INDEX

UNIT -1	**Introduction to SQLite**	1
	SQLite History	2
	Differences between SQL and SQLite	3
	Install SQLite on Windows	3
	SQLite Features	5
	SQLite Advantages	5
	SQLite Disadvantages	7
	SQLite Commands	7
	Data Definition Language	7
	Data Manipulation language	7
	Data Query Language	8
	SQLite data types	8
	SQLite Storage Classes	8
	SQLite Affinity Type	9
	SQLite manifest typing & type affinity	9
	SQLite – Transactions	10
	Properties of Transactions	10
	BEGIN TRANSACTION Command	11
	SQLite COMMIT Command	11
	SQLite ROLLBACK Command	12
	Data Filtering	12
	SQLite DISTINCT Clause	12
	SQLite WHERE Clause	14
	SQLite BETWEEN Clause	15
	SQLite IN Clause	16
	SQLite NOT IN Clause	16
	SQLite LIKE Clause	17

Scanned with OKEN Scanner

	SQLite - UNION Clause	18
	SQLite Intersect Clause	19
	SQLite Except	20
	SQLite LIMIT Clause	21
	SQLite GROUP BY Clause	22
	SQLite HAVING Clause	23
	SQLite CASE	23
	SQLite joins	25
	SQLite Inner Join	25
	SQLite Outer Join	25
	SQLite Left Outer Join	26
	The CROSS JOIN	26
	SQLite Trigger	27
	Usage of Triggers	27
	Advantages of using triggers	28
	SQLite CREATE TRIGGER	28
	SQLite DROP TRIGGER	30
	Exercise	30
UNIT - 2	**Database backup and CSV handling**	31
	SQLite dump	32
	Dump the entire database into a file	32
	Dump a specific table using the dump command	32
	Dump tables structure	33
	Dump data of one or more tables into a file	33
	2.2 CSV files handling	34
	Import a CSV file into a table	34
	Export a CSV file from table	35
	Exercise	37

Scanned with OKEN Scanner

UNIT - 3	**Python interaction with SQLite**	39
	Module	40
	Create a Module	40
	Loading the module	40
	The import statement	40
	The from-import statement	41
	PYTHONPATH	42
	Concepts of Namespace	44
	Scope of Namespace	45
	Packages in Python	46
	Importing sqlite3 module	47
	connect(), execute(),fetchone() and fetchall()	47
	Create a table using python	50
	Insert Records, Select Records and Delete Records	51
	commit()	53
	Exercise	54
UNIT - 4	**Python Interaction with Text and CSV**	55
	Opening a file	56
	close() method	58
	Reading and Writing Files	58
	The with statement	60
	CSV module	61
	Open a CSV File	61
	csv .reader(), csv.writer(), Writerows(), DictReader(), DictWriter()	62
	Close a CSV File	68
	Dataframe Handling using Panda and Numpy	69
	Python Pandas library	69
	Series	69

Scanned with OKEN Scanner

	DataFrame	70
	Operation on DataFrame	73
	head, tail, loc, iloc, value, to_numpy(), describe()	74
	Performing Queries on Data	81
	Sorting the Data	81
	Handling Missing Data	82
	Python NumPy	83
	NumPy Environment Setup	84
	NumPy Ndarray	85
	Numpy Methods : mean, median, mode, variance, Standard Deviation	87
	Exercise	92
UNIT - 5	**Data Visualization using data frame**	93
	Data Visualization	94
	Matplotlib Python Libraries	94
	install matplotlib	94
	Matplotlib.pyplot.legend()	96
	Subplots	98
	Scatter Plot in Matplotlib	99
	Line chart in Matplotlib	100
	Plot multiple lines in a single chart	101
	Plot Histogram in Matplotlib	102
	Bar Plot in Matplotlib	105
	Pie chart in Matplotlib	106
	Save Plot as a File	107
	Exercise	110

Scanned with OKEN Scanner

Unit - 1

Introduction to SQLite

Scanned with OKEN Scanner

INTRODUCTION

- SQLite is embedded relational database management system. It is self-contained, serverless, zero configuration and transactional SQL database engine.
- SQLite is free to use for any purpose commercial or private. In other words, "SQLite is an open source, zero-configuration, self-contained, stand alone, transaction relational database engine designed to be embedded into an application".
- SQLite is different from other SQL databases because unlike most other SQL databases, SQLite does not have a separate server process.
- It reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file.
- SQLite is written in ANSI-C and provides simple and easy-to-use API.
- SQLite is available on UNIX (Linux, Mac OS-X, Android, iOS) and Windows (Win32, WinCE, WinRT).

SQLite History

SQLite was designed originally on August 2000. It is named SQLite because it is very light weight (less than 500Kb size) unlike other database management systems like SQL Server or Oracle.

- **2000** - D. Richard Hipp designed SQLite for the purpose of no administration required for operating a program.
- **2000** - In August, SQLite 1.0 released with GNU Database Manager.
- **2011** - Hipp announced to add UNQl interface to SQLite DB and to develop UNQLite (Document oriented database).

Scanned with OKEN Scanner

Differences between SQL and SQLite

SQL	SQLite
• SQL is a Structured Query Language used to query a Relational Database System. It is written in C language.	• SQLite is an Embeddable Relational Database Management System which is written in ANSI-C.
• SQL is a standard which specifies how a relational schema is created, data is inserted or updated in the relations, transactions are started and stopped, etc.	• SQLite is file-based. It is different from other SQL databases because unlike most other SQL databases, SQLite does not have a separate server process.
• Main components of SQL are Data Definition Language(DDL) , Data Manipulation Language(DML), Embedded SQL and Dynamic SQL.	• SQLite supports many features of SQL and has high performance and does not support stored procedures.
• SQL is Structured Query Language which is used with databases like MySQL, Oracle, Microsoft SQL Server, IBM DB2, etc. It is not a database itself.	• SQLite is a portable database resource. You have to get an extension of SQLite in whatever language you are programming in to access that database. You can access all of the desktop and mobile applications.
• A conventional SQL database needs to be running as a .service like OracleDB to connect to and provide a lot of functionalities.	• SQLite database system doesn't provide such functionalities.
• SQL is a query language which is used by different SQL databases. It is not a database itself.	• SQLite is a database management system itself which uses SQL.

Install SQLite on Windows

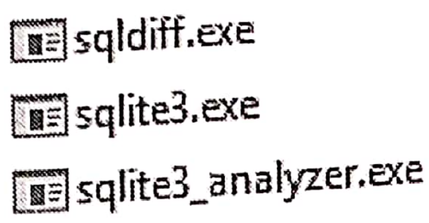
First, go to the https://www.sqlite.org website.

- **Step 1 –** Go to SQLite download page, "Download link :https://www.sqlite.org/download.html " and download precompiled binaries from Windows section.

Precompiled Binaries for Windows

sqlite-dll-win32-x86-3350500.zip (538.88 KiB)	32-bit DLL (x86) for SQLite version 3.35.5. (sha3: 55b49ce165984865d62918cbd0ad34fa7af82820e4f5d10c809e2191dc63877c)
sqlite-dll-win64-x64-3350500.zip (879.80 KiB)	64-bit DLL (x64) for SQLite version 3.35.5. (sha3: 15ba94e68c6596bb292ba6a4ce303f5beaf731754846d01fd7a55a6edf11c4d5)
sqlite-tools-win32-x86-3350500.zip (1.81 MiB)	A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff.exe program, and the sqlite3_analyzer.exe program. (sha3: 295214b0c3d6bfef32400632fc5237e1cb57c07ba0b76e94c98fb0c3fe92075c)

- **Step 2** – Download sqlite-shell-win32-*.zip and sqlite-dll-win32-*.zip zipped files.
- **Step 3** – Create a folder C:\>sqlite and unzip above two zipped files in this folder, which will give you sqlite3.def, sqlite3.dll and sqlite3.exe files.



- **Step 4** – Add C:\>sqlite in your PATH environment variable and finally go to the command prompt and issue sqlite3 command, which should display the following result.

