derived from the class object. It is the most base type in Python. So technically, all other class, either built-in or user-defines, are derived classes and all objects are instances of object class.

```
# Output: True
print(issubclass(list,object))

# Output: True
print(isinstance(5.5,object))
# Output: True
print(isinstance("Hello",object))
```

In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching same class twice.



So, in the above example of MultiDerived class the search order is [MultiDerived,Base1, Base2, object]. This order is also called linearization of MultiDerivedclass and the set of rules used to find this order is called Method Resolution Order (MRO).

MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents and in case of multiple parents, the order is same as tuple of base classes.

MRO of a class can be viewed as the __mro__ attribute or mro() method. Theformer returns a tuple while latter returns a list.

```
>>> MultiDerived.__mro__
(<class '__main__.MultiDerived'>,
<class '__main__.Base1'>,
<class '__main__.Base2'>,
<class 'object'>)

>>> MultiDerived.mro()
[<class '__main__.MultiDerived'>,
<class '__main__.Base1'>,
<class '__main__.Base2'>,
<class 'object'>]
```

Here is a little more complex multiple inheritance example and its visualization along with the MRO.

```
class X: pass
class Y: pass
class Z: pass

class A(X,Y): pass
class B(Y,Z): pass

class M(B,A,Z): pass
```

```
# Output:
# [<class '__main__.M'>, <class '__main__.B'>,
# <class '__main__.A'>, <class '__main__.X'>,
# <class '__main__.Y'>, <class '__main__.Z'>,
# <class 'object'>]
print(M.mro())
```

Refer to this, for further discussion on MRO and to know the actual algorithm how it is calculated.

Method Overriding in Python

In the above example, notice that __init__() method was defined in both classes, Triangleas well Polygon. When this happens, the method in the derived class overrides that in the base class. This is to say, __init__() in Triangle gets preference over the same in Polygon.

Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling Polygon.__init__() from __init__() in Triangle).

A better option would be to use the built-in function super(). So, super().__init__(3) is equivalent to Polygon.__init__(self,3) and is preferred.

Two built-in functions isinstance() and issubclass() are used to check inheritances. Function isinstance() returns True if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class object.

```
>>> isinstance(t,Triangle)
True
>>> isinstance(t,Polygon)
True
>>> isinstance(t,int)
False
>>> isinstance(t,object)
True
```

Similarly, issubclass() is used to check for class inheritance.

```
>>> issubclass(Polygon,Triangle)
False
>>> issubclass(Triangle,Polygon)
True
>>> issubclass(bool,int)
True
```

Operator Overloading

Python operators work for built-in classes. But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings.

This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

How to overload the operators in Python?

Consider that we have two objects which are a physical representation of a class (user-defined data type) and we have to add two objects with binary '+' operator it throws an error, because compiler don't know how to add two objects. So we define amethod for an operator and that process is called operator overloading. We canoverloadall existing operators but we can't create a new operator. To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.

Overloading binary + operator in Python :

When we use an operator on user defined data types then automatically a special function or magic function associated with that operator is invoked. 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. There by changing this magic method's code, we can give extra meaning to the + operator.

# to overload an binary + operator	**# + operator overloading.**

class A:
  def __init__(self, a):
  self.a = a
  # adding two objects
  def __add__(self, o):
  return self.a + o.a
 	
ob1 = A(1)
ob2 = A(2)
ob3 = A("JJKCC")
ob4 = A("RAJKOT")
print(ob1 + ob2)
print(ob3 + ob4)
	class complex:
	  def __init__(self, a, b):
	  self.a = a
	  self.b = b
	  # adding two objects
	  def __add__(self, other):
	  return self.a + other.a, self.b + other.b
	  def __str__(self):
	  return self.a, self.b
	Ob1 = complex(1, 2)
	Ob2 = complex(2, 3)
	Ob3 = Ob1 + Ob2
	print(Ob3)

# comparison operators	**# less than operators**

class A:
  def __init__(self, a):
  self.a = a
  def__eq__(self, other):
  if(self.a ==other.a):
  return"Both are equal"
  else:
  return"Not equal"
ob1 =A(4)
ob2 =A(4)
print(ob1 ==ob2)
	classA:
	  def__init__(self, a):
	  self.a =a
	  def__lt__(self, other):
	  if(self.a<other.a):
	  return"ob1 is lessthan ob2"
	  else:
	  return"ob2 is less than ob1"
	ob1 =A(2)
	ob2 =A(3)
	print(ob1 < ob2)

Binary Operators

OPERATOR	MAGIC METHOD
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)

Comparison Operators

OPERATOR	MAGIC METHOD
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

Assignment Operators

OPERATOR	MAGIC METHOD
-=	__isub__(self, other)
+=	__iadd__(self, other)
*=	__imul__(self, other)
/=	__idiv__(self, other)
//=	__ifloordiv__(self, other)
%=	__imod__(self, other)
**=	__ipow__(self, other)

Unary Operators

OPERATOR	MAGIC METHOD
–	__neg__(self, other)
+	__pos__(self, other)
~	__invert__(self, other)

Encapsulation

Encapsulation is the packing of data and functions operating on that data into a single component and restricting the access to some of the object's components. Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables etc.

Python follows the philosophy of we're all adults here with respect to hiding attributes and methods; i.e. you should trust the other programmers who will use your classes. Use plain attributes whenever possible.

You might be tempted to use getter and setter methods instead of attributes, but the only reason to use getters and setters is so you can change the implementation later if you need to. However, Python 2.2 and later allows you to do this with properties:

Protected members:

Protected member is (in C++ and Java) accessible only from within the class and it's subclasses. How to accomplish this in Python? The answer is -by convention. By prefixing the name of your member with a single underscore

Private members:

But there is a method in Python to define Private: Add "__" (double underscore) in front of the variable and function name can hide them when accessing them from out of class.

Python doesn't have real private methods, so one underline in the beginning of a method or attribute means you shouldn't access this method. But this is just convention.

I can still access the variables with single underscore. Also when using double underscore (__). we can still access the private variables

An example of accessing private member data.(Using name mangling)

```
class Person:
    def __init__(self):
        self.name = 'Manjula'
        self.__lastname = 'Dube'
    def PrintName(self):
        return self.name +' ' + self.__lastname
#Outside class
    P = Person()
    print(P.name)
    print(P.PrintName())
    print(P.__lastname)

#AttributeError: 'Person' object has no attribute '__lastname'
```

The __init__ method is a constructor and runs as soon as an object of a class is instantiated. Its aim is to initialize the object

Access public variable out of class, succeed

Access private variable our of class, fail

Access public function but this function access Private variable __B successfully since they are in the same class.

An example of accessing private member data.(Using name mangling technique)

```
class SeeMee:
    def youcanseeme(self):
        return 'you can see me'
    def __youcannotseeme(self):
        return 'you cannot see me'
#Outside class
Check = SeeMee()
    print(Check.youcanseeme())
# you can see me
    print(Check.__youcannotseeme())
#AttributeError: 'SeeMee' object has no attribute '__youcannotseeme'
```

If you need to access the private member function

```
class SeeMee:
    def youcanseeme(self):
        return 'you can see me'
    def __youcannotseeme(self):
        return 'you cannot see me'
 #Outside class
    Check = SeeMee()
    print(Check.youcanseeme())
    print(Check._SeeMee__youcannotseeme())
#Changing the name causes it to access the function
```

You can still call the method using its mangled name, so this feature doesn't provide much protection.

You should know the following for accessing private members and private functions:

When you write to an attribute of an object, which does not exist, the python system will normally not complain, but just create a new attribute.

Private attributes are not protected by the Python system. That is by design decision. Private attributes will be masked. The reason is, that there should be no clashes in the inheritance chain. The masking is done by some implicit renaming. Private attributes will have the real name

"__<className>_<attributeName>"

With that name, it can be accessed from outside. When accessed from inside the class, the name will be automatically changed correctly.

Hash Table

Hash tables are a type of data structure in which the address or the index value of the data element is generated from a hash function. That makes accessing the data faster as the index value behaves as a key for the data value. In other words Hash table stores key-value pairs but the key is generated through a hashing function.

So the search and insertion function of a data element becomes much faster as the key values themselves become the index of the array which stores the data.

In Python, the Dictionary data types represent the implementation of hash tables. The Keys in the dictionary satisfy the following requirements.

- The keys of the dictionary are hash-table i.e. there are generated by hashing function which generates unique result for each unique value supplied to the hash function.
- The order of data elements in a dictionary is not fixed.

So we see the implementation of hash table by using the dictionary data types as below.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

Example	Output
# Declare a dictionary **dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}** **# Accessing the dictionary with its key** **print ("dict['Name']: "), dict['Name']** **print ("dict['Age']: "), dict['Age']**	**dict['Name']: Zara** **dict['Age']: 7**

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example

Example	Output
# Declare a dictionary **dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}** **dict['Age'] = 8; # update existing entry** **dict['School'] = "DPS School"; # Add new entry** **print("dict['Age']: "), dict['Age']** **print ("dict['School']: "), dict['School']**	**dict['Age']: 8** **dict['School']: DPS School**

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation. To explicitly remove an entire dictionary, just use the del statement. –

Example	Output
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'} del dict['Name']; # remove entry with key 'Name' dict.clear(); # remove all entries in dict del dict ; # delete entire dictionary print "dict['Age']: ", dict['Age'] print "dict['School']: ", dict['School']	dict['Age']: Traceback (most recent call last): File "test.py", line 8, in print "dict['Age']: ", dict['Age']; TypeError: 'type' object is unsubscriptable

This produces the following result. Note that an exception is raisedbecauseafter del dict dictionary does not exist anymore.

Searching Algorithms

Searching is also a common and well-studied task. Given a list of values, a function that compares two values and a desired value, find the position of the desired value in the list.

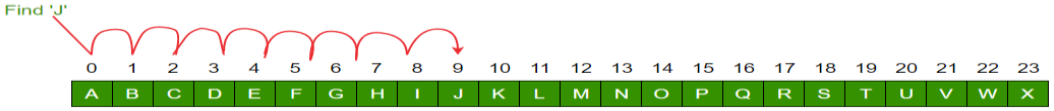
We will look at two algorithms that perform this task:

- **linear search**, which simply checks the values in sequence until the desired value is found
- **binary search**, which requires a sorted input list, and checks for the value in the middle of the list, repeatedly discarding the half of the list which contains values which are definitely either all larger or all smaller than the desired value

There are numerous other searching techniques. Often they rely on the construction of more complex data structures to facilitate repeated searching. Examples of such structures are hash tables (such as Python's dictionaries) and prefix trees. Inexact searches that find elements similar to the one being searched for are also an important topic.

Linear Search

In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the list.



I

in this to search an element from the unsorted list we are using this simplest technique. It simply traverses from top to bottom in the array and searches for the key value from the list and displayoutput as well. It is also called as sequential searching method. In this technique searching element is compared with the first element of the list, if match is found then the search is terminated. Otherwise next element from the list is fetched and compared with the searching element and this process is continued till

the searching element is found or list is completely traversed. At last, if searching element is not found from the list, it display a message that key element is not found in the list.

The time complexity of linear search algorithm is O(n). Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to linear search.

Algorithm	Example

1. Start
2. Initialize
p=0 and flag =1
3. Repeat through Step -4 k= 0,1,2 …… n-1
4. if (arr[p] == element)
flag =0
output " Successful searching element at " k+1 "location"
5. If flag ==1
output "Search is unseccessful"
6. Exit
	Sequential_Search(dlist, item):
	pos = 0
	found = False
	while pos < len(dlist) and not found:
	if dlist[pos] == item:
	found = True
	else:
	pos = pos + 1
	
	return found, pos

Binary search

Binary search is a more efficient search algorithm which relies on the elements in the list being sorted. We apply the same search process to progressively smaller sub-lists of the original list, starting with the whole list and approximately halving the search area every time.

We first check the ***middle*** element in the list.

- If it is the value we want, we can stop.
- If it is ***higher*** than the value we want, we repeat the search process with the portionof the list ***before*** the middle element.
- If it is ***lower*** than the value we want, we repeat the search process with the portionof the list ***after*** the middle element.

Algorithm	Example

1. Start
2. Initialize
first =0, last = n-1 , flag = False
3. Repeat through Step-5
While first <= lase and flag not found
Mid = (first + last)/2
4. If arr [mid] == element then
Found = true
else
if arr[mid] > element then
last = mid - 1
else
first = mid + 1
5. Return found
6. Exit
	def binary_search(item_list,item):
	first = 0
	last = len(item_list)-1
	found = False
	while(first<=last and not found):
	mid = (first + last)//2
	if item_list[mid] == item :
	found = True
	else:
	if item < item_list[mid]:
	last = mid - 1
	else:
	first = mid + 1
	return found

Sorting Algorithms

Sorting is the technique of arranging or ordering a collection of items such that each item and its successor satisfy a prescribed rule. The items can be simple values, such as integers and reals, or more complex types, such as student records or dictionary entries.

Ordering can be done on the basis of integer value, alphabetical order, date of creation, or any other comparison parameter.

We face the enormous requirement of sorting in everyday life. Consider the listings of a phone directory, the keys in a dictionary, or the terms in an index, all of which are organized in alphabetical order to make finding an entry much easier.

Bubble Sort

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you mustn't swap the element. Then, again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.

If there are *n* elements to be sorted then, the process mentioned above should be repeated *n-1* times to get required result. But, for better performance, in second step, last and second last elements are not compared because; the proper element is automatically placed at last after first step. Similarly, in third step, last and second last and second last and third last elements are not compared and so on.

Bubble sort algorithm is quite popular; there are many other better algorithm than bubble sort. Specially, bubble sort should not be used to sort large data if performance matters in that program.

Algorithm	**Example**

Input N numbers of an array A
Initialized i=0 and repeat through step 4 if (i<n)
Initialize j=0 and repeat through step 4
if (j<n-i-1)
If(A[i] >A[j])
(a) swap = A[j]
(b) A[j] =A[j+1]
(c) A[j+1] = swap
5. Display the sorted number of Array A
6. Exit
	def bobble_sort(item_list):
	for i in range(0,len(item_list)) :
	for j in range(0,len(item_list)-1) :
	if item_list[j] > item_list[j+1] :
	temp = item_list[j]
	item_list[j] = item_list[j+1]
	item_list[j+1] = temp

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- The subarray which is already sorted.
- Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted sub-array is picked and moved to the sorted sub-array.
The following example explains the above steps:

Algorithm	Example

1. Start
2. initialized PASS=0
3. repeat the Step - 9 while PASS < N -1
4. initialize MIN_INDEX = PASS
5. initialize i = PASS + 1
6. repeat the Step – 8 while I < N
7. if k[i] < k[MIN_INDEX] then
MIN_INDEX = 1
8. if MIN_INDEX != PASS
temp = k[PASS]
k[PASS] = k{[MIN_INDEX]
k[MIN_INDEX] = temp
9. i = i+1
10. PASS= PASS +1
11. Stop
	def selection_sort(item_list):
	for i in range(len(item_list) - 1, 0, -1):
	j = 0
	for k in range(1, i + 1):
	if item_list[k] > item_list[j]:
	j = k
	
	temp = item_list[i]
	item_list[i] = item_list[j]
	item_list[j] = temp

The selection sort, which makes n – 1 passes over the array to reposition n – 1values,is also O(n2). The difference between the selection and bubble sorts is that theselection sort reduces the number of swaps required to sort the list to O(n).

Insertion Sort

Insertion sort involves finding the right place for a given element in a sorted list. So in beginning we compare the first two elements and sort them by comparing them. Then we pick the third element and find its proper position among the previous two sorted elements. This way we gradually go on adding more elements to the already sorted list by putting them in their proper position.

Algorithm	Example

1. input an array A of N numbers
2. Initialize i=1 and repeat through Step 4 by incrementing i by 1
(1) if(i <= n-1)
(2) temp=A[i]
(3) Pos = i-1
3. Repeat the Step 3 if(tem < A[Pos]) and (Pos>=0)
(1) A[Pos+1] = A[Pos]
(2) Pos = Pos- 1
4. A[Pos+1] = Swap
5. Exit
	def insert_sort(item_list):
	for i in range(1,len(item_list)) :
	for j in range(i-1,-1,-1) :
	if item_list[j] > item_list[j+1] :
	temp=item_list[j]
	item_list[j]=item_list[j+1]
	item_list[j+1]=temp
	else:
	break

Quick Sort

- It is widely used sorting techniques which uses divide and conquer (also known as partition –exchange sort) mechanism.
- Quick sort is an effective developed by **C.A.R Hoarse** which passes a very good time complexity in average case.
- The Quick sort algorithm works by partitioning the array to be sorted and each partition are internally sorted recursively.
- In the quick sort mechanism, first all we have to select middle element form the list and is known as Pivot element.
- After that, the sort is divides the limit into sub lists.
- First list contains the element that are **Less than the pivot elements** and a second list contains the element that are **greater than or equal** to the **pivot element**
- Then recursively invokes itself with both lists in sorted order.
- Each time when the sort is invoked, it further divides the elements into smaller sub lists.

Algorithm	**Example**

Q_ sort(array, first, last)
1. Start
2. Low= 0, high=last,
pivot = array[(low+high)/2]
3. Repeat through step -8 while (low<= high)
4. Repeat step 5 while (array[low] <= pivot)
5. Low =low+1
6. Repeat step -7 while(array[high] > pivot
7. High = high -1
8. If (low <=high)
a. Temp= array[low]
b. Array[low] = array[high]
c. Array[high] = temp
d. Low=low+1
e. High-high-1
9. If (first < high)
Q_sort(array,first,high)
10. If (low < last)
Q_sort(array,low,last)
11. Exit
	def quick_sort(ar, first,last):
	low=first
	high=last
	pivot=ar[(low+high)//2]
	while(low<=high) :
	while(ar[low] < pivot) :
	low = low+1
	while(ar[high]> pivot) :
	high = high-1
	if(low <= high):
	temp=ar[low]
	ar[low]=ar[high]
	ar[high] = temp
	low=low+1
	high=high-1
	if (first < high) :
	quick_sort(ar,first,high)
	if (low< last) :
	quick_sort(ar,low,last)

Shell Sort

Shell-Sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shellSort is to allow exchange of far items. In shell-Sort, we make the array h-sorted for a large value of h. We keep reducing the value of h

until it becomes 1. An array is said to be h-sorted if all sub-lists of every h'th element is sorted.

