Chapter 1 : File Handling

Apart from quick console input and Output using input() and print(). Data can also flow from and into a file.

To do that we have read() and write() function. The data flow is considered in terms of system not the user so read() behaves like print() and write() behaves like input()

Additionally we have to do two more step before data manipulation.

1. Open a file

Syntax:
file_object=open("//file_path//filename.extension","mode")

Note:

1. file_object is a new variable name

2. open() function opens the file

3. 1st parameter is file name. If the file is in the same folder as the filename.py file manipulating it path is not needed else the absolute file path is needed.

4. An extra / is used to make special character / normal.

5. Filename must be written with extension

6. Windows uses both backslash('\') and forward slash('/') for file path but other OS like Mac Osx or Linux or Unix uses only forward slash('/'). So it is advised to used forward slash ('/') in file path.

7. Mode is the permission with which file will be opened. It is also enclosed in quotes.

2. Close a file

Syntax: file_object.close()

There are 6 different **modes to open a file**:

1) **r** : it stands for read only mode. It is the default mode if none is specified. The default cursor location inside file is at the beginning in read mode.

2) **w** : it stands for write only. It basically empties the file contents and then writes new content. It also creates a file if none exists. The default cursor location inside file is at the beginning in write mode.

3) **a** : it stand for append. It writes new content at the end of the existing file content. The default cursor location inside file is at the end in append mode

4) **r+** : it stands for read-write mode without new file creation.

5) **w+** : it stands for read-write mode with file creation if none exists.

5) **a+** : it stands for read-append mode.

Note:

On windows OS we also have rb, rb+, wb, wb+, ab, ab+ to manipulate the binary file. Windows treats some file as binary depending on the End of File (EOF)

Reading a file:

Syntax:file_object=open("//path//filename.extension","r or r+") variable =

```
file_object.read()
print(variable)
file_object.close()
```

Example

```
fo = open("example.txt","r") #fo =
open("example.txt") content = fo.read()
print(content)
fo.close()
```

Writing a file:

Syntax:file_object=open("//path//filename.extension","w or w+")

```
file_object.write("new content") file_object.close()
```

Example

```
fo = open("example.txt","w") fo.write("new content") fo.close()
```

Note:

1. w or w+ creates the file called if it doesnot exist 2.write is interpreted successfully only when the file is closed.

Appending to a file:

Syntax:file_object=open("//path//filename.extension","a or a+")

file_object.write("new content") file_object.close()

Example

fo = open("example.txt","a")

fo.write("new content") fo.close()

Function on file:

i) closed

Syntax: file_object.closed

Returns *True* if the file is closed *False* otherwise

Example

>>>print(fo.closed)

True

ii) name

Syntax: file_object.name

Returns the name of the file

Example

>>print(fo.name)

example.txt

iii) mode

Syntax: file_object.mode

Returns the mode in which file is opened

Example

```
>>>print(fo.mode)
r
```

OS module functions for directory manipulation:

Function for directory manipulation. We have import os before using them

os.mkdir() : creates a folder with given name

os.chdir() : switch to a folder with given name

os.getcwd() : print the current folder

os.rmdir() : removes the folder if empty

More at

```
>>>import os
>>>dir(os)
```

Creating a file with user defined name

A file is created if it does not exist and is opened in w or w+ mode. But we can also create a file with filename chosen by user like this:

```
filename = input()+".txt"
fo = open(filename,'w')
fo.close()
```

File Methods

1) **File.fileno()** – Returns an integer number (file descriptor) of the file.
2) **File.seek(offset,from=SEEK_SET)** – Change the file position to offset bytes, in reference to from (start,current,end).
3) **File.tell()** – Returns the current file location.
4) **File.readline()** – The method readlines() reads one entire line from the file. A trailing newline character is kept in the string. An empty string is returned only when EOF is encountered immediately.
5) **File.truncate([size])** – The method truncate() truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.
6) **File.readlines([sizeint])** – The method readlines() reads until EOF using readline() and returns a list containing the lines. Sizeint is an optional argument and if it is present, instead of reading up to EOF, whole line approximate to sizeint bytes are read. An empty string is returned if EOF is encountered immediately.

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

Directories in Python

All files will be saved in various directories, and python has efficient methods for handling directories and files. The os module has several methods that help to create, remove and change directories.

1) **mkdir() method** – The mkdir() method of the os module is used to create directories in the current directory. We need to supply an argument to this method which contains the name of the directory to be created.
2) **chdir() method** – To change the current directory, we can use the chdir() method. The chdir() method takes an argument, which is the name of the directory that we want to make the current directory.
3) **getcwd() method** – The getcwd() method displays the current working directory.
4) **rmdir() method** – The rmdir() method deletes the directory, which is passed as an argument in the method.

CSV File Handling in Python

CSV stands for Comma Separated Values. In python handling CSV files is quite easy. First we have to import csv module that provide all the necessary functions to handle CSV file.

Example

First create a CSV file "test.csv" which contains three columns by name 'A', 'B' and 'C' followed by two rows containing values.

test.csv

A,B,C

1,2,3

5,6,7

The following will reads it and display the contents of the file

import **csv**

```
ifile       = open('test.csv', "rb")
reader = csv.reader(ifile)
rownum = 0
for row in reader:
# Save header row.
    if rownum == 0:
        header = row
    else:
        colnum = 0
        for col in row:
            print '%-8s: %s' % (header[colnum], col)
            colnum += 1
        rownum += 1
ifile.close()
```

Output

```
A        : 1
B        : 2
C        : 4
A        : 5
B        : 6
C        : 7
```

Pickling

The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream and "unpickling" is the inverse operation, whereby a byte stream is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as "serialization", "marshaling," or "flattening", however, to avoid confusion, the terms used here are "pickling" and "unpickling".

Example

```
# Pickle or serialize means transform from obj to byte stream

# Pickled data is 50 times faster to process

# Picked data is like BYTECODE

# Unpickling is not safe


import pickle


pickle.dump("hello",open("pdata.p",'wb'))

# pickle.load(open('pdata.p', 'rb'))

# reader = pickle.load(open('save.p', 'rb'))


# p.close()
```

Chapter 2 : Exception Handling

An exception is an abnormal condition that is caused by a runtime error in the program. It disturbs the normal flow of the program. An example for an exception is division by 0. When a python script encounters a situation that it cannot deal with, it raises an exception. An exception is a python object that encounters an error.

If the exception object is not caught and handled properly, the interpreter will display an error message. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display appropriate message for taking corrective actions. This task is known as exception handling.

Built-in Exceptions

Python has a collection of built-in exceptions classes. If the runtime error belongs to any of the pre-defined built-in exception, it will throw the object of the appropriate exception. The following table gives a detailed explanation of built-in Exceptions and the situation in which they are invoked.

Exception Name	When it is raised
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EnvironmentError	Base class for all exceptions that occur outside the python environment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
Exception	Base class for all exceptions.
ImportError	Raised when import statement fails.
IndentationError	Raised when indentation is not specified properly.
IndexError	Raised when index is not found in a sequence.
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.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing ctrl+c.
KeyError	Raised when the specified key is not found in the dictionary.
LookupError	Base class for all lookup.