Open the command line window, and navigate to the C:\sqlite folder. Now type sqlite3 and press enter, you should see the following output

```
C:\sqlite>sqlite3
SQLite version 3.35.5 2021-04-19 18:32:05
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

.help command from the sqlite> prompt to see all available commands in sqlite3. quit the sqlite>, you use .quit command.

SQLite Features

- **SQLite is totally free:** SQLite is open-source. So, no license is required to work with it.
- **SQLite is server less:** SQLite doesn't require a different server process or system to operate.
- **SQLite is very flexible:** It facilitates you to work on multiple databases on the same session on the same time.
- **Configuration Not Required:** SQLite doesn't require configuration. No setup or administration required.
- **SQLite is a cross-platform DBMS:** You don't need a large range of different platforms like Windows, Mac OS, Linux, and Unix. It can also be used on a lot of embedded operating systems like Symbian, and Windows CE.
- **Storing data is easy:** SQLite provides an efficient way to store data.
- **Variable length of columns:** The length of the columns is variable and is not fixed. It facilitates you to allocate only the space a field needs. For example, if you have a varchar(200) column, and you put a 10 characters' length value on it, then SQLite will allocate only 20 characters' space for that value not the whole 200 space.
- **Provide large number of API's:** SQLite provides API for a large range of programming languages. **For example:** .Net languages (Visual Basic, C#), PHP, Java, Objective C, Python and a lot of other programming language.

SQLite Advantages

SQLite is a very popular database which has been successfully used with on disk file format for desktop applications.

There are a lot of advantages to use SQLite as an application file format:

Lightweight

- SQLite is a very light weighted database so, it is easy to use it as an embedded software with devices like televisions, Mobile phones, cameras, home electronic devices, etc.

Scanned with OKEN Scanner

SQLite Features

- **SQLite is totally free:** SQLite is open-source. So, no license is required to work with it.
- **SQLite is server less:** SQLite doesn't require a different server process or system to operate.
- **SQLite is very flexible:** It facilitates you to work on multiple databases on the same session on the same time.
- **Configuration Not Required:** SQLite doesn't require configuration. No setup or administration required.
- **SQLite is a cross-platform DBMS:** You don't need a large range of different platforms like Windows, Mac OS, Linux, and Unix. It can also be used on a lot of embedded operating systems like Symbian, and Windows CE.
- **Storing data is easy:** SQLite provides an efficient way to store data.
- **Variable length of columns:** The length of the columns is variable and is not fixed. It facilitates you to allocate only the space a field needs. For example, if you have a varchar(200) column, and you put a 10 characters' length value on it, then SQLite will allocate only 20 characters' space for that value not the whole 200 space.
- **Provide large number of API's:** SQLite provides API for a large range of programming languages. **For example:** .Net languages (Visual Basic, C#), PHP, Java, Objective C, Python and a lot of other programming language.

SQLite Advantages

SQLite is a very popular database which has been successfully used with on disk file format for desktop applications.

There are a lot of advantages to use SQLite as an application file format:

Lightweight

- SQLite is a very light weighted database so, it is easy to use it as an embedded software with devices like televisions, Mobile phones, cameras, home electronic devices, etc.

Better Performance

- Reading and writing operations are very fast for SQLite database. It is almost 35% faster than File system.
- It only loads the data which is needed, rather than reading the entire file and hold it in memory.
- If you edit small parts, it only overwrite the parts of the file which was changed.

No Installation Needed

- SQLite is very easy to learn. You don't need to install and configure it. Just download SQLite libraries in your computer and it is ready for creating the database.

Reliable

- It updates your content continuously so, little or no work is lost in a case of power failure or crash.
- SQLite is less bugs prone rather than custom written file I/O codes.
- SQLite queries are smaller than equivalent procedural codes so, chances of bugs are minimal.

Portable

- SQLite is portable across all 32-bit and 64-bit operating systems and big- and little-endian architectures.
- Multiple processes can be attached with same application file and can read and write without interfering each other.
- It can be used with all programming languages without any compatibility issue.

Accessible

- SQLite database is accessible through a wide variety of third-party tools.
- SQLite database's content is more likely to be recoverable if it has been lost. Data lives longer than code.

Scanned with OKEN Scanner

Reduce Cost and Complexity

- It reduces application cost because content can be accessed and updated using concise SQL queries instead of lengthy and error-prone procedural queries.
- SQLite can be easily extended in in future releases just by adding new tables and/or columns. It also preserves the backwards compatibility.

SQLite Disadvantages

- SQLite is used to handle low to medium traffic HTTP requests.
- Database size is restricted to 2GB in most cases.

SQLite Commands

SQLite commands are similar to SQL commands. There are three types of SQLite commands:

- **DDL:** Data Definition Language
- **DML:** Data Manipulation Language
- **DQL:** Data Query Language

Data Definition Language

There are three commands in this group:

CREATE: This command is used to create a table, a view of a table or other object in the database.

ALTER: It is used to modify an existing database object like a table.

DROP: The DROP command is used to delete an entire table, a view of a table or other object in the database.

Data Manipulation language

There are three commands in data manipulation language group:

INSERT: This command is used to create a record.

UPDATE: It is used to modify the records.

DELETE: It is used to delete records.

Data Query Language

SELECT: This command is used to retrieve certain records from one or more table.

SQLite data types

Data types in SQLite are different compared to other database management system. In SQLite, you can declare data types normally, but you still can store any value in any data type. SQLite is typed less. There are no data types, you can store any type of data you like in any column. This is called dynamic types.

In static types, like in other database management systems, if you declared a column with a data type integer, you can only insert values of data type integer. However, in dynamic types like in SQLite, the type of the column is determined by the value inserted. And then SQLite stores that value depending on its type.

SQLite Storage Classes

In SQLite there are different storage methods depending on the type of value, these different storage methods are called storage classes in SQLite.

The following are the storage classes available in SQLite:

- **NULL** – this storage class is used to store any NULL value.
- **INTEGER** – any numeric value is stored as a signed integer value (It can hold both positive and negative integer values). The INTEGER values in SQLite are stored in either 1, 2, 3, 4, 6, or 8 bytes of storage depending on the value of the number.
- **REAL** – this storage class is used to store the floating point values, and they are stored in an 8-bytes of storage.
- **TEXT** – stores text strings. It also supports different encoding like UTF-8, UTF-16 BE, or UTF-26LE.
- **BLOB** – used to store large files, like images or text files. The value is stored as byte array the same as the input value.

SQLite Affinity Type

SQLite supports the concept of type affinity on columns. Any column can still store any type of data but the preferred storage class for a column is called its affinity. Each table column in SQLite3 database is assigned one of the following type affinities:

TEXT

This column stores all data using storage classes NULL, TEXT or BLOB.

NUMERIC

This column may contain values using all five storage classes.

INTEGER

Behaves the same as a column with NUMERIC affinity, with an exception in a CAST expression.

REAL

Behaves like a column with NUMERIC affinity except that it forces integer values into floating point representation.

NONE

A column with affinity NONE does not prefer one storage class over another and no attempt is made to coerce data from one storage class into another.

BOLEAN

SQLite does not have a separate Boolean storage class. Instead, Boolean values are stored as integers 0 (false) and 1 (true).

SQLite manifest typing & type affinity

Other important concepts related to SQLite data types are manifest typing and type affinity:

- Manifest typing means that a data type is a property of a value stored in a column, not the property of the column in which the value is stored. SQLite uses manifest typing to store values of any type in a column.
- Type affinity of a column is the recommended type for data stored in that column. Note that the data type is recommended, not required, therefore, a column can store any type of data.

SQLite – Transactions

A transaction is a unit of work that is performed against a database.

Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

SQLite transaction means it's a set of T-SQL statements that will execute together as a unit like a single T-SQL statement.

If all these T-SQL statements executed successfully without having any errors then the transaction will be committed and all the changes made by the transaction will be saved to the database permanently. In case, if any error occurred while executing these SQLite statements then the complete transaction will be rollbacked.

Generally, the SQLite is in an auto-commit mode that means SQLite automatically starts a transaction for each command, process, and commit the transaction changes automatically to the database.

Properties of Transactions

Transactions have the following four standard properties, usually referred to by the acronym ACID.

- **Atomicity** – Ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure and previous operations are rolled back to their former state.
- **Consistency** – Ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation** – Enables transactions to operate independently of and transparent to each other.
- **Durability** – Ensures that the result or effect of a committed transaction persists in case of a system failure.

Following are the following commands used to control transactions:

- **BEGIN TRANSACTION** – To start a transaction.
- **COMMIT** – To save the changes, alternatively you can use **END TRANSACTION** command.
- **ROLLBACK** – To rollback the changes.

Scanned with OKEN Scanner

Transactional control commands are only used with DML commands INSERT, UPDATE, and DELETE. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

BEGIN TRANSACTION Command

The BEGIN command is used to start or open a transaction. Once an explicit transaction has been opened, it will remain open until it is committed or rolled back.

Following is the syntax of the SQLite BEGIN command.

```
BEGIN;

or

BEGIN [TRANSACTION];
```

Here **TRANSACTION** keyword is **optional.**

SQLite COMMIT Command

The COMMIT command is used to close out the current transaction and commit the changes to the database. We can also use the alias END.

Following is the syntax of COMMIT command in SQLite.