Algorithm	**Example**

1. Start
2. Initialize gap=size/2
3. Repeat through Step- 7 while (gap=gap/2)
4. Swap=0
5. Repeat through Step-7 while(swap)
6. Repeat through step -7 for i=0 to size – gap
7. If(arr[i]>arr[i+gap]
temp=arr[i]
arr[i]=arr[i+gap]
arr[i+gap]=temp
swap=1
8. Exit
	def shell_sort(item_list):
	length = len(item_list)
	gap = n//2
	while gap > 0:
	for i in range(gap,length):
	temp = item_list[i]
	j=i
	while j >= gap and item_list[j-gap]> temp:
	item_list[j] = item_list[j-gap]
	j = j-gap
	item_list[j]=temp
	gap=gap//2

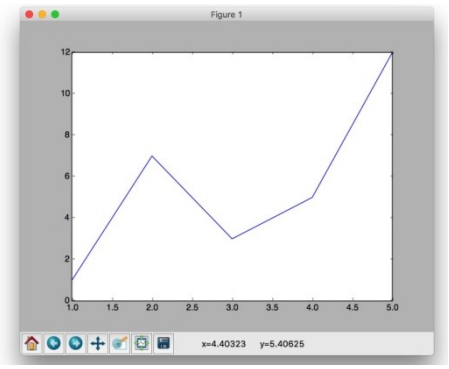
UNIT – 3 Plotting Using PyLab

PyLab is a Python standard library module that provides many of the facilities of MATLAB, "a high- level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation." Later in the book we will look at some of the more advanced features of PyLab, but in this chapter we focus on some of its facilities for plotting data.

There are also a number of Web sites that provide excellent tutorials. We will not try to provide a user's guide or a complete tutorial here. Instead, in this chapter we will merely provide a few example plots and explain the code that generated them. Other examples appear in later chapters.

simple example

```
import pylab
pylab.figure(1)
pylab.plot([1,2,3,4,5],[1,7,3,5,12])
pylab.show()
```



The bar at the top contains the name of the window, in this case "Figure."

The middle section of the window contains the plot generated by the invocation of pylab. plot. The two parameters of pylab.plot must be sequences of the same length. The first specifies the x-coordinates of the points to be plotted, and the second specifies the y-coordinates. Together, they provide a sequence of four <x ,y> coordinate pairs, [(1,1), (2,7), (3,3), (4,5)]. These are plotted in order. As each point is plotted, a line is drawn connecting it to the previous point.

The final line of code, pylab.show(), causes the window to appear on the computer screen. If that line were not present , the figure would still have been produced, but it would not have been displayed. This is not as silly as it at first sounds, since one might well choose to write a figure directly to a file , as we will do later, rather than display it on the screen.

The bar at the bottom of the window contains a number of push buttons. The rightmost button is used to write the plot to a file. The next button to the left is used to adjust the appearance of the plot in the window. The next four buttons are used for panning and zooming. And the button on the left is used to restore the figure to its original appearance after you are done playing with pan and zoom.

It is possible to produce multiple figures and to write them to files. These files can have any name you like, but they will all have the file extension .png. The file extension .png indicates that the file is in the Portable Networks Graphics format. This is a public domain standard for representing images.

In some operating systems, pylab.show() causes the process runningPython to be suspended until the figure is closed (by clicking on the round red button at the upper left- hand corner of the window). This is

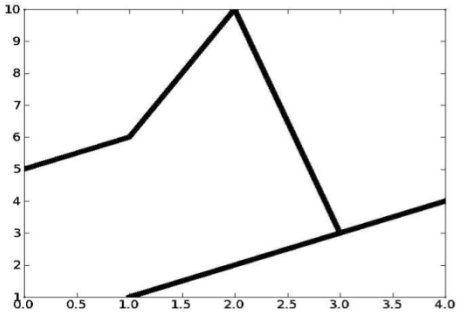
unfortunate. The usual workaround is to ensure that pylab.show() is the last line of code to be executed.

Example

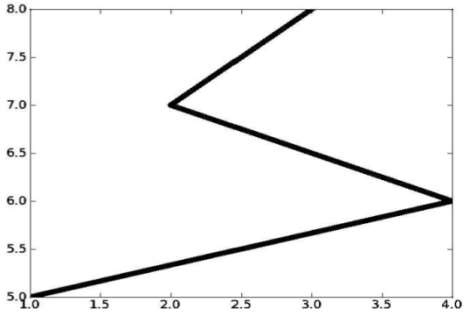
```
pylab.figure(1) #create figure 1
pylab.plot([1,2,3,4], [1,2,3,4]) #draw on figure 1
pylab.figure(2) #create figure 2
pylab.plot([1,4,2,3], [5,6,7,8]) #draw on figure 2
pylab.savefig('Figure-Addie') #save figure 2
pylab.figure(1) #go back to working on figure 1
pylab.plot([5,6,10,3]) #draw again on figure 1
pylab.savefig('Figure-Jane') #save figure 1
```

produces and saves to files named Figure-Jane.png and Figure-Addie.png the two plots below.

Observe that the last call to pylab.plot is passed only one argument. This argument supplies the y values. The corresponding x values default to range(len([5, 6, 10, 3])), which is why they range from 0 to 3 in this case.



ContentsofFigure-Jane.png



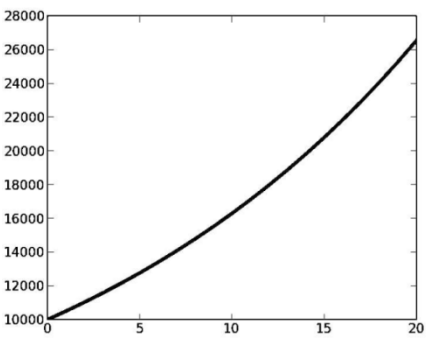
Contents ofFigure-Addie.png

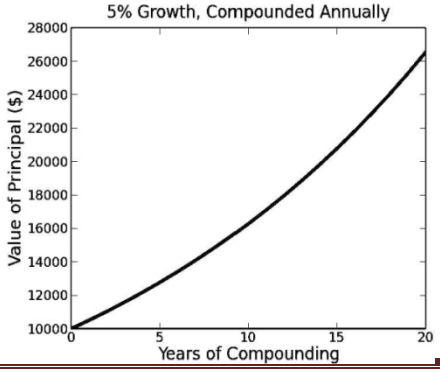
PyLab has a notion of "current figure." Executing pylab.figure(x) sets the current figure to the figure numbered x. Subsequently executed calls of plotting functions implicitly refer to that figure until another invocation of pylab.figure occurs. This explains why the figure written to the file Figure-Addie.pngwas the second figure created.

Example

```
principal = 10000 #initial investment
interestRate = 0.05
years = 0 values=[]
for i in range(years + 1):
        values.append(principal)
        principal += principal interestRate
pylab.plot(values)
```

produces the plot on the left below.



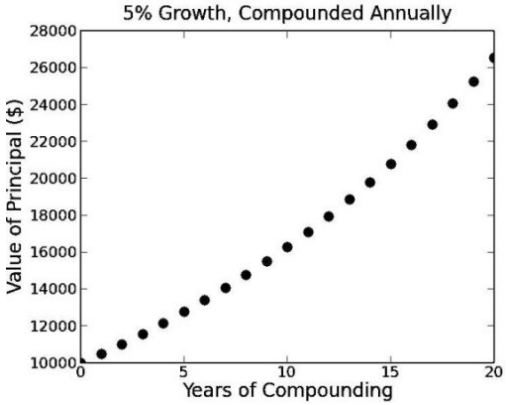


If we look at the code, we can deduce that this is a plot showing the growth of an initial investment of 10,000 at an annually compounded interest rate of 5%. However, this cannot be easily inferred by looking only at the plot itself. That's a badthing.Allplotsshouldhaveinformativetitles, andallaxesshouldbelabeled.

If we add to the end of our the code the lines

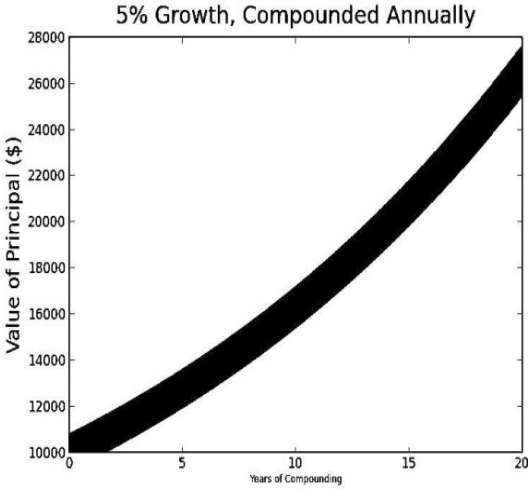
```
pylab.title('5% Growth, Compounded Annually')
pylab.xlabel('Years of Compounding')
pylab.ylabel('Value of Principal ($)')
```

we get the plot above and on the right. For every plotted curve, there is an optional argument that is a format string indicating the color and line type of the plot.[60] The letters and symbols of the format string are derived from those used in MATLAB, and are composed of a color indicator followed by a line-style indicator. The default format string is 'b-', which produces a solid blue line. To plot the above with red circles, one would replace the call pylab.plot(values) by pylab.plot(values, 'ro'),which produces the plot on the right. For a complete list of color and line-style indicators.



It's also possible to change the type size and line width used in plots. This can be done using keyword arguments in individual calls to functions, e.g., the code

```
principal = 10000 #initial investment
interestRate = 0.05
years = 20
values = []
for i in range(years + 1):
values.append(principal)
principal += principal interestRate
pylab.plot(values, linewidth = 30)
pylab.title('5% Growth, Compounded Annually',fontsize =
'xx-large')
pylab.xlabel('Years of Compounding', fontsize = 'x-small')
pylab.ylabel('Value of Principal ($)')
```



produces the intentionally bizarre-looking plot

It is also possible to change the default values, which are known as "rc

settings." (The name "rc" is derived from the .rc file extension used for runtimeconfiguration files in Unix.) These values are stored in a dictionary-like variablethat can be accessed via the name pylab.rcParams.So,for example, you canset the default line width to 6 pointsby executing the codepylab.rcParams['lines.linewidth'] =6.

The default values used in most of the examples in this book were set with the code

```
#set line width
        pylab.rcParams['lines.linewidth'] = 4
#set font size for titles
        pylab.rcParams['axes.titlesize'] = 20
#set font size for labels on axes
        pylab.rcParams['axes.labelsize'] = 20
#set size of numbers on x-axis
         pylab.rcParams['xtick.labelsize'] = 16
#set size of numbers on y-axis
        pylab.rcParams['ytick.labelsize'] = 16
#set size of ticks on x-axis
        pylab.rcParams['xtick.major.size'] = 7
#set size of ticks on y-axis
        pylab.rcParams['ytick.major.size'] = 7
#set size of markers
        pylab.rcParams['lines.markersize']= 10
```

If you are viewing plots on a color display, you will have little reasonto customize these settings. We customized the settings we used so that itwould be easier to read the plots when we shrank them and convertedthemto black and white.

Plotting Mortgages, an ExtendedExample

we worked our way through a hierarchy of mortgages as way of illustrating the use of sub classing. We concluded that chapter by observing that "our program should be producing plots designed to show how the mortgage behaves over time." Figureenhances class Mortgage by adding methods that make it convenient to produce such plots.

ThemethodsplotPaymentsandplotBalancearesimpleone-liners,buttheydouse a form of pylab.plot that we have not yet seen. When a figure contains multiple plots, it is useful to produce a key that identifies what each plot is intended to represent. In Figure 11.1, each invocation of pylab.plot uses the label keyword argument to associate a string with the plot produced by that invocation. (This and other keyword arguments must follow any format strings.) A key can then be added to the figure by calling the function pylab.legend,

The nontrivial methods in class Mortgage are plotTotPd and plotNet. The method plotTotPd simply plots the cumulative total of the payments made. The method plotNet plots an approximation to the total cost of the mortgage over timeby plotting the cash expended minus the equity acquired by paying off art of the loan.

It is an approximation because it does not perform a net present value calculation to take into account the time value of cash.

Class Mortgage with plotting methods

```
class Mortgage(object):
        """Abstract class for building different kinds of mortgages"""
    def __init__(self, loan, annRate, months):
        """Create a new mortgage"""
        self.loan = loan
        self.rate = annRate/12.0
        self.months = months
        self.paid = [0.0]
        self.owed = [loan]
        self.payment = findPayment(loan, self.rate, months)
        self.legend = None #description of mortgage

    def makePayment(self):
        """Make a payment"""
        self.paid.append(self.payment)
        reduction = self.payment - self.owed[-1] *self.rate
        self.owed.append(self.owed[-1] - reduction)

    def getTotalPaid(self):
        """Return the total amount paid so far"""
        return sum(self.paid)
    def__str__(self):
        return self.legend

    def plotPayments(self, style):
        pylab.plot(self.paid[1:], style, label = self.legend)

    def plotBalance(self, style):
        pylab.plot(self.owed, style, label = self.legend)

    def plotTotPd(self, style):
        """Plot the cumulative total of the payments made"""
        totPd = [self.paid[0]]
        for i in range(1, len(self.paid)):
            totPd.append(totPd[-1] + self.paid[i])
        pylab.plot(totPd, style, label = self.legend)

    def plotNet(self, style):
        """Plot an approximation to the total cost of the mortgage over time by
    plotting the cash expended minus the equity acquired by paying off part of
    the loan"""
        totPd = [self.paid[0]]
        for i in range(1, len(self.paid)):
            totPd.append(totPd[-1] + self.paid[i])
```

```
        #Equity acquired through payments is amount of original loan
        #paid to date, which is amount of loan minus what is still owed

        equityAcquired = pylab.array([self.loan]len(self.owed))
        equityAcquired = equityAcquired - pylab.array(self.owed)
net = pylab.array(totPd) - equityAcquired
        pylab.plot(net, style, label = self.legend)
```

The expression pylab.array(self.owed) in plotNet performs a type conversion. Thus far, we have been calling the plotting functions of PyLab with arguments of typelist. Underthecovers, PyLab has been converting these lists to a different type, **array**, which PyLab inherits from NumPy.63 The invocation pylab.array makes this explicit. There are a number of convenient ways to manipulate arrays that are not readily available for lists. In particular, expressions can be formed using arrays and arithmetic operators. Consider, for example, the code

```
a1 = pylab.array([1, 2, 4])
print 'a1 =', a1
a2 = a1*2
print 'a2 =', a2
print 'a1 + 3 =', a1 + 3
print '3 - a1 =', 3 - a1
print 'a1 - a2 =', a1 - a2
print 'a1 a2 =', a1 a2
```

The expression a1*2 multiplies each element of a1 by the constant 2. The expression a1+3 adds the integer 3 to each element of a1. The expression a1-a2 subtracts each element of a2 from the corresponding element of a1 (if the arrays had been of different length, an error would have occurred). The expression a1 a2 multiplies each element of a1 by the corresponding element of a2. When the above code is run it prints

```
a1 = [1 24]
a2 = [2 48]
a1 + 3 = [4 5 7]
3 - a1 =[2 1-1]
a1 - a2 = [-1 -2 -4]
a1 a2 =[2 832]
```

There are a number of ways to create arrays in PyLab, but the most common way istofirstcreatealist and then convertit.

repeats the three subclasses of Mortgage from Chapter 8. Each has a distinctinit that overrides theinit in Mortgage. The subclass TwoRate also overrides the makePayment method ofMortgage.

Subclasses of Mortgage

```
class Fixed(Mortgage):
def __init__(self, loan, r, months):
        Mortgage.__init__(self, loan, r, months)
        self.legend = 'Fixed, ' + str(r*100) + '%'
```

```
class FixedWithPts(Mortgage):
        def __init__(self, loan, r, months, pts):
        Mortgage.__init__(self, loan, r, months)
        self.pts = pts
        self.paid = [loan*(pts/100.0)]
        self.legend = 'Fixed, ' + str(r*100) + '%, '\+ str(pts) + ' points'

class TwoRate(Mortgage):
        def __init__(self, loan, r, months, teaserRate, teaserMonths):
        Mortgage.__init__(self, loan, teaserRate, months)
        self.teaserMonths = teaserMonths
        self.teaserRate = teaserRate
        self.nextRate =  r/12.0
        self.legend = str(teaserRate*100)+ '% for ' + str(self.teaserMonths)\+ ' months,
then ' + str(r*100) + '%'

def makePayment(self):
        if len(self.paid) == self.teaserMonths + 1:
                self.rate = self.nextRate
                self.payment = findPayment(self.owed[-1], self.rate,self.months -
        self.teaserMonths)
                Mortgage.makePayment(self)
```

contain functions that can be used to generate plots intended to provide insight about the different kinds of mortgages.

The function plotMortgages generates appropriate titles and axis labels for each plot, and then uses the methods in MortgagePlots to produce the actual plots. It uses calls to pylab.figure to ensure that the appropriate plots appear in a given figure. It uses the index i to select elements from the lists morts and styles in a way that ensures that different kinds of mortgages are represented in a consistent way across figures. For example, since the third element in morts is a variable- rate mortgage and the third element in styles is 'b:', the variable-rate mortgage isalways plotted using a blue dottedline.

