PYTHON

LECTURE 48

Today's Agenda

- **Operator Overloading**

- What Is Operator Overloading
- How To Perform Operator Overloading
- List Of Operators Which Can Be Overloaded
- Reverse Arithmetic Operators

What Is Operator Overloading?

- **Operator overloading** means **redefining** existing operators in **Python** to work on objects of **our classes**.
- For example, a **+** operator is used to **add** the **numeric values** as well as to **concatenate** the **strings**.
- That's because operator **+** is overloaded for **int** class and **str** class.

What Is Operator Overloading?

- But we can give **extra functionality** to this **+** operator and use it with **objects** of **our own class**.

- This method of giving **extra functionality** to the operators is called **operator overloading**.

Guess The Output ?

```
class Point:
    def __init__(self,x,y):
            self.x=x
            self.y=y
    def __str__(self):
            return f"x={self.x},y={self.y}"

p1=Point(10,20)
p2=Point(30,40)
p3=p1+p2
print(p3)
```

Why did TypeError occur?

TypeError was raised since Python didn't know how to add two Point objects together.

Output:

```
Traceback (most recent call last):
  File "opov11.py", line 10, in <module>
    p3=p1+p2
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

How To Perform Operator Overloading?

- There is an **underlying mechanism** related to **operators** in **Python**.

- The thing is when we use **operators**, a **special function** or **magic function** is automatically invoked that is associated with that **particular operator**.

How To Perform Operator Overloading?

- For example, when we use **+ operator**, the magic method **___add___** is automatically invoked in which the operation for **+ operator** is defined.

- So **by changing this magic method's code**, we can give extra meaning to the **+ operator**.

Example

```
class Point:
    def __init__(self,x,y):
            self.x=x
            self.y=y
    def __add__(self,other):
            x=self.x+other.x
            y=self.y+other.y
            p=Point(x,y)
            return p

    def __str__(self):
            return f"x={self.x},y={self.y}“
p1=Point(10,20)
p2=Point(30,40)
p3=p1+p2
print(p3)
```

Output:

```
x=40 , y=60
```

Explanation

- When we wrote **p1 + p2**, then **Python** did the following:
  - It searched for the magic method **___add___()** in our **Point** class since the left side operand i.e. **p1** is of **Point** class.
  - After finding **___add___()** in our class **Python converted** our statement **p1+p2** to **p1.___add___(p2)** which in turn is **Point.___add___(p1,p2)**.
  - So **p1** is passed as **self** and **p2** is passed to **other**
  - Finally **addition was done** and a new object **p** was returned which was copied to **p3**

Guess The Output

```
class Point:
    def __init__(self,x,y):
            self.x=x
            self.y=y
    def __add__(self,other):
            x=self.x+other.x
            y=self.y+other.y
            p=Point(x,y)
            return p

    def __str__(self):
            return f"x={self.x},y={self.y}"
p1=Point(10,20)
p2=Point(30,40)
p3=p1+p2
print(p3)
p4=p1+p2+p3
print(p4)
```

Output:

```
x=40,y=60
x=80,y=120
```

Exercise

- Write a program to create a class called **Distance** having **2 instance members** called **feet** and **inches** . Provide following methods in **Distance** class
  - **__init__() :** This method should accept 2 arguments and initialize **feet** and **inches** with it
  - **__str__():** This method should return string representation of **feet** and **inches**
  - **__add__() :** This method should add 2 **Distance objects** and return another **Distance object** as the result. While adding if sum of **inches** becomes **>=12** then it should be appropriately converted to **feet**

Solution

```
class Distance:
    def __init__(self,feet,inches):
            self.feet=feet
            self.inches=inches
    def __add__(self,other):
            feet=self.feet+other.feet
            inches=self.inches+other.inches
            if inches>=12:
                    feet=feet+inches//12
                    inches=inches%12
            d=Distance(feet,inches)
            return d

    def __str__(self):
            return f"feet={self.feet},inches={self.inches}“
```

```
d1=Distance(10,6)
d2=Distance(8,9)
d3=d1+d2
print(d1)
print(d2)
print(d3)
```

Output:

```
feet=10,inches=6
feet=8,inches=9
feet=19,inches=3
```

Guess The Output ?

```
class Distance:
    def __init__(self,feet,inches):
            self.feet=feet
            self.inches=inches
    def __add__(self,other):
            feet=self.feet+other.feet
            inches=self.inches+other.inches
            if inches>=12:
                    feet=feet+inches//12
                    inches=inches%12
            d=Distance(feet,inches)
            return d

    def __str__(self):
            return f"feet={self.feet},inches={self.i
```

```
d1=Distance(10,6)
d2=Distance(8,9)
d3=d1+d2
print(d1)
print(d2)
print(d3)
d4=d1+10
print(d4)
```

Why did AttributeError occur ?
This is because Python is trying to use the int object as Distance object and since int class has no feet data member the code is throwing AttributeError

Output:

```
feet=10,inches=6
feet=8,inches=9
feet=19,inches=3
Traceback (most recent call last):
  File "opoyl2.py", line 23, in <module>
    d4=d1+10
  File "opoyl2.py", line 6, in __add__
    feet=self.feet+other.feet
AttributeError: 'int' object has no attribute 'feet'
```

Solution

```
class Distance:
    def __init__(self,feet,inches):
            self.feet=feet
            self.inches=inches
    def __add__(self,other):
            if isinstance(other,Distance):
                    feet=self.feet+other.feet
                    inches=self.inches+other.inches
            else:
                    feet=self.feet+other
                    inches=self.inches+other
            if inches>=12:
                    feet=feet+inches//12
                    inches=inches%12
            d=Distance(feet,inches)
            return d
    def __str__(self):
            return f"feet={self.feet},inches={self.inches}“
```

```
d1=Distance(10,6)
d2=Distance(8,9)
d3=d1+d2
print(d1)
print(d2)
print(d3)
d4=d1+10
print(d4)
```

We have used isinstance() function to determine whether the argument other is of type Distance or not . If it is of type Distance we perform usual addition logic , otherwise we simply add the argument other to self.feet and self.inches as int value

Output:

```
feet=10,inches=6
feet=8,inches=9
feet=19,inches=3
feet=21,inches=4
```

List Of Arithmetic Operator For Overloading

Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)

Exercise

- Write a program to create a class called **Book** having **2 instance members** called **name** and **price** . Provide following methods in **Book** class
  - **__init__() :** This method should accept 2 arguments and initialize **name**and **price** with it
  - **__str__():** This method should return string representation of **name** and **price**
  - **__add__() :** This method should add price of 2 **Books** and return the **total price**

Solution

```
class Book:
    def __init__(self,name,price):
            self.name=name
            self.price=price

    def __add__(self,other):
            totalprice=self.price+other.price
            return totalprice

    def __str__(self):
            return f"name={self.name},price={self.price}"
```

```
b1=Book("Mastering Python",300)
b2=Book("Mastering Java",500)
print(b1)
print(b2)
print("Total price of books is:",b1+b2)
```

Output:

```
name=Mastering Python,price=300
name=Mastering Java,price=500
Total price of books is: 800
```

Guess The Output ?

```
class Book:
    def __init__(self,name,price):
            self.name=name
            self.price=price

    def __add__(self,other):
            totalprice=self.price+other.price
            return totalprice

    def __repr__(self):
            return f"name={self.name},price={self.price}"

b1=Book("Mastering Python",300)
b2=Book("Mastering Java",500)
b3=Book("Mastering C++",400)
print(b1)
print(b2)
print(b3)
print("Total price of books is:",b1+b2+b3)
```

Output:

```
name=Mastering Python,price=300
name=Mastering Java,price=500
name=Mastering C++,price=400
Traceback (most recent call last):
  File "opov14.py", line 19, in <module>
    print("Total price of books is:",b1+b2+b3)
TypeError: unsupported operand type(s) for +: 'int' and 'Book'
```

Why Did **TypeError** Occur ?

- **TypeError** occurred because **Python** evaluated the statement **b1+b2+b3** as follows:
  - At first it solved **b1+b2** , which became **b1.__add__(b2).**
  - So Python called **__add__()** method of **Book** class since the **left operand** is **b1** which is object of class **Book**
  - This call returned the **total price** of **b1** and **b2** which is **800**.
  - Now **Python** used **800** as the **calling object** and **b3** as argument so the call became **800.__add__(b3)**.
  - So Python now looks for a method **__add__()** in **int** class which can add an **int** and a **book** but it could not find such a method in **int** class which can take **Book** object as argument .
  - So the code threw **TypeError**

What Is The Solution To This Problem ?

- The solution to this problem is to provide **reverse special methods** in our class.

- The standard methods like **__add__(),__sub__()** only work when we have **object** of our class as **left operand** .

What Is The Solution To This Problem ?

- But they don't work when we have **object** of our class on **right side of the operator** and **left side operand** is not the **instance** of our class.

- **For example** : **obj+10** will call **___add___()** internally, but **10+obj** will not call **___add___()**

What Is The Solution To This Problem ?

- Therefore, to help us make our classes mathematically correct, Python provides us with **reverse/reflected special methods** such as **__radd__()**, **__rsub__()**, **__rmul__()**, and so on.

- These handle calls such as **x + obj**, **x - obj**, and **x * obj**, where **x** is **not an instance of the concerned class**.

Reflected Operators

Reflected arithmetic operators

__radd__(self, other)	b+a
__rsub__(self, other)	b-a
__rmul__(self, other)	b*a
__rfloordiv__(self, other)	b // a
__rdiv__(self, other)	b / a
__rtruediv__(self, other)	b / a (from __furture__ import division)
__rmod__(self, other)	b % a
__rdivmod__(self, other)	divmod(b, a)
__rpow__	b ** a

Modified Example

```
class Book:
    def __init__(self,name,price):
            self.name=name
            self.price=price
    def __add__(self,other):
            totalprice=self.price+other.price
            return totalprice
    def __radd__(self,other):
            totalprice=self.price+other
            return totalprice
    def __str__(self):
            return f"name={self.name},price={self.price}"
```

```
b1=Book("Mastering Python",300)
b2=Book("Mastering Java",500)
b3=Book("Mastering C++",400)
print(b1)
print(b2)
print(b3)
print("Total price of books
is:",b1+b2+b3)
```

Output:

```
name=Mastering Python,price=300
name=Mastering Java,price=500
name=Mastering C++,price=400
Total price of books is: 1200
```

List Of Relational Operator For Overloading

Operator	Expression	Internally
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

Exercise

- Write a program to create a class called **Distance** having **2 instance members** called **feet** and **inches** . Provide following methods in **Distance** class
  - **__init__() :** This method should accept 2 arguments and initialize **feet** and **inches** with it
  - **__str__():** This method should return string representation of **feet** and **inches**
  - **__eq__() :** This method should compare 2 **Distance** objects and return **True** if they are equal otherwise it should return **False**

Solution

```
class Distance:
    def __init__(self,feet,inches):
            self.feet=feet
            self.inches=inches
     def __eq__(self,other):
            x=self.feet*12+self.inches
            y=other.feet*12+other.inches
            if x==y:
                    return True
            else:
                    return False
    def __str__(self):
            return f"feet={self.feet},inches={self.inches}“
```

```
d1=Distance(0,12)
d2=Distance(1,0)
print(d1)
print(d2)
if d1==d2 :
        print("Distances are equal")
else:
        print("Distances are not equal")
```

Output:

```
feet=0,inches=12
feet=1,inches=0
Distances are equal
```

List Of Shorthand Operator For Overloading

Operator	Expression	Internally
-=	p1-=p2	p1.__isub__(p2)
+=	p1+=p2	p1.__iadd__(p2)
*=	p1*=p2	p1.__imul__(p2)
/=	p1/=p2	p1.__idiv__(p2)
//=	p1//=p2	P1.__ifloordiv__(p2)
%=	p1%=p2	p1.__imod__(p2)
**=	p1**=p2	p1.__ipow__(p2)

List Of Special Functions/Operators For Overloading

Function/Operator	Expression	Internally
len( )	len(obj)	obj.__len__(self)
[ ]	obj[0]	obj.__getitem__(self,index)
in	var in obj	obj.__contains__(self,var)
str( )	str(obj)	obj.__str__(self)

PYTHON

LECTURE 49

Today's Agenda

- **Database Programming In Python-I**
  - What Is Data And Database ?
  - What Is DBMS ?
  - What Is SQL ?
  - **How To Configure Our System For Database Programming In Python ?**

Introduction

- Before we learn about **Database Programming In Python**, let's first understand -

- **What is Data?**

- **What is Database ?**

Introduction

- **What is Data?**
  - In simple words **data** can be **facts** or **information.**
  - For example **your name**, **population** of a **country** , **names** of political parties in your **country** , **today's temperature** etc
  - A **picture** , **image** , **file** , **pdf** etc can also be considered data.

Introduction

- **What is a Database?**
  - A **database** is a **collection** of **inter-related data** or **information** that is organized so that it can easily be **accessed**, **managed**, and **updated** .
  - Let's discuss few examples.
    - Your **mobile's phone book** is a **database** as it stores data pertaining to people like their **phone numbers**, **name** and **other contact details** etc.
    - Your **University** uses **database** to store **student details** like **enrollment no**, **name** , **address** , **academic performance** etc
    - Let's also consider the **Facebook**. It needs to store, manipulate and present data related to **members**, their **friends**, **member activities**, **messages**, **advertisements** and lot more. Here also **database** is used

How Databases Store The Data ?

- Most of the **databases** store their data in the form of **tables**

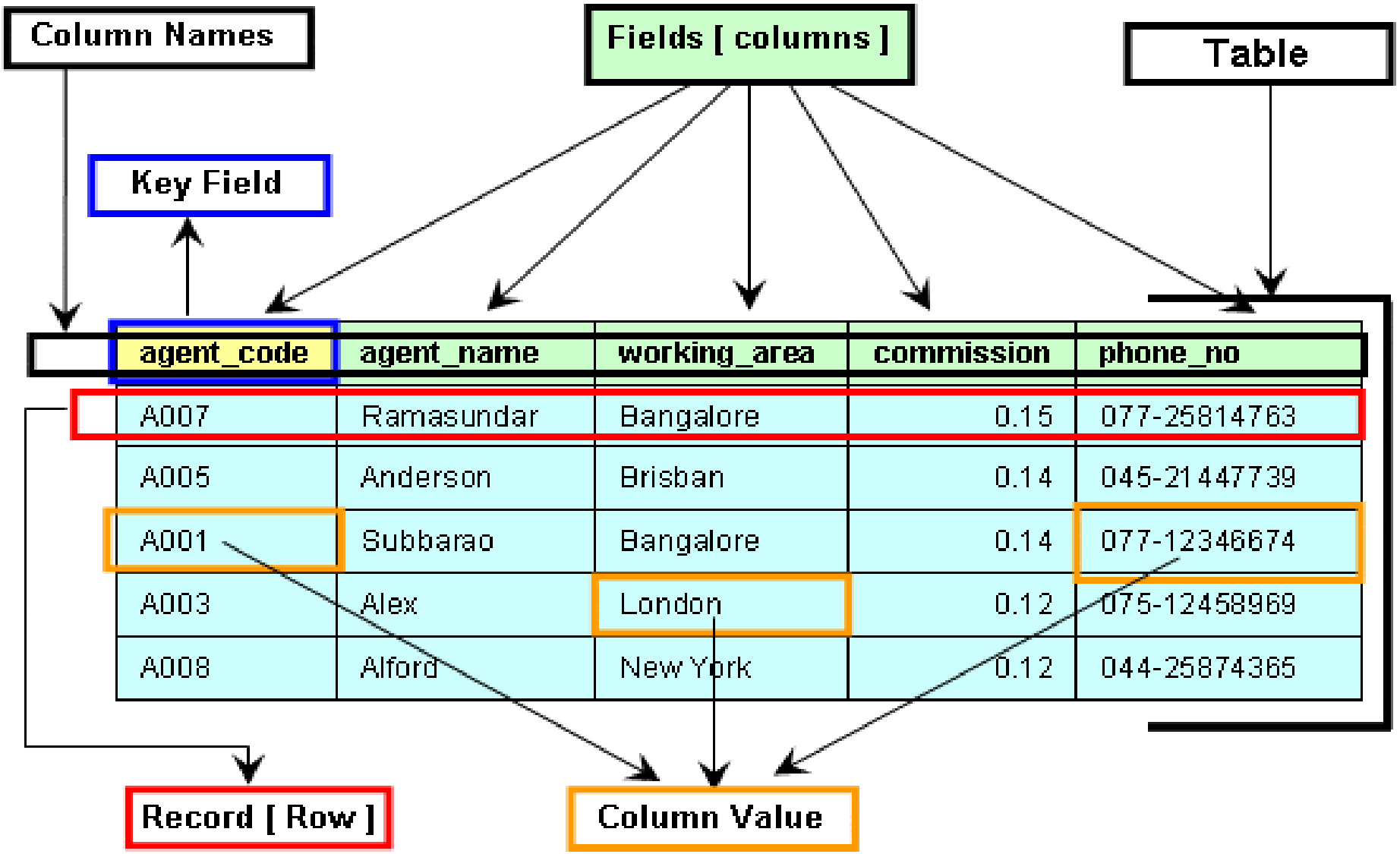
- Each **table** in a database has **one or more columns**, and each column is assigned a specific **data type**, such as an integer number, a sequence of characters (for text), or a date.

- Each **row** in the table has a value for each **column**.

How Databases Store The Data ?

Name	FName	City	Age	Salary
Smith	John	3	35	$280
Doe	Jane	1	28	$325
Brown	Scott	3	41	$265
Howard	Shemp	4	48	$359
Taylor	Tom	2	22	$250

Components Of A Table

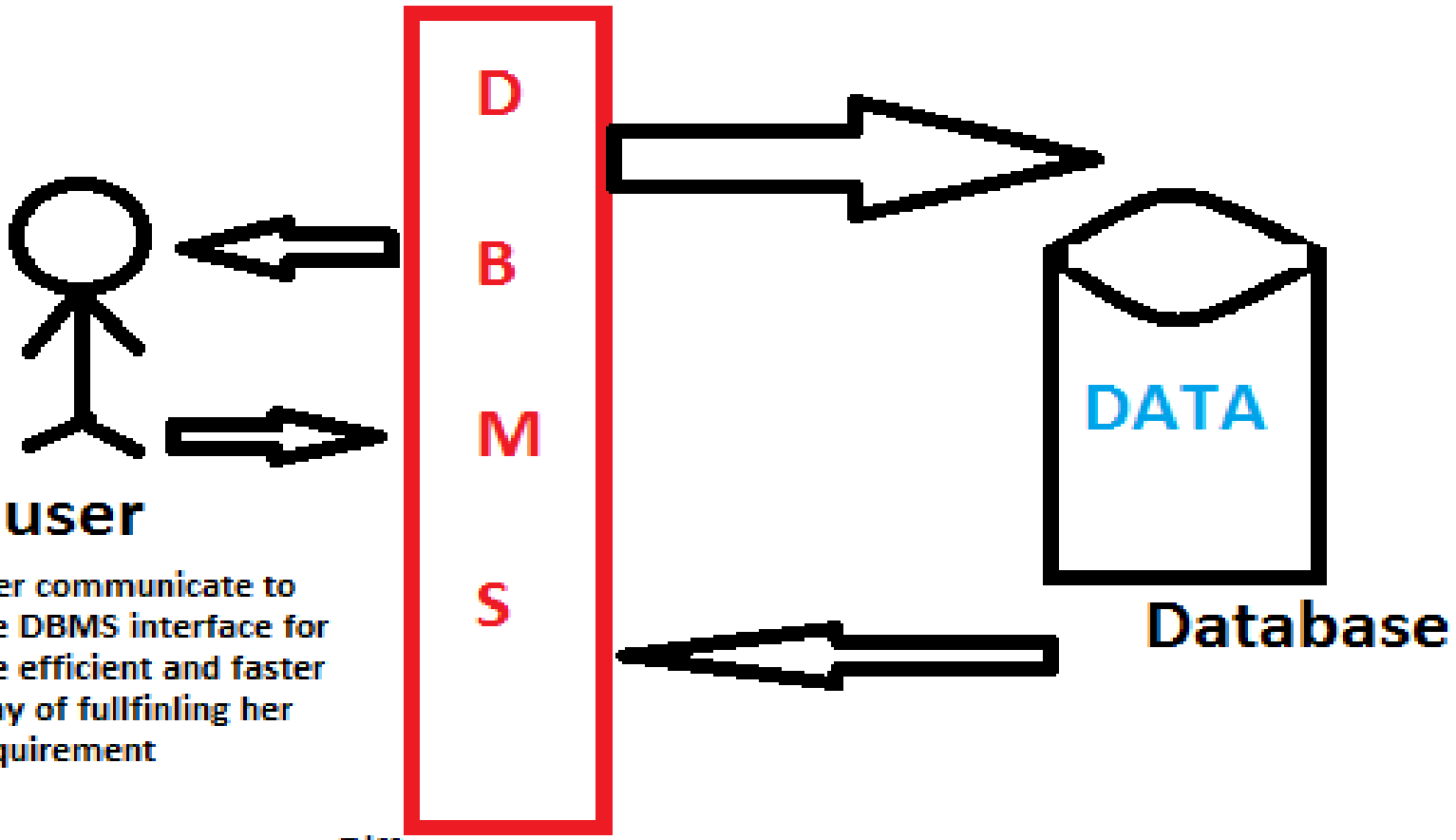


What Is A DBMS ?

- A **DBMS** is a program or a software that allows users to perform different **operations** on a database.

- These **operations** include:
  - **Creating the database/tables**
  - **Inserting records into these tables**
  - **Selecting records from these tables for displaying**
  - **Updating / Deleting the records**

What Is A DBMS ?



Some Popular DBMS

- Some of the most popular **DBMS** are:
  - **Oracle**
  - **MySQL**
  - **MS SQL Server**
  - **SQLite**
  - **PostgreSQL**
  - **IBM DB2**

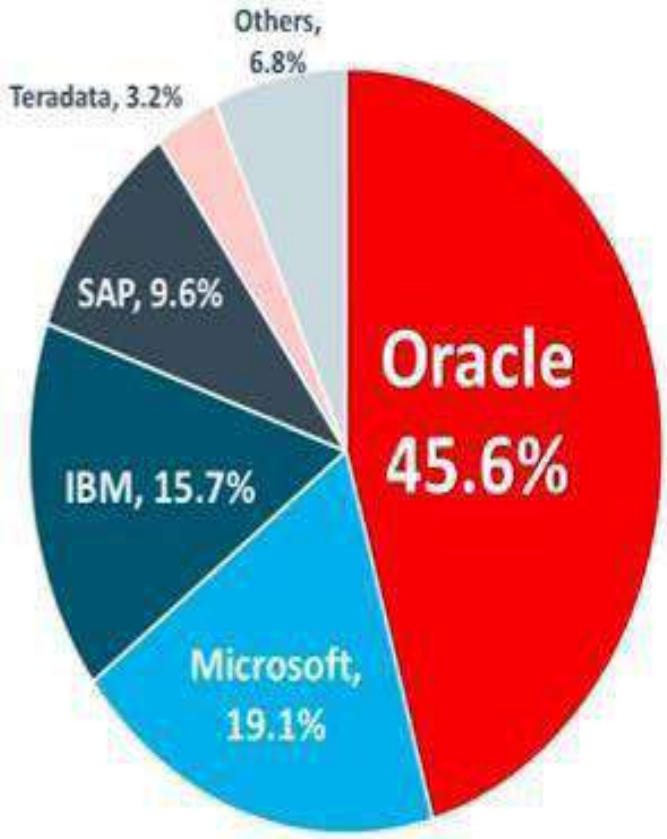
and many more

The Market Leader

python

Oracle Continues Have Largest **RDBMS** Market Share by Wide Margin

More than Double Sales of Nearest Competitor



Graphic created by Oracle based on Gartner

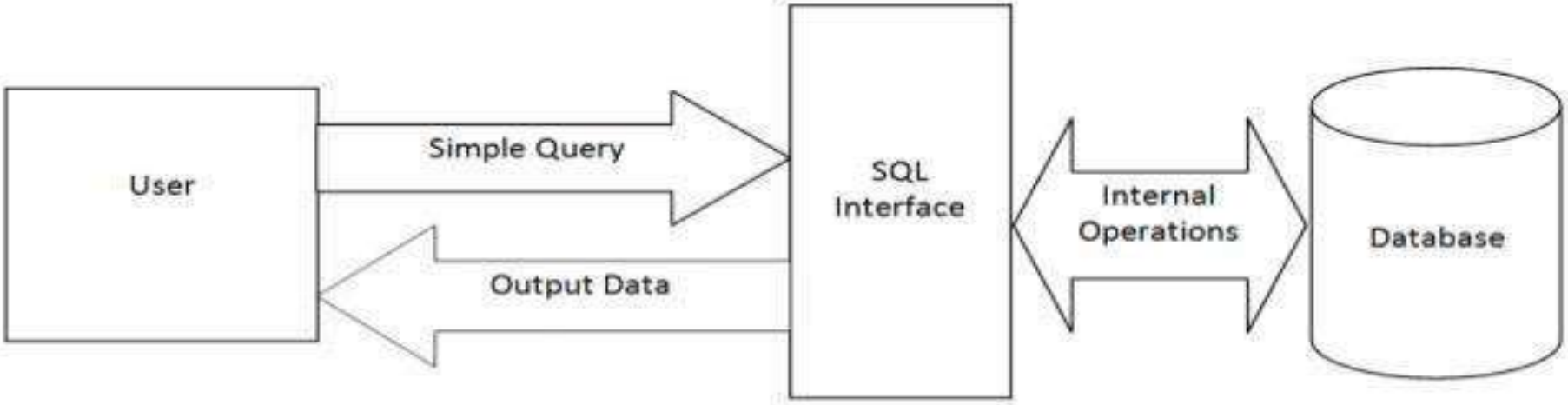
What Is SQL ?

- **SQL** is an abbreviation for "**Structured Query Language**".
- It is a language used by **EVERY DBMS** to interact with the database.
- It provides us **COMMANDS** for **inserting data** to a database, **selecting data** from the database and **modifying data** in the database

Pictorial View Of SQL

What is SQL?

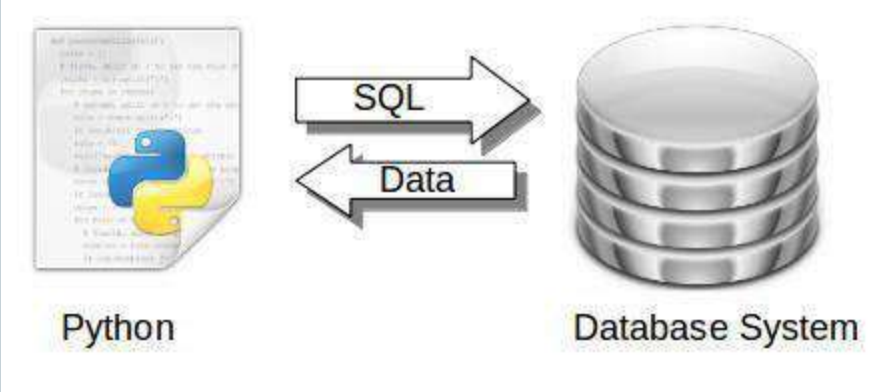
SQL is simply a language that makes it easy to pull data from your application's database.



Database Programming In Python

- **Python** is wonderfully able to interact with **databases**, and this is what we will learn in this chapter.

- **Advantages:**
  - **Platform-independent**
  - **Faster and more efficient**
  - **Easy to migrate and port database application interfaces**



How Python Connects To Database?

- Python uses the ***Python Database API*** in order to interact with databases.

- An **API** stands for **Application Programming Interface**.

- It is a **set** of **predefined functions** , **classes** and **methods** given by the **language** for a **particular task** and the programmer can use it whenever he wants to perform that task in his code.

How Python Connects To Database?

- The ***Python Database API*** allows us to handle **different database management systems** (DBMS) in our **Python code**.

- However the **steps at the code level remain altogether same**.

- That is using the same steps we can connect to **Oracle** or **MySQL** or **SQLite** or any other **DBMS**

PYTHON

LECTURE 50

Today's Agenda

- **Database Programming In Python-II**
  - Introduction To SQLite
  - Steps Needed For Connecting To SQLite From Python
  - Exploring Connection And Cursor Objects
  - Executing The SQL Queries
  - Different Ways Of Fetching The Data

Introduction To SQLite

- **SQLite** is an **Open Source Database** developed in **C** programming language.

- It stores data to a **text file** on the device and is **a popular choice** for application softwares such as web browsers as well as mobile platforms like **Android** and **iOS**.

SQLite Features

- **Extremely light-weighted** (not more than 500 KBs)
- **No complex setup**
- **Fully transactional.**
- It supports all standard relational database features like **SQL Queries**, **Joins** , **Constraints** etc

Who Uses Sqlite ?

Following are well known companies/products that use **SQLite**:

Adobe

Apple

McAfee

Microsoft

DropBox

Facebook

Google

Python

PHP

SQLite Limitations

- **SQLite** supports neither **RIGHT OUTER JOIN** nor **FULL OUTER JOIN**. It supports only **LEFT OUTER JOIN**.

- With **ALTER TABLE** statement in **SQLite** we can only **add a column** or **rename a table** or **column**.

- However, we can't do the following:

  - **DROP a column.**

  - **ADD a constraint.**

SQLite Limitations

- **GRANT** and **REVOKE** commands are not implemented in **SQLite**.

- **VIEWs** are **read-only** – we can't write **INSERT**, **DELETE**, or **UPDATE** statements into the view.

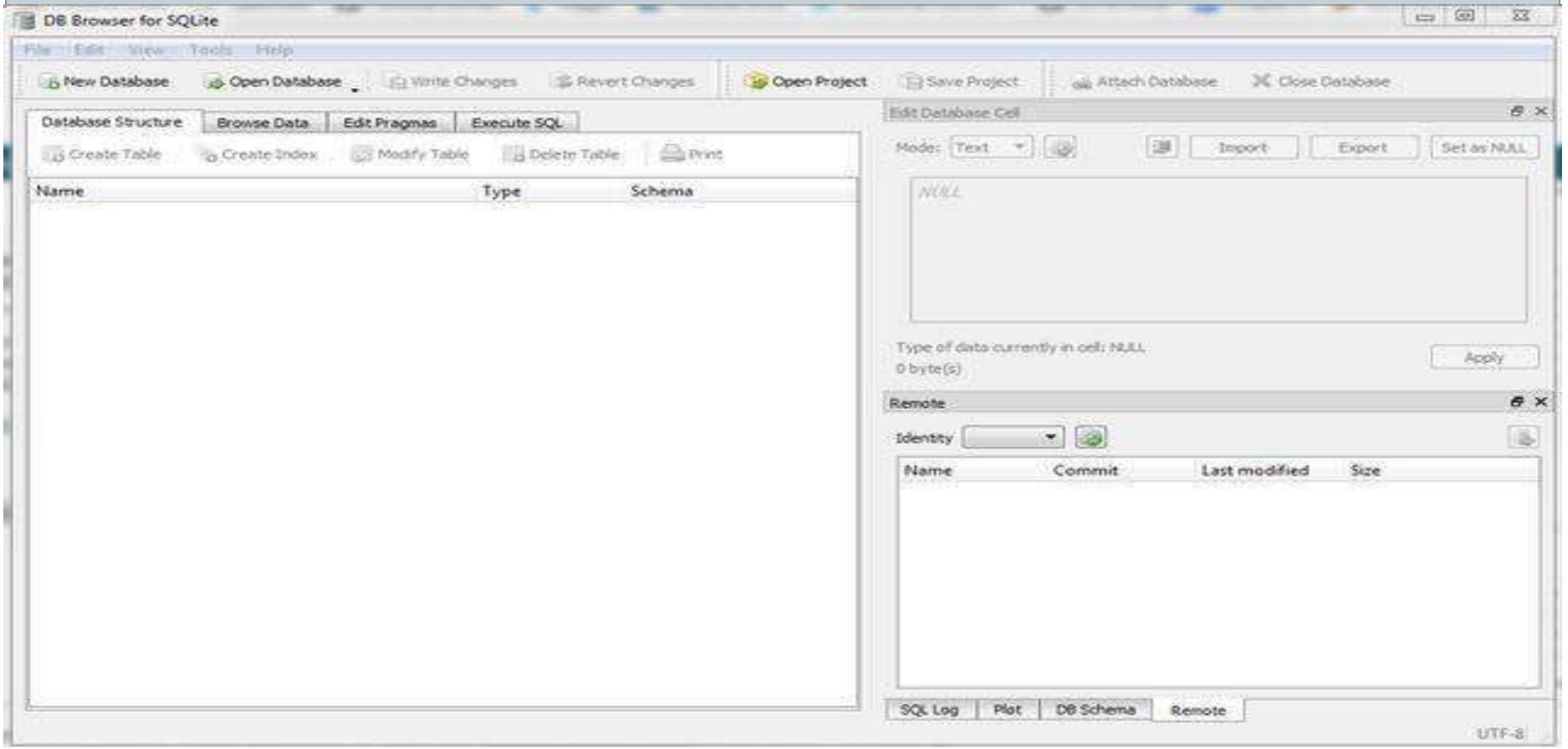
- **SQLite** only supports **FOR EACH ROW** triggers, and it doesn't support **FOR EACH STATEMENT** triggers.

SQLite Installation

- **Installing** and **setting up SQLite** takes a matter of a few minutes.
- We can use **SQLite** from the command line tools, but there is a **GUI-based utility** which lets us use **SQLite** through a decent graphical interface.
- This is called **DB Browser for SQLite**.
- We can download it from **https://sqlitebrowser.org/dl/** with respect to our OS platform.

SQLite Installation

- On Windows platform, the interface for DB Browser for SQLite looks like this –



Steps Required For Connecting Python Code To SQLite

- Connecting our **Python app** to any **database** involves total **6 important steps**.

- Also remember that ***these steps will always remain*** same ***irrespective of the database*** we are trying to connect

- The only ***difference*** will be the ***change*** in the ***name of the module***

Steps Required For Connecting Python Code To Sqlite

- For connecting to **SQLite** , the steps are:
  - Import the module **sqlite3**
  - Establish a ***connection*** to the database.
  - Create a ***cursor*** to communicate with the data.
  - ***Execute*** the SQL query
  - ***Fetch*** the result returned by the SQL query
  - ***Close*** the ***cursor*** and ***connection*** to the database.

Step 1-
Importing The Module

- Since we are connecting to **SQlite** , so all the **functions** and **classes** we will be using will be supplied by the **module** called **sqlite3**.

- So the first step will be to **import** this **module** by writing the following statement:

```
import sqlite3
```

Step 2-
Establishing The Connection

- After importing the module ,we must open the connection to the **SQlite**.
- This can be done by calling the function **connect( )** of **sqlite3** module having the following syntax:

sqlite3.connect("path to the db file")

- Following is the description of this function:
  - It accepts a **connection string** as argument mentioning the path to the db file
  - If a connection is established, then a **Connection** object is returned.
  - If there is any problem in connecting to the database , then this function throws the exception called **sqlite3.DatabaseError**

Important Attributes/Methods Of Connection Object

- When the connection is successful , we get back the **sqlite3.Connection** object.

- This object provides us some important methods which are as follows:

  - **cursor()** : Return a new **Cursor** object using the **connection.**
  - **close()**: **Closes** the **connection**
  - **commit()**: **Commits** any pending **transactions** to the database.
  - **rollback()**: **Rollbacks** any pending **transactions**.

Step 3-
Creating The Cursor

- Once we have a connection, we need to get a ***cursor***
- A **Cursor** allows us to send all the **SQL commands** from our Python code to the database.
- It can also hold the set of rows returned by the query and lets us work with the records in that set, in sequence, one at a time.
- To get a **Cursor** object we call the method **cursor( )** of the **Connection** object as follows:

```
cur=conn.cursor()
```

Important Attributes/Methods Of Cursor Object

- A **Cursor** object provides us some attributes and methods to execute the **SQL query** and get back the results

- Following are it's important **attributes**:
  - **rowcount**: Returns the number of rows fetched or affected by the last operation, or -1 if the module is unable to determine this value.

- Following are it's important **methods**:
  - **execute(statement)** : Executes an **SQL *statement*** string on the DB
  - **fetchall()**: Returns all remaining result rows from the last query as a sequence of tuples
  - **fetchone()**: Returns the next result row from the last query as a tuple
  - **fetchmany(n)**: Returns up to n remaining result rows from the last query as a sequence of tuples.
  - **close()**: Closes the cursor

Step 4-
The execute( ) Method

- **Syntax:**

execute(SQL statement, [parameters], **kwargs)

- This method can accept an **SQL statement** - to be run directly against the database. It executes this SQL query and stores the result.

- **For example:**

cur.execute('select * from allbooks')

The **execute( )** Method

- It can also accept **Bind variables** assigned through the **parameters** or **keyword arguments.**

- We will discuss this later

Step 5-
Fetching The Result

- Once we have executed the **SELECT query** , we would like to retrieve the rows returned by it.
- There are numerous ways to do this:
  - By iterating directly over the **Cursor** object
  - By calling the method **fetchone()**
  - By calling the method **fetchall()**
- We will discuss each of these methods after **step 6**

Step 6-
Fetching The Result

- The **final step** will be to **close the cursor** as well as **close the connection** to the database once we are done with processing.

- This is done by calling the method **close()** on both the objects.

- **Example:**

cur.close()

conn.close()

An Important Point!

- During communication with **Sqlite** , if any problem occurs the methods of the module **sqlite3** throw an exception called **DatabaseError**.

- So it is a best practice to execute **sqlite3** methods that access the database within a **try..except** structure in order to catch and report any exceptions that they might throw.

Directly Iterating Over The Cursor

- The **Cursor** object holds all the rows it retrieved from the database as **tuples**.

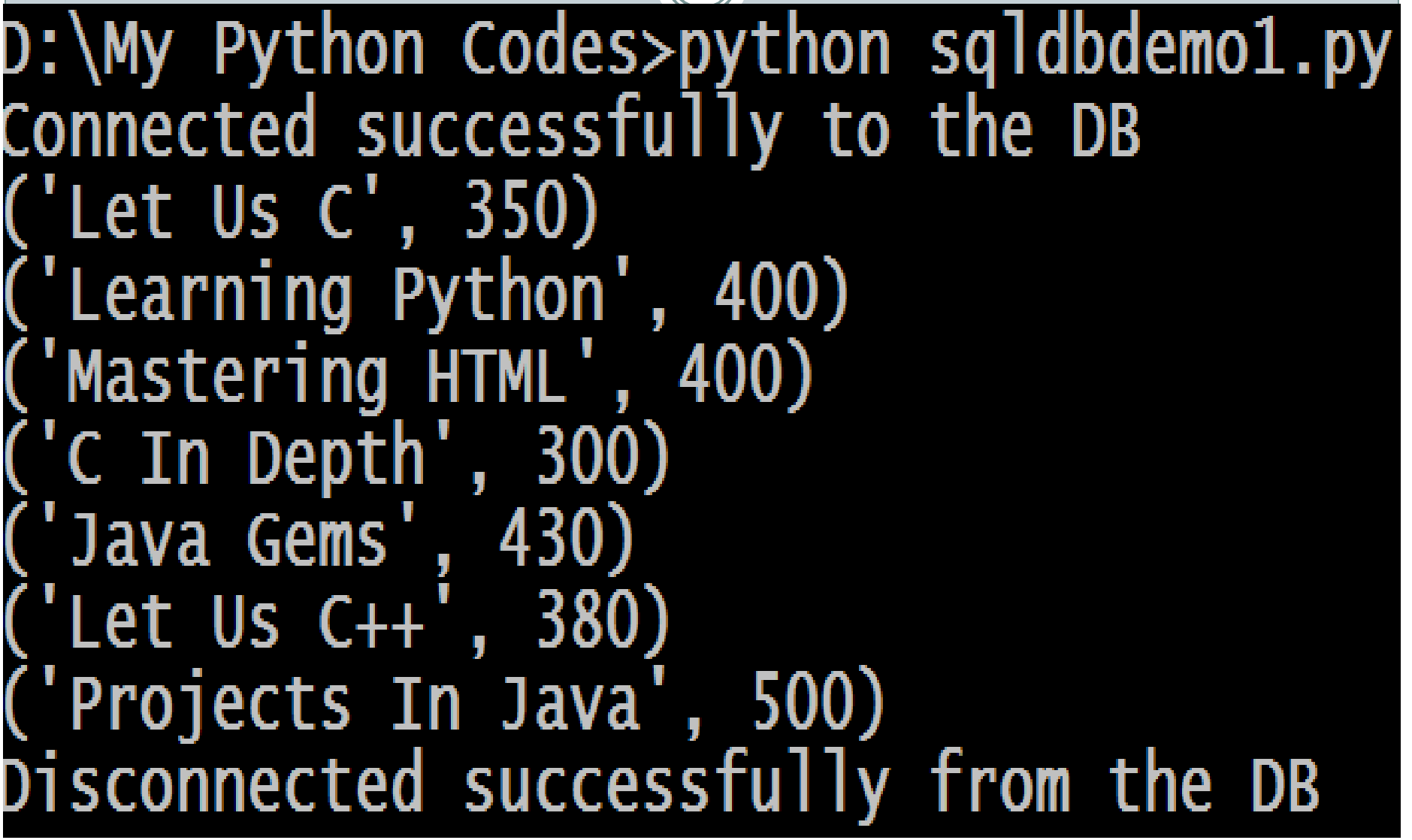
- So if we **iterate** over the **Cursor** object using the **for loop** , then we can retrieve these rows

Exercise

- Assume you have a table called **Allbooks** in the database which contains **4 columns** called **bookid**, **bookname**, **bookprice** and **subject**.

- Write a **Python** code to do the following:
  - Connect to the database
  - Execute the query to select **name of the book** and **it's price** from the table **Allbooks**
  - Display the records

Sample Output



Solution

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqlitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Select bookname,bookprice from allbooks")
    for x in cur:
            print(x)
except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
            if conn is not None:
                    conn.close()
                    print("Disconnected successfully from the DB")
```