```
COMMIT [TRANSACTION]

or

END [TRANSACTION]
```

Here **TRANSACTION** keyword is **optional.**

Scanned with OKEN Scanner

SQLite ROLLBACK Command

By using ROLLBACK command we can cancel the transaction and roll back all the proposed changes.

Following is the syntax of the ROLLBACK command.

ROLLBACK

or

ROLLBACK [TRANSACTION]

Here also **TRANSACTION** keyword is optional.

The ROLLBACK command will revert all the proposed changes made by the current transaction and then close the transaction

Both COMMIT and ROLLBACK will end the current transaction, putting SQLite back into Auto commit mode.

Data Filtering

SQLite DISTINCT Clause

The SQLite DISTINCT clause is used with SELECT statement to eliminate all the duplicate records and fetching only unique records.

It is used when you have multiple duplicate records in the table.

Syntax:

SELECT DISTINCT column1, column2,.....columnN

FROM table_name

WHERE [condition]

Scanned with OKEN Scanner

Example:

We have a table named "STUDENT", having the following data:

```
Command Prompt - sqlite3 JTP.db
sqlite> SELECT * FROM STUDENT;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
3|Mark|23|USA|2000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
7|Ajeet|27|Delhi|10000.0
8|Mark|23|USA|5000.0
9|Mark|23|USA|9000.0
sqlite> _
```

First Select NAME from "STUDENT" without using DISTINCT keyword. It will show the duplicate records:

Output:

```
Command Prompt - sqlite3 JTP.db
sqlite> SELECT NAME FROM STUDENT ;
Ajeet
Akash
Mark
Chandan
Kunwar
Ajeet
Mark
Mark
sqlite> _
```

Now, select NAME from "STUDENT" using DISTINCT keyword.

SELECT DISTINCT NAME FROM STUDENT;

Output:

```
Command Prompt - sqlite3 JTP.db
sqlite> SELECT DISTINCT NAME FROM STUDENT;
Ajeet
Akash
Mark
Chandan
Kunwar
sqlite> _
```

SQLite WHERE Clause

The SQLite WHERE clause is generally used with SELECT, UPDATE and DELETE statement to specify a condition while you fetch the data from one table or multiple tables.

If the condition is satisfied or true, it returns specific value from the table. You would use WHERE clause to filter the records and fetching only necessary records.

WHERE clause is also used to filter the records and fetch only specific data.

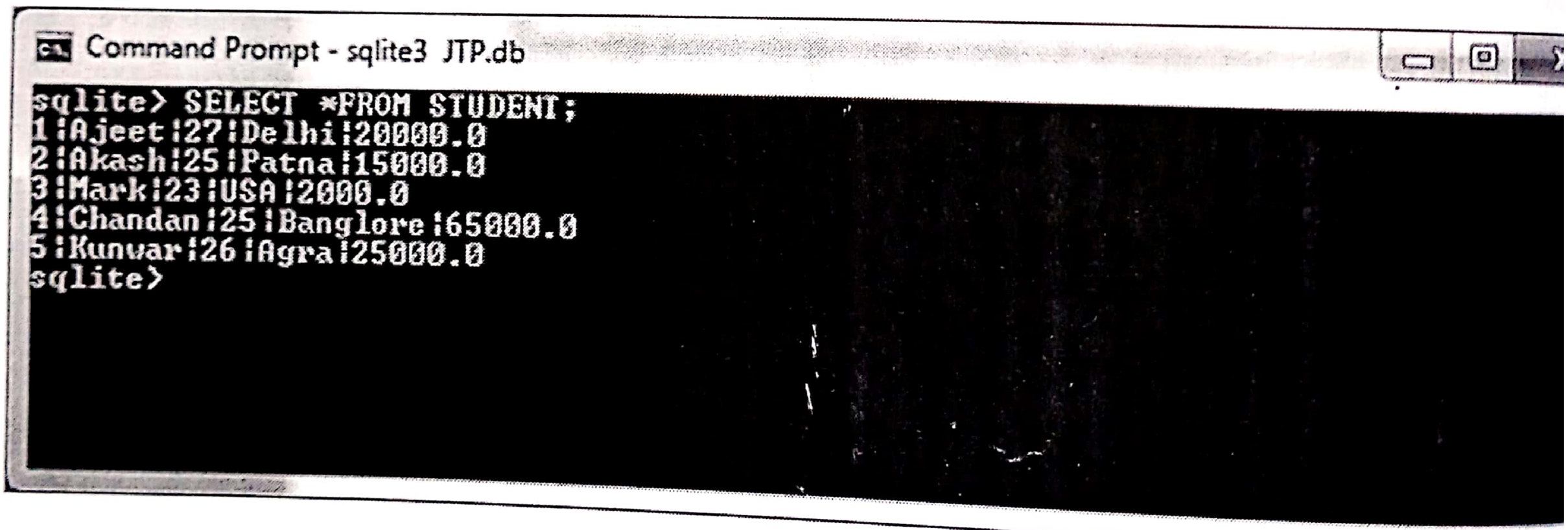
Syntax:

```
SELECT column1, column2, columnN

FROM table_name

WHERE [condition]
```

Example:

In this example, we are using WHERE clause with several comparison and logical operators. like >, <, =, LIKE, NOT, etc.

We have a table student having the following data:



Example1:

Select students WHERE age is greater than or equal to 25 and fees is greater than or equal to 10000.00

```
SELECT * FROM STUDENT WHERE AGE >= 25 AND FEES >= 10000.00;
```

Output:

```
Command Prompt - sqlite3 JTP.db
sqlite> SELECT * FROM STUDENT WHERE AGE >= 25 AND FEES >= 10000.00;
1|Ajeet|27|Delhi|20000.0
2|Akash|25|Patna|15000.0
4|Chandan|25|Banglore|65000.0
5|Kunwar|26|Agra|25000.0
sqlite>
```

SQLite BETWEEN Clause

The SQLite BETWEEN is used to select those values that are within a range of two values. For example, "**X BETWEEN Y AND Z**" will return true (1) if the value X is between the two values Y and Z. Otherwise, it will return false (0). "**X BETWEEN Y AND Z**" is equivalent to "**X >= Y AND X <= Z**", X must be greater than or equal to Y and X is less than or equal to Z.

Example:

In the following example query, we will write a query to get students with Id value between 5 and 8:

```
SELECT *
FROM Students
WHERE StudentId BETWEEN 5 AND 8;
```

This will give only the students with ids 5, 6, 7, and 8:

```
sqlite> SELECT *
   ...> FROM Students
   ...> WHERE StudentId BETWEEN 5 AND 8;
StudentId   StudentName   DepartmentId   DateOfBirth
----------  -----------   -------------  ----------
5           Sally         2              1998-10-12
6           Jena                         1998-10-12
7           Nancy         2              1998-10-12
8           Adam          3              1998-10-12
sqlite>
```

SQLite IN Clause

The SQLite IN clause takes one operand and a list of operands. It will return true if the first operand value equal to one of the operands' value from the list. The IN operator returns true (1) if the list of operands contains the first operand value within its values. Otherwise, it will return false (0).

Like this: "**col IN(x, y, z)**". This is equivalent to " **(col=x) or (col=y) or (col=z)** ".

Example:

The following query will select students with ids 2, 4, 6, 8 only:

```
SELECT *
FROM Students
WHERE StudentId IN(2, 4, 6, 8);
```

```
sqlite> SELECT *
   ...> FROM Students
   ...> WHERE StudentId IN(2, 4, 6, 8);
StudentId   StudentName  DepartmentId  DateOfBirth
----------  -----------  ------------  -----------
2           John         1             1998-10-12
4           Sara         2             1998-10-12
6           Jena                       1998-10-12
8           Adam         3             1998-10-12
```

The previous query will give the exact result as the following query because they are equivalent:

```
SELECT *
FROM Students
WHERE (StudentId = 2) OR (StudentId =  4) OR (StudentId =  6) OR (StudentId =
8);
```

SQLite NOT IN Clause

The SQLite NOT IN clause operand is the opposite of the IN operator. But with the same syntax, it takes one operand and a list of operands. It will return true if the first operand value not equal to one of the operands' value from the list. i.e., it will return true (0) if the list of operands doesn't contain the first operand. Like this: "**col NOT IN(x, y, z)**". This is equivalent to "**(col<>x) AND (col<>y) AND (col<>z)**".

Example:

The following query will select students with ids not equal to one of these Ids 2, 4, 6, 8:

```
SELECT *
FROM Students
WHERE StudentId NOT IN(2, 4, 6, 8);
```

```
sqlite> SELECT *
   ...> FROM Students
   ...> WHERE StudentId NOT IN(2, 4, 6, 8);
StudentId   StudentName   DepartmentId   DateOfBirth
----------  -----------   ------------   -----------
1           Michael       1              1998-10-12
3           Jack          1              1998-10-12
5           Sally         2              1998-10-12
7           Nancy         2              1998-10-12
9           Stevens       3              1998-10-12
10          George                       1998-10-12
11          guru11        1              1999-10-12
```

SQLite LIKE Clause

SQLite **LIKE** operator is used to match text values against a pattern using wildcards. If the search expression can be matched to the pattern expression, the LIKE operator will return true, which is 1. There are two wildcards used in conjunction with the LIKE operator –

- The percent sign (%)
- The underscore (_)

The percent sign represents zero, one, or multiple numbers or characters. The underscore represents a single number or character. These symbols can be used in combinations.

Syntax:

Following is the basic syntax of % and _.