Generate Mortgage Plots

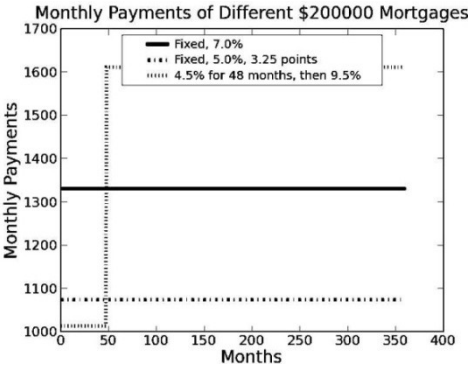
```
def plotMortgages(morts, amt):
        styles = ['b-', 'b-.', 'b:']
        cost = 1
        balance = 2
        netCost = 3 pylab.figure(payments)
        pylab.title('Monthly Payments of Different $' + str(amt)+ ' Mortgages')
        pylab.xlabel('Months')
        pylab.ylabel('Monthly Payments')
        pylab.figure(cost)
        pylab.title('Cash Outlay of Different $' + str(amt) + ' Mortgages')
        pylab.xlabel('Months')
        pylab.ylabel('Total Payments')
        pylab.figure(balance)
```

```
        pylab.title('Balance Remaining of $' + str(amt) + ' Mortgages')
        pylab.xlabel('Months')
        pylab.ylabel('Remaining Loan Balance of $')
        pylab.figure(netCost)
        pylab.title('Net Cost of $' + str(amt) + ' Mortgages')
        pylab.xlabel('Months')
        pylab.ylabel('Payments - Equity $')

        for i in range(len(morts)):
                pylab.figure(payments)
                morts[i].plotPayments(styles[i])
                pylab.figure(cost)
                morts[i].plotTotPd(styles[i])
                pylab.figure(balance)
                morts[i].plotBalance(styles[i])
                pylab.figure(netCost)
                morts[i].plotNet(styles[i])
        pylab.figure(payments)
        pylab.legend(loc = 'upper center') pylab.figure(cost)
        pylab.legend(loc = 'best')
        pylab.figure(balance)
        pylab.legend(loc ='best')

def compareMortgages(amt, years, fixedRate, pts, ptsRate,varRate1, varRate2,
varMonths):
        totMonths = years 12
        fixed1 = Fixed(amt, fixedRate, totMonths)
        fixed2 = FixedWithPts(amt, ptsRate, totMonths, pts)
        twoRate = TwoRate(amt, varRate2, totMonths, varRate1, varMonths)
        morts = [fixed1, fixed2, twoRate]
        for m in range(totMonths):
                for mort in morts:
                        mort.makePayment()
plotMortgages(morts, amt)

def compareMortgages(amt, years, fixedRate, pts, ptsRate,varRate1, varRate2,
varMonths):
        totMonths = years 12
        fixed1 = Fixed(amt, fixedRate, totMonths)
        fixed2 = FixedWithPts(amt, ptsRate, totMonths, pts)
        twoRate = TwoRate(amt, varRate2, totMonths, varRate1, varMonths)
        morts = [fixed1, fixed2, twoRate]
        for m in range(totMonths):
                for mort in morts:
                        mort.makePayment()
         plotMortgages(morts, amt)
```

The function compareMortgages generates a list of different mortgages, and simulates making a series of payments on each, It then calls plotMortgages

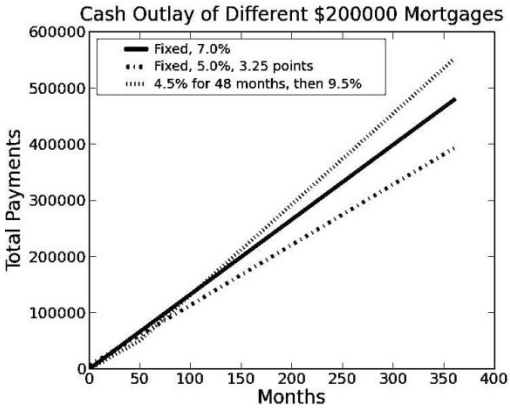
to produce theplots.

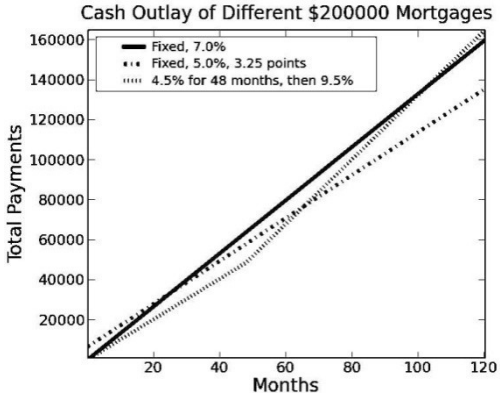
compareMortgages(amt=200000, years=30, fixedRate=0.07, pts = 3.25, ptsRate=0.05, varRate1=0.045, varRate2=0.095, varMonths=48)



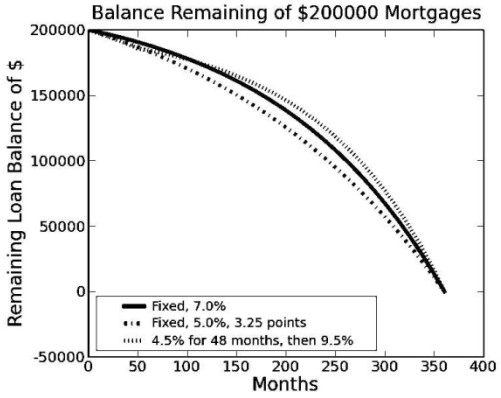
The first plot, which was produced by invocations of plotPayments, simply plots each payment of each mortgage against time. The box containing the key appears where it does because of the value supplied to the keyword argument loc used in the call to pylab.legend. When loc is bound to 'best' the location is chosen automatically. This plot makes it clear how the monthly payments vary (or don't) overtime,but doesn't shed much light on the relative costs of each kind of mortgage.

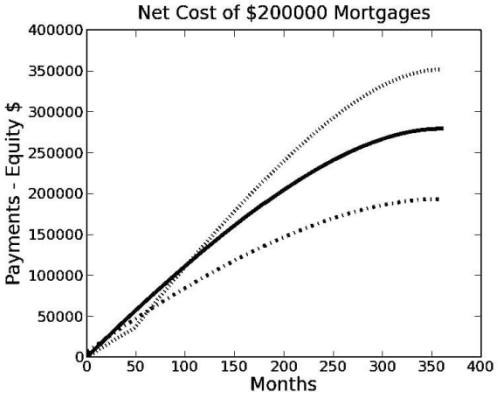
The next plot was produced by invocations of plotTotPd. It sheds some light on the cost of each kind of mortgage by plotting the cumulative costs that have been incurred at the start of each month. The entire plot is on the left, and an enlargement of the left part of the plot is on the right.





The next two plots show the remaining debt (on the left) and the total net cost of having the mortgage (on the right).





Dynamic Programming

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into sub-problems and stores the results of sub-problems to avoid computing the same results again. Following are the two main properties of a problem that suggests that the given problem can be solved using Dynamic programming.

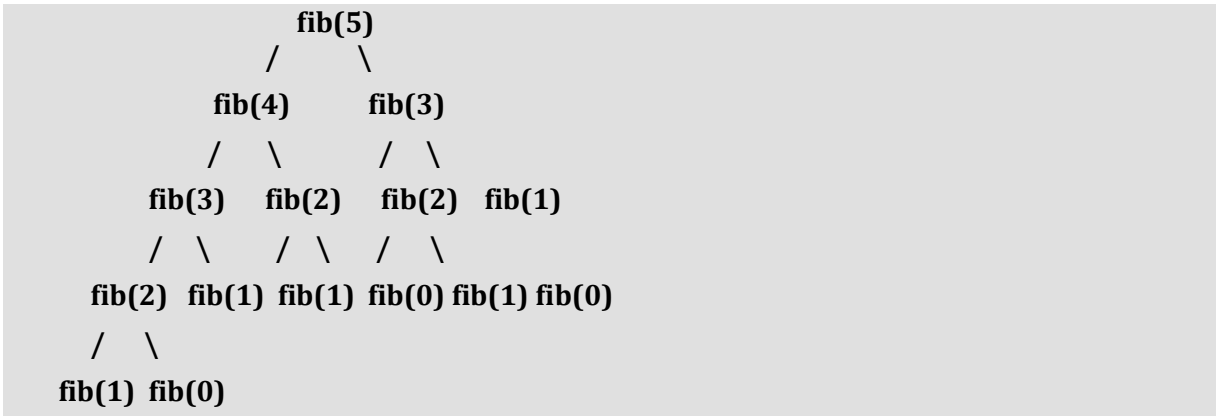
- Overlapping Sub-problems
- Optimal Substructure

1) Overlapping Sub-problems:

Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same sub-problems are needed again and again. In dynamic programming, computed solutions to sub-problems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) sub-problems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common sub-problems. If we take an example of following recursive program for Fibonacci Numbers, there are many sub-problems which are solved again and again.

```
intfib(intn)
{
  if( n <= 1 )
        returnn;
  returnfib(n-1) + fib(n-2);
}
```

Recursion tree for execution of *fib(5)*



We can see that the function fib(3) is being called 2 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:

Memoization (Top Down):

The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need the solution to a sub-problem, we first look into the lookup table. If the pre-computed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

Following is the memoized version for nth Fibonacci Number.

```
# Function to calculate nth Fibonacci number
deffib(n, lookup):
# Base case
        ifn ==0orn ==1:
                arr[n] =n
# If the value is not calculated previously then calculate it
        if arr[n] isNone:
                arr[n] =fib(n-1, arr)  +fib(n-2,arr)
# return the value corresponding to that value of n
                returnarr[n]
# end of function

# Driver program to test the above function
defmain():
        n =34
# Declaration of lookup table
# Handles till n = 100
        arr =[None]*(101)
        print"Fibonacci Number is ", fib(n, arr)
        if__name__=="__main__":
                main()
```

Tabulation (Bottom Up):

The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of sub-problems bottom-up.

Following is the tabulated version for nth Fibonacci Number.

```
# Python program Tabulated (bottom up) version
deffib(n):
# array declaration
        f =[0]*(n+1)
# base case assignment
        f[1] =1
# calculating the fibonacci and storing the values
        fori inxrange(2, n+1):
                f[i] =f[i-1] +f[i-2]
                returnf[n]
# Driver program to test the above function
defmain():
        n =9
        print"Fibonacci number is ", fib(n)
 if__name__=="__main__":
        main()
```

Output:

Fibonacci number is 34

Both Tabulated and Memoized store the solutions of sub-problems. In Memoized version, table is filled on demand while in Tabulated version, starting from the first entry, all entries are filled one by one. Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version.

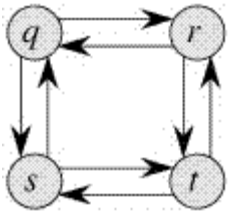
To see the optimization achieved by Memoized and Tabulated solutions over the basic Recursive solution, see the time taken by following runs for calculating 40th Fibonacci number:

2) Optimal Substructure: A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its sub-problems.

For example, the Shortest Path problem has following optimal substructure property:

If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithms like Floyd–Warshall and Bellman–Ford are typical examples of Dynamic Programming.

On the other hand, the Longest Path problem doesn't have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following un-weighted graph given in the CLRS book. There are two longest paths from q to t: **q→r→t** and **q→s→t**. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path **q→r→t** is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is **q→s→t→r** and the longest path from r to t is **r→q→s→t.**



Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

Limitations and Characteristics Dynamic Programming

As we have just found out, there are two key characteristics that a problem must have in order for us to try to solve it using dynamic programming:

- The optimal sub-structure: It should be possible to compose the optimal solution to the problem from the optimal solution of its sub-problems.

- Intersecting sub-problems: The problem must be broken down into sub-problems, which in turn are re-used. In other words, a recursive approach to

solving a problem would imply multiple (not single) solutions of the same sub-problem, instead of generating all the new and unique sub-problems in each recursive cycle.

Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a sub-problem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Problem Scenario

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are **n** items in the store
- Weight of **ith** item *wi*>0

Profit for **ith** item *pi*>0

- AndCapacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of **ith** item.

0 ≤ xi ≤ 1

The **ith** item contributes the weight *xi*.*wi* to the total weight in the knapsack and profit *xi*.*pi* to the total profit. Hence, the objective of this algorithm is to **maximize∑n=1n(xi.pi)** subject to **constraint, ∑n=1n(xi.wi)W**

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit. Thus, an optimal solution can be obtained by **∑n=1n(xi.wi)=W**

In this context, first we need to sort those items according to the value of *piwi* , so that **pi+1wi+1 ≤ piwi**. Here, ***x*** is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
  do x[i] = 0
weight = 0
for i = 1 to n
  if weight + w[i] ≤ W then
    x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
    weight = W
    break
return x
```

Analysis

If the provided items are already sorted into a decreasing order of **piwi** , then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Example

Let us consider that the capacity of the knapsack ***W* = 60** and the list of provided items are shown in the following table –

Item	**A**	**B**	**C**	**D**
Profit	280	100	120	120
Weight	40	10	20	24
Ratio (*piwi*)	7	10	6	5

As the provided items are not sorted based on **piwi** . After sorting, the items are as shown in the following table.

Item	**B**	**A**	**C**	**D**
Profit	100	280	120	120
Weight	10	40	20	24
Ratio (*piwi*)	10	7	6	5

Solution

After sorting all the items according to *piwi*. First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***. Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more items can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60** And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either ***0*** or ***1***, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example-1

Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio p_i/w_i. Let us consider that the capacity of the knapsack is $W = 60$ and the items are as shown in the following table.

Item	A	B	C
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6

Using the Greedy approach, first item ***A*** is selected. Then, the next item ***B*** is chosen. Hence, the total profit is **100 + 280 = 380**. However, the optimal solution of this instance can be achieved by selecting items, ***B*** and ***C***, where the total profit is **280 + 120 = 400**. Hence, it can be concluded that Greedy approach may not give an optimal solution. To solve 0-1 Knapsack, Dynamic Programming approach is required.

Problem Statement

A thief is robbing a store and can carry a max$_i$mal weight of ***W*** into his knapsack. There are ***n*** items and weight of **ith** item is w_i and the profit of selecting this item is p_i. What items should the thief take?

Dynamic-Programming Approach

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution S is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the max$_i$mum weight **w**.

The algorithm takes the following inputs

- The max$_i$mum weight **W**
- The number of items **n**
- The two sequences **v = <v$_1$, v$_2$, …, v$_n$>** and **w = <w$_1$, w$_2$, …, w$_n$>**