Exercise

- Modify the code so that values are displayed without **tuple symbol** i.e. without the symbol of **( )**
- **Sample Output**

```
D:\My Python Codes>python sqldbdemo2.py
Connected successfully to the DB
Let Us C 350
Learning Python 400
Mastering HTML 400
C In Depth 300
Java Gems 430
Let Us C++ 380
Projects In Java 500
Disconnected successfully from the DB
```

Solution

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqlitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Select bookname,bookprice from allbooks")
    for name,price in cur:
            print(name,price)
except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
            if conn is not None:
                    conn.close()
                    print("Disconnected successfully from the DB")
```

We will just have to unpack each row of the tuple to get the individual values

Using The Method **fetchone( )**

- Sometimes we may want to pull just one record at a time from the table .
- As a result **Cursor** object provides us a method called **fetchone( )** .
- This method returns **one record** as a **tuple**, and if there are no more records then it returns **None**

Exercise

- Modify the previous code to display the name and price of the **costliest** book from the table **Allbooks**

Sample Output

```
D:\My Python Codes>python sqldbdemo3.py
Connected successfully to the DB
('Projects In Java', 500)
Disconnected successfully from the DB
```

Solution

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqlitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Select bookname,bookprice from allbooks order by bookprice desc")
    x=cur.fetchone()
    if x is not None:
            print(x)

except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
            if conn is not None:
                    conn.close()
                    print("Disconnected successfully from the DB")
```

Using The Method **fetchall( )**

- **Syntax:**

cur.fetchall()

- The method fetches all rows of a query result set and returns it as a **list** of **tuples**.

- If no more rows are available, it returns an **empty list**.

Sample Code

```
import cx_Oracle
conn=None
cur=None
try:
    conn=cx_Oracle.connect("scott/tiger@Sachin-PC/orcl")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Select bookname,bookprice from allbooks order by bookprice desc")
    x=cur.fetchall()
    print(x)
except (cx_Oracle.DatabaseError)as ex:
    print("Error in connecting to Oracle:",ex)
finally:
        if cur is not None:
                cur.close()
                print("Cursor closed successfully")
        if conn is not None:
                conn.close()
                print("Disconnected successfully from the DB")
```

Output

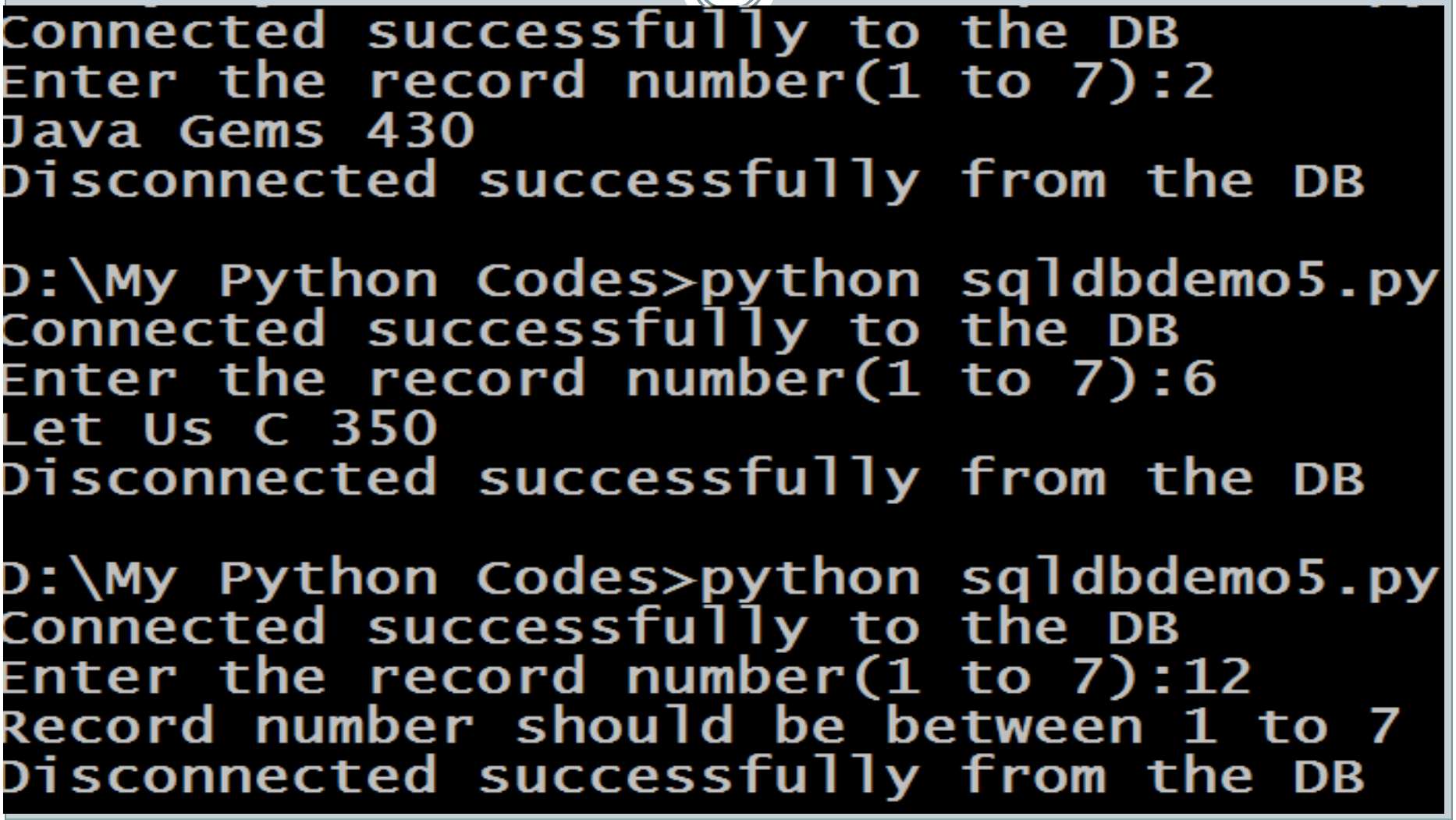
```
Connected successfully to the DB
[('Projects In Java', 500), ('Java Gems', 430), ('Learning Python', 400), ('Mast
ering HTML', 400), ('Let us C++', 380), ('Let us C', 350), ('C In Depth', 300)]
Disconnected successfully from the DB
```

Exercise

- Modify the previous book application so that your code asks the user to enter a record number and displays only that book .

- Make sure if the record number entered by the user is wrong then display appropriate error message

Output


Connected successfully to the DB
Enter the record number(1 to 7):2
Java Gems 430
Disconnected successfully from the DB
D:\My Python Codes>python sqldbdemo5.py
Connected successfully to the DB
Enter the record number(1 to 7):6
Let Us C 350
Disconnected successfully from the DB
D:\My Python Codes>python sqldbdemo5.py
Connected successfully to the DB
Enter the record number(1 to 7):12
Record number should be between 1 to 7
Disconnected successfully from the DB

Solution

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqlitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Select bookname,bookprice from allbooks
    order by bookprice desc")
    booklist=cur.fetchall()
    recnum=int(input("Enter the record number(1 to
    "+str(len(booklist))+"):"))
```

Solution

```
    if recnum <1 or recnum>len(booklist):
            print("Record number should be between 1 to
    "+str(len(booklist)))
    else:
            row=booklist[recnum-1]
            print(row[0],row[1])


except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
            if conn is not None:
                    conn.close()
                    print("Disconnected successfully from the DB")
```

PYTHON

LECTURE 51

Today's Agenda

- **Database Programming In Python-III**
  - Executing INSERT Command
  - Executing Dynamic Queries
  - Concept Of Bind Variables
  - Executing Update Command
  - Executing Delete Command

Inserting Record

- To insert a new record in the table we have to execute the **INSERT INTO** command.

- It has **2 syntaxes**:
  - **Insert into <table_name> values(<list of values>)**
  - **Insert into <table_name>(<list of cols>) values(<list of values>)**

Inserting Record

- To insert a record in the table from our Python code we simply pass the **insert query** as argument to the **execute( )** method of **cursor** object.

- It's general syntax is:

cur.execute("insert query")

- **Two important points:**

  - After executing insert if we want to get the number of row inserted we can use the **cursor** attribute **rowcount**

  - Unless we call the method **commit( )** of **connection** object , the record we insert does not get saved in the table

Steps Required For Inserting Records

- For inserting record the overall steps are:
  - Import the module **sqlite3**
  - Establish a ***connection*** to the database.
  - Create a ***cursor*** to communicate with the data.
  - ***Execute*** the ***Insert*** query
  - ***Commit*** the changes
  - ***Close*** the connection to the database.

Example

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqlitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Insert into allbooks values(108,'Python Web
    Prog',500,'Python')")
    n=cur.rowcount
    print(n," row inserted")
    conn.commit()

except (sqlite3.DatabaseError)a
    print("Error in connecting to Sqlite:",ex)
finally:
        if conn is not None:
                conn.close()
                print("Disconnected successfully from the DB")
```

```
Connected successfully to the DB
1  row inserted
Disconnected successfully from the DB
```

Executing Dynamic Queries

- **Dynamic queries** are those where we **set the values** to be **passed in the query** at **run time**.

- For example , we would like to **accept the values** of the **record to be inserted from the user** and then pass it to the insert query.