NameError	Raised when an identifier is not found in the local or global namespace.
NotImplemntedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.
OSError	Raised for operating system-related errors.
RuntimeError	Raised when a generated error does not fall into a category.
SyntaxError	Raised when there is an error in python Syntax
SystemError	Raised when the python interpreter finds an internal problem,
	but when this error is encountered the python interpreter does not exist.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
UnboundLocalError	Raised when trying to access a local variable in a function or a method but no value has been assigned to it.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid value specified.
ArithemeticError	Base class for all errors that occur for numeric calculation.
FloatingPointError	Raised when a floating point calculation Fails.
StopIteration	Raised when the next()method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except stopIteration and SystemExit.
ValueZero	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.

Handling Exceptions

Python uses a keyword try to prepare a block of code that is likely to cause an error and throw an exception. An except block is defined which catches the exception thrown by the try block and handles it. The try block can have one or more statements that could generate an exception. If anyone statement generates an exception, then the remaining

statements in the block are skipped and execution jumps to the except block that is placed next to the try block.

The except block can have more than one statement and if the except parameter matched with the type of the exception object, then the exception is caught and statements in the except block will be executed. Every try block should be followed by at least one except statement. **try.... except**

The following gives the Syntax of try....except statement.

Syntax

```
try:
    suite; except
Exception1:
    Exception1_suite
except Exception2:
    Exception2_suite else:
    else_suite
```

When an exception occurs inside a try block, it will go to the corresponding except block. If no exception is raised inside a try block then after the try block, the statements in the else block is executed. The else block can also be considered a place to put the code that does not raise any exception.

Example program 1

```
try:
    a = int(input("First number")) b =
int(input("Second number")) result= a/b
print("result=",result) except ZeroDivisionError:
    print("Division by Zero") else:
    print("successful Division")
```

Output

First number:10 second

number:0

Division by zero

In the above example, the second number is 0. Since division by zero is not possible, an Exception is thrown and the execution goes to the except block. All the rest of the statements are bypasses in the try block. Hence the Output is displayed as division by zero.

Example program 2

```
try:
a = int(input("First number")) b =
int(input("Second number")) result= a/b
print("result=",result) except
ZeroDivisionError:
print("Division by Zero") else:
print("successful Division")
```

Output

First number:20 second

number:10 Result= 2

successful Division

In the above example statement exception is not thrown and hence all the statements in the try block are executed. After executing try block, the control passes on to the else block. Hence the print statement in the else block is executed.

except clause with no exception

We can use except statement with no exceptions, this kind of try-except statement catches all the exceptions. Since it catches all the exceptions, it will not help the programmer to exactly identify what is the cause for the error occurred. Hence it is not

considered as a good programming practice. The following shows the Syntax for except clause with no exception.

Syntax

```
try:
    suite; except
Exception1:
    Exception1_suit else:
    else_suite
```

Example Program

```
try:
    a   = int(input("First number"))
    b   = int(input("Second number"))
result= a/b print("result=",result) except:
print("Error occurred") else:
    print("successful Division")
```

Output

```
Error occurred
```

except clause with multiple exceptions

This is used when we want to give multiple exceptions in one except statement. The following shows the Syntax of the except clause with multiple exceptions

Syntax try:

```
    suite; except(Exception1[, Exception2[,.....ExceptionN])):
    Exception1_suit else:
    else_suite
```

Example program try:

```
        a = int(input("First number")) b =
int(input("Second number")) result= a/b
print("result=",result) except
(ZeroDivisionError,TypeError):
        print("Error occurred")
else:
        print("successful Division")
```

Output

```
First number:10 second
number:0
Division by zero
```

try....finally

A finally block can be used with a try block. The code placed in the finally block is executed no matter exception is caused or caught. We cannot use except clause and finally clause together with a try block. It is also not possible to use else clause and finally together with a try block. The following gives the Syntax for a try....finally block

Syntax try: suite;

```
finally:
        finally_suite
```

Example program 1

```
try:
        a = int(input("First number")) b =
int(input("Second number")) result= a/b
print("result=",result) finally:
        print("executed always")
```

Output

```
First number:10 second

number:0

Division by zero occurred
Traceback (most recent call last):

  File "main.py", line 5, in ,module> result=a/b

      ZeroDivisionError: integer division or module by zero
```

In the above example, an error occurred in the try block. It caught by python's error handling mechanism. But the statement in the finally block is executed even though an error has occurred.

Example program 2

```
try:

    a = int(input("First number")) b =

int(input("Second number")) result= a/b

print("result=",result) finally:

    print("executed always")
```

Output

```
First number:10 second

number:5 result = 2

executed always
```

In the above program there was no exception thrown. Hence after the try block, the finally block is executed.

Raising an exception

We have discussed about raising built-in exception. It is also possible to define a new exception and raise it if needed. This is done using the raise statement. The following shows the Syntax of raise statement. An exception can be a string, a class or an object.

Most of the exceptions that the python raises are classes, with an argument that is an instance of the class. this is equivalent to throw clause in java.

Syntax

rasie [exception [,arguments [[, traceback]]]

Here, exception is the type of exception (for example, IOError) and arguments is a value for the exception argument. This argument is optional. The exception argument is None, if we do not supply any argument. The argument, traceback, is also optional. If this is used, then the traceback object is used for the exception. This is rarely used in programming.

In order to catch the exceptions defined using raise statement, we need to use the except clause which is already discussed, the following code shows how an exception define using the raise statement can be used.

Example program a= int(input(Enter the parameter vaule:")) try:

if a<0:

raise ValueError("Not a positive integer")

except ValueError as err:

print(err) else: print("Positive Integer=",

a)

Output enter the parameter value:-9

Not a positive Integer

User-defined Exception

Python also allows us to create our own exceptions by deriving classes from the standard built-in exceptions. In the try block, the user-defined exceptions is raised and caught in the except block. This is useful when we need to provide more specific information when an exception is caught. The following shows an example for user-defined exception

Example program class

Error(Exception):

```
        pass class
ValueTooSmallError(Error): pass class
ValueTooLargeError(Error):
        pass
number=10 while
True: try:
            i_num = int(input("enter a number:")) if
            i_num<number: raise ValueTooSmallError elif
            i_num>number:
                raise ValueTooLargeError break except
        ValueTooSmallError: print("This value is too small, try again!")
        except ValueTooLargeError:
            print("This value is too large, try again!")
    print("congratulations! you guessed it correctly.")
```

Output

```
Enter a number: 13
This value is too large, try again!
Enter a number: 9
This value is too small, try again! Enter the number: 10

congratulations! you guessed it correctly.
```

Assertions in Python

An assertion is a checking in python that can be turned on or off while testing a program. In assertion, an expression is tested and if the result is false, an exception is

raised. Assertions are done using the assert statement. The application of assertion is to check for a valid input or for a valid Output, The following shows the Syntax for asset statement.

When python interpreter encounters an assertion statement, it evaluates the accompanying expression. If the expression is evaluated to False, python raise an Assertion Error exception. AssertionError exceptions can be caught and handled like any other exceptions using try-except statement, but if not handled, they will terminate the program and produce a traceback.