```
Dynamic-0-1-knapsack (v, w, n, W)
    for w = 0 to W do
      c[0, w] = 0
    for i = 1 to n do
      c[i, 0] = 0
      for w = 1 to W do
        if wi ≤ w then
          if vi + c[i-1, w-wi] then
```

c[i, w] = v_i + c[i-1, w-w_i]
else c[i, w] = c[i-1, w]
else
c[i, w] = c[i-1, w]

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from. If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, w-W]**.

Analysis

This algorithm takes θ(*n*, *w*) times as table *c* has (*n* + 1).(*w* + 1) entries, where each entry requires θ(1) time to compute.

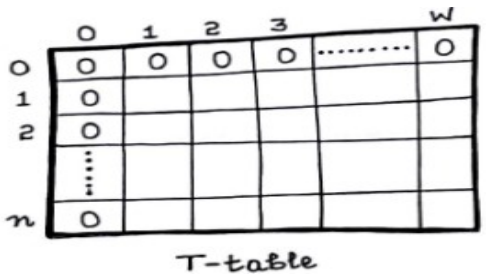
0/1 Knapsack Problem

In 0/1 Knapsack Problem,

- As the name suggests, items are indivisible i.e. here we have to either take an item completely or leave it completely but we cannot take the fraction of any item.
- 0/1 Knapsack Problem is solved using dynamic programming approach.

Steps for solving 0/1 Knapsack Problem

Consider a knapsack of weight capacity 'W' and 'n' number of items with some weights and values are given.



Step-01:

Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns and fill all the boxes of 0^{th} row and 0^{th} column with zeroes as shown below-

Step-02:

Now, start to fill the table row wise from left to right using the following formula-

T (i , j) = max { T (i-1 , j) , $value_i$ + T(i-1 , j – $weight_i$) }

T(i , j) = maximum value of the selected items if we can take items 1 to i and we have weight restrictions of j.

Time Complexity of 0/1 Knapsack Problem

- It takes θ(nw) time for filling the (n+1)(w+1) table entries where each entry takes constant time θ(1) for its computation.
- Then, it takes θ(n) time for tracing the solution as the tracing process traces the n rows of the table starting from row n and then moving up by 1 row at every step.
- Thus, overall θ(nw) time is taken to solve 0/1 Knapsack Problem using dynamic programming approach.

PRACTICE PROBLEM BASED ON 0/1 KNAPSACK PROBLEM

Problem

For the given set of items and knapsack capacity = 5 kg, find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.

Item	Weight	Value
1	2	3
2	3	4

3	4	5
4	5	6

OR

Find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach. Consider-

n = 4
w = 5 kg
(w1, w2, w3, w4) = (2, 3, 4, 5)
(b1, b2, b3, b4) = (3, 4, 5, 6)

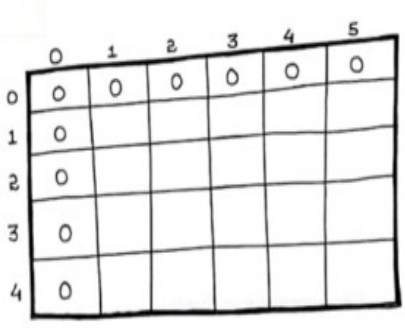
OR

A thief enters a house for robbing it. He can carry a maximal weight of 5 kg into his bag. There are 4 items in the house with the following weights and values. What items should thief take if he either takes the item completely or leaves it completely?

Item	Weight (kg)	Value ($)
Mirror	2	3
Silver nugget	3	4
Painting	4	5
Vase	5	6

Solution

- Knapsack capacity (w) = 5 kg
- Number of items (n) = 4



Step-01:

Draw a table say 'T' with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns and fill all the boxes of 0^{th} row and 0^{th} column with 0.

Step-02:

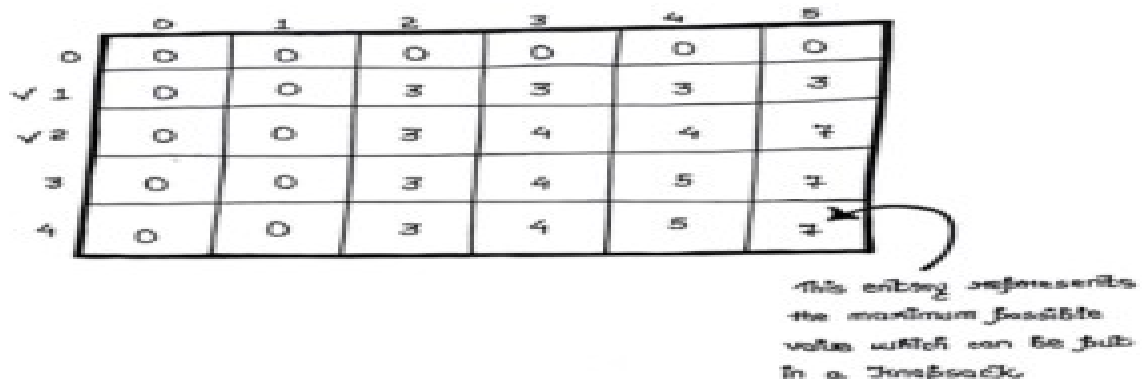
Now, start to fill the table row wise from left to right using the following formula-

$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$

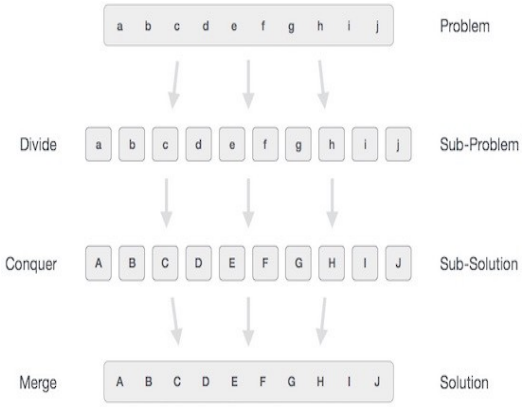
Finding T(1,1)	Finding T(1,2)
We have, i = 1 j = 1 $(value)_i = (value)_1 = 3$ $(weight)_i = (weight)_1 = 2$ Substituting the values, we get- T(1,1) = max { T(1-1 , 1) , 3 + T(1-1 , 1-2) } T(1,1) = max { T(0,1) , 3 + T(0,-1) } T(1,1) = T(0,1) { Ignore T(0,-1) } T(1,1) = 0	We have, i = 1 j = 2 $(value)_i = (value)_1 = 3$ $(weight)_i = (weight)_1 = 2$ Substituting the values, we get- T(1,2) = max { T(1-1 , 2) , 3 + T(1-1 , 2-2) } T(1,2) = max { T(0,2) , 3 + T(0,0) } T(1,2) = max {0 , 3+0} T(1,2) = 3
Finding T(1,3)	**Finding T(1,4)**
We have, i = 1 j = 3 $(value)_i = (value)_1 = 3$ $(weight)_i = (weight)_1 = 2$	We have, i = 1 j = 4 $(value)_i = (value)_1 = 3$ $(weight)_i = (weight)_1 = 2$

Substituting the values, we get- T(1,3) = max { T(1-1 , 3) , 3 + T(1-1 , 3-2) } T(1,3) = max { T(0,3) , 3 + T(0,1) } T(1,3) = max {0 , 3+0} T(1,3) = 3	Substituting the values, we get- T(1,4) = max { T(1-1 , 4) , 3 + T(1-1 , 4-2) } T(1,4) = max { T(0,4) , 3 + T(0,2) } T(1,4) = max {0 , 3+0} T(1,4) = 3
Finding T(1,5)	**Finding T(2,1)**
We have, i = 1 j = 5 $(value)_i = (value)_1 = 3$ $(weight)_i = (weight)_1 = 2$ Substituting the values, we get- T(1,5) = max { T(1-1 , 5) , 3 + T(1-1 , 5-2) } T(1,5) = max { T(0,5) , 3 + T(0,3) } T(1,5) = max {0 , 3+0} T(1,5) = 3	We have, i = 2 j = 1 $(value)_i = (value)2 = 4$ $(weight)_i = (weight)2 = 3$ Substituting the values, we get- T(2,1) = max { T(2-1 , 1) , 4 + T(2-1 , 1-3) } T(2,1) = max { T(1,1) , 4 + T(1,-2) } T(2,1) = T(1,1) { Ignore T(1,-2) } T(2,1) = 0
Finding T(2,2)	**Finding T(2,3)**
We have, i = 2 j = 2 $(value)_i = (value)2 = 4$ $(weight)_i = (weight)2 = 3$ Substituting the values, we get- T(2,2) = max { T(2-1 , 2) , 4 + T(2-1 , 2-3) } T(2,2) = max { T(1,2) , 4 + T(1,-1) } T(2,2) = T(1,2) { Ignore T(1,-1) } T(2,2) = 3	We have, i = 2 j = 3 $(value)_i = (value)2 = 4$ $(weight)_i = (weight)2 = 3$ Substituting the values, we get- T(2,3) = max { T(2-1 , 3) , 4 + T(2-1 , 3-3) } T(2,3) = max { T(1,3) , 4 + T(1,0) } T(2,3) = max { 3 , 4+0 } T(2,3) = 4
Finding T(2,4)	**Finding T(2,5)-**
We have, i = 2 j = 4 $(value)_i = (value)2 = 4$ $(weight)_i = (weight)2 = 3$ Substituting the values, we get- T(2,4) = max { T(2-1 , 4) , 4 + T(2-1 , 4-3) } T(2,4) = max { T(1,4) , 4 + T(1,1) } T(2,4) = max { 3 , 4+0 } T(2,4) = 4	We have, i = 2 j = 5 $(value)_i = (value)2 = 4$ $(weight)_i = (weight)2 = 3$ Substituting the values, we get- T(2,5) = max { T(2-1 , 5) , 4 + T(2-1 , 5-3) } T(2,5) = max { T(1,5) , 4 + T(1,2) } T(2,5) = max { 3 , 4+3 } T(2,5) = 7

Similarly, after computing all the values and filling them in the table, we get Thus, Maximum possible value which can be put in Knapsack = 7



Dynamic Programming With Divide and Conquer



In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub-problems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problems (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.

Broadly, we can understand **divide-and-conquer** approach in a three-step process.

- ***Divide:*** **Break the given problem into su-bproblems of same type.**
- ***Conquer:*** **Recursively solve these sub-problems**
- ***Combine:*** **Appropriately combine the answers**

Divide/Break

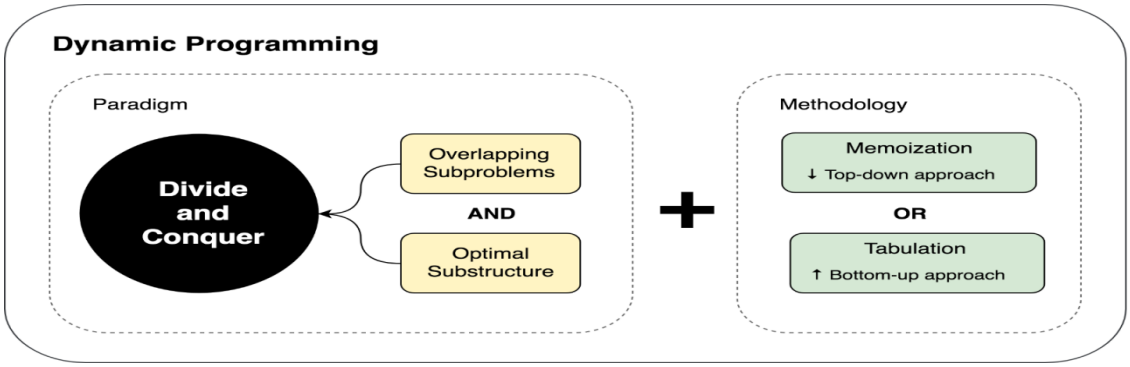
This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

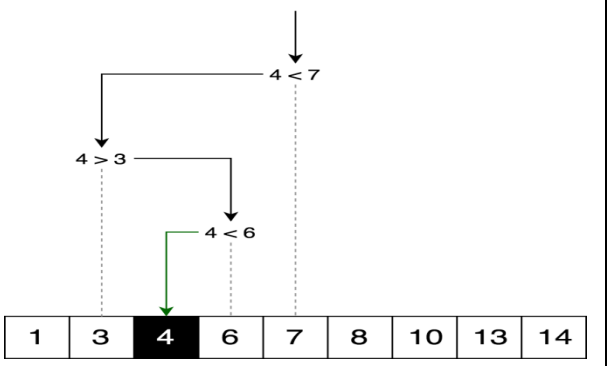
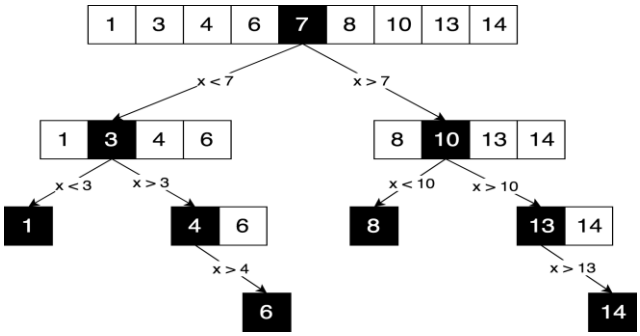
Merge/Combine

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquers & merges steps works so close that they appear as one.



Examples

The following program is an example of **divide-and-conquer** programming approach where the binary search is implemented using python.

PART -1	PART -2
	

Divide & Conquer Method	Dynamic Programming
It deals (involves) three steps at each level of recursion: • **Divide** the problem into a number of sub-problems. • **Conquer** the sub-problems by solving them recursively. • **Combine** the solution to the sub-problems into the solution for original sub-problems.	It involves the sequence of four steps: • Characterize the structure of optimal solutions. • Recursively defines the values of optimal solutions. • Compute the value of optimal solutions in a Bottom-up minimum. • Construct an Optimal Solution from computed information.
It is Recursive.	It is non Recursive.
It does more work on sub-problems and hence has more time consumption.	It solves sub-problems only once and then stores in the table.
It is a top-down approach.	It is a Bottom-up approach.
In this sub-problems are independent of each other.	In this sub-problems are interdependent.
For example: Merge Sort & Binary Search etc.	**For example:** Matrix Multiplication.

Unit -4 Network Programming and GUI using Python

PROTOCOL

A network protocol is an established set of rules that determine how data is transmitted between different devices in the same network. Essentially, it allows connected devices to communicate with each other, regardless of any differences in their internal processes, structure or design. Network protocols are the reason you can easily communicate with people all over the world, and thus play a critical role in modern digital communications.

Similar to the way that speaking the same language simplifies communication between two people, network protocols make it possible for devices to interact with each other because of predetermined rules built into devices' software and hardware. Neither local area networks (LAN) nor wide area networks (WAN) could function the way they do today without the use of network protocols.

The Following are two types of protocol models based on which other protocols are developed

1. TCP/IP Protocols
2. UDP

TCP/IP PROTOCOLS

The 4 Layers of the TCP/IP Model

The TCP/IP model defines how devices should transmit data between them and enables communication over networks and large distances. The model represents how data is exchanged and organized over networks. It is split into four layers, which set the standards for data exchange and represent how data is handled and packaged when being delivered between applications, devices, and servers.

The four layers of the TCP/IP model are as follows:

1. **Datalink layer**: The datalink layer defines how data should be sent, handles the physical act of sending and receiving data, and is responsible for transmitting data between applications or devices on a network. This includes defining how data should be signaled by hardware and other transmission devices on a network, such as a computer's device driver, an Ethernet cable, a network interface card (NIC), or a wireless network. It is also referred to as the link layer, network access layer, network interface layer, or physical layer and is the combination of the physical and data link layers of the Open Systems Interconnection (OSI) model, which standardizes communications functions on computing and telecommunications systems.
2. **Internet layer**: The internet layer is responsible for sending packets from a network and controlling their movement across a network to ensure they reach their destination. It provides the functions and procedures for transferring data sequences between applications and devices across networks.
3. **Transport layer:** The transport layer is responsible for providing a solid and reliable data connection between the original application or device and its intended destination. This is the level where data is divided into packets and numbered to create a sequence. The transport layer then determines how much data must be

sent, where it should be sent to, and at what rate. It ensures that data packets are sent without errors and in sequence and obtains the acknowledgment that the destination device has received the data packets.

4. **Application layer:** The application layer refers to programs that need TCP/IP to help them communicate with each other. This is the level that users typically interact with, such as email systems and messaging platforms. It combines the session, presentation, and application layers of the OSI model.

User Datagram Protocol (UDP)

UDP is another protocol that transfers data in a connection less and unreliable manner. It will not check how many bits are sent or how many bits are actually received at the other side. During transmission of data, there may be loss of some bits. Also, the data sent may not be received in the same order. Hence UDP is generally not used to send text. UDP is used to send images, audio files, and video files. Even if some bits are lost, still the image or audio file can be composed with a slight variation that will not disturb the original image or audio.

SOCKETS

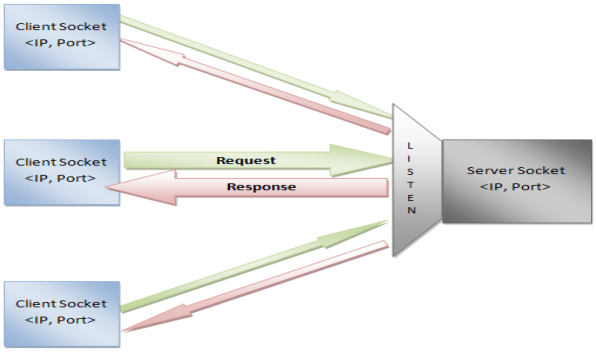
It is possible **to establish a logical connecting point between a server and a client so that communication can be done through that point. This point is called 'socket'**. Each socket is given an identification number, which is called '**port number'**. Port number takes 2 bytes and can be from 0 to 65,535. Establishing communication between a server and a client using sockets is called 'socket programming'.

We should use a new port number for each new socket. Similarly, we should allot a new port number depending on the service provided on a socket. Every new service on the net should be assigned a new port number. Please have a look at some already allotted port numbers for the services shown in Table. The port numbers from 0 to 1023 are used by our computer system for various applications (or services) and hence we should avoid using these port numbers in our networking Programs.

Table: Some Reserved Port Numbers and Associated Services

PORT	Application or services
13	Date and time Protocol
20	File Transfer Protocol (FTP) Data Transfer
21	File Transfer Protocol (FTP) Command Control
22	Secure Shell (SSH) Secure Login
23	Telnet remote login service, unencrypted text messages
25	Simple Mail Transfer Protocol (SMTP) email delivery
53	Domain Name System (DNS) service
67, 68	Dynamic Host Configuration Protocol (DHCP) Which provides configuration at boot time
80	Hypertext Transfer Protocol (HTTP) used in the World Wide Web
109	POP2, which is mailing services
110	Post Office Protocol (POP3) which is mailing services
119	Network News Transfer Protocol (NNTP)
123	Network Time Protocol (NTP)
143	Internet Message Access Protocol (IMAP) Management of digital mail
161	Simple Network Management Protocol (SNMP)
194	Internet Relay Chat (IRC)
443	HTTP/HTTPS transfers Secure web page

The points of the story are that a server on a network uses socket to connect to a client. When the server wants to communicate with several clients simultaneously, several sockets are needed. Every socket will have a different port number.



The socket Module

To create a socket, you must use the **socket.socket()** function available in the socket module, which has the general syntax

s = socket.socket (socket_family, socket_type, protocol = 0)

- **First argument** socket_family indicates which version of the IP address should be used whether IP address version 4 or 6. This argument can take either of the following two values
  Socket.AF_INT → version 4
  Socket.AF_INT6 → version 6
- **Second argumenent** is type which represent the type of the protocol to be used whether TCP/IP or UDP. This can assume one of the following two values.
  socket.SOCK_STREAM → FOR TCP/IP
  socket.SOCK_DGRAM → FOR UDP
- Last argument is protocol. This is usually left out, defaulting to 0.

Once you have socket object, then you can use the required functions to create your client or server program.

Knowing IP ADDRESS

To find IP address of a website, we can use gethostname() function available in socket module. This function takes the website name and retuens its IP address.

address = socket.gethostname('www.google.com')

above code will return the IP address of www.google.com website into address variable. If there is no such website on internet then it returns **gaierror(Get Address Information Error)**

```
# W.A.Python script to find IP ADDRESS
import socket
side = 'www.google.com'
try:
   ipaddr = socket.gethostbyname(side)
   print("GOOGLE IP ADDRESS IS :"+ipaddr)
except socket.gaierror:
   print("Google does not Exist")
```

URL

URL (Uniform Resource Locator) represents the address that is specified to access information or resource 'on Internet. Look at the example URL:

https://www.google.co.in/

The URL contains four parts:.

1. The protocol to use (http://)
2. The Server name or IP address of the. server (**https://www.google.co.in/**)
3. The Third part represents port number, which is optional (:80).
4. The last part is the file that is referred. This would be general index.html home.htm file (/index. html)

When a URL is given, we can parse the URL and find out all the parts of the URL with the help of urlparse() function of ***urllib.parse*** module in Python. We can pass the URL to **'urlparse() function**, it returns a tuple containing t the parts of the URL.

We can retrieve the individual parts of the URL from the tuple using the following attributes:

- scheme = this gives the protocol name used in the URL.
- netloc = gives the website name on the net with port number if present.
- path = gives the path of the web page.
- Port = gives the port number.

We can also get the total URL from the tuple by calling geturl() function. These details are shown Program 2.

```
import urllib
url = ""
d_p_url =  = urllib.parse.urlparse(url)
# DISPLAY FULL URL
print("FULL URL"+d_p_url)

# DISPLAY DIFFERENT PART OF URL
print("PROTOCOL USED URL NAME:- "+ d_p_url.scheme)
print("Net Location:- ",+d_p_url_netloc)
print("Web Page Path : -",+d_p_url.path)
print("Parameters :- ",d_p_url.params)
print("Port Number :- ",d_p_url.port)
print("Total URL :- ",d_p_url.geturl)
```

Reading the Source Code of a Web Page

If we know the URL of a Web page, it is possible to get the source code of the web page with the help of **urlopen()** function. This function belongs to urllib.request module. When we provide the URL of the web page, this function stores its source code into a file-like object and returns it as:

file = urllib.request.urlopen("https:\\python.org/")

Now, using read() method on file' object, we car: read the data of that object. This is shown in Program. Please remember that while executing this program, we should have Internet connection switched on in our computer.