```
SELECT FROM table_name

WHERE column LIKE 'XXXX%'
```

Or

```
SELECT FROM table_name
WHERE column LIKE '_XXXX_'
```

Example:

```
sqlite> SELECT * FROM COMPANY WHERE ADDRESS  LIKE '%-%';
```

Output:

```
ID        NAME      AGE       ADDRESS      SALARY
---------- ---------- ---------- ---------- ----------
3         raj       66        Rander-Road  5000.0
8         deep      12        ram-nagar    7000.0
```

SQLite - UNION Clause

SQLite **UNION** clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order, but they do not have to be of the same length.

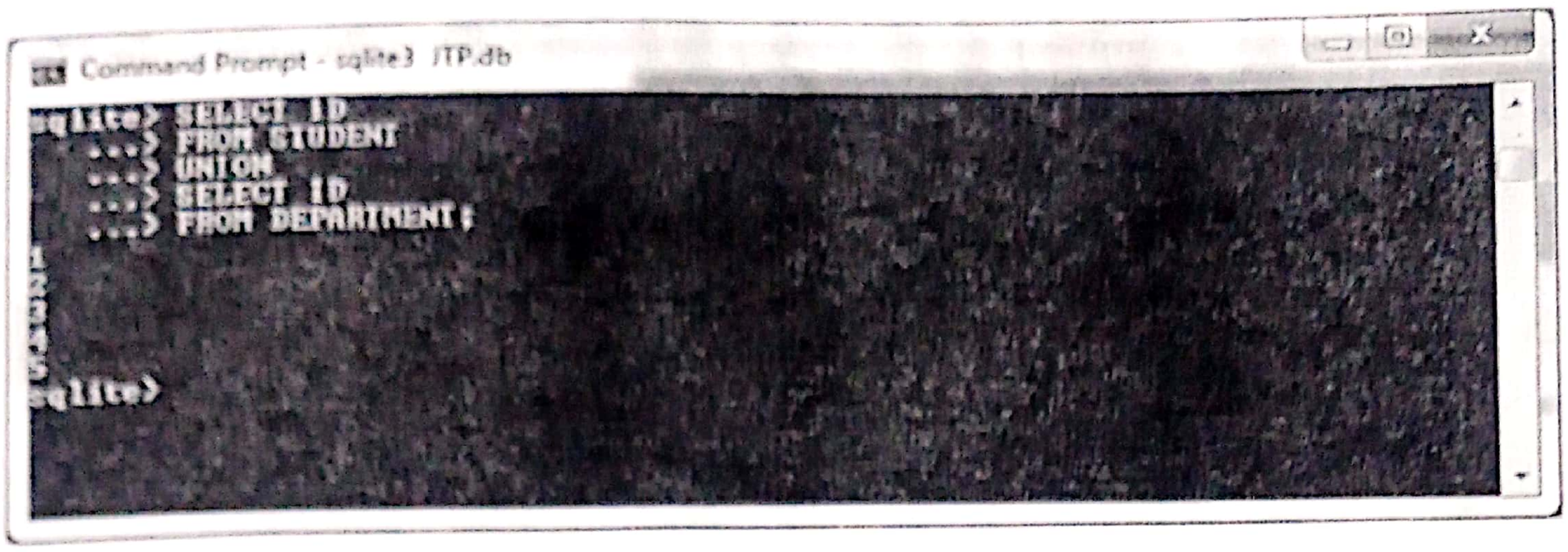
Syntax

Following is the basic syntax of **UNION**.

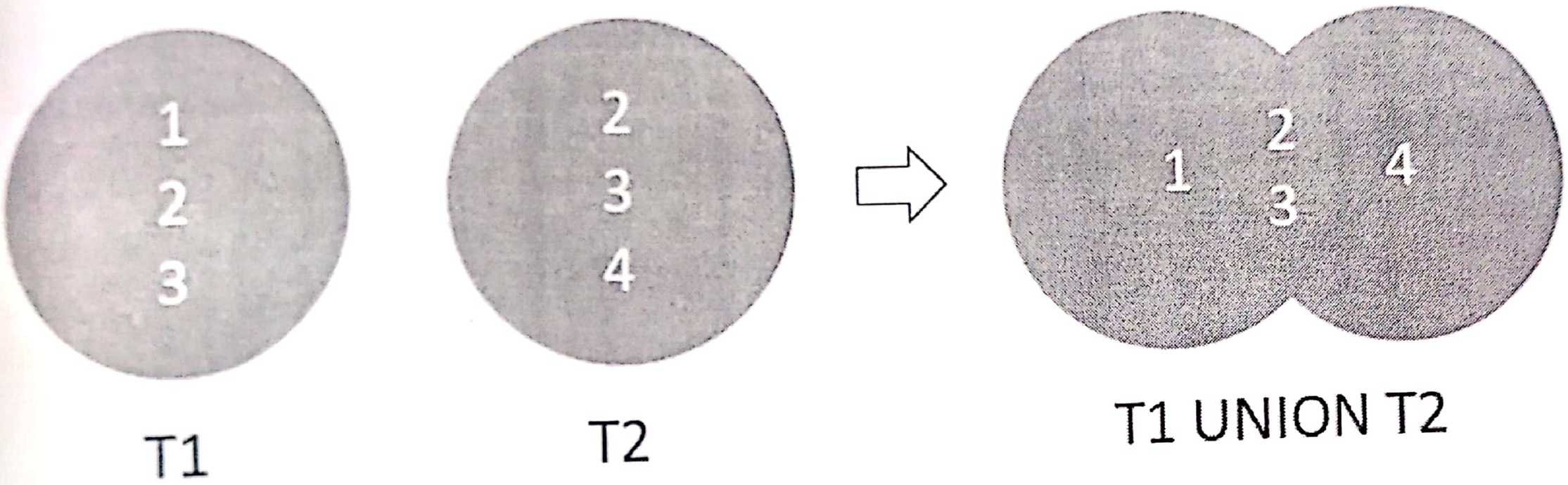
```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Example:

```
SELECT ID FROM STUDENT
UNION
SELECT ID FROM DEPARTMENT;
```



The following picture illustrates the UNION operation of t1 and t2 tables:



SQLite Intersect

SQLite INTERSECT operator compares the result sets of two queries and returns distinct rows that are output by both queries.

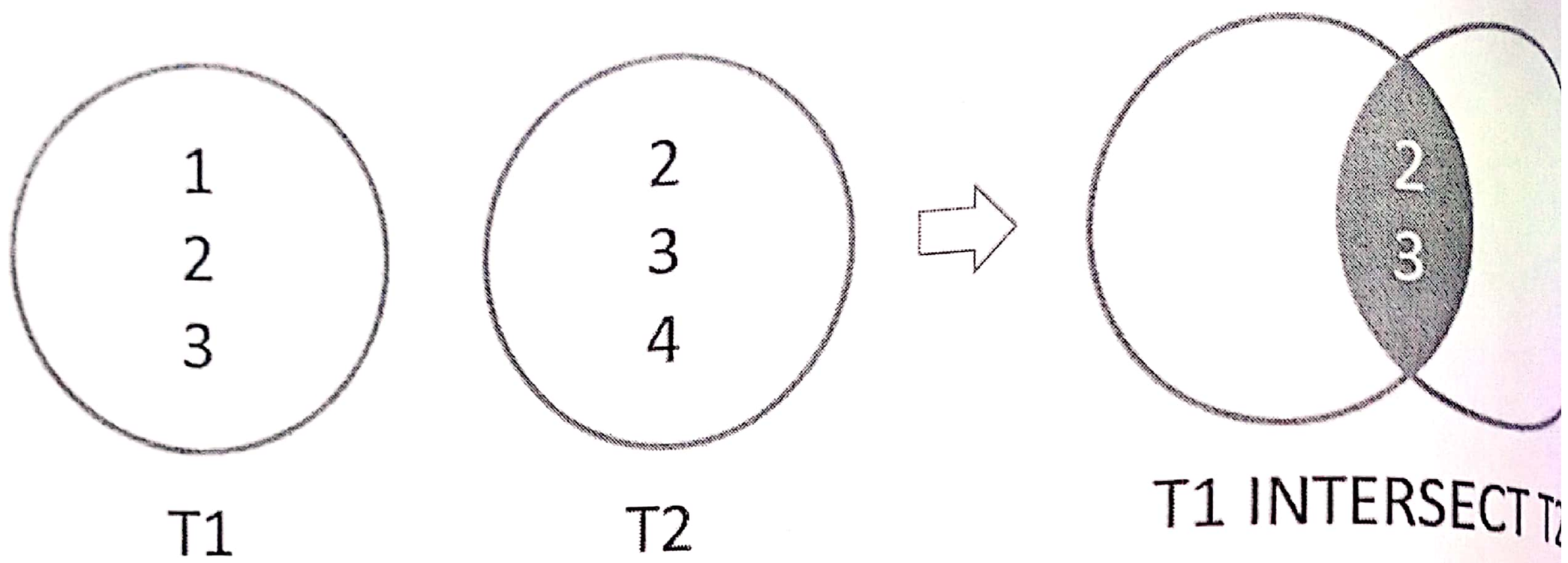
Syntax:

```
SELECT select_list1

FROM table1

INTERSECT

SELECT select_list2

FROM table2
```

The basic rules for combining the result sets of two queries are as follows:

- First, the number and the order of the columns in all queries must be the same.
- Second, the data types must be comparable.