Example program

```
def sum(a,b):
        sum=a+b assert(sum>0),"too low
value" return(sum) a= int(input("first
number") b= int(input("second
number") print(sum(a,b))
```

Output first number:-8

```
second number:-2
        Traceback (most recent call last):
  File "main.py", line 8, in <module> print(sum(a,b))
  File "main.py", line 4, in sum assert(sum>0),
        "two low value"
AssertionError: two low value
```

Chapter 3 : Object Oriented Programming

A class contains a collection of data (variables) and methods (functions) that act on those data. Class is considered as a blueprint for an object. For example consider the design of a house with details of window, doors, roofs and floors. We can build the house based on the descriptions. This can be considered as a class while the object is house itself. An object is said to be an instance of a class and the process of creating a class is called instantiation. The basic concepts related to OOP are as follows:

1. Classes

2. Objects

3. Encapsulation

4. Data hiding

5. Inheritance

6. Polymorphism

Advantages of Object Oriented Programming

Objects oriented programming has following advantages:

1) Simplicity

2) Modifiability

3) Extensibility and Maintainability

4) Re-usability

5) Security

Class Definition

The class definition in python begins with the keyword class. The first statement after class definition will be the docstring which gives brief description about the class. The following shows the Syntax for defining a class.

Syntax

class ClassName:

'Optional class documentation string' class_suite

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions. There are also special attributes in it that begins with double underscore (__). For example, __doc__ gives us the docstring of that class. As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

Example program class

Student:

"Common base class for all stuudents" studentcount = 0

def __init__(self, rollno, name, course):

self.rollno = rollno self.name = name

self.course = course

Student.studentcount += 1 def displaycount(self) print("Total students=", Student.studentcount) def displaystudent(self) print("Roll Number:",self.rollno) print("Name:",self.name)

print("Salary:",self.course)

In the above program, we have created a class called Student. The variables studentcount is a class variable whose value is shared among all instance of this class. This can be accessed from inside the class or outside the class by giving Student.studentcount.

The first method __init__() is a special method, which is called class constructor or initialization method that python calls when we create a new instance of this class.

Class methods have only one specific difference from ordinary functions - they must have an extra argument in the beginning of the parameter list. This particular argument is self which is used for referring to the instance. But we need not give any value for this parameter when we call the method. Python provides it automatically. Self

is not a reserved word in python but just a strong naming convention and it is always convenient to use conventional names as it makes the program more readable. So while defining our class methods, we must explicitly list self as the first argument for each method, including __init__.

It also implies that if we have a method which takes no arguments, then we still have to define the method to have a self-argument. Self is an instance identifier and is required so that the statements within the methods can have automatic access to the current instance attributes.

Creating Objects

An object is created by calling the class name with the arguments defined in the __init__ method. We can access a method by using the dot operator along with the object. Similarly a class name can be accessed by specifying the method name with the dot operator with the class.

Example program stud1 = Student(10, "Jack", "MS") stud2 = Student(20, "Jill", "BE") stud1.displayStudent() stud2.displayStudent() print("Total number of students:",Student.studentcount)

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

Roll number: 10

Name: Jack

Salary: MS

Roll number: 20

Name: Jill

Salary: BE

Total number of students: 2

Built-in Class Attributes

Python contains several built-in class attributes. These attributes are accessed using the dot operator. The following gives the list of built-in class attributes.

1. __dict__: This Attribute contains the dictionary containing the class's namespace.

2. _doc_: It describes the class or it contains the class documentation string. If undefined, it contains none.

3. _name_: This attribute contains the class name.

4. _module_: This contains the module name in which the class is defined. This attribute is "_main_" in interactive mode.

5. _base_: This contains an empty tuple containing the base classes, in the order of their occurrences in the base class list.

The following program shows the usage of built-in class attributes.

Example program class

Student:

"Common base class for all stuudents" def _init_(self,

rollno, name, course):

self.rollno = rollno self.name = name self.course = course

def displayStudent(self) print("Roll Number:",self.roolno)

print("Name:",self.name) print("Course:",self.course) stud1 =

Student(10, "Jack", "MS") stud1.displayStudent()

print("Student._doc_:",Student._doc_)

print("Student._name_:",Student._name_)

print("Student._module_:",Student._module_)

print("Student._base_:",Student._base_)

print("Student._dict_:",Student._dict_)

Output

Roll Number: 10

Name: Jack

Course: MS

Student.__doc__: Common base class for all students Student.__name__: Student

Student.__module__: __main__

Student.__base__: ()

Student.__dict__:{'displayStudent':<function displayStudent at 0x7efdcc047758>, '__module__':'__main__','__doc__':'Comman base class for all students', '__init__':<function __init__at 0x7edcc0476e0>}

Destructors in Python

Python automatically deletes an object that is no longer in use. This automatic destroying of objects is known as garbage collection. Python periodically performs the garbage collection to free the blocks of memory that are no longer in use. 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. The following program shows the destructors in python.

Example program class

```
Student:
"Common base class for all stuudents" def __init__(self,
rollno, name, course):
        self.rollno = rollno self.name = name
self.course = course def displayStudent(self)
print("Roll Number:",self.roolno)
print("Name:",self.name) print("Course:",self.course)
def __del__(self):
        class_name = self.__class__.__name__
print(class_name, "destroyed") stud1 = Student(10,
"Jack", "MS") stud1.displayStudent() del Stud1
```

Output

Roll Number: 10

Name: Jack

Course: MS

Student destroyed

Encapsulation

Encapsulation is the most basic concept of OOP. It is the combining of data and functions associated with that data in a single unit. In most of the languages including python, this unit is called a class. In simple terms we can say that encapsulation is implemented through classes. In fact the data members of a class can be accessed through its member functions only. It keeps the data safe from any external interference and misuse. The only way to access the data is through the functions of the class. In the example of the class Student, the class encapsulates the data (rollno, name, course) and the associated functions into a single independent unit.

Data Hiding

We can hide data in python. For this we need to prefix double underscore for an attribute. Data hiding can be defined as the mechanism of hiding the data of a class from outside world or to be precise, from other classes. This is done to protect the data from any accidental or intentional access. In most of the object oriented programming languages, encapsulation is implemented through classes. In a class, data may be made private or public. Private data or function of a class cannot be accessed from outside the class while public data or functions can be accessed from anywhere. So data hiding is achieved by making the members of the class private. Access to private members is restricted and is only available to the member functions of the same class. However the public part of the object is accessible outside the class. Once the attributes are prefixed with the double underscore, it will not be visible outside the class. The following program shows an example for data hiding in python

Example program class

HidingDemo:

"Program for hiding data"

```
        self.__num+=1 print("Number count=",
    self.__num) number = HidingDemo()
    number.numbercount() print(number.__num)
```

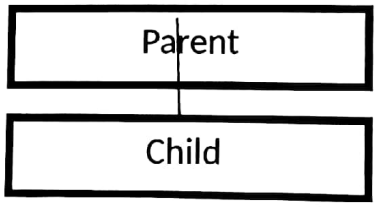
Output

```
    Number count= 1
    Traceback (most recent call last):
      File "main.py", line 9, in <module> print number.__num
    AttributeError: HidingDemo instance has no attribute '__num'
```

The Output shows that numbercount is displayed once. When it is attempted to call the variable number.__num from outside the function, it resulted in an error.

Inheritance

The mechanism of deriving a new class from an old class is known as inheritance. The old class is known as base class or super class or parent class. The new one is called the subclass or derived class or child class. Inheritance allows subclasses to inherit all the variables and methods of their parent class. The advantage of inheritance is the reusability of the code. Following figure illustrates the single inheritance. The attributed and methods of parent class will be available in child class after inheritance.



Deriving a child class

The following shows the Syntax for deriving a child class in python.