```
import urllib.reuest
op_file = urllib.request.urlopen("http://python.org/")
print("READ CODE FROM WEBSIDE")
print(op_file.read())
```

Downloading a Web Page from Internet .

It is also possible to download the entire web page from Internet and save it into our computer system. Of course, the images of the web page may not be downloaded. To Download the webpage, we should have Internet connection switched on in our computer.

First we should open the web page using the **urlopen()** function. This function returns the content of the web page into a file Eke object. Now, we can read from this object using **read()** method as:

File = urllib.request.urlopen(https://python.org/)

The urlopen() function can raise "urllib.error.HTTPError". if the web page is not found. The next step is to open a file and write the 'content' data into that file. Since WebPages contain binary data, to store the web pages, we have to open the file in binary write mode

```
f=open('myfile.html','wb')
f.write(content)

import urllib.reuest
try:
   file = urllib.request.urlopen("https://www.python.org/")
   file_data = file.read()
except urllib.error.HttpError:
   print("Web Side Does Not Open")
   exit()

f = open("webcopy.html",wb)
f.write(file_data)
f.close()
```

Downloading an Image from Internet.

We can connect our computer, to Internet and. download images like **.jpg , .gif or .png files** from Internet into our Computer: system by writing a simple Python program For this purpose, we can use uilretrieve() function of urllib.request module. This Function takes the URL of the image, file and our, own image, file name as arguments.

```
download = urllib.request.urlretrieve(url,'myimage.jpg')
```

Here: download is a tuple object. URL represerts the URL string of the image location on internet. Myimage.jpg is the name of the file on which the image will be downloaded.

```
import urllib.request
url = "" # image file URL Link
download = urllib.request.urlretrieve(url,"image1.jpg")
```

A TCP/IP Server:

A server is a program that provides services to other computers on the network or Internet. Similarly a client is a program that receives services from the servers. When a server wants to communicate with a client, there is a need of a socket. A socket is a point of connection between the server and client. The following are the general steps to be used at server side

1. Create a TCP/IP socket at server side using socket() function.

```
S = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Here, socket.AF_INET represents IP Address version 4 and socket.SOCK_STREAM indicates that we are using TCP/IP protocol for communication with client. Anyhow, a socket uses IP address version 4 and TCP/IP protocol by default. Hence, we can create a socket without giving protocol version n and type of the Protocol as:

2. Bind the socket with host name and port number using bind() method. Here, host name can be an IP Address or a website name. If we want to run this program in an individual computer that is not connected in any network, then we can take hose name as 'localhost', this represents that the server is running in, the local system and not on any network; The IP Address of the 'localhost' is 127.0.0.1. As an alternative to 'localhost, we can also mention this IP Address. bind() method is used in the following format:

   s.bind((host,post)) # here host ,post is tuple

3. we can specify maximum number of connection using listen() method as:

   s.listen(5) # maximum connection allowed are 5

4. The server should wait till a client accept connection. thie is done using accept() method as:

   c , addr = s.accept() # this method returns c and address

   Here, 'c is connection abject that can be used' to sern messages to the client. 'addr' is the address of the client that has accepted the connecton.

5. Finally, using send() method, we can send message string to client. The message strings should be sent in the form of byte streams as they are used by the TCP/IP Protocol.

   c.send(b"message string")

   observe the b prefixed before the message string. This indicates that the string is a binary string. The other way to convert a string into binary format is using encode() method as

   string.encode()

6. After sending the messages to the client, the server can be disconnected by closing the connection object as

   c.close()

```
# server send message to cleint
import socket
# take a server naem and port number
host='localhost'
port=5000
# blind the stock with server side using TCP/IP protocol
s= socket.socket(socket.AF_INET,socket.SOCK_STREAM)
# blind the socket with server and port number
s.bind((host,port))
#allow maximum socket connect to socket
s.listen(1)
# allow till a client accept connection
c , address  = s.accept()
#display client address
print("Connection from :- " , str(address))
# send message to client
c.send(b"HELLO CLIENT, HOW ARE YOU")
```

```
msg = "BYE"
c.send(msg.encode())
#  disconnect the seerver
c.close()
```

A TCP/IP Client

A client is a program that receives the data or services from the server. We use following general steps in the client program;

1 At the client side, we should first create the socket object that uses TCP/IP protocol to receive data:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

2. Connect the socket to the server and port number using connect () method.

```
s.connect((host,port))
```

3. To receive the messages from the server, we can use recv () method as:

```
msg  = s.recv(1024)
```

Here, 1024 indicates the buffer size. This is the memory used while receiving the data. It means, at a time, 1024 bytes of data can be received from the server. We can change this limit in multiples of 1024. For example, we can use 2048 or 3072 etc. The received strings will be in binary format. Hence they can be converted into normal strings using decode () method.

4. Finally, we should disconnect the client by calling close() method on the socket object as:

```
s.close()
```

```
# cleint received message to server
import socket
# take a server naem and port number
host='localhost'
port=5000
# blind the stock with server side using TCP/IP protocol
s= socket.socket(socket.AF_INET,socket.SOCK_STREAM)
# connect it to server and port number
s.connect((host,port))
# received message string from server 1024 bytes
msg = s.recv(1024)
# repeat as long as message string are not empty
while msg:
   print("Received :- ", msg.decode())
   msg = s.recv(1024)

#  disconnect the seerver
s.close()
```

Save the above program as **Client1.py.** The Programs sever1 and client1 should be opened in two separate DOS windows and then executed: First we run the **Serverl.py** program in the left side DOS window and then we run the Client program in the right-side DOS Window. We can see the client's IP-Address and the post, number being displayed at the Server side window. The messages sent by the server are displayed at the client side window, as shown in the outputs.

A UDP Server

If we want to create a server that uses UDP protocol to send messages, we have to specify socket SOCK_DGRAM while creating the socket object, as:

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

It means the server will send data in the form of packets called 'datagrams', Now, using sendto() function, the server can send data to the client. Since UDP is a connection-less protocol, server does not know where the data should be sent. Hence we have to specify the client address in sendto() function, as

s.sendto('message string',(host,port))

Here, the "message string" is the binary string to be sent. The tuple (host, port) represents the host name and port number of the client. :

In Program, we are creating a server to send data to the client. The server will wait for 5 seconds after running it and then it send two messages to the client.

```
# UDP server that sends message to client
import socket as so
import time
 # take the server name and port number
 host ='localhost'
 port = 5000
 # create a socket at server side to use UDP protocol
 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
 # let the server waits for 5 seconds
 time.sleep(5)
 # send message to the client after encoding into binary string
 s.sendto(b"Hello Client, How Are You?",(host,port))
 msg = "BYE...!"
 s.sendto(msg.encode(),(host,post))
 # disconnect the server
 s.close()
```

Save this program as UDPserver. We will run this program only after the client program is also reply

A UDP Client

At the client side, the socket should be created to use UDP protocol as

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

The socket should be bound to the server using bind) method as:

s.bind(host,port)

Now, the client can receive messages with the help of recvfrom() method as

msg, addr = s.recvform(1024)

This method receives 1024 bytes at a time from the server which is called buffer size. This method returns the received message into 'msg' and the address of the server into 'addr'. Since the client does not know how many messages (i.e. strings) the server sends, we can use recvfrom() method inside a while loop and receive all the messages as:

```
While msg:
        print('Received : -'+msg.decode())
        msg, addr = s.recvfrom(1024)
```

But the problem here is that the client will hang when the server disconnects. To rectify this problem, we have to set some time for the socket so that the client will

automatically disconnect after that time elapses. This ig done using settimeout() method.

s.settimeout()

This method instructs the socket to block when 5 seconds time is elapsed. Hence if the server disconnects, the client will wait for another 5 seconds time and disconnects. The settimeout() method can raise socket.timeout exception.

```
import socket
# take the server name and port number
host = 'localhost'
post = 5000
# create a client side socket that uses UDP protocol
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# connect it to server with host name and post number
s.blind((host,port))
# receive message string from server 5 second at a time 1024 Bytes
msg, addr = s.recvfrom(1024)
try:
  # let the socket after 5 second if the server disconnect
  s.settimeout(5)

  while msg:
    print('Received :' + msg.decode())
    msg. addr = s.recvfrom(1024)

  except socket.timeout:
    print("Time Over and hence terminating")

# disconnect the client
s.close()
```

save this program as UDPclient. Now run the UDPclient program in dos windows and the UDPserver program in another DOS windows. It is better to run the UDPclient program first so that it will for the server. Then run the UDPserver program. This program will send two strings to the UDPclient after 5 second from the time of its execution. The UDPclient will wait for another 5 seconds then terminates.

If you remove the settimeout() method from UDPclient program and re-run both the UDPclient and UDPserver. You can observe the client hangs and it will never terminate.

File server

A file server is a server program that: accepts a file name from a client, searches for the file on the hard disk and sends the content of the file to the client. When the requested file is not found at server side, the server sends a message to the client saying File does not exist'. We can create a file server using TCP/IP type of socket connection and save the program as Fileserver.

```
# server that sends files contents to client
import socket
#take server name and port number
host='localhost'
```

```
port=6767
# create a TCP socket
s =socket.socket()
#bind socket to host and post number
s.bind((host,port))
#maximum 1 connection is accepted
s.listen(1)
# wait till client accepts a connection
c,addr = s.accept()
# accept file name from client
fname = c.recv(1024)
#convert file name into a normal string
fanme=str(fname.decode())
print('file name received from client :' + fname)

try:
   # open the file at server side
   f = open(fname,'rb')
   #read content of the file
   content =  f.read()
   # send file content to client
   c.send(content)
   print('File content sent to client')
   # close the file
   f.close()
except FileNotFoundError:
   c.send(b'File does not exist')

# dis connect server
c.close()
```

now, we write a code for the File client.

File Client

A file client is a client side program that sends a request to the server to search for a file and send the file contents. We have to type the file name from the key beard at the file client. This file name is sent to the file server and the file contents are received by the client in turn. We can create a file client program using TCP/IP type of socket connection and save the program as a FileClient

```
# client sned and received data
import socket
# tak server name and post
host='localhost'
port=6767
```

```
# create a TCP socket
s = socket.socket()
# connect to server
s.connect((host,port))
# type file name from keyboard
filename=input("Enter File Name : ")
#send file name to server
s.send(filename.encode())
# receive file content from server
content = s.recv(1024)
print(content.decode())
# dis connect the client
s.close()
```

Now, run the first **fileserver** and **fileclient** program in separate **Dos windows**. Enter file name in the fileserver program and you can see the file contents display at a client.

Two-Way Communication between Server and Client

It is possible to send data from client to the server and make the server respond to the client's request. After receiving the response from server, the client can again ask for some information from the server. In this way both the server and client can establish two way communications with each other. It means sending and receiving data will be done by both the server and client. The following programs will give you an idea of how to achieve this type of communication. These programs are at very fundamental level and need refinement. When refined, we can get programs for server and client which can take part in real chatting.

Chat Server Code	Chat Client Code

import socket
host='127.0.0.1'
port = 9000
 	
# create server side socket 	
s = socket.socket()
s.bind((host,port))
 	
# let maximum connect are 14 only 	
s.listen(1)
 	
# wait till a client connects 	
c , addr = s.accept()
print("A client Connected")
while True:
  # receive data from client
	import socket
	host='127.0.0.1'
	port = 9000
	
	# create client side socket
	s = socket.socket()
	s.connect((host,port))
	
	# enter Data at client
	str = input(" Enter Data")
	
	# continue as long as exit not entered by user
	while str != 'exit':
	  # send data from client to server
	  s.send(str.encode())

```
   data = c.recv(1024)
   # if client sends empty string, come out
   if not data:
      break
   print("From client: " +
str(data.decode())
   # enter response data from server
    data1 = input("Enter response")
   # send that data to client
   c.send(data1.decode())

# close connection
c.close()
```

```
   # receive data from client
   data = c.recv(1024)
   data = data.decode()
   print("form server : " + data)

   # enter data
   str=input("Enter Data")

# close connection
c.close()
```

Sending a Simple Mail

Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending e-mail and routing e-mail between mail servers.

Hear, being a powerful language don't need any external library to import and offers a native library to send emails- "SMTP lib". "smtplib" creates a Simple Mail Transfer Protocol client session object which is used to send emails to any valid email id on the internet. Different websites use different port numbers.

Syntax to create one SMTP object, which can later be used to send an e-mail

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

Here is the detail of the parameters –

- **host** – This is the host running your SMTP server. You can specify IP address of the host or a domain name like gmail.com. This is optional argument.
- **port** – If you are providing *host* argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.
- **local_hostname** – If your SMTP server is running on your local machine, then you can specify just *localhost* as of this option.

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

- **The sender** - A string with the address of the sender.
- **The receivers** - A list of strings, one for each recipient.
- **The message** - A message as a string formatted as specified in the various RFCs.

Here, we are using a Gmail account to send a mail. Port number used here is '587'. And if you want to send mail using a website other than Gmail, you need to get the corresponding information.

Steps to send mail from Gmail account:

1. First of all, "smtplib" library needs to be imported.
2. After that, to create a session, we will be using its instance SMTP to encapsulate an SMTP connection.

   s = smtplib.SMTP('smtp.gmail.com', 587)

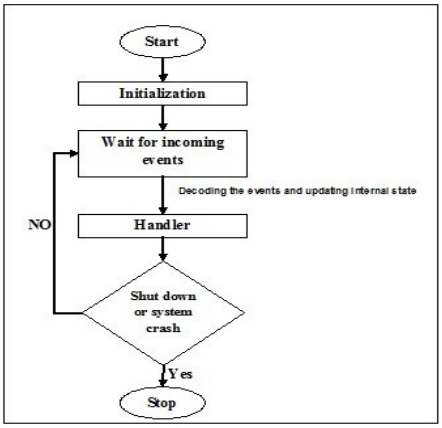
In this, you need to pass the first parameter of the server location and the second parameter of the port to use. For Gmail, we use port number 587.

3. For security reasons, now put the SMTP connection in the TLS mode. TLS (Transport Layer Security) encrypts all the SMTP commands. After that, for security and authentication, you need to pass your Gmail account credentials in the login instance. The compiler will show an authentication error if you enter invalid email id or password.
4. Store the message you need to send in a variable say, message. Using the sendmail() instance, send your message. sendmail() uses three parameters: **sender_email_id, receiver_email_id and message_to_be_sent**. The parameters need to be in the same sequence.

```
import smtplib
from email.mine.text import MIMEText

# first type the body text for our mail
TEXT = "This is my email. \nThis is sent to you from my python program \n I think you appreciated me."

# create MIMEtext class the email is sent
msg = MIMEText(TEXT)

# from which address the mail is sent
fromaddr = "jjkcc.bca@gmail.com"

# which adress the mail is sent
toaddr = "jjkcc.bca@yahoo.com"

# Stop the adress into msg object
msg['From']=fromaddr
msg['To']=toaddr
msg['Subject']= 'Hello Friends'

# connect to gmail.com server using 587 port number
server = smtplib.SMTP('smtp.gmail.com',587)

# put the smtp connection in TLS mode.
server.starttls()

# login to the server with your correct password
server.login(fromaddr,"mypassword")

# send the message to the server
server.send_message(msg)
print("Mail Sent....")

# close connection
server.quit()
```

GUI PROGRAMING

What is Event-Driven Programming



An event-driven program is also known as an event-driven application is a program designed to react to specific kinds of user input such as clicks on a command button, choosing a choice from a drop-down list, adding an entry into a text box, or other kinds of user events.

Event-driven programming separates event-processing logic from the rest of a program's code. The event-driven approach contrasts with batch processing. Because event-driven programming is an approach rather than a type of language, event-driven apps can be created in any programming language. Depending on the specific application, event-driven processing can improve responsiveness, throughput and flexibility.

What is event handling?

Event Handling is having a function or method containing program statements that are executed when an event occurs. An event handler typically is a software routine that processes actions such as keystrokes and mouse movements. Event-driven programming in python depends upon an event loop that is always listening for the new incoming events.

Events and Binds

Tkinter uses event sequences to define which events binds to handlers. It is the first argument "event" of the bind method. The event sequence is given as a string, using the following syntax:

<modifier-type-detail>

The type field is the essential part of an event specifier, whereas the "modifier" and "detail" fields are not obligatory and are left out in many cases. They are used to provide additional information for the chosen "type". The event "type" describes the kind of event to be bound, e.g. actions like mouse clicks, key presses or the widget got the input focus.

GUI

Tkinter is a Python Package for creating GUI applications. Python has a lot of GUI frameworks, but Tkinter is the only framework that's built into the Python standard library. Tkinter has several strengths; it's cross-platform, so the same code works on Windows, macOS, and Linux. Tkinter is lightweight and relatively painless to use compared to other frameworks. This makes it a compelling choice for building GUI applications in Python, especially for applications where a modern shine is unnecessary, and the top priority is to build something that's functional and cross-platform quickly.

To understand Tkinter better, we will create a simple GUI.

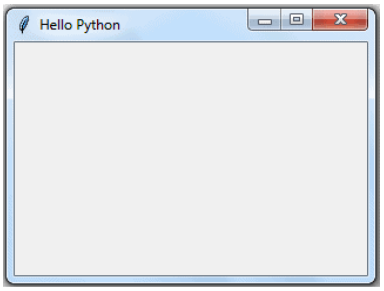
Getting Started

1. Import Tkinter package and all of its modules.
2. Create a root window. Give the root window a title (using title()) and dimension (using geometry()). All other widgets will be inside the root window.

3. Use mainloop() to call the endless loop of the window. If you forget to call this nothing will appear to the user. The window will wait for any user interaction till we close it.

Basic GUI Application

GUI elements and their functionality are defined in the Tkinter module. The following code demonstrates the steps in creating a UI.




```
from tkinter import *
window=Tk()
# add widgets here
window.title('Hello Python')
window.geometry("300x200+10+20")
window.mainloop()
```

First of all import the Tkinter module. After importing, setup the application object by calling the Tk() function. This will create a top-level window (root) having a frame with a title bar, control box with minimizes and close buttons, and a client area to hold other widgets.

The geometry() method defines the width, height and coordinates of the top left corner of the frame as below (all values are in pixels):

window.geometry("widthxheight+XPOS+YPOS")

The application object then enters an event listening loop by calling the **mainloop() method**. The application is now constantly waiting for any event generated on the elements in it. The event could be text entered in a text field, a selection made from the dropdown or radio button, single/double click actions of mouse, etc.

The application's functionality involves executing appropriate callback functions in response to a particular type of event. The event loop will terminate as and when the close button on the title bar is clicked. The above code will create the following window:

Buttons

The button widget is used to add various types of buttons to the python application. Python allows us to configure the look of the button according to our requirements. Various options can be set or reset depending upon the requirements.

We can also associate a method or function with a button which is called when the button is pressed.