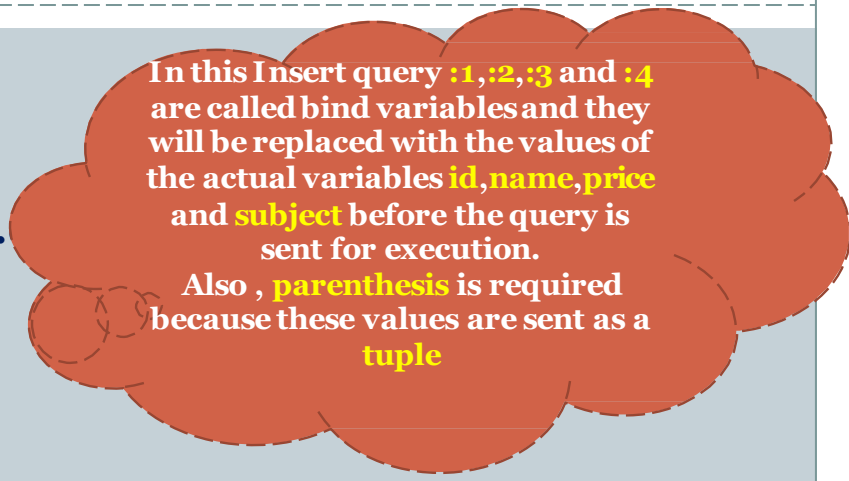
- Such queries are called **dynamic query**

Executing Dynamic Queries

- **Dynamic queries** for **Sqlite** can be set using the concept of **bind variables**

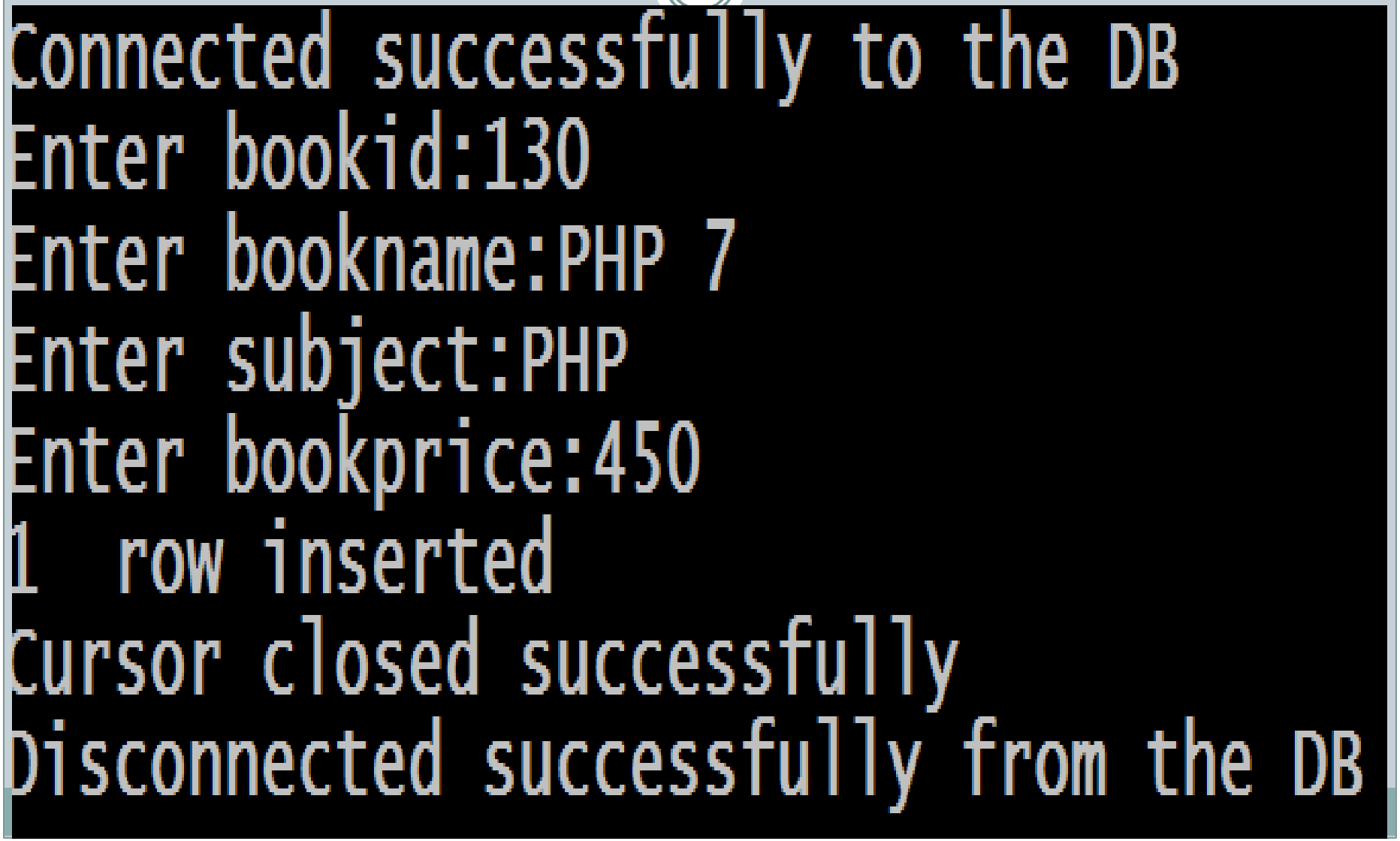
- **Bind variables** are like **placeholders** used in a query , represented using **:some_number**, and are replaced with actual values before query execution.

Using bind variables By Position




```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqlitedb/library.
    print("Connected successfully to the DB")
    cur=conn.cursor()
    id=int(input("Enter bookid:"))
    name=input("Enter bookname:")
    price=int(input("Enter bookprice:"))
    subject=input("Enter subject:")
    cur.execute("Insert into allbooks values(:1,:2,:3,:4)",(id,name,price,subject))
    n=cur.rowcount
    print(n," row inserted")
    conn.commit()
except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
        if conn is not None:
                conn.close()
                print("Disconnected successfully from the DB")
```

Sample Output

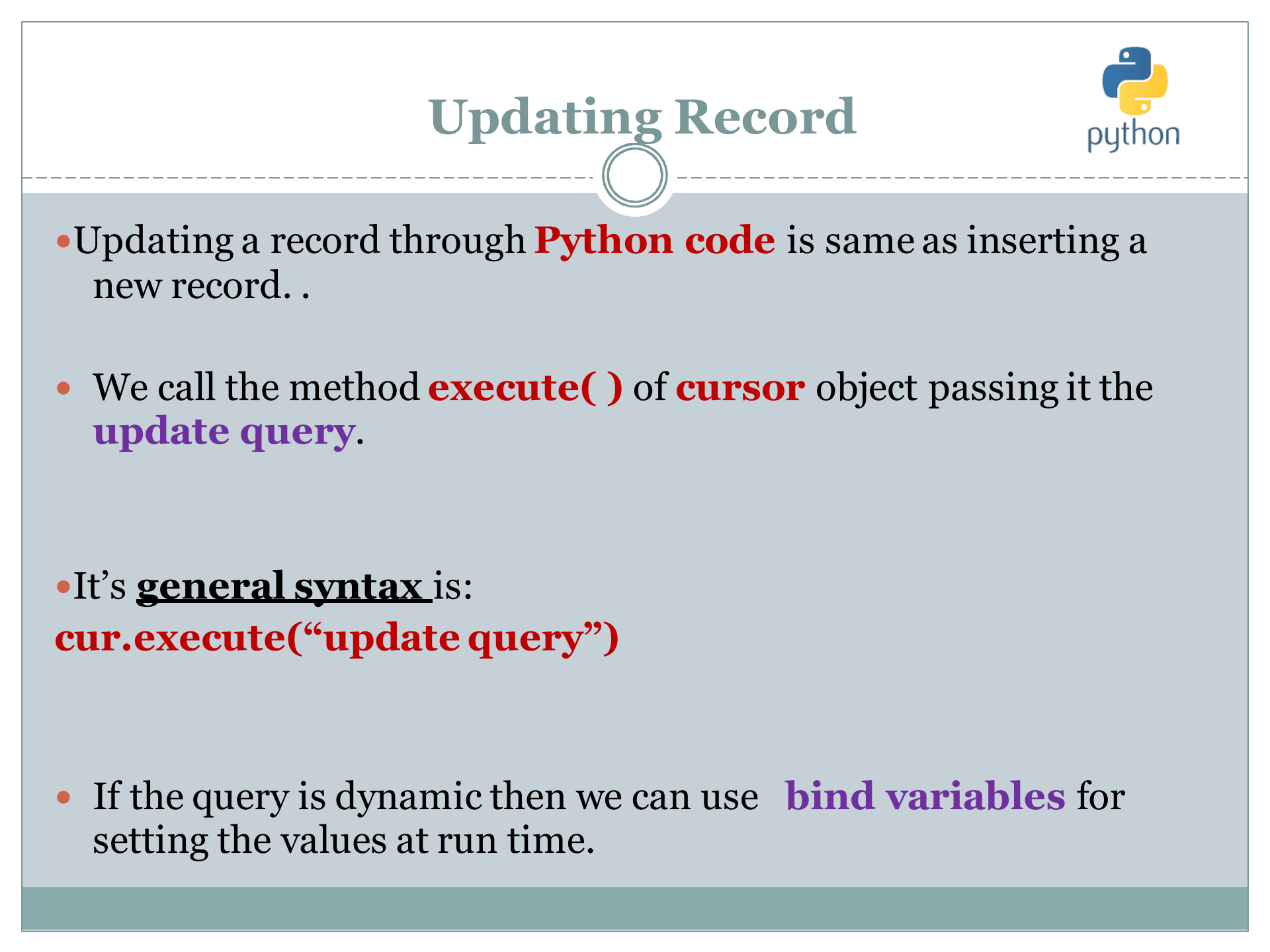


Updating Record

- To update a record in the table we have to execute the **UPDATE** command.

- It has **2 syntaxes**:
  - **Update <table name> set <col name>=<value>**
  - **Update <table name> set <col name>=<value> where <test condition>**

Updating Record



- Updating a record through **Python code** is same as inserting a new record. .
- We call the method **execute( )** of **cursor** object passing it the **update query**.
- It's **general syntax** is:

cur.execute("update query")

- If the query is dynamic then we can use **bind variables** for setting the values at run time.

Example

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqlitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Update allbooks set bookprice=500 where bookid=201 ")
    n=cur.rowcount
    print(n," row updated")
    conn.commit()
except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
        if conn is not None:
                conn.close()
                print("Disconnected successfully from the DB")
```

Exercise

- Write a program to accept a **subject name** and an **amount** from the user and **increase the price** of all the books of the **given subject** by adding the amount in the **current price**. Finally display whether books were updated or not and how many books were updated

Sample Output

Sample Run 1

```
Connected successfully to the DB
Enter subject name:Python
Enter the amount to increase:200
2 rows updated
Cursor closed successfully
Disconnected successfully from the DB
```

Sample Run 2

```
Connected successfully to the DB
Enter subject name:RoR
Enter the amount to increase:200
No rows updated
Cursor closed successfully
Disconnected successfully from the DB
```

Solution

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqlitedb/library.db")
    print("Connected successfully to the DB")
    subject=input("Enter subject name:")
    amount=int(input("Enter the amount to increase:"))
    cur=conn.cursor()
    cur.execute("Update allbooks set bookprice=bookprice+:1 where
    subject=:2",(amount,subject))
    n=cur.rowcount
    if n==0:
            print("No rows updated")
    else:
            print(n, "rows updated")
            conn.commit()

except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
            if conn is not None:
                    conn.close()
                    print("Disconnected successfully from the DB")
```

Deleting Record

- To delete a record from the table we have to execute the **DELETE** command.

- It has **2 syntaxes**:
  - **Delete from <table name>**
  - **Delete from <table name> where <test condition>**

Deleting Record

- Deleting a record through **Python code** is same as updating/inserting a record. .
- We call the method **execute( )** of **cursor** object passing it the **delete query**.
- It's **general syntax** is:

cur.execute("delete query")

- If the query is dynamic then we can use **bind variables** for setting the values at run time.

Example

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqlitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    cur.execute("Delete from allbooks where bookid=109")
    n=cur.rowcount
    print(n," row deleted")
    conn.commit()

except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
        if conn is not None:
                conn.close()
                print("Disconnected successfully from the DB")
```

Exercise

- Write a program to accept a **subject name** from the user and **delete** all the books of the **given subject .**Finally display whether books were deleted or not and how many books were deleted

Sample Output

Sample Run 1

```
Connected successfully to the DB
Enter subject name:JS
1  rows deleted
Cursor closed successfully
Disconnected successfully from the DB
```

Sample Run 2

```
Connected successfully to the DB
Enter subject name:JS
No rows deleted
Cursor closed successfully
Disconnected successfully from the DB
```

Solution