Syntax

```
    class subClassName(ParentClass1[,ParentClass2, .....]):
```

'Optional class documentation string' class_suite

The following program shows an example for single inheritance

Example program class

Student:

"Common base class for all stuudents" def __init__(self,

rollno, name, course):

self.rollno = rollno self.name = name

self.course = course def displayStudent(self)

print("Roll Number:",self.roolno)

print("Name:",self.name) print("Course:",self.course)

#Inheritance class

Test(Student):

def getMarks(self,marks):

self.marks=marks def

displayMarks(self):

print("Total marks:", self.marks) r =

int(input("Enter Roll Number:")) n = input("Enter

Name:") c = input("Enter Course Name:") m =

int(input("Enter Marks:")) #creating the object

print("Result") stu1=Test()

stud1.getData(r,n,c) stud1.getMarks(m)

stud1.displayStudent() stud1.displayMarks()

Output

Enter Roll Number:20 Enter

Name:Smith

Enter Course Name:MS

Enter Marks:200 Result

Roll Number: 20

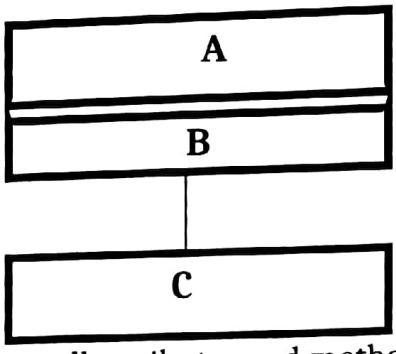
Name: Smith

Course: MS

Total Marks: 200

Multilevel Inheritance

We have already discussed the mechanism of deriving a new class from one parent class. We can further inherit the derived class to form a new child class. Inheritance which involves more than one parent class but at different levels is called multilevel inheritance. Following figure illustrates multilevel inheritance.

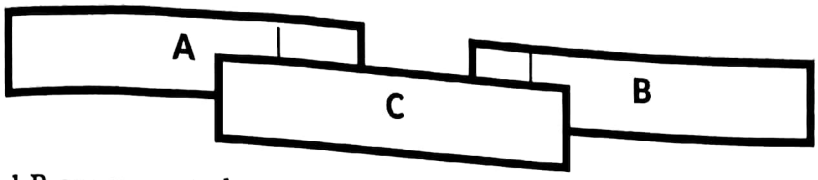


Here B is derived from A. Hence all attributes and methods that are available in A is now Available in B also. C is derived from B. Hence all methods and attributes of B and A is now available in C.

Multiple Inheritance

It is possible to inherit from more than one parent class. Such type of inheritance is called multiple inheritance. In this case all the attributes and methods of both the parent

class will be available in the child after inheritance. Following Figure illustrates multiple inheritance.



Here A and B are parent classes and C is the child class. The attributes and methods of both classes A and B are now available in C after inheritance.

Method Overriding

Polymorphism is an important characteristic of object oriented programming language. In object-oriented programming, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes. We can override methods in the parent class.

Method overriding is required when we want to define our own functionality in the child class. This is possible by defining a method in the child class that has the same name, same arguments and same return type as a method in the parent class. When this method is called, the method defined in the child is invoked and executed instead of the one in the parent class. The following shows an example program for method overriding

Example program class Parent: "Base class" def

init(self,name): self.name=name def

displayName(self): print("Name: ",self.name) def

del(self) class_name = self._class_._name_

print(class_name., "destroyed") class Child(Parent):

def _init_(self,name,address):

self.name=name

```
        self.address=address def

    displyName(self):

        print("Name: ",self.name)

    print("Address:",self.address) def __del__(self):

        class_name=self.__class__.__name__ print(class_name,"destroyed)

    #Main program n = input("Enter

    Name:") a = input("Enter Address:")

    obj = child(n,a) obj.displayName() del

    obj
```

Output

```
    Enter Name: Jill

    Enter Address: Klos

    Name: Jill

    Address: Klos

    Child destroyed
```

In the above program, even though there are two methods with the same name dispalyName() is available in child class, the method in the child class will be invoked. The method in the child class has overridden the method in the parent class.

Operator Overloading

Operator overloading is one of the important feature of object oriented programming. C++ and python supports operator overloading, while Java does not support operator overloading. The mechanism of giving special meanings to an operator is known as operator overloading. For example the operator '+' is used for adding two numbers. In python this operator can also be used for concatenating two strings. This'+' operator can be used for adding member variable of two different class. The following example shows how '+' operator can be overloaded.

Example program class

```
Abc:
        def __init__(self, a, b):
                self.a = a self.b = b
                def __str__(self): return "Abc (%d, %d)" % (self.a, self.b)
        def __add__(self,other):
                return Abc(self.a + other.a, self.b + other.b) a1 = Abc(2, 4) a2 =
Abc(5, -2) print(a1 + a2)
```

Output

Abc(7,2)

Practice Questions

1. Write a python class to convert an integer to roman numerals.
2. Write a python class to convert a roman numeral to an integer.
3. Write a program to find area and perimeter of a rectangle using classes and objects.
4. Define a class to represent a bank account. Include the following details like name of the depositor, account number, type of account, balance amount in the account. Write methods to assign initial values, to deposit an amount, withdraw an amount after checking the balance, to display name, account number, account type and balance.
5. Create a class called staff with code and name. Create classes Teacher(subject, publication),Typist(speed), Officer(grade). Using the typist class as base class, create 2 classes Regular(salary) and Casual(daily wages). Implement a menu driven program for the same.

Chapter 4 : Regular Expressions

Regular Expressions are essentially a tiny, highly specialized programming language embedded inside python and made available through the re module. A regular expression helps to match or find other strings or set of strings, using a specialized Syntax held in a pattern. They are widely used in UNIX world.

The re module raises the exception re.error if an error occurs while compiling or using a regular expression. There are various characters, which have a special meaning when they are used in regular expression. While dealing with regular expression, we use raw strings as r'expression'.

The match()Function

The purpose of match()function is to match regular expression pattern to a string with optional flags. For using this function, the re module needs to be imported. The following shows the Syntax of match() function.

Syntax

re.match(pattern, string, flags=0)

Where pattern is the regular expression to be matched, string is searched to match the pattern at the beginning of string, flags are modifiers specified using bitwise OR(|). The re.match() function returns a match object if the meaning was success and return None, if matching was a failure.

Two functions group (n) and groups () of match object are used to get matched expressions. The group (n) method returns entire match (or specific subgroup n) and groups () method returns all matching subgroups in a tuple or empty if there weren't any matches.

Example program import re line="python programming is fun"

matchobj = re.match(r'fun', line, re.M|re.I) if matchobj:

print("match-->matchobj.group():", matchobj.group()) else:

print("no match")

Output

no match

The match() function always checks the beginning of the string. Even though the keyword fun is there in the string it is not considered as a match since it does not appear at the beginning of the string. Consider the below example

Example program import re line="python programming is fun"

matchobj = re.match(r'python', line, re.M|re.I) if matchobj:

print("match-->matchobj.group():", matchobj.group()) else:

print("no match")

Output

match -->matchobj.group() : python

In the above example the word python is found as a match since it appears at the beginning of the string.

The search() function

The search () function searches for first occurrence of regular expression pattern within a string with optional flags. The following shows the Syntax for search () function.

Syntax re.search(pattern,string,flags=0)

Where pattern, string and flags have the same meaning as that of match() function. The re.search()function returns a match object if the meaning was success and return None, if matching was failure.