Syntax

W = Button(parent, options)

- **Parent** - This represents the parent window.
- **Options** - Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

SN	Option	Description
1	activebackground	It represents the background of the button when the mouse hover the button.
2	activeforeground	It represents the font color of the button when the mouse hover the button.
3	Bd	It represents the border width in pixels.
4	Bg	It represents the background color of the button.

5	Command	It is set to the function call which is scheduled when the function is called.
6	Fg	Foreground color of the button.
7	Font	The font of the button text.
8	Height	The height of the button. The height is represented in the number of text lines for the textual lines or the number of pixels for the images.
10	Highlightcolor	The color of the highlight when the button has the focus.
11	Image	It is set to the image displayed on the button.
12	justify	It illustrates the way by which the multiple text lines are represented. It is set to LEFT for left justification, RIGHT for the right justification, and CENTER for the center.
13	Padx	Additional padding to the button in the horizontal direction.
14	pady	Additional padding to the button in the vertical direction.
15	Relief	It represents the type of the border. It can be SUNKEN, RAISED, GROOVE, and RIDGE.
17	State	This option is set to DISABLED to make the button unresponsive. The ACTIVE represents the active state of the button.
18	Underline	Set this option to make the button text underlined.
19	Width	The width of the button. It exists as a number of letters for textual buttons or pixels for image buttons.
20	Wraplength	If the value is set to a positive number, the text lines will be wrapped to fit within this length.

Label

The Label is used to specify the container box where we can place the text or images. This widget is used to provide the message to the user about other widgets used in the python application.

There are the various options which can be specified to configure the text or the part of the text shown in the Label.

Syntax

W = Label (master, options)

SN	Option	Description
1	anchor	It specifies the exact position of the text within the size provided to the widget. The default value is CENTER, which is used to center the text within the specified space.
2	Bg	The background color displayed behind the widget.
3	bitmap	It is used to set the bitmap to the graphical object specified so that, the label can represent the graphics instead of text.
4	bd	It represents the width of the border. The default is 2 pixels.
5	cursor	The mouse pointer will be changed to the type of the cursor specified, i.e., arrow, dot, etc.
6	font	The font type of the text written inside the widget.
7	fg	The foreground color of the text written inside the widget.
8	height	The height of the widget.
9	image	The image that is to be shown as the label.
10	justify	It is used to represent the orientation of the text if the text contains multiple lines. It can be set to LEFT for left justification, RIGHT for right justification, and CENTER for center justification.

11	padx	The horizontal padding of the text. The default value is 1.
12	pady	The vertical padding of the text. The default value is 1.
13	relief	The type of the border. The default value is FLAT.
14	text	This is set to the string variable which may contain one or more line of text.
15	textvariable	The text written inside the widget is set to the control variable StringVar so that it can be accessed and changed accordingly.
16	underline	We can display a line under the specified letter of the text. Set this option to the number of the letter under which the line will be displayed.
17	width	The width of the widget. It is specified as the number of characters.
18	wraplength	Instead of having only one line as the label text, we can break it to the number of lines where each line has the number of characters specified to this option.

Entry

The Entry widget is used to prove the single line text-box to the user to accept a value from the user. We can use the Entry widget to accept the text strings from the user. It can only be used for one line of text from the user. For multiple lines of text, we must use the text widget.

Syntax

w = Entry (parent, options)

SN	Option	Description
1	bg	The background color of the widget.
2	bd	The border width of the widget in pixels.
3	cursor	The mouse pointer will be changed to the cursor type set to the arrow, dot, etc.
4	exportselection	The text written inside the entry box will be automatically copied to the clipboard by default. We can set the exportselection to 0 to not copy this.
5	fg	It represents the color of the text.
6	font	It represents the font type of the text.
7	highlightbackground	It represents the color to display in the traversal highlight region when the widget does not have the input focus.
8	highlightcolor	It represents the color to use for the traversal highlight rectangle that is drawn around the widget when it has the input focus.
9	highlightthickness	It represents a non-negative value indicating the width of the highlight rectangle to draw around the outside of the widget when it has the input focus.
10	insertbackground	It represents the color to use as background in the area covered by the insertion cursor. This color will normally override either the normal background for the widget.
11	insertborderwidth	It represents a non-negative value indicating the width of the 3-D border to draw around the insertion cursor. The value may have any of the forms acceptable to Tk_GetPixels.
12	insertofftime	It represents a non-negative integer value indicating the number of milliseconds the insertion cursor should remain "off" in each blink cycle. If this option is zero, then the cursor doesn't blink: it is on all the time.

13	insertontime	Specifies a non-negative integer value indicating the number of milliseconds the insertion cursor should remain "on" in each blink cycle.
14	insertwidth	It represents the value indicating the total width of the insertion cursor. The value may have any of the forms acceptable to Tk_GetPixels.
15	justify	It specifies how the text is organized if the text contains multiple lines.
16	relief	It specifies the type of the border. Its default value is FLAT.
17	selectbackground	The background color of the selected text.
18	selectborderwidth	The width of the border to display around the selected task.
19	selectforeground	The font color of the selected task.
20	show	It is used to show the entry text of some other type instead of the string. For example, the password is typed using stars (*).
21	textvariable	It is set to the instance of the StringVar to retrieve the text from the entry.
22	width	The width of the displayed text or image.
23	xscrollcommand	The entry widget can be linked to the horizontal scrollbar if we want the user to enter more text then the actual width of the widget.

Entry widget methods

Python provides various methods to configure the data written inside the widget. There are the following methods provided by the Entry widget.

SN	Method	Description
1	delete(first, last = none)	It is used to delete the specified characters inside the widget.
2	get()	It is used to get the text written inside the widget.
3	icursor(index)	It is used to change the insertion cursor position. We can specify the index of the character before which, the cursor to be placed.
4	index(index)	It is used to place the cursor to the left of the character written at the specified index.
5	insert(index,s)	It is used to insert the specified string before the character placed at the specified index.
6	select_adjust(index)	It includes the selection of the character present at the specified index.
7	select_clear()	It clears the selection if some selection has been done.
8	select_form(index)	It sets the anchor index position to the character specified by the index.
9	select_present()	It returns true if some text in the Entry is selected otherwise returns false.
10	select_range(start,end)	It selects the characters to exist between the specified range.
11	select_to(index)	It selects all the characters from the beginning to the specified index.
12	xview(index)	It is used to link the entry widget to a horizontal scrollbar.
13	xview_scroll(number,what)	It is used to make the entry scrollable horizontally.

Dialogs:

in Python, There are serval Libraries for Graphical User Interface. Tkinter is one of them that is most useful. It is a standard interface. Tkinter is easy to use and provides serval functions for building efficient applications.

MESSAGEBOX

In Every Application, we need some Message to Display like "Do You Want To Close " or showing any warning or Something information. For this Tkinter provide a library like **messagebox**. By using the message box library we can show serval Information, Error, Warning, Cancelation ETC in the form of Message-Box. It has a Different message box for a different purpose.

Sr	Function name	Description
1	showinfo()	To display some important information.
2	showwarning()	To display some type of Warning.
3	showerror()	To display some Error Message.
4	askquestion()	To display a dialog box that asks with two options YES or NO.
5	askokcancel()	To display a dialog box that asks with two options OK or CANCEL.
6	askretrycancel()	To display a dialog box that asks with two options RETRY or CANCEL.
7	askyesnocancel()	To display a dialog box that asks with three options YES or NO or CANCEL.

Syntax of the Message-Box

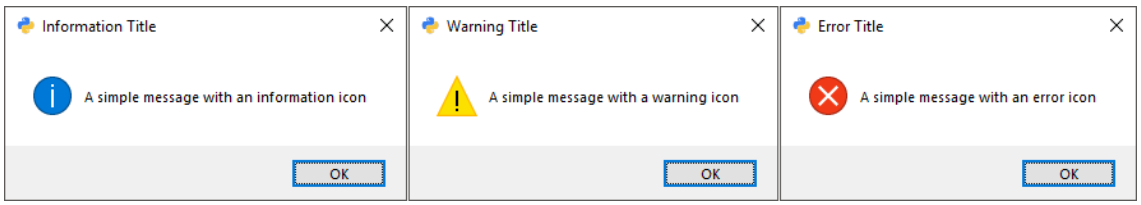
messagebox.name_of_function(Title, Message, [, options])

- name_of_function – Function name that which we want to use.
- Title – Message Box's Title.
- Message – Message that you want to show in the dialog.
- Options –To configure the options.

```
from tkinter import messagebox
# After creating parent...
info = messagebox.showinfo('Information Title', 'A simple message with an information icon', parent=parent)
warn = messagebox.showwarning('Warning Title', 'A simple message with a warning icon', parent=parent)
error = messagebox.showerror('Error Title', 'A simple message with an error icon', parent=parent)
```

Looking at the values of each info, warn and error after executing the lines, you will notice that they all have been assigned value "ok". If the user clicks the OK button or even closes the window using the cross, these dialogs will always return "ok".

Method	OK Clicked	Dialog Closed (X)
messagebox.showinfo	"ok"	"ok"
messagebox.showwarning	"ok"	"ok"
messagebox.showerror	"ok"	"ok"

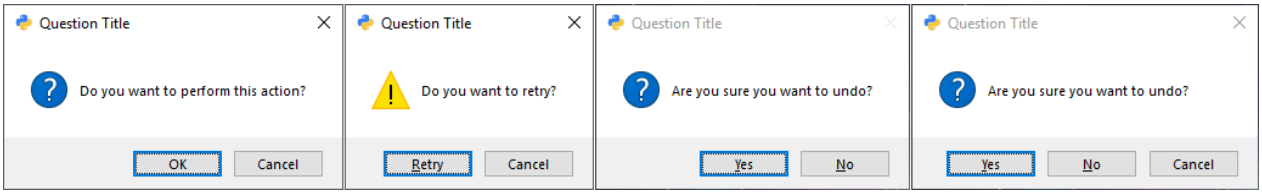


Question Dialogs

Question dialogs allow you to ask a user a particular question with a title and have them reply in one of two ways (or three in one case).

```
from tkinter import messagebox
# After creating parent...
okcancel = messagebox.askokcancel('Question Title', 'Do you want to perform this action?', parent=parent) # OK / Cancel
retrycancel = messagebox.askretrycancel('Question Title', 'Do you want to retry?', parent=parent) # Retry / Cancel
yesno = messagebox.askyesno('Question Title', 'Are you sure you want to undo?', parent=parent) # Yes / No
yesnocancel = messagebox.askyesnocancel('Question Title', 'Are you sure you want to undo?', parent=parent) # Yes / No / Cancel
```

Method	OK	Retry	Yes	No	Cancel	Dialog Closed (X)
messagebox.askokcancel	True				False	False
messagebox.askretrycancel		True			False	False
messagebox.askyesno			True	False		Not Possible
messagebox.askyesnocancel			True	False	None	None

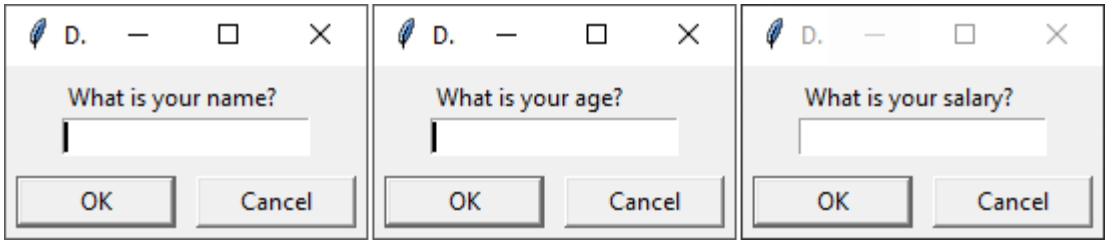


Input Dialogs

Input dialogs ask for a particular type of value; in this case that is either of type string, integer or float. Unfortunately like all the other dialogs, the icon from the parent will not be used due to how these dialogs are constructed. If you plan on consistently taking input from a user I recommend creating your own input dialog.

```
from tkinter import simpledialog
# After creating parent...
string_value = simpledialog.askstring('Dialog Title', 'What is your name?', parent=parent)
integer_value = simpledialog.askinteger('Dialog Title', 'What is your age?', minvalue=0, maxvalue=100, parent=parent)
float_value = simpledialog.askfloat('Dialog Title', 'What is your salary?', minvalue=0.0, maxvalue=100000.0, parent=parent)
```

Method	OK	Cancel	Dialog Closed (X)
simpledialog.askstring	Input String	None	None
simpledialog.askinteger	Input Integer	None	None
simpledialog.askfloat	Input Float	None	None



File / Folder Dialogs

File and folder dialogs allow you to ask the user select a folder, single file, multiple files or chose a location to save a file as.

```
from tkinter import filedialog
# After creating parent...
directory = filedialog.askdirectory(title='Select a folder', parent=parent)
# Ask the user to select a single file name.
file_name = filedialog.askopenfilename(title='Select a file', parent=parent)
# Ask the user to select a one or more file names.
file_names = filedialog.askopenfilenames(title='Select one or more files', parent=parent)
# Ask the user to select a single file name for saving.
save_as = filedialog.asksaveasfilename(title='Save as', parent=parent)
```

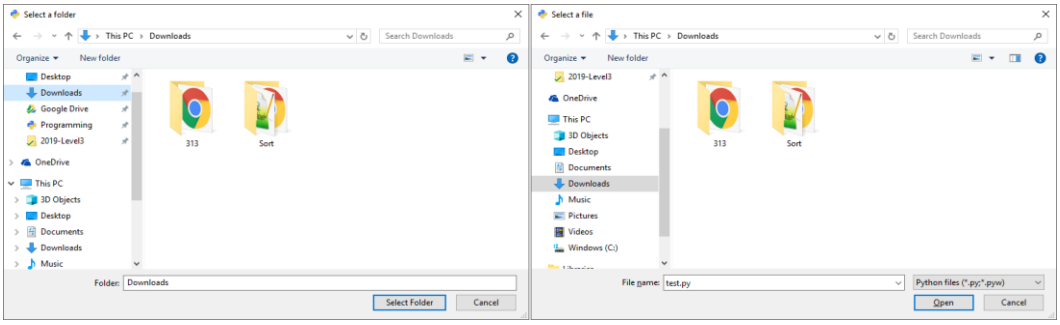
Method	OK	Cancel	Dialog Closed (X)
filedialog.askdirectory	Directory Path	''	''
filedialog.askopenfilename	File Path	''	''
filedialog.askopenfilenames	List of File Paths	''	''
filedialog.asksaveasfilename	File Path	''	''

When using any of these file dialog calls, you can also supply a string to the parameter initialdir which will put the user in a particular directory when the dialog first opens; for example:

```
directory_to_start_from = 'C:/Users/User/Downloads/'
directory = filedialog.askdirectory(initialdir=directory_to_start_from, title='Please select a folder:', parent=parent)
```

When asking for a file to open or save as, you can specify what type of files the user can select (based off file extension) using the filetypes argument. This argument takes a list of tuples, where each of these tuples contain a string of the group name and a string of file extensions separated by ; if required. The tuple that is the first in the list is the default selection. An example of the usage of filetypes:

```
file_types = [('Python files', '*.py;*.pyw'), ('All files', '*')] # Initially can select any .py or .pyw files but can change the selection to anything (*)
file_name = filedialog.askopenfilename(title='Select a file', filetypes=file_types, parent=parent)
```



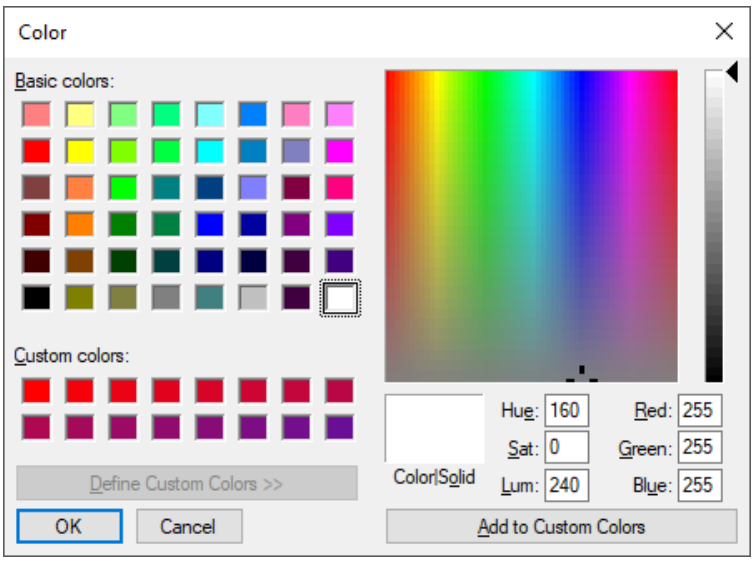
Color Picker Dialog

Color pickers are used to ask a user to select a color. These aren't seen in many places but when a color needs to be selected, this is a great option as it contains everything needed.

from tkinter import colorchooser
 # After creating parent...
color = colorchooser.askcolor(initialcolor=(255, 255, 255), parent=parent) # Returns a tuple

Method	OK	Cancel	Dialog Closed (X)
filedialog.askdirectory	((Red Int, Green Int, Blue Int), 'Hex Value')	(None, None)	(None, None)

As an example of the output, if I select cyan, I will get ((0.0, 255.99609375, 255.99609375), '#00ffff'). The numbers are large as they are floats but this means 0 red, 256 green and 256 blue. We can also see that the second value in the tuple is the hex value for cyan.



Event handling:

The Application object is always anticipating events as it runs an event listening loop. Each type of GUI widget is capable of identifying certain type of user interaction called event. User's actions such as mouse button click or double click, text typed in entry box, a widget comes in or goes out of focus etc. creates an event object. This event is notified to the application object which maps it to a user-defined event handler function.

Event is identified by its type, modifier and qualifier in the string format as <modifier-type-qualifier>

Here is a list of different tkinter events:

Event	modifier	type	qualifier	action
<Button-1>		Button	1	Mouse left button click.
<Button-2>		Button	2	Mouse middle button click.
<Destroy>		Destroy		Window is being destroyed
<Double-Button-1>	Double	Button	1	Double-click mouse button 1.
<Enter>	Enter			Cursor enters window/widget
<Expose>		Expose		Window fully or partially exposed.
<KeyPress-a>		KeyPress	a	Any key pressed.
<KeyRelease>		KeyRelease		Any key released.
<Leave>		Leave		Cursor leaves window.
<Print>			Print	PRINT key has been pressed
<FocusIn>		FocusIn		Widget gains focus
<FocusOut>		FocusOut		widget loses focus

Any event should be registered with one or more GUI widgets for it to be processed. Otherwise, it will be ignored. In tkinter, there are two ways to register event with a widget.