```
import sqlite3
conn=None
try:
    conn=sqlite3.connect("d:/mysqlitedb/library.db")
    print("Connected successfully to the DB")
    cur=conn.cursor()
    subject=input("Enter subject name:")
    cur.execute("Delete from allbooks where subject=:1",(subject,))
    n=cur.rowcount
    if n==0:
            print("No rows deleted")
    else:
            print(n," rows deleted")
            conn.commit()
except (sqlite3.DatabaseError)as ex:
    print("Error in connecting to Sqlite:",ex)
finally:
            if conn is not None:
                    conn.close()
                    print("Disconnected successfully from the DB")
```

PYTHON

LECTURE 52

Today's Agenda

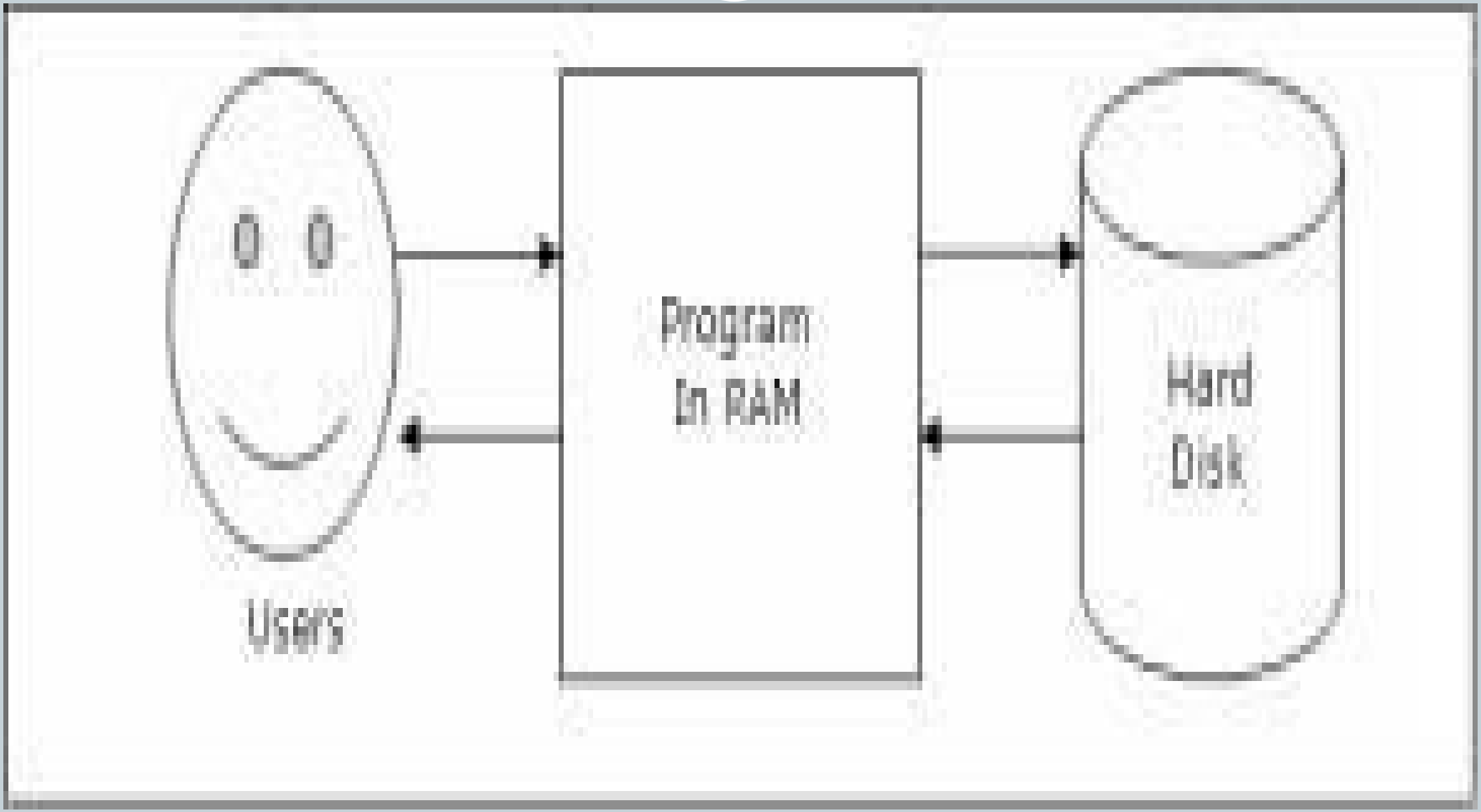
- **File Handling**
  - What Is File Handling ?
  - What Is The Need Of File Handling ?
  - Examples Where Files Are Used?
  - Python's Way Of Handling Files
  - File Opening Modes
  - Writing In A File
  - Different Ways For Reading From A File

What Is File Handling ?

- **File handling** is the process of accessing data files stored in the **secondary memory** of our computer.

- It allows us to **perform various operations** on these files **through our program** like **renaming files**, **deleting file**, **moving file** and above all **reading** and **writing** the contents in a File

What Is File Handling ?



Real Life Examples Of File Handling

- **Mobile's Phonebook**
- **Computer/Mobile Games**
- **Call Logs**
- **Gallery In Mobile**
- **User Accounts In Operating System**
- **Windows Registry**

Steps Needed For File Handling

- Broadly , file handling involves **3 steps**:
  - **Open the file.**
  - **Process file i.e perform read or write operation.**
  - **Close the file.**

Step -1 : Opening The File

- Before we can perform any operation on a file, we must open it.
- **Python** provides a **function** called **open()** to open a file.
- **Syntax:**

fileobject = open(filename, mode)

  - The **filename** is the name or path of the file.
  - The **mode** is a string which specifies the type operation we want to perform on the file (i.e **read**, **write**, **append**, etc). Default is **read**

File Opening Modes

Mode	Description
"r"	Opens the file for reading. If the file doesn't already exists we will get **FileNotFoundError** exception
"w"	Opens the file for writing. In this mode, if file specified doesn't exists, it will be created. If the file exists, then it's data is destroyed. If the path is incorrect then we will get **FileNotFoundError** exception
"a"	Opens the file in append mode. If the file doesn't exists this mode will create the file. If the file already exists then it appends new data to the end of the file rather than destroying data as "w" mode does.

File Opening Modes

Mode	Description
“r+”	Opens file for both reading and writing.
“w+”	Opens a file for both writing and reading. If the file exists then it will overwrite it otherwise it will create it.
“a+”	Opens file for appending and reading. If the file already exists then pointer will be set at the end of the file otherwise a new file will be created.

Examples Of Opening File

- **Example 1:**
  - **f = open("employees.txt", "r")**
- This statement opens the file **employees.txt** for **reading**.

- **Example 2:**
  - **f = open("teams.txt", "w")**
- This statement opens the file **teams.txt** in **write mode**.

- **Example 3:**
  - **f = open("teams.txt", "a")**
- This statement opens the file **teams.txt** in **append mode**.

Examples Of Opening File

- Instead of using **relative file paths** we can also use **absolute file paths**.

- **For example:**
  - **f = open("C:/Users/sachin/documents/README.txt", "w")**

- This statements opens the text file **README.txt** that is in **C:\Users\sachin\documents** directory in **write mode**.

Examples Of Opening File

- We can also use something called "**raw string**" by specifying **r** character in front of the **string** as follows:
  - **f = open(r"C:\Users\sachin\documents\README.txt", "w")**
- The **r** character causes the **Python** to treat every character in string as literal characters.

Step -3 : Closing The File

- Once we are done working with the file or we want to open the file in some other mode, we should close the file using **close()** method of the file object as follows:
  - **f.close()**

The TextIOWrapper Class

- The file object returned by **open()** function is an object of type **TextIOWrapper**.
- The class **TextIOWrapper** provides methods and attributes which helps us to **read** or **write** data from and to the file.
- In the next slide we have some commonly used methods of **TextIOWrapper** class.

Methods Of The TextIOWrapper Class

Method	Description
read([num])	Reads the specified number of characters from the file and returns them as string. If **num** is omitted then it reads the entire file.
readline()	Reads a single line and returns it as a string.
readlines()	Reads the content of a file line by line and returns them as a list of strings.
write(str)	Writes the string argument to the file and returns the number of characters written to the file.
seek(offset, origin)	Moves the file pointer to the given offset from the origin.
tell()	Returns the current position of the file pointer.
close()	Closes the file

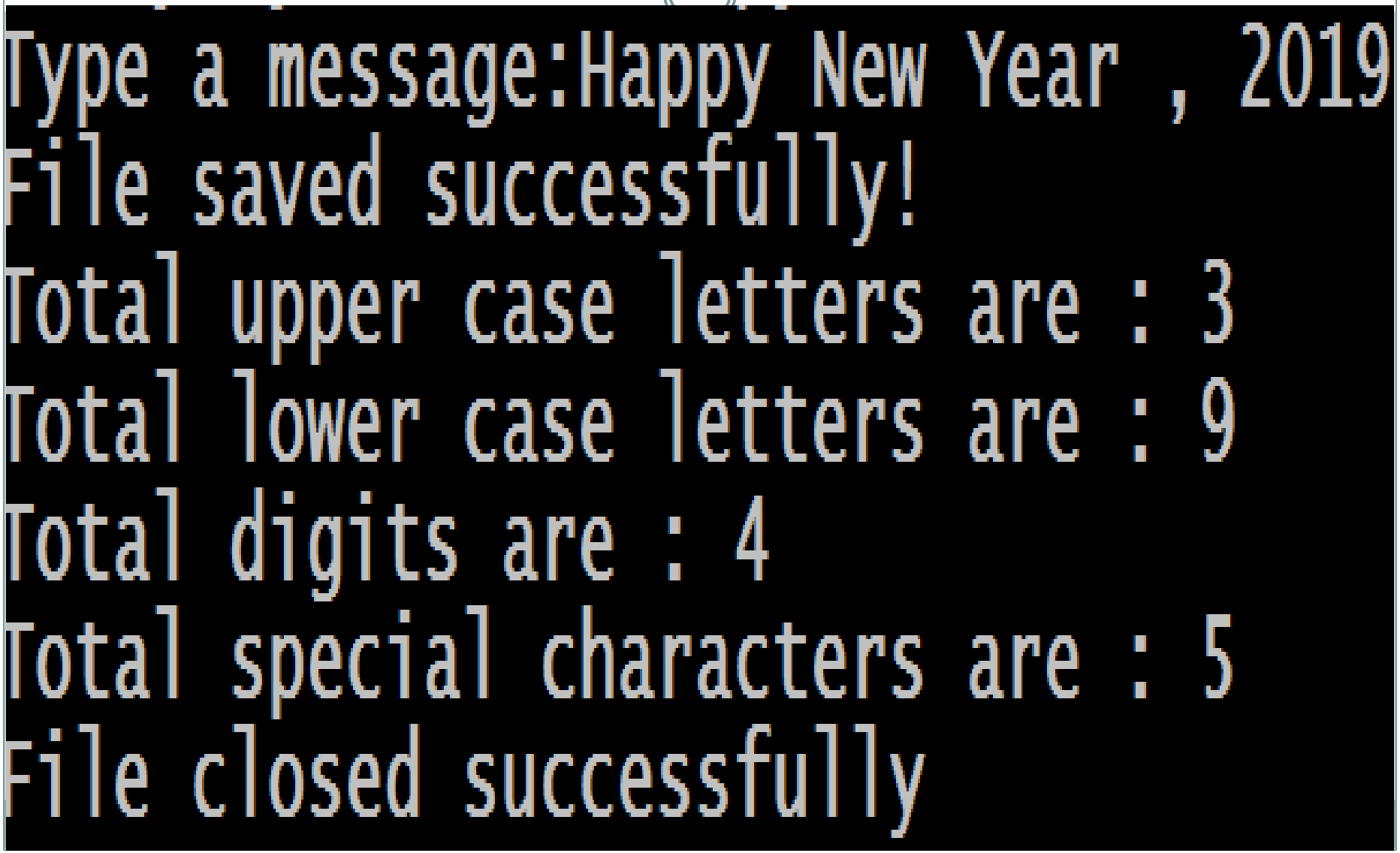
Exceptions Raised In File Handling

- **Python** generates **many exceptions** when something goes wrong while interacting with files.
- The 2 most common of them are:
  - **FileNotFoundError**: Raised when we try to open a file that doesn't exist
  - **OSError**: Raise when an operation on file cause system related error.

Exercise

- Write a program to create a file called **message.txt** in **d:** of your computer .
- Now ask the user to **type a message** and **write it** in the file .
- Finally display **how many capital letters**, **how many small letters** , **how many digits** and **how many special characters** were written in the file.
- Also properly handle every possible exception the code can throw

Sample Output



Solution

```
fout=None
try:
    fout=open("d:\\message.txt","w")
    text=input("Type a message:")
    upper=0
    lower=0
    digits=0
    for ch in text:
            fout.write(ch)
            if 65<=ord(ch)<=90:
                    upper+=1
            elif 97<=ord(ch)<=122:
                    lower+=1
            elif 48 <=ord(ch)<=57:
                    digits+=1
            print("File saved successfully!")
            print("Total upper case letters are :",upper)
            print("Total lower case letters are :",lower)
            print("Total digits are :",digits)
            print("Total special characters are :",
            len(text)-(lower+upper+digits))
except FileNotFoundError as ex:
            print("Could not create the file: ",ex)

except OSError:
            print("Some error occurred while writing")