The difference between re.match() and search() is that match() checks for a match only at the beginning of the string, while search() checks for a match anywhere in the string. The following shows an example between match() and search() functions.

Example program

import re line="python programming is fun" matchobj =

re.match(r'fun', line, re.M|re.I) if matchobj:

print("match --> matchobj.group():", matchobj.group()) else:

```
print("no match") searchobj = re.search(r'fun', line, re.M|
re.I) if searchobj:
print("search-->searchobj.group():",searchobj.group(
)) else:
print("nothing found")
```

Output

```
no match search-->searchobj.group(): fun
```

In the above example, match() function returns nothing while search() function searches the entire string to find a match.

Search and Replace

Search and replace is done in python with the help of sub method in re module. The following shows the Syntax of sub method.

Syntax

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the regular expression pattern in string with repl, substituting all occurrences. If value for max is provided, it will replace only the number of occurrences specified in max. This method returns modified string. The following example shows the application of sub method.

Example program

```
import re zipcode="2004-995-559" num =
re.sub(r'\D', "", zipcode) print("zip code:",
num)
```

Output

```
zip code: 2004995559
```

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. We can provide multiple modifiers using exclusive OR (|), as shown in the previous examples. The following table shows the various modifiers and their description available in the re module.

Modifiers	Description
re.I	Performs case-sensitive matching.
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group(\w and \W), as well as word boundary behavior(\b and \B)
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line(not just start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.X	Permits "cuter" regular expression Syntax. It ignores whitespaces(except inside s set[] or when escaped by a backslash) and treats unescaped # as a comment marker.

Regular Expression Patterns

All characters matches themselves except for flow control characters (+ ? . * ^ $ () {} | \). We can escape a control character by preceding it with a backslash. Following table shows the regular patterns and their descriptions.

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[....]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets.
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrences of preceding expression.
re?	Matches 0 or 1 occurrences of preceding expression.
re{n}	Matches exactly n number of occurrences of preceding expression.
re{n,}	Matches n or more occurrences of preceding expression.
re{n,m}	Matches at least n and mast m occurrences of preceding expression.

a|b	Matches either a or b
(re)	Groups regular expression and remembers matched text.
(?imx)	Temporarily toggles on i, m, or x options within a regular expression =. If n parentheses, only that are is affected.
(?-imx)	Temporarily toggles off i, m, or x options within a regular expression =. If n parentheses, only that are is affected.
(?:re)	Group's regular expression without remembering matche dtext.
(?imx:re)	Temporarily toggles on i, m, or x options within parentheses.
(?-imx:re)	Temporarily toggles off i, m, or x options within parentheses.
(?#...)	Comment
(?= re)	Specifies position using a pattern. Doesn't have a range.
(?! re)	Specifies position using a pattern negation. Doesn't have a range.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word character.
\W	Matches non word character.
\s	Matches whitespaces. Equivalent to [\t\n\r\f].
\S	Matches nonwhite spaces.
\d	Matches digits. Equivalent to [0-9].
\D	Matches no digits.
\A	Matches beginning of the string.
\Z	Matches end of the string. If a newline exists, it matches just before newline.
\z	Matches end of the string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspaces (0x08) when inside brackets.
\B	Matches no word boundaries.
\n, \t, etc.	Matches newline, carriage returns, tabs, etc.
\1....\9	Matches nth grouped sub expression.
\10	Matches nth grouped sub expression. If it matched already. Otherwise refers to the octal representation of a character code.

Character Classes

The table shows example of various character classes and their description used in regular expressions of python

Example	Description
[Pp]ython	Match "Python" or "python"

rub[ye]	Match "ruby" or rube"
[aeiou]	Match any lowercase vowel
[0-9]	Match any digit.
[a-z]	Match any lowercase ASCII letters.
[A-Z]	Match any uppercase ASCII letters.
[a-zA-Z0-9]	Match any of the above.
[^aeiou]	Match anything other than a lowercase vowels
[^0-9]	Match anything other than digits.

Repetition Cases

Following Table shows the example of various repetition cases and their description used in regular expression of python.

Example	Description
ruby?	Match "rub" or "ruby": the y id optional
ruby*	Match "rub" plus 0 or more y's
ruby+	Match "rub" plus 1 or more y's
\d{3}	Match exactly 3 digits
\d{3,}	Match 3 0r more digits.
\d{3,5}	Matches 3,4, or 5 digits.

findall() method

The findall() method searches all patterns and returns a list. The following shows an example for findall() method.

Example program import re a = "hello

1234 i'am 234"

print(re.findall("\d",a)

Output

['1234', '234'] **compile() method**

The compile() method compiles a regular expression pattern into regular expression object, which can be used for matching using its match() and search() methods . The Syntax for compile() is as follows.

Syntax

re.compile(pattern, flags=0)

The expression's behavior can be modified by specifying a flags value. Values can be any of the following variables, combined using bitwise OR (the | operator).

prog = re.compile(pattern) result = prog.match(string) is equvalent to

result = re.match(pattern, string) Similarly the sequence

prog = re.compile(pattern) result = prog.search(string) is equvalent to

result = re.search(pattern, string)

using re.compile() and saving the resulting expression object for reuse is more efficient when the expression will be used several times in a single program.

Example program import re file =

open('abc.txt','r') text= file.readlines()

file.close() keyword = re.compile(r'a

') for lines in text:

if keyword.search(lines):

print(lines)

Practice Questions

1) Given a string like, "1,3-7,12,15,18-21", produce the list [1,3,4,5,6,7,12,15,18,19,20,21]
2) Write a python program to check the validity of password input by user. Validation (At least 1 letter between [a-z] and 1 letter between [A-Z]. At least 1

number between [0-9]. At least 1 character from [$#@]. Minimum length 6 characters. Maximum length 16 characters.

3) Create a file test.txt. Retrieve all lines that contain "the".
4) Retrieve all lines that contain "the" with lower or upper case letter .
5) Retrieve lines that have two consecutive o's.
6) Retrieve lines that contain a three letter string consisting of "s", then any character, then "e", such as "she".
7) Retrieve lines with a three letter word that starts with s and ends with e.
8) Retrieve lines that contain a word of any length that starts with s and ends e .
9) Retrieve lines that contains a word of any length that starts with s and ends with e and the word has at least four characters. 10) Retrieve lines that start with a.

Chapter 5 : Introduction to GUI

In python we can provide GUI using a library called "tkinter". tkinter provides a powerful object-oriented interface. One can create GUI application using tkinter by the following steps

1) First import tkinter module
2) Create GUI application main window.
3) Enter the mainloop and add widgets related to your application
4) Close the mainloop Example

```
From tkinter import *

Obj = Tk()

#code to add widgets

Obj.mainloop()
```

tkinter

widgets

tkinter provides different widgets for control flow, input and Output. There are roughly fifteen types of widgets in tkinter, they are as follows

1) Buttons

This widget is used to display buttons in your application. The Syntax to create a button is as follows

```
B = Button(obj,option=value)
```

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the button function.

2) Checkbutton

This widget is used to display the number of options as checkboxes. User can select multiple options. The Syntax to create a checkbutton is as follows

```
c = Checkbutton(obj,option=value)
```

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the checkbutton function.

3) Frame

This widget is used as a container widget to organize other widgets. The Syntax to create a frame is as follows

f = Frame(obj,option=value)

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the frame function.