The following picture illustrates the INTERSECT operation:



SQLite Except

SQLite EXCEPT operator compares the result sets of two queries and returns distinct rows from the left query that are not output by the right query.

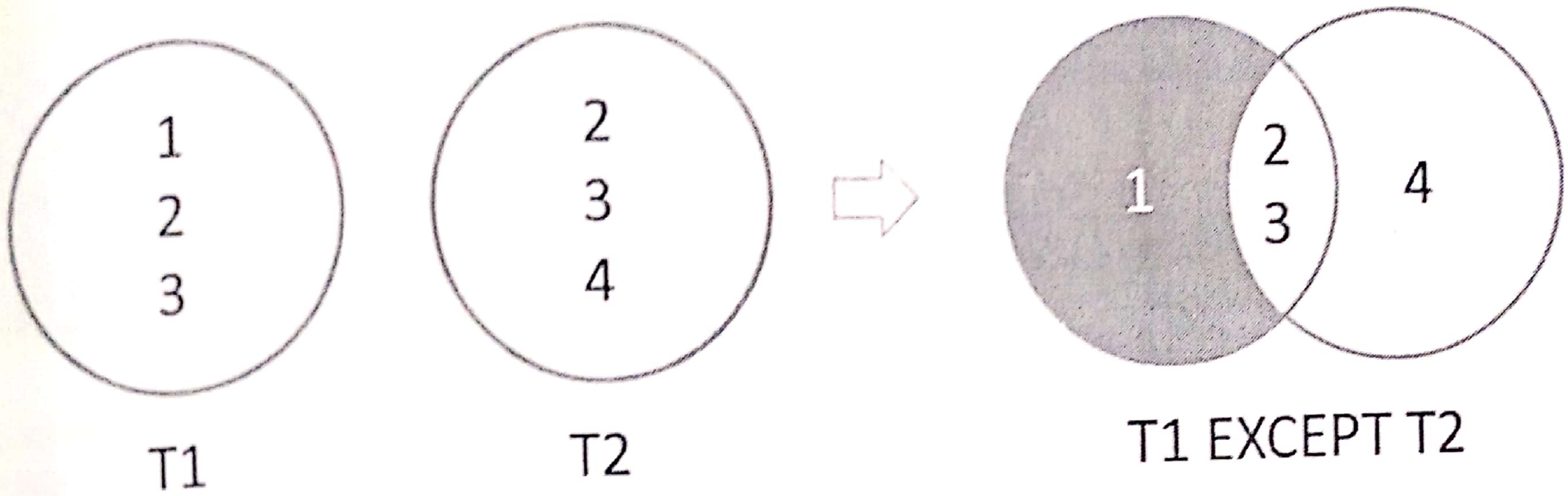
Syntax:

```
SELECT select_list1

FROM table1

EXCEPT

SELECT select_list2

FROM table2
```

This query must conform to the following rules:

- First, the number of columns in the select lists of both queries must be the same.
- Second, the order of the columns and their types must be comparable.

The following picture illustrates the EXCEPT operation:



SQLite LIMIT Clause

The SQLite LIMIT clause is used to limit the data amount fetched by SELECT command from a table.

Syntax:

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows]
```

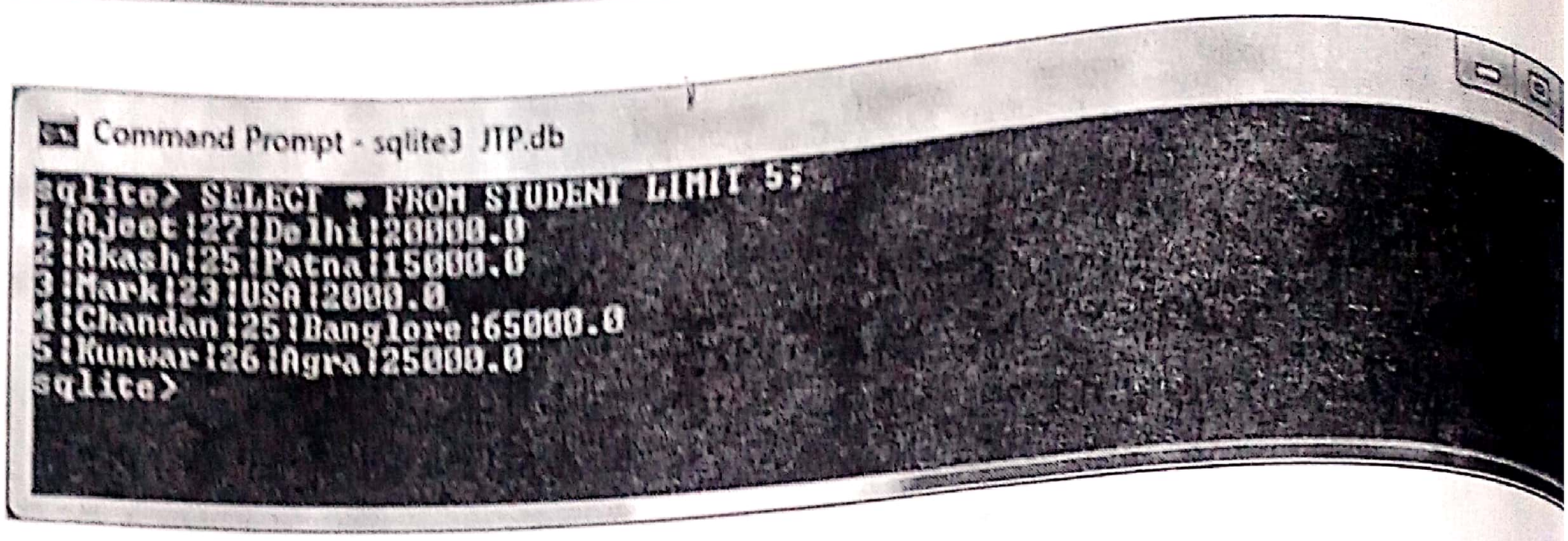
The LIMIT clause can also be used along with OFFSET clause.

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows] OFFSET [row num]
```

Example1:

Fetch the records from the "STUDENT" table by using LIMIT according to your need of number of rows.

```
SELECT * FROM STUDENT LIMIT 5;
```



SQLite GROUP BY Clause

The SQLite GROUP BY clause is used with SELECT statement to collaborate the same identical elements into groups.

The GROUP BY clause is used with WHERE clause in SELECT statement and precedes the ORDER BY clause.