finally:
            if not fout is None:
                    fout.close()
                    print("File closed successfully")
```

Exercise

- Write a program to open the **message.txt** created by the previous code.
- Now read and display the contents of the file.

Click to add text

- Also properly handle every possible exception the code can throw

Solution

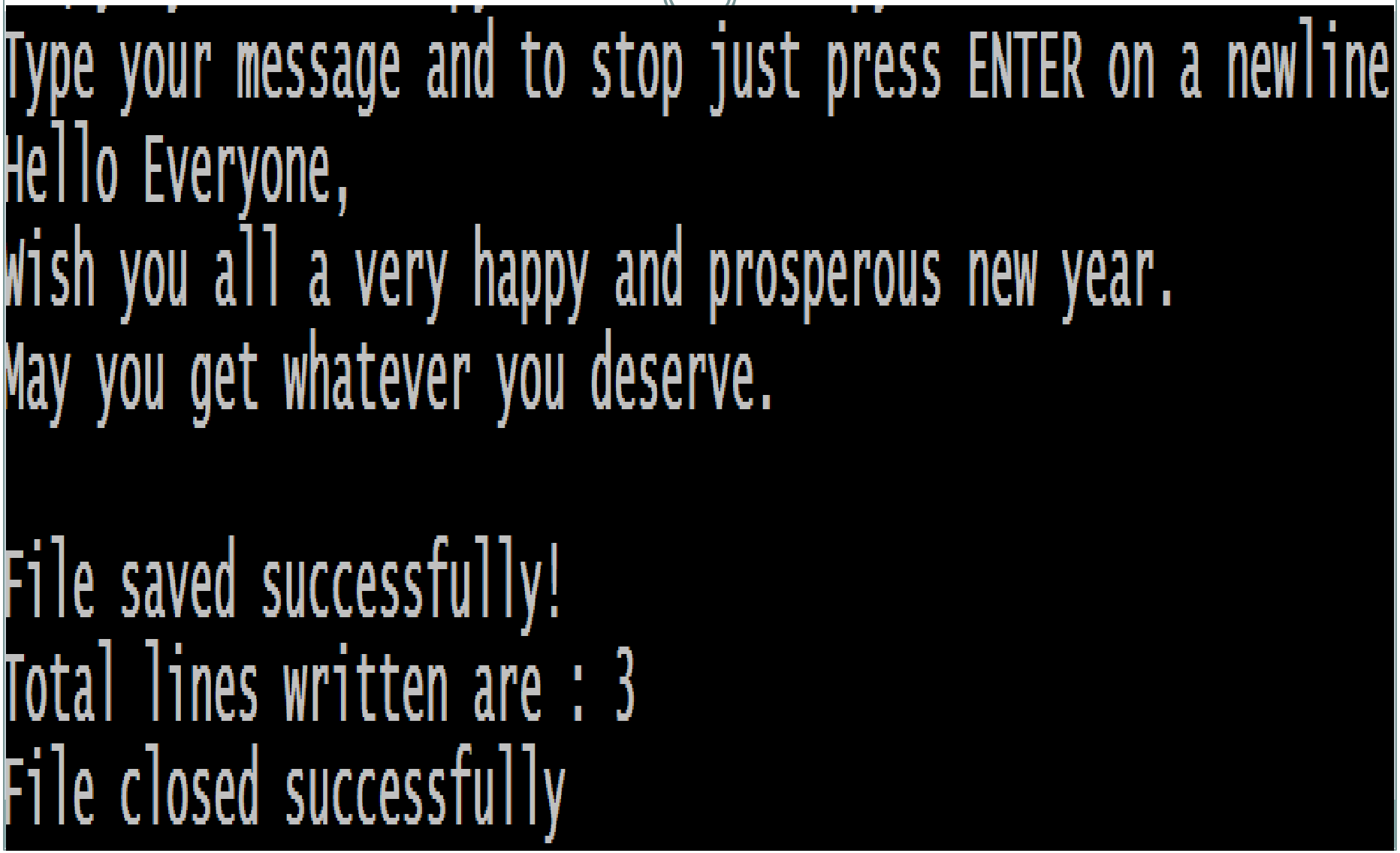
```
fin=None
try:
    fin=open("d:\\message.txt","r")
    text=fin.read()
    print(text)
except FileNotFoundError as ex:
    print("Could not open the file: ",ex)

finally:
    if  fin is not None:
            fin.close()
            print("File closed successfully")
```

```
Happy New Year , 2019
File closed successfully
```

Exercise

- Write a program to create a file called **message.txt** in **d:** of your computer .
- Now ask the user to continuously type messages and save them in the file line by line.
- Stop when the user strikes an **ENTER** key on a **new line**
- Finally display **how many lines** were written in the file.
- Also properly handle every possible exception the code can throw

Sample Output



Solution

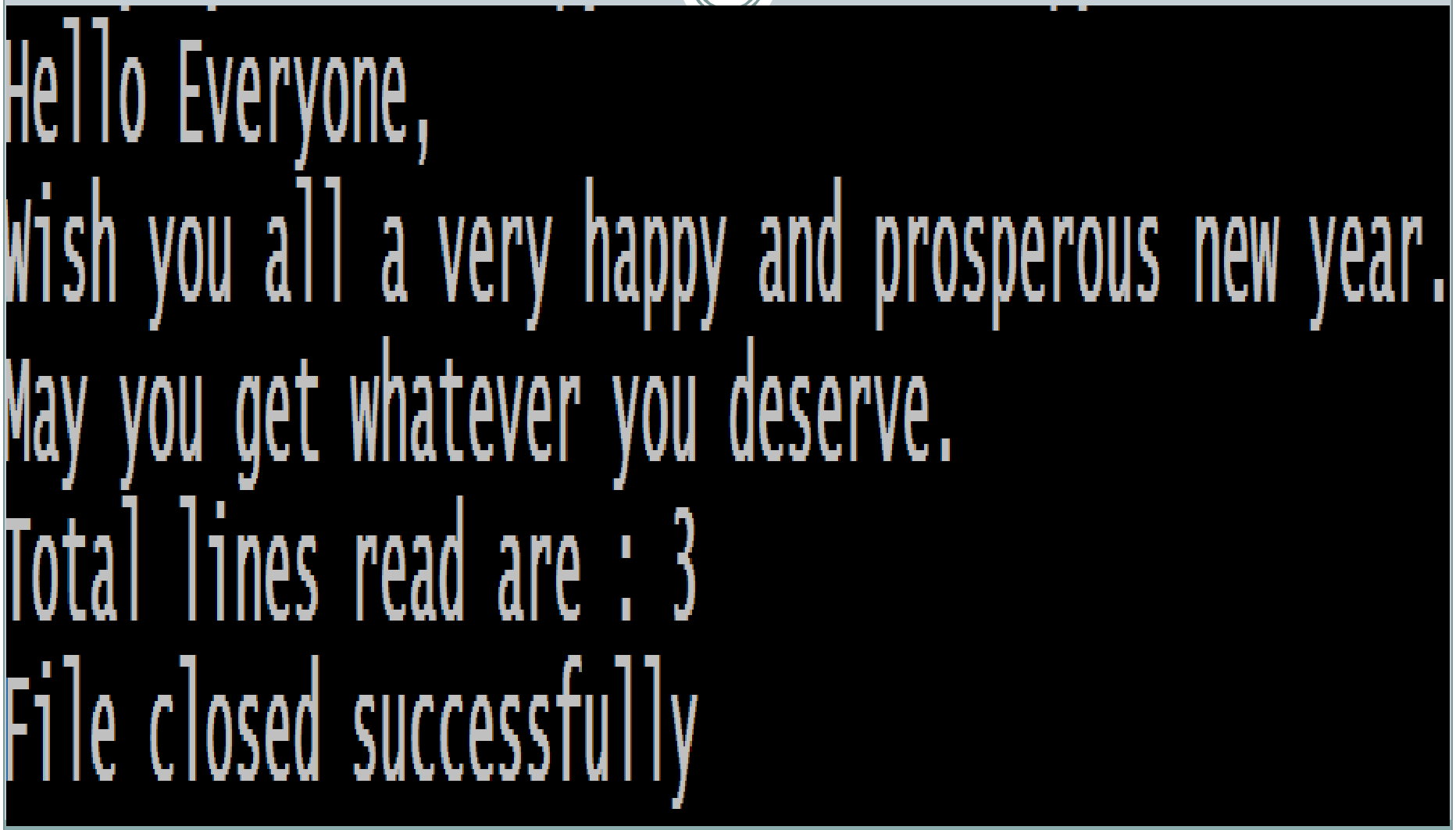
```
fout=None
try:
    fout=open("d:\\message.txt","w")
    text=input("Type your message and to stop just press ENTER on a newline\n")
    lines=0
    while True:
            if text=="":
                    break
            lines+=1
            fout.write(text+"\n")
            text=input()

    print("File saved successfully!")
    print("Total lines written are :",lines)

except FileNotFoundError as ex1:
    print("Could not create the file: ",ex1)

except OSError as ex2:
    print(ex2)

finally:
    if  fout is not None:
            fout.close()
            print("File closed successfully")
```

```
Type your message and to stop just press ENTER on a newline
Hello Everyone,
Wish you all a very happy and prosperous new year.
May you get whatever you deserve.

File saved successfully!
Total lines written are : 3
File closed successfully
```

Exercise

- Write a program to open the **message.txt** created by the previous code.
- Now **read** and **display** the contents of the **file** line by line.
- Finally also display **total number of lines** read from the file.
- Also properly handle every possible exception the code can throw

Sample Output



Solution

```
try:
    fin=open("d:\\message.txt","r")
    lines=0
    while True:
            text=fin.readline()
            if text=="":
                    break
            lines+=1
            print(text,end="")

    print("Total lines read are :",lines)

except FileNotFoundError as ex:
    print("Could not open the file: ",ex)

finally:
     if  fin is not None:
            fin.close()
            print("File closed successfully")
```

```
Hello Everyone,
Wish you all a very happy and prosperous new year.
May you get whatever you deserve.
Total lines read are : 3
File closed successfully
```

Using for Loop To Read The File

- Python allows us to use **for loop** also to read the contents of the **file line by line**.

- This is because the object of **TextIOWrapper** is also a kind of **collection/sequence** of characters fetched from the file.

- The only point is that when we use **for loop** on the **file object** , Python reads and returns **one line at a time**.

Exercise

- Write a program to open the **message.txt** created by the previous code.
- Now **read** and **display** the contents of the **file** line by line.
- Finally also display **total number of lines** read from the file.
- Also properly handle every possible exception the code can throw

Solution

```
fin=None
try:
    fin=open("d:\\message.txt","r")
    lines=0
    for text in fin:
            print(text,end="")
            lines+=1
    print("Total lines read are :",lines)

except FileNotFoundError as ex:
    print("Could not open the file: ",ex)

finally:
    if  fin is not None:
            fin.close()
            print("File closed successfully")
```

```
Hello Everyone,
Wish you all a very happy and prosperous new year.
May you get whatever you deserve.
Total lines read are : 3
File closed successfully
```

PYTHON

LECTURE 53

Today's Agenda

- **File Handling**
  - The seek( ) Method
  - Appending In A File
  - Using with Statement

The seek( ) Method

- To reset the internal file pointer we use the **seek()** method.
- **Syntax:**
  seek(offset, whence).
- **whence** is optional, and determines where to seek from.
  - If whence is 0, the bytes/letters are counted from the beginning.
  - If it is 1, the bytes are counted from the current cursor position.
  - If it is 2, then the bytes are counted from the end of the file.
  - If nothing is put there, 0 is assumed.
- **offset** describes how far from whence that the cursor moves.

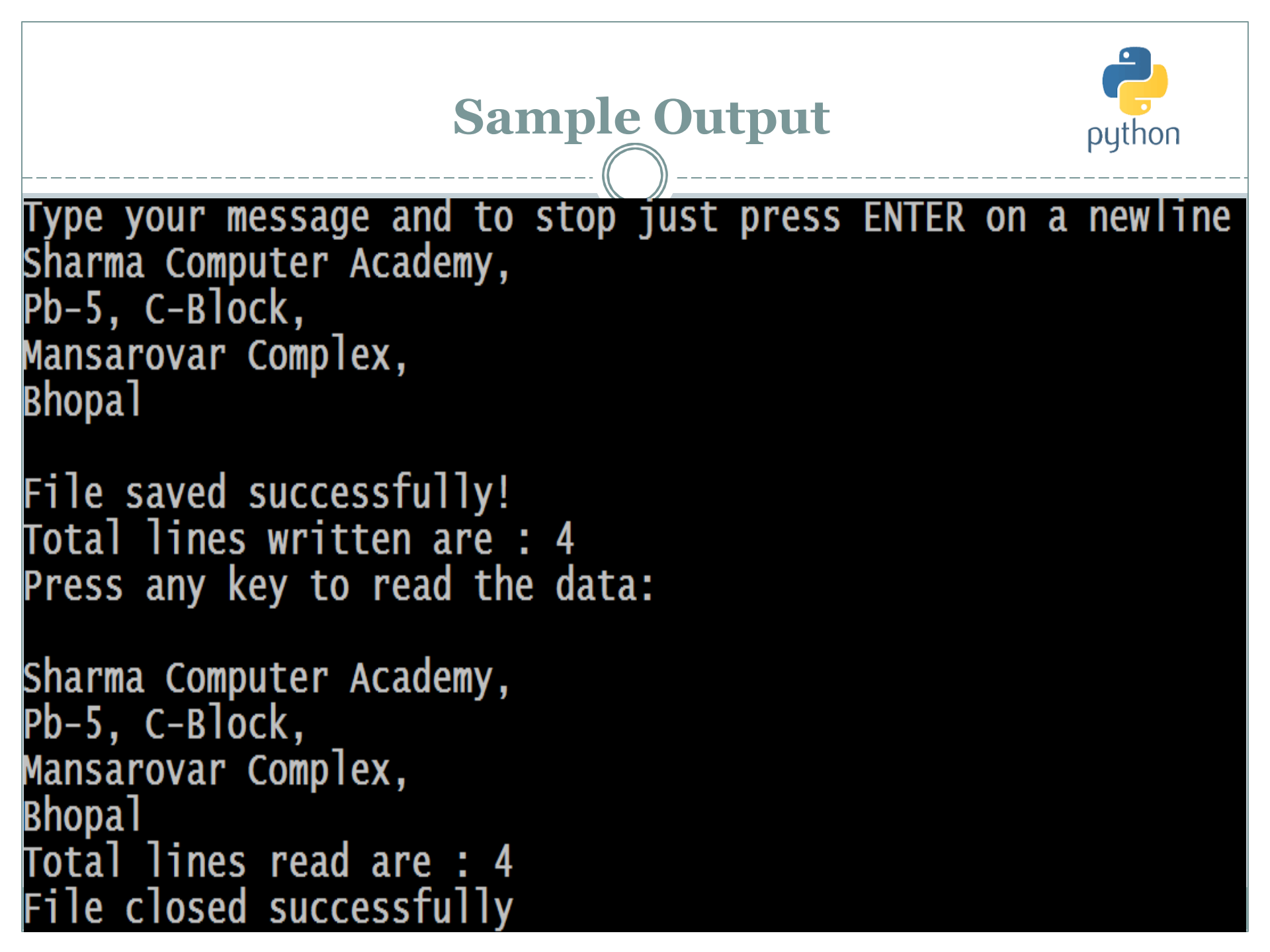
seek( ) Examples

- **f.seek(45,0)**
  - would move the cursor to **45** bytes/letters after the **beginning** of the file.
- **f.seek(10,1)**
  - would move the cursor to **10** bytes/letters after the **current** cursor position.
- **f.seek(-77,2)**
  - would move the cursor to **77** bytes/letters before the **end** of the file (notice the - before the 77)
- Special Note:
  - **Python 3** only supports **text file** seeks from the beginning of the file.
  - If we try to call **seek()** with **non-zero value** for offset from **cur** or **end** then the code will throw an exception called:
    - **UnsupportedOperation: can't do nonzero end-relative seeks**
  - Thus for **seek()** to work , the file must be a **binary file like** image file, music file etc.

Exercise

- Write a program to create the file **message.txt** in **d:** of your computer
- Now ask the user to continuously type messages and save them in the file line by line.
- Stop when the user strikes an ENTER key on a new line
- Finally display **how many lines** were written in the file.
- Now read and display the contents of the file line by line.
- Finally also display total number of lines read from the file.
- Also properly handle every possible exception the code can throw

Sample Output



Solution