4) Canvas

This widget is used to draw different shaped in your application. The Syntax to create a canvas is as follows

B = Canvas(obj,option=value)

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the canvass function.

5) Entry

This widget is used to display single-lined text field to accept input from the user. The Syntax to create a entry is as follows

e = Entry(obj,option=value)

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the entry function.

6) Label

This widget is used to display a single line text in the application. The Syntax to create a label is as follows

l = Label(obj,option=value)

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the label function.

7) Listbox

The listbox widget provide a list of options to choose from. The Syntax to create a listbox is as follows

l = Listbox(obj,option=value)

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the listbox function.

8) Menubutton

This widget is used to display the menu bar and different menus in an application. The Syntax to create a menubutton is as follows

m = Menubutton(obj,option=value)

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the menubutton function.

9) Menu

This widget is used to provide various functionalities to the menu present in the menubar. The Syntax to create a menu is as follows

m = Menu(obj,option=value)

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the menu function.

10) Message

This widget is used to display multiline text field for accepting values from the user. The Syntax to create a message is as follows

m = Message(obj,option=value)

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the message function.

11) Radiobutton

This widget is used to display number of options in the form of radio buttons from which user can select one. The Syntax to create a radiobutton is as follows

r = Radiobutton(obj,option=value)

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the radiobutton function.

12) Scale

This widget is used to provider a slider. The Syntax to create a scale is as follows

```
s = scale(obj,option=value)
```

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the scale function.

13) Scrollbar

The scrollbar widget is used to add scrolling capability to various widgets. The Syntax to create a scrollbar is as follows

```
B = Scrollbar(obj,option=value)
```

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the scrollbar function.

14) Text

This widget is used to display text in multiple lines. The Syntax to create a text is as follows

```
B = Text(obj,option=value)
```

Here obj is the main tkinter object and options take different parameter that can be set or assigned to a value to make the text function.

15) tkMessagBox

This module is used to display message box in your application. The Syntax to create a tkMessageBox is as follows

```
tkMessageBx.FunctionName(title, message [, options])
```

Here function name is the name of the message box, title is the text to b displayed in the title bar of the message box, message is the text to be displayed as a message and options takes different parameter to configure the message box.

GeometryManagement

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

- The pack() Method - This geometry manager organizes widgets in blocks before placing them in the parent widget.
- The grid() Method - This geometry manager organizes widgets in a table-like structure in the parent widget.
- The place() Method -This geometry manager organizes widgets by placing them in a specific position in the parent widget.

Chapter 6 : Database programming

Python allows us to connect to various databases through database interfaces. Python's database interface is DB-API (Data Base - Application Programming Interface). We can choose the database which we want to connect to python. Python DB-API supports a large number of databases like oracle, MS-SQL server 200, mySQL, mSQL, Sybase etc. For using each database we need to download separate DB-API module.

Connecting to a Database

1. Create a database called SAMPLE in MySQL.
2. The user id and password used to access SAMPLE database is "user" and "pass" respectively.
3. Create a table STUDENT in SAMPLE.
4. The table STUDENT has a field ROLLNO, NAME, AGE, COURSE, GRADE.

The following code shows how to connect MySQL database with python.

Example program

```
    import MySQLdb      db =
MySQLdb.connect("localhost","user","pass","SAMPLE")  cursor = db.cursor()
    cursor.execute("SELECT VERSION()")      data = cursor.fetchone()
    print("database version:",data)      db.close()
```

While running this script, it is producing the following Output in linux machine.

Output

```
        Database version: 5.0.45
```

In the above example, a connection is established with the data source and a connection object is returned. In our example, the connection object is saved to db, If connection is not properly established, db will store the value None. The database object db is used to create a cursor object cursor and this cursor is used for executing SQL queries. At the end, database connection is closed and resources are released.

Creating Tables

Once a connection is successfully established, we can create the tables. Creation of tables, insertion, updating and deletion operations are performed in python with the help of execute() statement. The following example shows how to create a table in MySQL from python. **Example program**

```
import MySQL        db =
MySQLdb.connect("localhost","user","pass","SAMPLE")   cursor = db.cursor()
sql = """CREATE TABLE DEPT(
                DEPTNO INT,
                DEPT_NAME CHAR(20),
LOCATION CHAR(25))"""
cursor.execute(sql)   db.close()
```

INSERT Operation

Once a table is created, we need to insert values to the table. The values can be inserted using the INSERT statement of SQL. The following example shows an example for insert operation.

Example program

```
import MySQL        db =
MySQLdb.connect("localhost","user","pass","SAMPLE")   cursor = db.cursor()
sql = """INSERT INTO DEPT(DEPTNO,DEPT_NAME,LOCATION)
VALUES (10,'sales','chennai')"""      cursor.execute(sql)   db.close()
```

UPDATE Operation

UPDATE operation is done to modify existing values available in the table. We can update one or more records at the same time. The following example shows how to update records in a table from python. **Example program**

```
import MySQL          db =
MySQLdb.connect("localhost","user","pass","SAMPLE")   cursor = db.cursor()
sql = "UPDATE DEPT SET LOCATION = 'bangalore'                    WHERE
DEPT_NAME = 'sales'          cursor.execute(sql)   db.close()
```

DELETE Operation

DELETE operation is required when we want to delete some undesired or unwanted records from a table. We can specify DELETE operation with or without conditions. The following shows how to delete a record from a table. **Example program**

```
import MySQL          db =
MySQLdb.connect("localhost","user","pass","SAMPLE")   cursor = db.cursor()
sql = "DELETE FROM DEPT WHERE LOCATION = 'bangalore'
cursor.execute(sql)   db.close()
```

READ Operation

READ operation is used to fetch desired records from a database. There are several methods for fetching records from a database. Once a connection is established, we can make queried to a database. The following methods are used in READ operation.

1. **fetchone():** It fetches the next row of a query result set. A result set is an object is returned when a cursor object is used to query a table. **Example program**

```
import MySQL          db =
MySQLdb.connect("localhost","user","pass","SAMPLE")   cursor = db.cursor()
sql = "SELECT LOCATION FROM DEPT"     cursor.execute()       row =
cursor.fetchone()     if row:
        print("location:",row.LOCATION)   db.close()
```

Output

```
location:bangalore
```

2. **fetchall():** It fetches all the row in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.

Example program

```
import MySQL        db =
MySQLdb.connect("localhost","user","pass","SAMPLE")  cursor = db.cursor()
sql = "SELECT LOCATION FROM DEPT"    cursor.execute()      row =
cursor.fetchall()      if row:
        print("location:",row.LOCATION)   db.close()
```

Output

```
location:bangalore
location:chennai      location:mumbai
```

3. **rowcount():** This is a read-only attribute and returns the number of rows that were affected by an execute() method. **Example program**

```
import MySQL        db =
MySQLdb.connect("localhost","user","pass","SAMPLE")  cursor = db.cursor()
sql = "SELECT LOCATION FROM DEPT"    numrows = cursor.execute(sql)
print("number of records: ", numrows.rowcount)          db.close()
```

Output

```
number of records: 3
```

Transaction Control

A transaction is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit. The effects of all the SQL statement in a transaction can be either all committed (applied to the database) or all rolled back(undone from the database). Transaction is a mechanism to ensure data consistency. Transaction ensures 4 properties generally referred to as ACID properties.