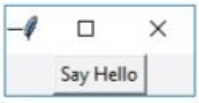
bind() function:

This function associates an event to an event handler function.

Widget.bind(event, handler)

Example

```
from tkinter import *
def hello(event):
  print("Hello World")
top=Tk()
B = Button(top, text='Say Hello')
B.bind('<Button-1>', hello)
B.pack()
top.mainloop()
```



The handler function event object as the argument and is bound with button object. The function gets called when left button of mouse (identified as <Button-1>) is clicked.

Widget Attributes

Colors

Tkinter represents colors with strings. There are two general ways to specify colors in Tkinter –

- You can use a string specifying the proportion of red, green and blue in hexadecimal digits. For example, "#fff" is white, "#000000" is black, "#000fff000" is pure green, and "#00ffff" is pure cyan (green plus blue).
- You can also use any locally defined standard color name. The colors "white", "black", "red", "green", "blue", "cyan", "yellow", and "magenta" will always be available.

Options

The common color options are

- **activebackground** – Background color for the widget when the widget is active.
- **activeforeground** – Foreground color for the widget when the widget is active.
- **background** – Background color for the widget. This can also be represented as *bg*.
- **disabledforeground** – Foreground color for the widget when the widget is disabled.
- **foreground** – Foreground color for the widget. This can also be represented as *fg*.
- **highlightbackground** – Background color of the highlight region when the widget has focus.
- **highlightcolor** – Foreground color of the highlight region when the widget has focus.
- **selectbackground** – Background color for the selected items of the widget.
- **selectforeground** – Foreground color for the selected items of the widget.

Dimensions

Various lengths, widths, and other dimensions of widgets can be described in many different units.

- If you set a dimension to an integer, it is assumed to be in pixels.
- You can specify units by setting a dimension to a string containing a number followed by.

Sr.No.	Character	Description
1	**C**	Centimeters
2	**I**	Inches
3	**M**	Millimeters
4	**P**	Printer's points (about 1/72")

Length options

Tkinter expresses a length as an integer number of pixels. Here is the list of common length options

- **borderwidth** – Width of the border which gives a three-dimensional look to the widget.
- **highlightthickness** – Width of the highlight rectangle when the widget has focus .

- **padX padY** – Extra space the widget requests from its layout manager beyond the minimum the widget needs to display its contents in the x and y directions.
- **selectborderwidth** – Width of the three-dimentional border around selected items of the widget.
- **wraplength** – Maximum line length for widgets that perform word wrapping.
- **height** – Desired height of the widget; must be greater than or equal to 1.
- **underline** – Index of the character to underline in the widget's text (0 is the first character, 1 the second one, and so on).
- **width** – Desired width of the widget.

Fonts

There may be up to three ways to specify type style.

Simple Tuple Fonts

As a tuple whose first element is the font family, followed by a size in points, optionally followed by a string containing one or more of the style modifiers bold, italic, underline and overstrike.

Example

- ("Helvetica", "16") for a 16-point Helvetica regular.
- ("Times", "24", "bold italic") for a 24-point Times bold italic.

Font object Fonts

You can create a "font object" by importing the tkFont module and using its Font class constructor

import tkFont
font = tkFont.Font (option, ...)

Here is the list of options –

- **family** – The font family name as a string.
- **size** – The font height as an integer in points. To get a font n pixels high, use -n.
- **weight** – "bold" for boldface, "normal" for regular weight.
- **slant** – "italic" for italic, "roman" for unslanted.
- **underline** – 1 for underlined text, 0 for normal.
- **overstrike** – 1 for overstruck text, 0 for normal.

Example

helv36 = tkFont.Font(family = "Helvetica",size = 36,weight = "bold")

X Window Fonts

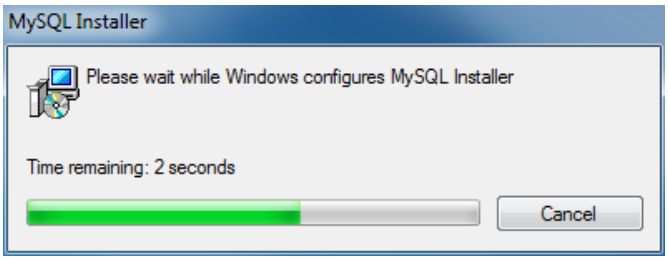
If you are running under the X Window System, you can use any of the X font names.

For example, the font named "-*-lucidatypewriter-medium-r-*-*-*-140-*-*-*-*-*-*" is the author's favorite fixed-width font for onscreen use. Use the *xfontsel* program to help you select pleasing fonts.

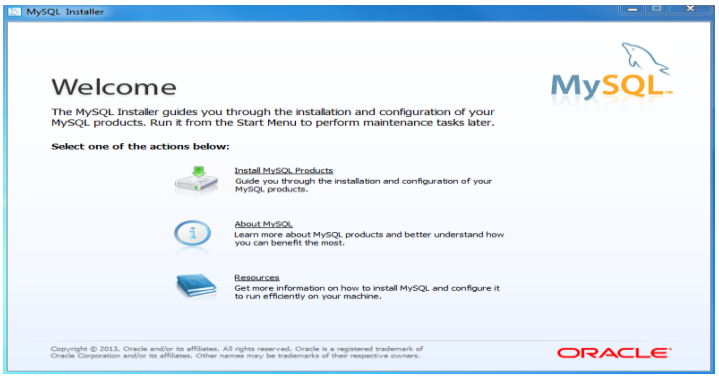
Unit -5 Connecting with Database

Install MySQL in Windows Operating System

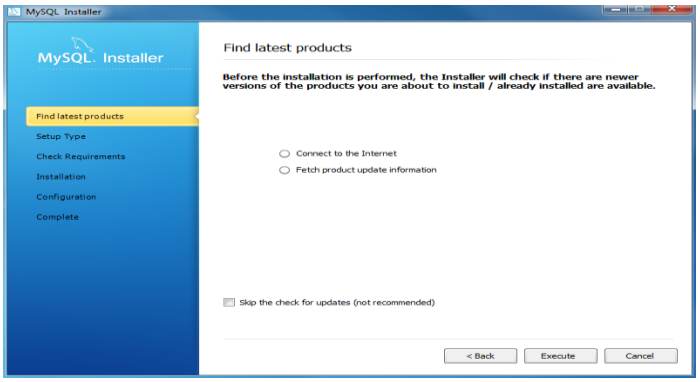
To install MySQL using the MySQL installer, double-click on the MySQL installer file and follow the steps below:



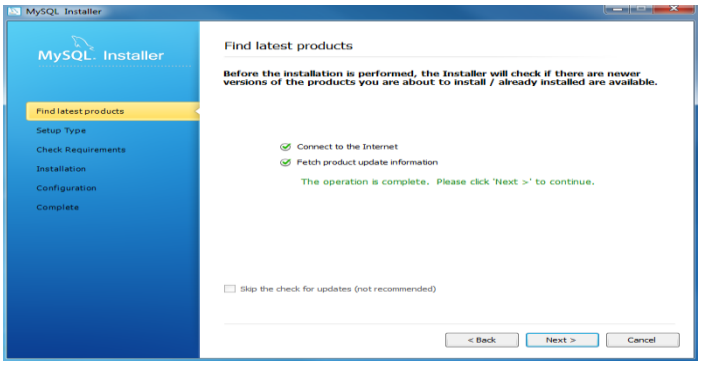
Step 1: Windows configures MySQL Installer



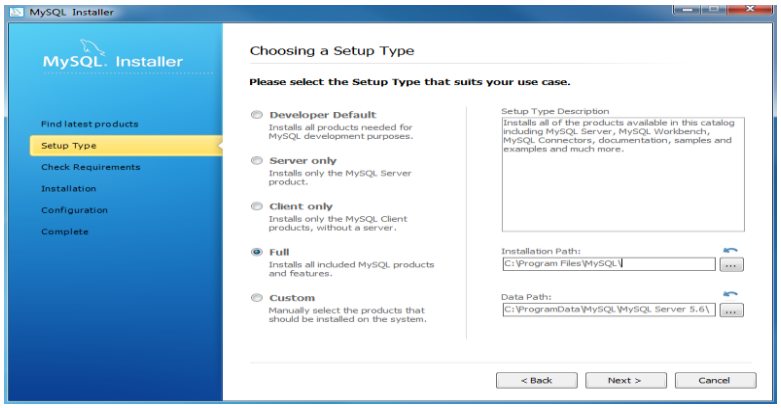
Step 2 – Welcome Screen: A welcome screen provides several options. Choose the first option: Install MySQL Products



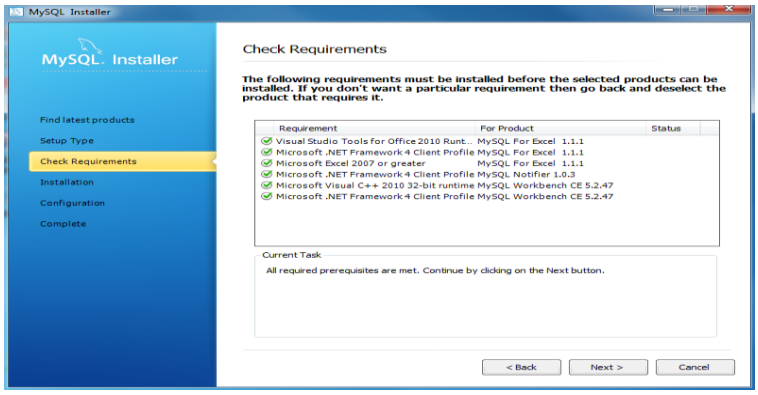
Step 3 – Download the latest MySQL products: MySQL installer checks and downloads the latest MySQL products including MySQL server, MySQL Workbench, etc.



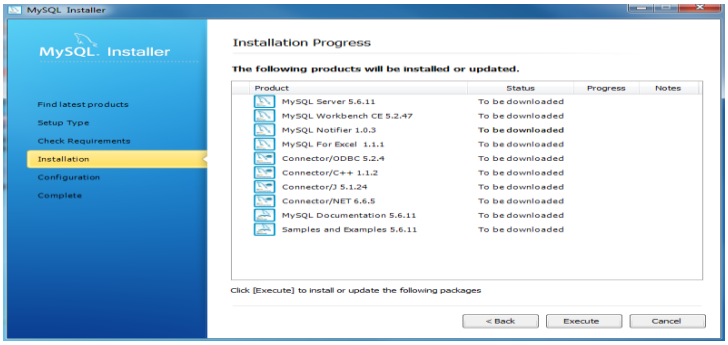
Step 4: Click the Next button to continue



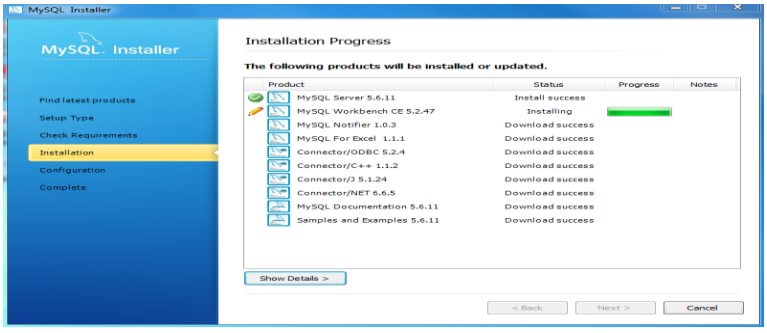
Step 5 – Choosing a Setup Type: there are several setup types available. Choose the Full option to install all MySQL products and features.



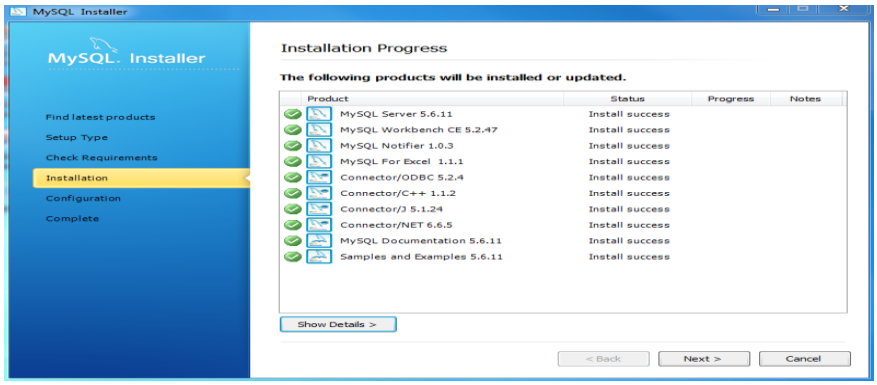
Step 6 – Checking Requirements



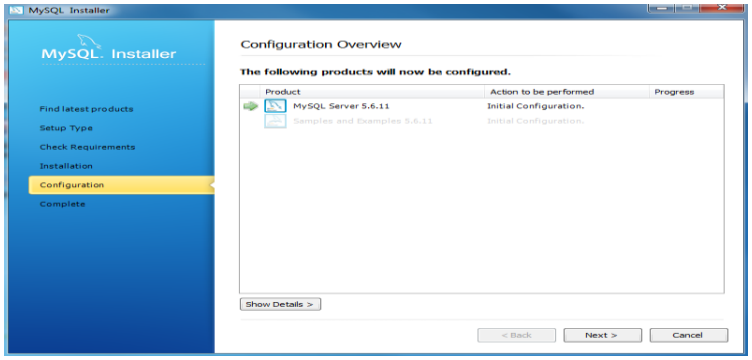
Step 7 – Installation Progress: MySQL Installer downloads all selected products. It will take a while, depending on which products you selected and the speed of your internet connection.



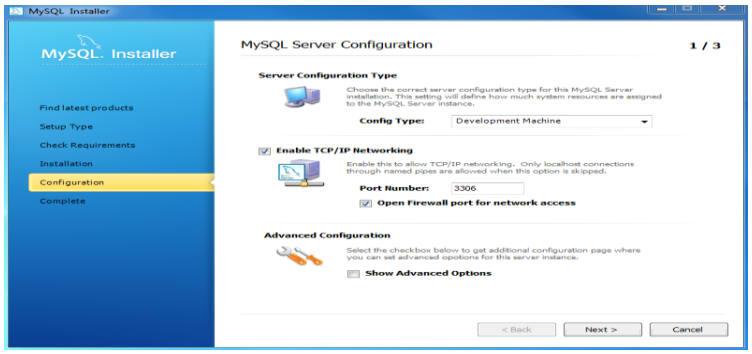
Step 7 – Installation Progress: downloading Products in progress.



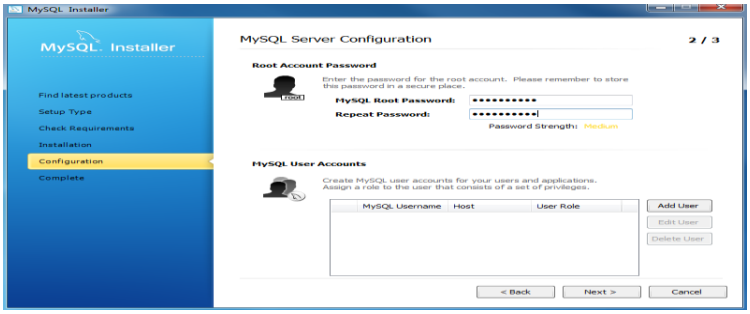
tep 7 – Installation Progress: Complete Downloading. Click the **Next** button to continue…



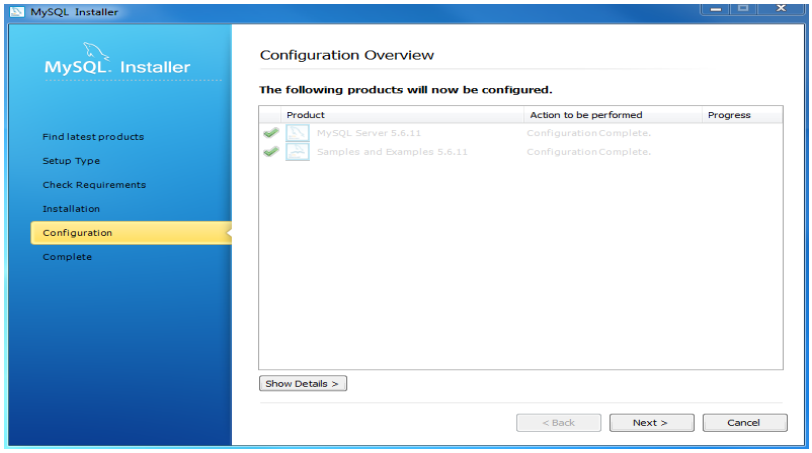
Step 8 – Configuration Overview. Click the Next button to configure MySQL Database Server



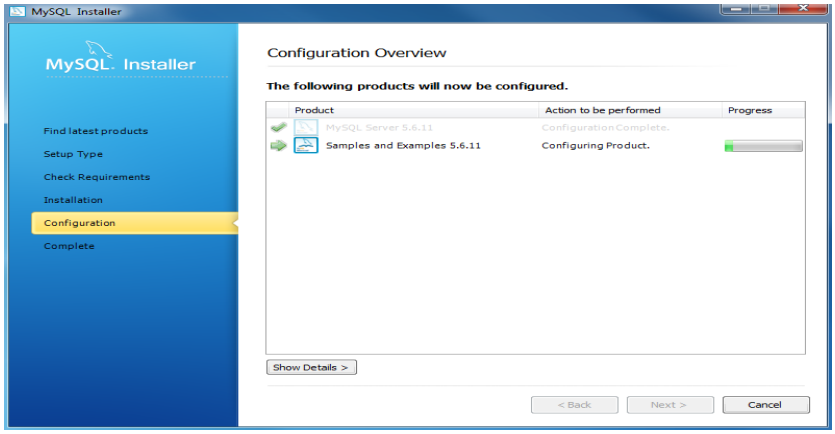
Step 8.1 – MySQL Server Configuration: choose Config Type and MySQL port (3006 by default) and click Next button to continue.