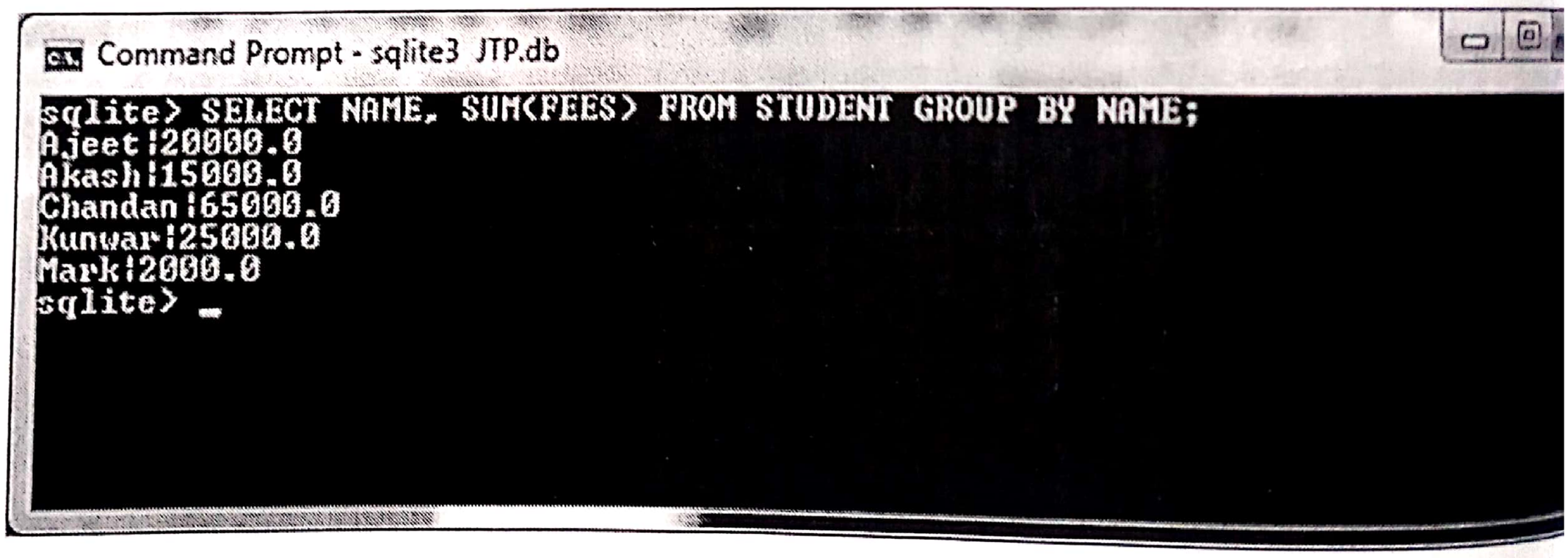
Syntax:

```
SELECT column-list
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2....columnN
ORDER BY column1, column2....columnN
```

Example:

Use the GROUP BY query to know the total amount of FEES of each student:

SELECT NAME, SUM(FEES) FROM STUDENT GROUP BY NAME;



SQLite HAVING Clause

The SQLite HAVING clause is used to specify conditions that filter which group results appear in the final results. The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

The position of HAVING clause in a SELECT query:

Syntax:

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
ORDER BY column1, column2
```

Example:

```
sqlite > SELECT * FROM COMPANY GROUP BY name HAVING count(name) > 2;
```

output:

```
ID         NAME       AGE        ADDRESS    SALARY
---------- ---------- ---------- ---------- ----------
45         rajan      56         Surat      7500
```

SQLite CASE

The SQLite CASE expression evaluates a list of conditions and returns an expression based on the result of the evaluation.

The CASE expression is similar to the IF-THEN-ELSE statement in other programming languages.

You can use the CASE expression in any clause or statement that accepts a valid expression. For example, you can use the CASE expression in clauses such as WHERE, ORDER BY, HAVING, SELECT and statements such as SELECT, UPDATE, and DELETE.

Syntax:

```
CASE case_expression
    WHEN when_expression_1 THEN result_1
    WHEN when_expression_2 THEN result_2
    ...
    [ ELSE result_else ]
END
```

The simple CASE expression compares the case_expression to the expression appears in the first WHEN clause, when_expression_1, for equality.

If the case_expression equals when_expression_1, the simple CASE returns the expression in the corresponding THEN clause, which is the result_1.

Otherwise, the simple CASE expression compares the case_expression with the expression in the next WHEN clause.

In case no case_expression matches the when_expression, the CASE expression returns the result_else in the ELSE clause. If you omit the ELSE clause, the CASE expression returns NULL.

The simple CASE expression uses short-circuit evaluation. In other words, it returns the result and stop evaluating other conditions as soon as it finds a match.

Example:

```
SELECT customerid,
    firstname,
    lastname,
    CASE country
        WHEN 'USA'
            THEN 'Domestic'
        ELSE 'Foreign'
    END CustomerGroup
FROM
    customers
```

SQLite joins:

In SQLite, JOIN clause is used to combine records from two or more tables in a database. It unites fields from two tables by using the common values of the both table.

There are mainly three types of Joins in SQLite:

- SQLite INNER JOIN
- SQLite OUTER JOIN
- SQLite CROSS JOIN

SQLite Inner Join

The SQLite Inner join is the most common type of join. It is used to combine all rows from multiple tables where the join condition is satisfied.

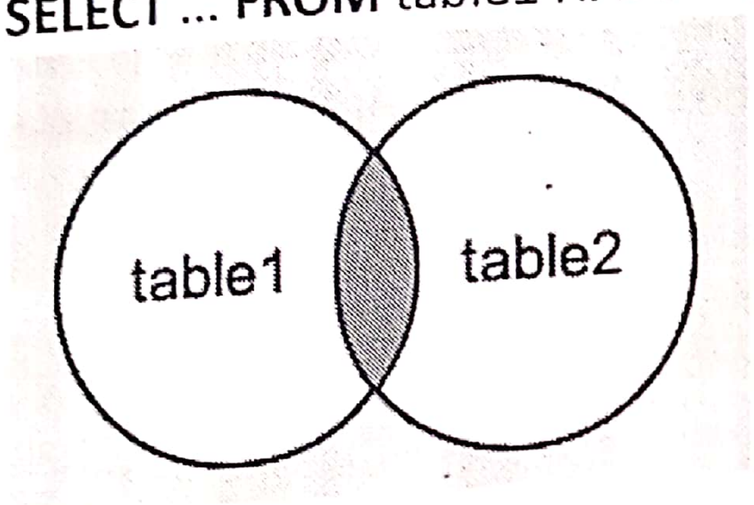
The SQlite Inner join is the default type of join.

Syntax:

SELECT ... **FROM** table1 [**INNER**] JOIN table2 **ON** conditional_expression ...

OR

SELECT ... **FROM** table1 NATURAL JOIN table2...



Example:

SELECT EMP_ID, NAME, DEPT FROM STUDENT INNER JOIN DEPARTMENT
ON STUDENT.ID = DEPARTMENT.EMP_ID;

SQLite Outer Join

In SQL standard, there are three types of outer joins:

- Left outer join
- Right outer join
- Full outer join.

But, SQLite supports only Left Outer Join.

SQlite Left Outer Join

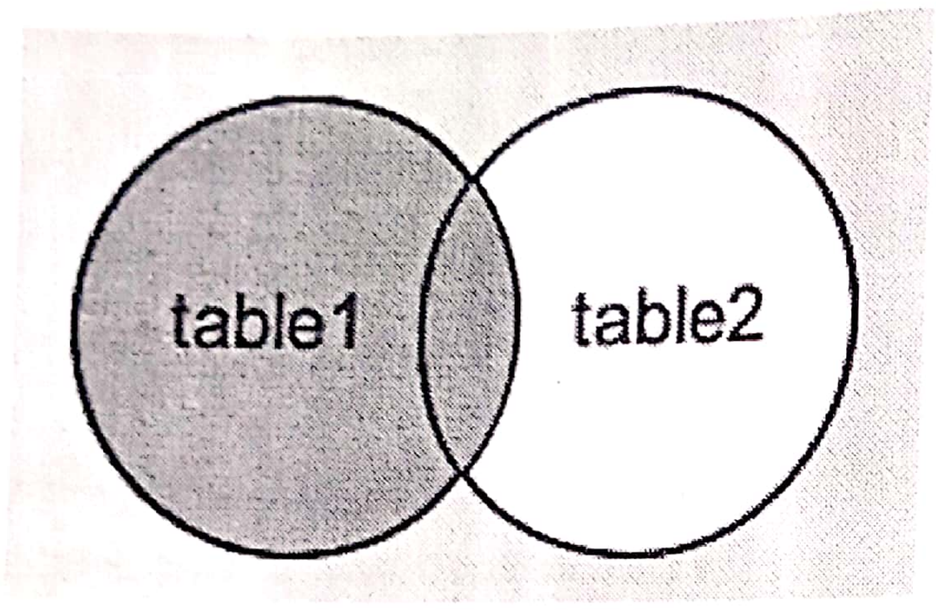
The SQLite left outer join is used to fetch all rows from the left hand table specified in the ON condition and only those rows from the right table where the join condition is satisfied.

Syntax:

SELECT ... FROM table1 LEFT OUTER JOIN table2 ON conditional_expression

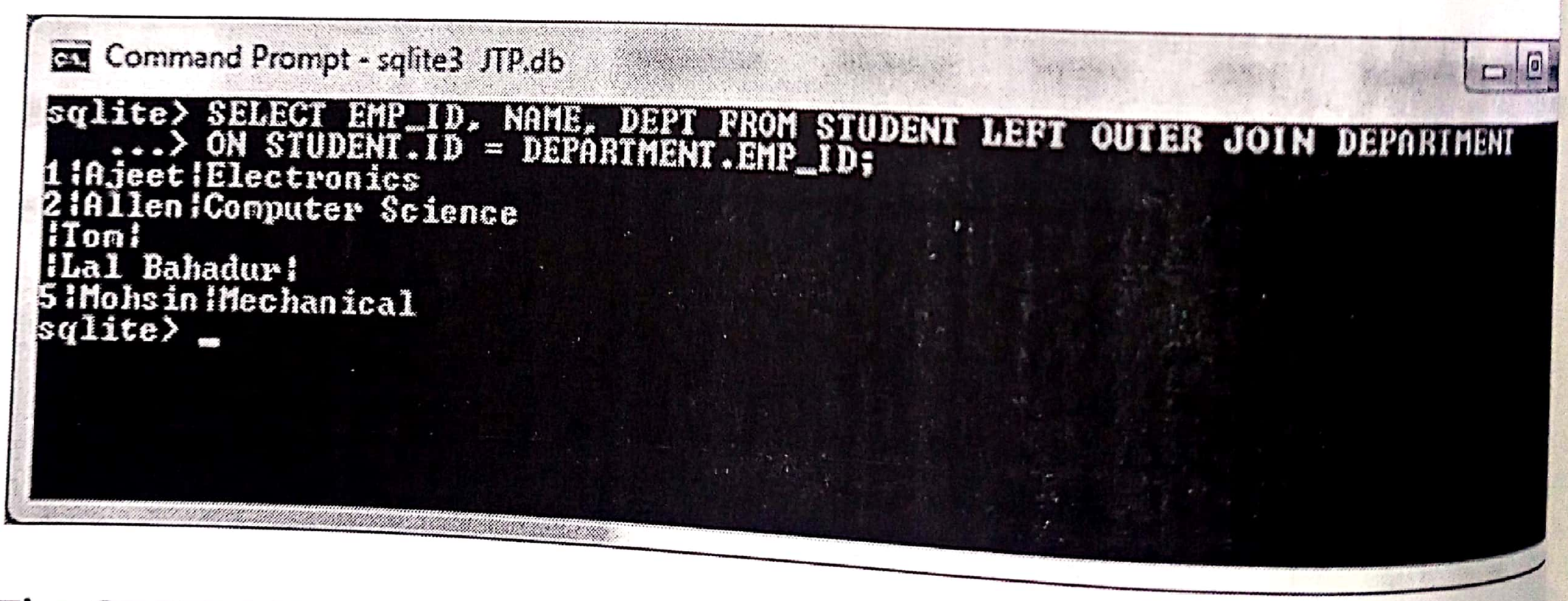
Or

SELECT ... FROM table1 LEFT OUTER JOIN table2 USING (column1 ,.....



Example:

SELECT EMP_ID, NAME, DEPT FROM STUDENT LEFT OUTER JOIN DEPARTMENT ON STUDENT.ID = DEPARTMENT.EMP_ID;



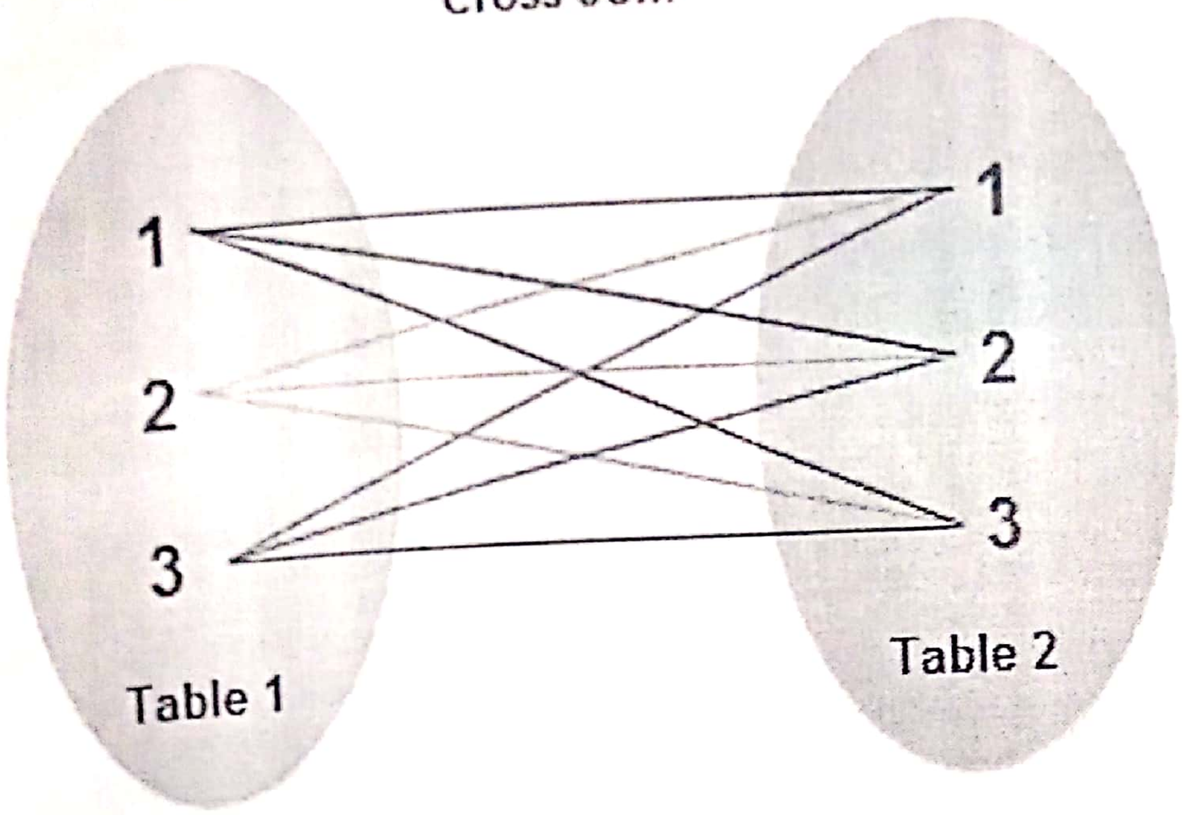
The CROSS JOIN

CROSS JOIN matches every row of the first table with every row of the second table. If the input tables have x and y row, respectively, the resulting table will have x*y row. Because CROSS JOINs have the potential to generate extremely large tables, care must be taken to only use them when appropriate.

Syntax :

SELECT ... FROM table1 CROSS JOIN table2 ...

Cross Join



Example:

sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY CROSS JOIN DEPARTMENT;

SQLite Trigger

SQLite Trigger is an event-driven action or database callback function which is invoked automatically when an INSERT, UPDATE, and DELETE statement is performed on a specified table.

The main tasks of triggers are like enforcing business rules, validating input data, and keeping an audit trail.

Usage of Triggers:

- Triggers are used for enforcing business rules.
- Validating input data.
- Generating a unique value for a newly-inserted row in a different file.
- Write to other files for audit trail purposes.
- Query from other files for cross-referencing purposes.
- Used to access system functions.
- Replicate data to different files to achieve data consistency.

Advantages of using triggers:

- Triggers make the application development faster. Because the database stores triggers, you do not have to code the trigger actions into each database application.
- Define a trigger once and you can reuse it for many applications that use the database.
- Maintenance is easy. If the business policy changes, you have to change only the corresponding trigger program instead of each application program.

SQLite CREATE TRIGGER

To create a new trigger in SQLite, you use the CREATE TRIGGER statement as follows:

Syntax:

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
 [BEFORE|AFTER|INSTEAD OF] [INSERT|UPDATE|DELETE]
 ON table_name
 [WHEN condition]
 BEGIN
  statements;
 END;
```

In this syntax:

- First, specify the name of the trigger after the CREATE TRIGGER keywords.
- Next, determine when the trigger is fired such as BEFORE, AFTER, or INSTEAD OF. You can create BEFORE and AFTER triggers on a table. However, you can only create an INSTEAD OF trigger on a view.
- Then, specify the event that causes the trigger to be invoked such as INSERT, UPDATE, or DELETE.
- After that, indicate the table to which the trigger belongs.
- Finally, place the trigger logic in the BEGIN END block, which can be any valid SQL statements.

If you combine the time when the trigger is fired and the event that causes the trigger to be fired, you have a total of 9 possibilities:

- BEFORE INSERT
- AFTER INSERT
- BEFORE UPDATE
- AFTER UPDATE

Scanned with OKEN Scanner

- BEFORE DELETE
- AFTER DELETE
- INSTEAD OF INSERT
- INSTEAD OF DELETE
- INSTEAD OF UPDATE

Suppose you use a UPDATE statement to update 10 rows in a table, the trigger that associated with the table is fired 10 times. This trigger is called FOR EACH ROW trigger. If the trigger associated with the table is fired one time, we call this trigger a FOR EACH STATEMENT trigger.