1. Atomicity

2. Consistency

3. Isolation

4. Durability

COMMIT Operation

The COMMIT command is the transactional command used to serve changes invoked by a transaction to the database. The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command. the Syntax for a commit statement is db.commit()

ROLLBACK Operation

The ROLLBACK command is the transaction command used to undo transactions that have not already been saved to the database. The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued. The Syntax for rollback is db.rollback()

Disconnecting from a Database

A close() method is called to disconnect from the database. The Syntax for closing is db.close(). If the connection to a database is closed by the user with the close() method, any outstanding transactions are rolled back by the database. However, instead of depending on any of the database lower level implementation details, our application would be better to call commit or rollback explicitly.

Exception Handling in Database

There are many sources of errors in databases. A few examples are a Syntax error in an executed SQL statement, a connection failure or calling the fetch method for an already canceled or finished statement handle. The database API defined a number of errors that must exist in each database module. The following table lists the exceptions and their descriptions related to database

Exceptions	Description
DatabaseError	Used for error in the database. Must subclass Error.
DataError	Subclass of DatabaseError that refers to error in the data.
Error	Base class for errors. Must subclass StandardError.
IntegrityError	Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys.

InterfaceError	Used for errors in the database module, not the database itself, Must subclass Error.
InternalError	Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.
NotSupportedError	Subclass of DatabaseError that refers to trying to call unsupported functionality.
OperationalError	Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally outside the control of the python script.
ProgrammingError	Subclass of DtabaseError that refers to errors such as a bad table name.
Warning	Used for non-fatal issues. Must subclass StandardError.

Working with mysql database and mysql connector

[Note: >>> import mysql.connector should not give any error, otherwise install the relevant drivers and connector]

---------- Reading Records ---------- **# 1. Make connection**

object from mysql.connector import connect

conn=connect(host="127.0.0.1",database="db2",user="root",passw ord="")

 # 2. Make cursor object cursor =

conn.cursor() **# 3. Execute query** sql =

"select * from mytable"

cursor.execute(sql) **# 4. Read operation**

for i in cursor.fetchall():

  print(i) **# 5. Close**

conection conn.close()

---------- Writing Records ---------- **# 1. Make connection**

object from mysql.connector import connect

```
conn=connect(host="127.0.0.1",database="db2",user="root",passw ord="")
# 2. Make cursor object cursor =
conn.cursor() # 3. Execute query sql =
'insert into mytable
values("Ram","Kumar")'
cursor.execute(sql) # 4. Publish changes
conn.commit() # 5. Close conection
conn.close()
```

---------- Popular mysql commands ----------

1. To enter mysql prompt mysql -u root
2. See all databases show databases;
3. To select a database; use <dbname>; 4. To create table

```
create table
<tablename>( attribute
datatype(size), attribute
datatype(size)
);
```

5. To read from table select * from <tablename>; 6. To insert values in table insert into <tablename> values("string_value", int);
7. Delete table drop table <tableame>
8. Update table
alter table <old_table_name> rename <new_table_name>;

Practice Questions

1. Create a table sailor with fields sid:integer, sname:char, age:int, rating:int, rating:int with sid as the primary key.
2. Insert values into sailor.
3. Update sailor with rating incremented by one.
4. Delete sailor whose age>40
5. Find the result of following SQL queries.
   i) find details of sailor ii) find the sailor names with age>20 and age<40 iii) find the name and rating of the sailors with age>40

Chapter 7 : Socket Programming

Sockets are nothing but logical end points of a bidirectional communication channel. In python to create a socket you must use socket.socket() function that is socket module. The syntax to create a simple socket is as follows.

s = socket.socket(socket_family, socket_type, protocol=0)

Here socket_family represents the family of socket it can be either AF_UNIX or AF_INET. socket_type represents the type of the socket it can be either SOCK_STREAM or SOCK_DGRAM. protocol is typically initialized to zero, this can be used to identify a variant of protocol within a socket domain and type.

Socket methods are broadly classified into three types

1) server socket methods
2) client socket methods
3) general socket methods

Server Socket Methods

Server is nothing but a system that responds to the request sent by the client. The request can be to do some computation, or to fetch the contents of a particular file, or just a message. The methods that are used to setup a server after creating a socket are as follows

1) s.bind() : This method is used to bind the address which contains the hostname and port number to the socket
2) s.listen() : This method is used to start TCP listener, which waits till the connection request is sent from the client.
3) s.accept() : This method is used to accept the TCP client connection request.

Client Socket Methods

Client is nothing but a system that sends request to the server. The request can be to do some computation, or to fetch the contents of a particular file, or just a message. The methods that are used to setup a client after crating a socket are as follows.

1) s.connect() : This method is used to initiate the communication with TCP server.

General Socket Methods

General socket methods are the methods that are used to establish communication between client and server. The methods are as follows

1) s.recv() : This method is used to receive TCP messages.

2) s.send() : This method is used to send TCP messages.
3) s.recvfrom() : This method is used to receive UDP messages.
4) s.sendto() : This method is used to send UDP messages.
5) s.close() : This method is used to close communication.
6) socket.gethostname() : This method returns the hostname.

Following is an example program to create a simple server and simple client.

Server.py

```
import socket,time print("\n********** SERVER

**********\n")

# 1. Create the socket object s = socket.socket()

print("Stage 1: Socket object Created.\n") time.sleep(2) #

2. Setup Connection host = "192.168.0.104" port = 5555

s.bind((host,port)) print("Stage 2: Binding

complete.\n") time.sleep(2)

# 3. Listen to client request

s.listen(100)

print("Stage 3: Server is now listening for client reuquest:
\n") for i in range(5):

time.sleep(1)

print(".",end="") counter = 0

while True:    if counter ==

10:

    break
  client_ip, client_id =s.accept()

  print("Conected to client %s at ip %s" %
(client_id,client_ip))    while True:

    data = client_ip.recv(1024)       print(data.decode('ascii'))

info = input("Server> ")
```

```
client_ip.send(info.encode('ascii'))    counter += 1
client_ip.close()
s.close() # Shutdown socket. print("Conection terminated")
```

client.py

```
import socket,time print("\n********** Client **********\n")
# 1. Create the socket object s = socket.socket()
print("Stage 1: Socket object Created.\n") time.sleep(1) #
2. Setup Coennection host = "192.168.0.104" port = 5555
time.sleep(1)
# 3. Make conection request print("Stage 3: Trying to connect to
server: \n") for i in range(3):
  time.sleep(1)    print(".",end="")
s.connect((host,port)) while True:
 data = input('Client> ')   if not
data:break
 s.send(data.encode('ascii'))   data =
s.recv(1024)   if not data:break   print
(data.decode('ascii')) s.close()
```

Chapter 8 : XML parsing

XML is a portable, open source language that allows programmers to develop applications that can be read by other applications, regardless of operating system and/or developmental language.

Python provides standard libraries and set of interfaces to work with XML files. One such interface is DOM (Document Object Model). In DOM the entire file is read into memory and stored in a hierarchical form to represent all the features of an XML document. DOM is extremely useful for random-access applications.