```
try:
    fobj=open("d:\\message.txt","w+")
    text=input("Type your message and to stop just press ENTER on a newline\n")
    lines=0
    while True:
            if text=="":
                    break
            lines+=1
            fobj.write(text+"\n")
            text=input()

    print("File saved successfully!")
    print("Total lines written are :",lines)
    print("Press any key to read the data:")
    input()
    fobj.seek(0)
    lines=0
```

Solution

```
while True:
        text=fobj.readline()
        if text=="":
                break
        lines+=1
        print(text,end="")

    print("Total lines read are :",lines)

except FileNotFoundError as ex1:
    print("Could not open the file: ",ex1)

except OSError as ex2:
    print("Error in file I/O: ",ex2)

finally:
    if 'fobj' in globals():
        fobj.close()
        print("File closed successfully")
```

Appending Data In a File

- We can use **"a"** mode to **append** data to end of the **file**.
- When we open the file in **“a”** or **“a+”** mode , the internal **file pointer** is placed **at the end** while **writing** the new data.
- Thus , the new text we write **does not overwrite** the previous contents of the file.
- Moreover if the file is **not present** it gets **created**.

Example

```
days=["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"]
fobj=None
try:
    fobj=open("d:\\days.txt","a")
    items=0
    for day in days[:3]:
            fobj.write(day+"\n")
            items+=1
    fobj.close()
    print(items," values written")
    print("File closed")
    items=0
    fobj=open("d:\\days.txt","a")
    for day in days[3:]:
            fobj.write(day+"\n")
            items+=1
    fobj.close()
    print(items," more values written")
    print("File closed")
    print("Press any key to read back the file")
    input()
```

```
3  values written
File closed
4  more values written
File closed
Press any key to read back the file
```

Example

```
    fobj=open("d:\\days.txt","r")
    for day in fobj:
            print(day,end="")

except FileNotFoundError as ex:
    print("Could not open the file: ",ex)

finally:
    if  fobj is not None:
            fobj.close()
            print("File closed successfully")
```

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
File closed successfully
```

Using with Statement

- **Python** also provides a nice shortcut for **file handling** using the **with** statement.

- The following is the general form of the **with** statement when used with files.

```
with open(filename, mode) as file_object:
        # body of with statement
        # perform the file operations here
```

Important Points About **with** Statement

- It automatically closes the file without requiring any work on our part.
- The scope of **file_object** variable is only limited to the body of the with statement.
- So , if we try to call **read()** or **write()** method on it outside the block we will get **NameError**.

Example

```
try:
    with open("d:\\message.txt","r") as fin:
            lines=0
            for x in fin:
                    print(x,end="")
                    lines+=1
    print("Total lines read are :",lines)
    print("File closed successfully!")
except FileNotFoundError as ex:
    print("Could not open the file: ",ex)
```

PYTHON

LECTURE 54

Today's Agenda

- **Modules**
  - What is a module ?
  - Different ways of importing module
  - The concept of the variable **__name__**

What Is A **Module** ?

- Python **modules** are **.py** files that consist of **Python code**.

- So all the **Python programs** that we have written can be called a **module**.

From Where These **Modules** Come ?

- Modules are available in **3 ways**:
  - Some modules are available through the **Python Standard Library** and are therefore installed with our Python installation. Examples of such modules are: **math**, **sys**,**random** etc
  - We also can install modules developed by other programmers using **Python's package manager** called **pip**.
  - Additionally, we can create **our own Python modules** since modules are comprised of Python **.py** files.

What A **Module** Can Contain ?

- A **Python** module can contain any valid **Python** element like:
  - **variables**
  - **functions**
  - **classes**

Defining A Module

```
def greet(name):
    print("Good Morning ",name,"!")
```

- Save the above code in a file called **welcome.py**

welcome.py

```
def greet(name):
   print("Good Morning ",name,"!")
```

Using A Module

- To use a module we must **import** it in our program which can be done in **4** ways:
  - Using import
  - Using aliasing
  - Using from
  - Using wildcard

First Way
(Direct import)

- Create another **Python** file called **test.py** and import the module **welcome.py** in it.

- Now we can call the function **greet()** by using the syntax as shown below:

- **Syntax:**

welcome.greet(<arg>)

First Way
(Direct import)

test.py

```
import welcome
name=input("What is your name?")
welcome.greet(name)
```

Run:

python test.py

Output:

What is your name?Sachin

Good Morning Sachin !

Second Way
(Using aliasing)

- In the previous example we had to prefix the name of the module called **welcome** before the name of it's function **greet()**

- In order to shorten this syntax , **Python** allows us to **alias** the module name using the keyword **as**

- **Syntax:**

import <module_name> as <new_name>

Second Way
(Using aliasing)

test.py

```
import welcome as w
name=input("What is your name?")
w.greet(name)
```

Run:

python test.py

Output:

What is your name?Sachin

Good Morning Sachin !

Guess The Output ?

test.py

```
import welcome as w
name=input("What is your name?")
welcome.greet(name)
```

Run:

python test.py

Output:

What is your name?Sachin

NameError: name 'welcome' is not defined

Guess The Output ?

test.py

```
import welcome
import welcome as w
name=input("What is your name?")
welcome.greet(name)
w.greet(name)
```

Run:

python test.py

Output:

What is your name?Sachin
Good Morning Sachin!
Good Morning Sachin!

Third Way

- If we do not want to prefix the module name at all with any prefix then we must **import** specific members of a **module**
- To do this , **Python** provides us **from keyword**
- **Syntax:**
  - **from modname import name1[, name2[, ... nameN]]**

Third Way

test.py

```
from welcome import greet
name=input("What is your name?")
greet(name)
```

Run:

python test.py

Output:

What is your name?Sachin

Good Morning Sachin !

Guess The Output ?

m1.py

```
def greet(name):
    print("Good Morning ",name,"!"))
```

m2.py

```
def greet(name):
    print("Good Evening ",name,"!"))
```

test.py

```
from m1 import greet
from m2 import greet
name=input("What is your name?")
greet(name)
```

Execution

Run:

python test.py

Output:

What is your name?Sachin

Good Evening Sachin !

Guess The Output ?

m1.py

```
def greet(name):
    print("Good Morning ",name,"!"))
```

m2.py

```
def greet(name):
    print("Good Evening ",name,"!"))
```

test.py

```
from m1 import greet
from m2 import greet
name=input("What is your name?")
m1.greet(name)
```

Execution

Run:

python test.py

Output:

What is your name?Sachin

NameError: name 'm1' is not defined

Guess The Output ?

welcome.py

```
def greet1(name):
    print("Good Morning ",name,"!")

def greet2(name):
    print("Good Afternoon ", name, "!")
```

test.py

```
from welcome import greet1
name=input("What is your name?")
greet1(name)
greet2(name)
```

Execution

Run:

python test.py

Output:

What is your name?Sachin

Good Morning Sachin !

NameError: name 'greet2' is not defined

Fourth Way

- It is also possible to **import all names** from a **module** into the current file by using the **wildcard character** *

- **Syntax:**
  - **from modname import ***

- This allows us to use all the items from a **module** into the **current file**

Fourth Way

welcome.py

```
def greet1(name):
    print("Good Morning ",name,"!")

def greet2(name):
    print("Good Afternoon ", name, "!")
```

test.py

```
from welcome import *
name=input("What is your name?")
greet1(name)
greet2(name)
```

Execution

Run:

python test.py

Output:

What is your name?Sachin
Good Morning Sachin !
Good Afternoon Sachin !

Defining A Class In A Module

- It is also possible to define a **class** inside a **module** containing **attributes** and **methods**

- It can be accessed in the same way like we can access functions

Defining A Class In A Module

shape.py

```
import math
class Circle:
    def __init__(self,radius):
        self.radius=radius
    def area(self):
        print("Area is",math.pi*math.pow(self.radius,2))
```

Using The Class Outside The Module

useshape.py

```
import shape
radius=int(input("Enter radius:"))
obj=shape.Circle(radius)
obj.area()
```

Run:

python useshape.py

Output:

Enter radius:3
Area is 28.274333882308138

The Concept Of __name__=='__main__'

- Now after learning basics of modules , let us discuss another very important concept in **Python**.

- All programmers who write **standard code** in **Python** always include a test condition in their module which is something like :

```
if __name__=='__main__':
    Some code
```

- So let us understand what it is all about

The Concept Of __name__=='__main__'

- Before we start the discussion , guess the output of the following code:

calculate.py

```
def add(a, b):
    print("Sum of",a,"and",b,"is",a+b)


def subtract(a, b):
    print("Diff of", a, "and", b, "is", a - b)
```

Run:

python calculate.py

Output:

Why did the code not produce any output?

This is because add() and subtract() are functions and functions are executed only when they are called .

But since we didn't call any of these functions so no output was produced

The Concept Of __name__=='__main__'

- Now , guess the output:

calculate.py

```
def add(a, b):
    print("Sum of",a,"and",b,"is",a+b)


def subtract(a, b):
    print("Diff of", a, "and", b, "is", a - b)

add(10,20)
subtract(10,20)
```

Run:

python calculate.py

Output:

Sum of 10 and 20 is 30

Diff of 10 and 20 is -10

As expected the code is showing the results of both add() and subtract() functions as we have called both of them

The Concept Of __name__=='__main__'

- Now suppose we **import** the previous **calculate** module in another file and call only the **add()** function.

- **Can you guess the output then ?**

Guess The Output

calculate.py

```
def add(a, b):
   print("Sum of",a,"and",b,"is",a+b)

def subtract(a, b):
   print("Diff of", a, "and", b, "is", a - b)

add(10, 20)
subtract(10, 20)
```

test.py

```
import calculate
calculate.add(5,7)
```

Execution

Run:

python test.py

Output:

Sum of 10 and 20 is 30
Diff of 10 and 20 is -10
Sum of 5 and 7 is 12

Why 3 Outputs ?

- Whenever we import a **module** , Python immediately executes it.
- So as soon as **Python** found the statement:

  import calculate

- It imported the module **calculate** and executed all the **global statements** in it.
- And since our module **calculate** contained 2 function calls **add(10,20)** and **subtract(10,20)** so the overall output contains outputs of these calls also

How To Overcome This Problem ?

- This is where the concept of a special variable called **__name__** comes into picture.
- For every module , **Python** creates a variable called **__name__**, which contains the **name of the module**.
- But the point to understand is that:
  - **When we run a module as a stand alone file then this variable has the value "__main__"**
  - **Otherwise this variable has the name of the file as it's value**

The Concept Of __name__=='__main__'

- To understand the concept **look carefully** at the output of the code:

demo.py

```
print("demo.py module name is:",__name__)
```

Run:

python demo.py

Output:

demo.py module name is: __main__

Since we are running the file demo.py as a stand alone module so the variable __name__has been set to the value "__main__"

The Concept Of __name__=='__main__'

- Now consider the code below and it's output:

demo.py

```
print("demo.py module name is:",__name__)
```

sample.py

```
import demo
```

Run:

python sample.py

Output:

demo.py module name is: demo

Now the output is different because we are importing the module demo in another module.

As mentioned previously in this case the value of the variable __name__ is the name of the file so the output is demo

The Concept Of __name__=='__main__'

- Now , we can solve our problem , so that if we run the module **calculate** as a stand alone module then both the functions , **add()** and **subtract()** , should run and if we run it as a part of another module then only that function should run which we have called.

Guess The Output

calculate.py

```
def add(a, b):
    print("Sum of",a,"and",b,"is",a+b)

def subtract(a, b):
    print("Diff of", a, "and", b, "is", a - b)

if __name__=="__main__":
    add(10, 20)
    subtract(10, 20)
```

test.py

```
import calculate
calculate.add(5,7)
```

Execution

Run:

python test.py

Output:

Sum of 5 and 7 is 12