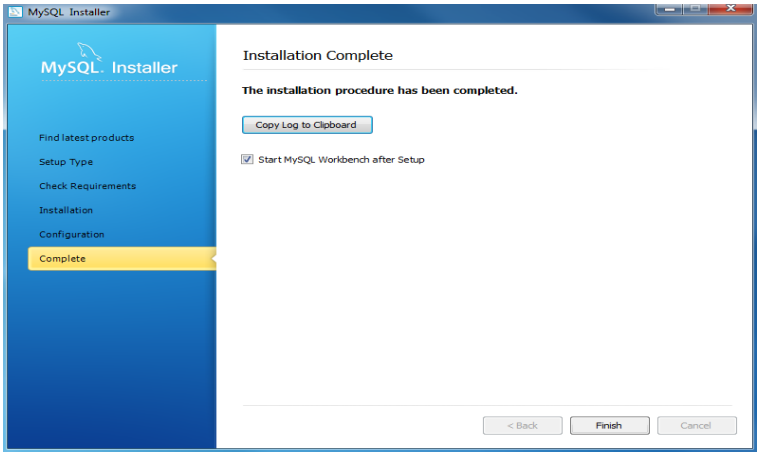
Step 8.1 – MySQL Server Configuration – In Progress: MySQL Installer is configuring MySQL database server. Wait until it is done and click the Next button to continue.



Step 8.1 – MySQL Server Configuration – Done. Click the Next button to continue.



Step 8.2 – Configuration Overview: MySQL Installer installs sample databases and sample models.



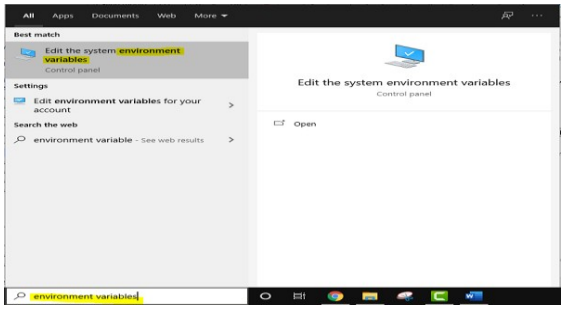
Step 9 – Installation Completes: the installation completes. Click the **Finish** button to close the installation wizard and launch the MySQL Workbench.

Setting the path to MYSQL server

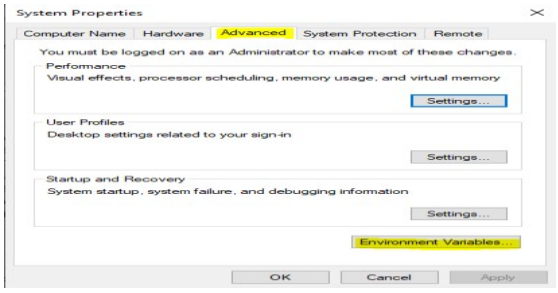
Step1 – Locate the mysql.exe file. We found in the following location –

C:\Program Files\MySQL\MySQL Server 5.6\bin

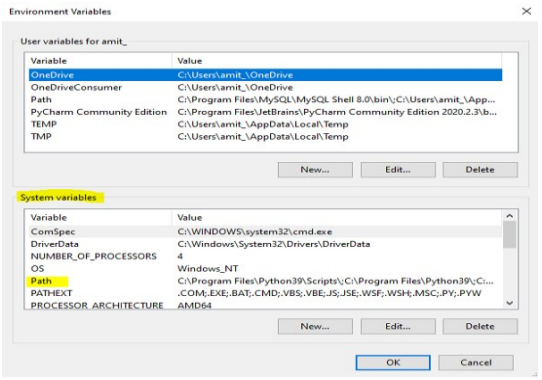
Step 2 – Press Start and type "Environment Variables". Click



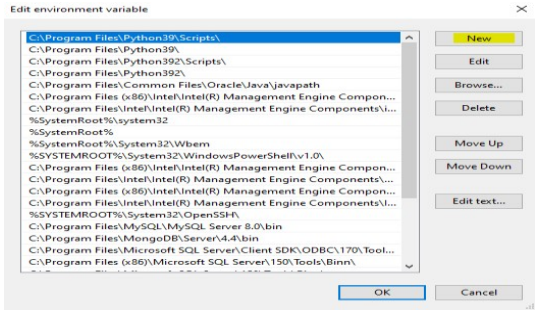
Step 3 – Under 'Advanced', click on 'Environment Variables'



Step 4 – Locate the 'System variables' and double-click on "Path":



Step 5 – Click on "New" –
Add the same path and click OK –
C:\Program Files\MySQL\MySQL Server 5.6\bin



The new PATH value should be available to any new command shell the user opens now. This will allow the user to invoke any MySQL executable program by typing its name at the DOS prompt from any directory on the system.

Installing MySQL Connector

Before installing MySQL Connector Python, you need Root or Administrator privileges in order to perform the installation. Python installed on your system

Installation

Python needs a MySQL driver to access the MySQL database. The MySQL Python Connector is available on **pypi.org**, therefore, you can install it using the pip command. The pip command allows you to install MySQL Python connector on any Operating system including Windows. Navigate your command line to the location of PIP, and type the following. Download and install "MySQL Connector": (default root of python)

C:\Users\Your Name\AppData\Local\Programs\Python\Python37\Scripts>
python -m pip install mysql-connector-python

Now you have downloaded and installed a MySQL driver.

Verifying MySQL Connector/Python installation

After installing the MySQL Python connector, we can see a new module by the name MySQL is added to the existing modules in python library. Go to python command line or IDEL Shell and type

>>> **help ("modules")**

As a result, it will display all available modules of python. We should be able to locate MySQL module among them. This represents that the MySQL connection for python has been successfully installed.

Now, you need to test it to make sure that it is working correctly and you are able to connect to the MySQL database server without any issues. To verify the installation, you use the following steps: Open Python command line or IDEL Shell and write the following code

>>> import mysql.connector

>>>mysql.connector.connect(host='localhost',database='python',user='root',password='')

If you see the following output, it means that you have been successfully installing the MySQL Connector/Python on your system.

<mysql.connector.connection.MySQLConnection object at 0x0187AE50>

Working with MySQL Database

MySQL is the world's most popular open-source database. Despite its powerful features, MySQL is simple to set up and easy to use. Once your MySQL server is up and running, you can connect to it as the super user **root** with the MySQL client.

On Windows, click **Start -> All Programs -> MySQL -> MySQL 5.6 Command Line** Client. If you did not install MySQL with the MySQL Installer, open a command prompt, go to the bin folder under the base directory of your MySQL installation, and issue the following command:

C:\> mysql -u root -p

You are then asked for the root password, which was assigned in different manners according to the way you installed MySQL.

Once you are connected to the MySQL server, a welcome message is displayed and the mysql> prompt appears, which looks like this:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4 Server version: 5.6.0 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

At this point, if you have logged in using a temporary root password that was generated during the installation or initialization process, change your root password by typing the following statement at the prompt:

mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'new_password';

Until you change your root password, you will not be able to exercise any of the super user privileges, even if you are logged in as root.

Here are a few useful things to remember when using the MySQL client:

- Client Commands (for example- help, quit, and clear) and keywords in SQL statements (for example,-SELECT, CREATE TABLE, and INSERT) are not case-sensitive.
- Column names are case-sensitive. Table names are case-sensitive on most, but not case-sensitive on Windows platforms. Case-sensitivity during string comparison depends on the character collation you use. In general, it is a good idea to treat all identifiers (database names, table names, column names, etc.) and strings as case-sensitive. See Identifier Case Sensitivity and Case Sensitivity in String Searches for details.
- You can type your SQL statements on multiple lines by pressing **Enter** in the middle of it. Typing a **semicolon (;)** followed by an Enter ends an SQL statement and sends it to the server for execution; the same happens when a statement is ended with \g or \G (with the latter, returned results are displayed vertically). However, client commands (for example, help, quit, and clear) do not require a terminator.
- To disconnect from the MySQL server, type QUIT or \q at the client:

  mysql> QUIT

Some Basic Operations with MySQL

Showing existing databases:- Use a SHOW DATABASES statement:

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.00 sec)
```

Creating a new database:- Use a CREATE DATABASE statement:

```
mysql> CREATE DATABASE PYTHON;
Query OK, 1 row affected (0.01 sec)
```

Check if the database has been created:

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| PYTHON             |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

Creating a table inside a database. First, pick the database in which you want to create the table with a USE statement:

mysql> USE PYTHON;

Database changed

The USE statement tells MySQL to use python as the default database for subsequent statements. Next, create a table with a CREATE TABLE statement:

```
CREATE TABLE emp
(
        id     INT unsigned NOT NULL AUTO_INCREMENT,   # Unique ID for the record
        name   VARCHAR(30) NOT NULL,                   # Name of the emp
        address      VARCHAR(50) NOT NULL,             # address of the emp
        birth   DATE NOT NULL,                         # Birthday of the emp
        PRIMARY KEY     (id)                          # Make the id the primary key
);
```

Data types you can use in each column are explained in Data Types. Primary Key Optimization explains the concept of a primary key. What follows # on each line is a comment, which is ignored by the MySQL client. Check if the table has been created with a SHOW TABLES statement:

```
mysql> SHOW TABLES;
| Tables_in_python |
+----------------+
| emp          |
+----------------+
```

1 row in set (0.00 sec)

DESCRIBE shows information on all columns of a table:

```
mysql> DESCRIBE emp;
| Field | Type            | Null | Key | Default | Extra          |
+-------+-----------------+------+-----+---------+----------------+
| id    | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| name  | varchar(30)     | NO   |     | NULL    |                |
|address | varchar(50)     | NO   |     | NULL    |                |
| birth | date            | NO   |     | NULL    |                |
+-------+-----------------+------+-----+---------+----------------+
```

4 rows in set (0.00 sec)

Adding records into a table. Use, for example, an INSERT...VALUES statement:

```
INSERT INTO emp ( name, address, birth) VALUES
  ( 'jay', 'limda chock, Rajkot', '2015-01-03' ),
  ( 'ajay', 'Suchak road, Rajkot', '2013-11-13' ),
  ( 'sanjay', ' shashtri medan same, Rajkot', '2016-05-21' );
```

See Literal Values for how to write string, date, and other kinds of literals in MySQL.

Retrieving records from a table: - Use a SELECT statement, and "*" to match all columns:

```
mysql> SELECT * FROM emp;
| id | name    | address | birth      |
+----+---------+--------+------------+
|  1 | jay   | limda chock, Rajkot | 2015-01-03 |
|  2 | ajay  | Suchak road, Rajkot | 2013-11-13 |
|  3 | sanjay | shashtri medan same, Rajkot | 2016-05-21 |
```

3 rows in set (0.00 sec)

MySQL Database Connection using in Python

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

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

Creating the connection

To create a connection between the MySQL database and the python application, the connect() method of **mysql.connector** module is used. Pass the database details like HostName, username, and the database password in the method call. The method returns the connection object.

syntax

Connection-Object= mysql.connector.connect(host = <host-name> ,
user = <username>, passwd = <password>)

Example

```
import mysql.connector
#Create the connection object
myconn = mysql.connector.connect(host="localhost", user = "root",passwd = "")
#printing the connection object
print(myconn)
```

Output:

<mysql.connector.connection.MySQLConnection object at 0x7fb142edd780>

Here, we must notice that we can specify the database name in the connect() method if we want to connect to a specific database.

Example

```
import mysql.connector
 #Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "", databa
se = "python")
 #printing the connection object
print(myconn)
```

Output:

<mysql.connector.connection.MySQLConnection object at 0x7ff64aa3d7b8>

Creating a cursor object

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

syntax

<my_cur> = conn.cursor()

Example

```
import mysql.connector
#Create the connection object
```

```
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "", databa
se = "python")
#printing the connection object
print(myconn)
#creating the cursor object
cur = myconn.cursor()
print(cur)
```

Output:

<mysql.connector.connection.MySQLConnection object at 0x7faa17a15748>

Retrieving All Rows From a Table

The SELECT statement is used to read the values from the databases. We can restrict the output of a select query by using various clause in SQL like where, limit, etc. Python provides the fetchall() method returns the data stored inside the table in the form of rows. We can iterate the result to get the individual rows. we will extract the data from the database by using the python script. We will also format the output to print it on the console.

```
import mysql.connector
# connect to mysql database
conn = mysql.connector.connect(host='localhost',database='', user ='root',password='')

if conn.is_connected():                          #check if conncted ot not
   print("My Sql Is Connected")
cursor.execuet(" select * form emp")             # prepare a cursor using cursor() method
rows = cursor.fetchall()                         # get all row
print("total number of rows =", cursor.rowcount)      # display the number of row
while row is rows:                                    # if the row exits
   print(row)
cursor.close()
conn.close()                                          # close connection
```

fetchone() method

The fetchone() method is used to fetch only one row from the table. The fetchone() method returns the next row of the result-set.

```
# connect to mysql database
conn = mysql.connector.connect(host='localhost',database='', user
='root',password='')

if conn.is_connected():                          #check if conncted ot not
   print("My Sql Is Connected")

cursor.execuet(" select * form emp")     # prepare a cursor using cursor() method
row = cursor.fetchone()                  # get first row from the cursor object
 print(row)
cursor.close()
conn.close()                                  # close connection
```

Inserting Rows into a Table

To insert new rows into a MySQL table, you follow these steps:

- Connect to the MySQL database server by creating a new conn object.
- Initiate a Cursor object from the conn object.
- Execute the INSERT statement to insert data into the table.
- Close the database connection.

MySQL Connector/Python provides API that allows you to insert one or multiple rows into a table at a time.

```
import mysql.connector
# connect to mysql database
conn = mysql.connector.connect(host='localhost',database='python', user
='root',password='')
#check if conncted ot not
if conn.is_connected():
   print("My Sql Is Connected")
# prepare a cursor object using cursor method
cursor =conn.cursor()
#prepare SQL query string to insert a row
str = " insert into emp(eno.ename,sal) values (9999,'abc',99.99)"
try:
   # execute the SQL query using execute () method
   cursor.execute(str)

   # save the change to the database
   conn.commit()
   print("1 row inserted...")
except:
   # rollback if there is any error
   conn.rollback()
#close connection
cursor.close()
conn.close()
```

- First, import MySQL Connector package.
- Next, to check connection, than after construct an INSERT statement with data for inserting into the EMP table.
- Then next, execute the statement, and commit or rollback the change in the try except block. That you have to explicitly call the commit () or rollback () method in order to make the changes to the database. In case a new row is inserted successfully.
- After that, close the cursor and database connection at the end.
- Finally, to insert a new row into the emp table.

Deleting Rows from the table

DELETE operation is required when you want to delete some records from your database. To delete the records in a table

- import mysql.connector package.
- Create a connection object using the mysql.connector.connect() method, by passing the user name, password, host (optional default: localhost) and, database (optional) as parameters to it.
- Create a cursor object by invoking the cursor() method on the connection object created above.
- Then, execute the DELETE statement by passing it as a parameter to the execute() method.

```
import mysql.connector
def delete_rows(eno):
   # connect to mysql database
   conn = mysql.connector.connect(host='localhost',database='python', user
='root',password='')

   #check if conncted ot not
   if conn.is_connected():
      print("My Sql Is Connected")

   # prepare a cursor using cursor() method
   cursor = conn.cursor()

   # prepare sql statement
   str = "delete from emp where empno= %d"

   # define the argument
   args = (eno)

   try:
      # execute the SQL statement
      cursor.execute(str%args)
      # save the changes
      conn.commit()
      print("1 row Deleted")

   except :
      # rollback if there is any error
      conn.rollback()

   finally:
      # close connection
      cursor.close()
      conn.close()

# enter employee number whose row is to be deleted
x = int(input("Enter Emp Number for delete record"))

# pass empno to delete_rows() function
delete_rows(x)
```

Updating rows in a Table

UPDATE Operation on any database implies modifying the values of one or more records of a table, which are already available in the database. You can update the values of existing records in Mysql using the UPDATE statement. To update specific rows, you need to use the WHERE clause along with it.

- Import mysql.connector package.
- Create a connection object using the mysql.connector.connect() method, by passing the user name, password, host (optional default: localhost) and, database (optional) as parameters to it.
- Create a cursor object by invoking the cursor() method on the connection object created above.
- Then, execute() method on the cursor object, by passing an UPDATE statement as a parameter to it.

```
import mysql.connector
def update_rows(eno):
    # connect to mysql database
   conn = mysql.connector.connect(host='localhost',database='python', user
='root',password='')

   #check if conncted ot not
   if conn.is_connected():
      print("My Sql Is Connected")

   # prepare a cursor using cursor() method
   cursor = conn.cursor()

    # prepare sql statement
   str = "update emp1 set sal = sal+1000 where empno=%d"

   # define the argument
   args = (eno)
   try:
      # execute the SQL statement
      cursor.execute(str%args)
      # save the changes
      conn.commit()
      print("1 row Deleted")

   except :
      # rollback if there is any error
      conn.rollback()

   finally:
      # close connection
      cursor.close()
      conn.close()
```

```
# enter employee number whose row is to be deleted
x = int(input("Enter Emp Number for update record"))

# pass empno to delete_rows() function
update_rows(x)
```

Creating Database Tables through Python

Once a database connection is established, you can create tables by passing the CREATE TABLE query to the execute() method.

In short, to create a table

- Import mysql.connector package.
- Create a connection object using the mysql.connector.connect() method, by passing the user name, password, host (optional default: localhost) and, database (optional) as parameters to it.
- Create a cursor object by invoking the cursor() method on the connection object created above.
- Then, execute the CREATE TABLE statement by passing it as a parameter to the execute() method.

```
# write a python program to create table in MySQL database
import mysql.connector

# connect to mysql database
conn = mysql.connector.connect(host='localhost',database='python', user ='root',password='')

#check if conncted ot not
if conn.is_connected():
   print("My Sql Is Connected")

# prepare a cursor using cursor() method
cursor = conn.cursor()

# drop table if already exists
cursor.execute("drop table if exists emp1")

# prepare sql statement
str = "create table emp1(empno int, ename char(3), salary float)"

# execute the query to create table
cursor.execute(str)
print("Table Creted")

# close connection
cursor.close()
conn.close()
```