Following program explains the way to load an XML document. First we need to create a minidom object using xml.dom module. The minidom object provides a simple parser method that quickly creates a DOM tree from XML file. **Example program**

test.xml

```
<<collection main_attr="Contact Information">
        <element ele_attr="Python Student">
          <name>Alok</name>
           <email>a@gmail.com</email>
          <phone>12345</phone>
           <address>Abc</address>
    </element>
      <element ele_attr="Trainer">
        <name>Ram</name>
           <email>r@gmail.com</email>
         <phone>23456</phone>
           <address>Bcd</address>
    </element>
```

```
        <element ele_attr="Django Student">
          <name>Suman</name>
            <email>s@gmail.com</email>
          <phone>34567</phone>
            <address>Cde</address>
      </element>
</collection>
```

myxmlparser.py

```
from xml.dom.minidom import parse obj =
parse('test.xml')  collection =
obj.documentElement
print("\nMain attribute is:
",collection.getAttribute('main_attr'))  elements =
collection.getElementsByTagName('element') print("\n ********** Element level
details **********\n") for element in elements:
  print("\nThis detail is for: ",element.getAttribute('ele_attr'))    name =
element.getElementsByTagName('name')[0]    print("Name:
",name.childNodes[0].data)    email =
element.getElementsByTagName('email')[0]    print("Email:
",email.childNodes[0].data)    phone =
element.getElementsByTagName('phone')[0]    print("Phone:
",phone.childNodes[0].data)    address =
element.getElementsByTagName('address')[0]    print("Address:
",address.childNodes[0].data)
```

Output

```
Main attribute is:  Contact Information
********** Element level details **********
This detail is for:  Python Student
Name:  Alok
Email:  a@gmail.com
Phone:  12345
Address:  Abc

This detail is for:  Trainer
Name:  Ram
Email:  r@gmail.com
Phone:  23456
Address:  Bcd

This detail is for:  Django Student
Name:  Suman
Email:  s@gmail.com
Phone:  34567
Address:  Cde
```

Chapter 9 : Iterators Generators and Decorators

Iterators

An iterator can be seen as a pointer to a container, e.g. a list structure that can iterate over all the elements of this container. The iterator is an abstraction, which enables the programmer to access all the elements of a container (a set, a list and so on) without any deeper knowledge of the data structure of this container object.

Python iterator objects are required to support two methods while following the iterator protocol. *__iter__* returns the iterator object itself. This is used in **for** and **in** statements.

__next__ method returns the next value from the iterator. If there are no more items to return then it should raise **StopIteration** exception. **Example program** class Counter(object): def __init__(self, low, high):

self.current = low self.high = high

def __iter__(self):

'Returns itself as an iterator object' return self

def __next__(self):

'Returns the next value till current is lower than high' if self.current > self.high:

raise StopIteration else:

self.current += 1 return self.current - 1

After defining the above iterator class it can be used as follows

>>> c = Counter(5,10) >>> for i in c:

... print(i, end=' ')

...

5 6 7 8 9 10

Remember that an iterator object can be used only once. It means after it raises **StopIteration** once, it will keep raising the same exception.

Example

```
>>> c = Counter(5,6)
>>> next(c) 5
>>> next(c)
6
>>> next(c)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 11, in next
StopIteration
>>> next(c)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 11, in next
StopIteration
```

Generators

A generator is a function that produces a sequence of results instead of a single value. Generators are a simple and powerful possibility to create or to generate iterators. A simple generator can be created using **yield** keyword.

Example program 1

```
>>> def my_generator():
...     print("Inside my generator")
```

```
... yield 'a'
... yield 'b' ... yield 'c' ...
>>> my_generator()
<generator object my_generator at 0x7fbcfa0a6aa0>
```

In the above example we create a simple generator using the yield statements. We can use it in a for loop just like we use any other iterators.

```
>>> for char in my_generator():
... print(char) ...
Inside my generator a b c
```

Example program 2 def counter_generator(low, high): while low <= high:

```
    yield low
        low += 1
```

Output

```
>>> for i in counter_generator(5,10):
... print(i, end=' ')
...
5 6 7 8 9 10
```

In the above program, inside the while loop when the control reaches to the **yield** statement, the value of low is returned and the generator state is suspended. During the second **next** call the generator resumed where it freeze-ed before and then the value of **low** is increased by one. It continues with the while loop and comes to the **yield** statement again.

Decorators

A decorator in Python is any callable Python object that is used to modify a function or a class. A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class. The modified functions or classes usually contain calls to the original function "func" or class "C".

The following example program gives a simple decorator

Example program

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

def foo(x):
    print("Hi, foo has been called with " + str(x))

print("We call foo before decoration:")
foo("Hi")

print("We now decorate foo with f:")
foo = our_decorator(foo)

print("We call foo after decoration:")
foo(42)
```

Output

```
We call foo before decoration:
Hi, foo has been called with Hi
We now decorate foo with f:
We call foo after decoration:
Before calling foo
Hi, foo has been called with 42
After calling foo
```

Chapter 10 : Date and Time

Python provides various modules like time, calendar, datetime for parsing formatting and performing calculations on date and time.

The time Module

The time module contains a lot of functions to process time. Time intervals are represented as floating point number in seconds. The time since January 1, 1970, 12:00 a.m is available and is termed as an epoch. The following example shows how many seconds have elapsed since the epoch. **Example** import time seconds = time.time()

print("Number of seconds since 12:00am, January 1, 1970:",seconds)

Output

Number of seconds since 12:00am, January 1, 1970:1448080451.357909

The float representation is useful when sorting or comparing dates, but not as useful for producing human readable representations. For printing current ctime() can be more useful. The following shows an example ctime() method. **Example** import time print("The time is:", time.ctime()) later = time.time()+30 print("30 secs from now:", time.ctime(later))

Output

The time is: Sat Nov 21 10:10:08 2015

30 secs from now: Sat Nov 21 10:10:38 2015

Struct_time Structure

Storing times as elapsed seconds is useful in some situations, but there are times when you need to have access to the individual fields of a date (year, month, etc.). The time module defines struct_time for holding date and time values with components broken out so they

are easy to access. There are several functions that work with struct_time values instead of floats. The struct_time has the following attributes.

Attribute	Values
tm_year	Current year
tm_mon	1 to 12
tm_mday	1 to 31
tm_hour	0 to 23
tm_min	0 to 59
tm_sec	0 to 61(60 and 61 are leap seconds)
tm_yday	0 to 6 (0 is Monday)
tm_wday	1 to 366 (Julian day)
tm_isdst	-1,0,1 -1 if library determines DST

Parsing and Formatting Time

The two functions strptime() and strftime() convert between struct_time and string representations of time values. There is a long list of formatting instructions available to support input and output in different styles. The following example converts the current time from a string, to a struct_time instance, and back to a string.

Example

```
import time now = time.ctime() print(now) parserd =

time.strptime(now) print(parsed) print(time.strftime("%a %b %d %H

%M %S %Y", parsed))
```

Output

```
Sat Nov 21 11:16:24 2015
Time.stuct_time(tm_year=2015, tm_mon=11, tm_mday=21, tm_hour=11, tm_min=16,
tm_sec=24, tm_wdays=5, tm_yday=325, tm_isdst=-1)

Sat Nov 21 11:16:24 2015
```

The calendar module

The calendar module has many built-n functions for printing and formatting the output. By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call calendar.setfirstweekday() function. The following example shows a formatted text calendar.

Example import calendar c =

calendar.TextCalendar(calendar.SUNDAY)

c.prmonth(2015,10)

Output

October 2015

Su Mo Tu We Th Fr Sa

1 2 3

4 5 6 7 8 9 10

11 12 13 14 15 16 17

18 19 20 21 22 23 24 25 26 27 28 29 30 31