You can access the data of the row being inserted, deleted, or updated using the OLD and NEW references in the form: OLD.column_name and NEW.column_name.

The following table illustrates the rules.:

Action	Reference
INSERT	NEW is available
UPDATE	Both NEW and OLD are available
DELETE	OLD is available

Example:

SQLite BEFORE INSERT trigger.

Suppose you want to validate the email address before inserting a new lead into the leads table. In this case, you can use a BEFORE INSERT trigger.

First, create a BEFORE INSERT trigger as follows:

```
CREATE TRIGGER validate_email_before_insert_leads
  BEFORE INSERT ON leads
BEGIN
  SELECT
    CASE
        WHEN NEW.email NOT LIKE '%_@__%.__%' THEN
         RAISE (ABORT,'Invalid email address')
    END;
END;
```

SQLite DROP TRIGGER statement

To drop an existing trigger, you use the DROP TRIGGER statement as follows:

Syntax:

```
DROP TRIGGER [IF EXISTS] trigger_name;
```

In this syntax:

- First, specify the name of the trigger that you want to drop after the DROP TRIGGER keywords.
- Second, use the IF EXISTS option to delete the trigger only if it exists.

Note that if you drop a table, SQLite will automatically drop all triggers associated with the table.

Exercise

Short Question

1. What is SQLite?
2. What datatype does SQLite supports?
3. What is the use of LIMIT clause with SELECT query?
4. Why is ORDER BY clause used with SELECT statement?
5. What is the use of SQLite GROUP BY clause?
6. What is the use of DISTINCT clause in SQLite?
7. Explain Commit and rollback.

Long Question

1. What are the advantages of using SQLite?
2. Explain SQLite commands.
3. What is the difference between SQL and SQLite?
4. Explain the different type of join in SQLite.
5. What is UNION operator? How does it work?
6. Explain Triggers and types of trigger with example.

Scanned with OKEN Scanner

Unit - 2

Database backup and CSV handling

SQLite dump

The SQLite dump command to backup and restore a database.

.dump command that gives you the ability to dump the entire database or tables into a text file.

Dump the entire database into a file

To dump a database into a file, you use the .dump command. The .dump command converts the entire structure and data of an SQLite database into a single text file.

By default, the .dump command outputs the SQL statements on screen. To issue the output to a file, you use the **.output** FILENAME command.

The following command opens a new SQLite database connection to the student.db file.

```
C:\sqlite>sqlite3 c:/sqlite/ student.db
SQLite version 3.33.0 2021-06-05 10:57:30
Enter ".help" for usage hints.
sqlite>
```

The following commands specify the output of the dump file to student.sql and dump the student database into the student.sql file.

```
sqlite> .output c:/sqlite /student.sql
sqlite> .dump
sqlite> .exit
```

Dump a specific table using the SQLite dump command

If you want to dump a specific table, you need to specify the table name followed the .dump command. For example, the following command saves the albums table to the albums.sql file.

```
sqlite> .output c:/sqlite/albums.sql
sqlite> .dump albums
sqlite> .quit
```


Scanned with OKEN Scanner

The following picture shows the content of the albums.sql file.

```
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE "albums"
(
    [AlbumId] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    [Title] NVARCHAR(160)  NOT NULL,
    [ArtistId] INTEGER  NOT NULL,
    FOREIGN KEY ([ArtistId]) REFERENCES "artists" ([ArtistId])
        ON DELETE NO ACTION ON UPDATE NO ACTION
);
INSERT INTO "albums" VALUES(1,'For Those About To Rock We Salute You',1);
INSERT INTO "albums" VALUES(2,'Balls to the Wall',2);
INSERT INTO "albums" VALUES(3,'Restless and Wild',2);
INSERT INTO "albums" VALUES(4,'Let There Be Rock',1);
INSERT INTO "albums" VALUES(5,'Big Ones',3);
INSERT INTO "albums" VALUES(6,'Jagged Little Pill',4);
INSERT INTO "albums" VALUES(7,'Facelift',5);
INSERT INTO "albums" VALUES(8,'Warner 25 Anos',6);
```

Dump tables structure

If you want to dump the structures of tables in a database, you use the .schema command. The following commands set the output file to student.sql file and save the structures of tables into the student.sql file.

```
sqlite> .output c:/sqlite/ student.sql
sqlite> .schema
sqlite> .quit
```

Dump data of one or more tables into a file

To dump the data of a table into a text file, you use these steps:

First, set the mode to insert using the .mode command as follow

```
sqlite> .mode insert
```

From now on, every SELECT statement will issue the result as the INSERT statements instead of pure text data.

Second, set the output to a text file instead of the default standard output. The following command sets the output file to the record.sql file.

```
sqlite> .output record.sql
```

Third, issue the SELECT statements to query data from a table that you want to dump. The following command returns data from the student table.

```
sqlite> select * from student;
```

Check the content of the record.sql file.

To dump data from other tables, you need to issue the SELECT statements to query data from those tables.

CSV files handling

Import a CSV file into a table

We want to import data from CSV file into a table that does not exist in the SQLite database.

- First, the sqlite3 tool creates the table. The sqlite3 tool uses the first row of the CSV file as the names of the columns of the table.
- Second, the sqlite3 tool import data from the second row of the CSV file into the table.

We will import a CSV file named city.csv with two columns: name and population.

To import the c:\sqlite\city.csv file into the cities table:

First, set the mode to CSV to instruct the command-line shell program to interpret the input file as a CSV file. To do this, you use the .mode command as follows:

```
sqlite> .mode csv
```

Second, use the command .import FILE TABLE to import the data from the city.csv file into the cities table.

```
sqlite>.import c:/sqlite/city.csv cities
```

To verify the import, you use the command .schema to display the structure of the cities table.

```
sqlite> .schema cities
CREATE TABLE cities
(
  "name" TEXT,
  "population" TEXT
);
```

The table is already available in the database and you just need to import the data.

First, use the following CREATE TABLE statement to create the table cities.

```
CREATE TABLE cities
(
 name TEXT NOT NULL,
 population INTEGER NOT NULL
);
```

If the table already exists, the sqlite3 tool uses all the rows, including the first row, in the CSV file as the actual data to import. Therefore, you should delete the first row of the CSV file.

The following commands import the city_without_header.csv file into the cities table.

```
sqlite> .mode csv
sqlite> .import c:/sqlite/city_no_header.csv cities
```

Export a CSV file from table

By using the sqlite3 tool, you can use the SQL statements and dot-commands to interact with the SQLite database.

To export data from the SQLite database to a CSV file, you use these steps:

1. Turn on the header of the result set using the .header on command.
2. Set the output mode to CSV to instruct the sqlite3 tool to issue the result in the CSV mode.
3. Send the output to a CSV file.
4. Issue the query to select data from the table to which you want to export.

The following commands select data from the customers table and export it to the data.csv file.

```
>sqlite3 c:/sqlite/chinook.db
sqlite> .headers on
sqlite> .mode csv
sqlite> .output data.csv
sqlite> SELECT customerid,
   ...>      firstname,
   ...>      lastname,
   ...>      company
```

```
...>  FROM customers;
sqlite> .quit
```

If you check the data.csv file, you will see the following output.

```
CustomerId,FirstName,LastName,Company
1,"Luís","Gonçalves","Embraer - Empresa Brasileira de Aeronáutica S.A."
2,Leonie,"Köhler",
3,"François",Tremblay,
4,"Bjørn",Hansen,
5,"František","Wichterlová","JetBrains s.r.o."
6,Helena,"Holý",
7,Astrid,Gruber,
8,Daan,Peeters,
9,Kara,Nielsen,
10,Eduardo,Martins,"Woodstock Discos"
11,Alexandre,Rocha,"Banco do Brasil S.A."
```

Besides using the dot-commands, you can use the options of the sqlite3 tool to export data from the SQLite database to a CSV file.

For example, the following command exports the data from the tracks table to a CSV file named tracks.csv.

>sqlite3 -header -csv c:/sqlite/chinook.db "select * from tracks;" > tracks.csv

```
TrackId,Name,AlbumId,MediaTypeId,GenreId,Composer,Milliseconds,Bytes,UnitPrice
1,"For Those About To Rock (We Salute You)",1,1,1,"Angus Young, Malcolm Young, Br
2,"Balls to the Wall",2,2,1,,342562,5510424,0.99
3,"Fast As a Shark",3,2,1,"F. Baltes, S. Kaufman, U. Dirkscneider & W. Hoffman",2
4,"Restless and Wild",3,2,1,"F. Baltes, R.A. Smith-Diesel, S. Kaufman, U. Dirkscn
5,"Princess of the Dawn",3,2,1,"Deaffy & R.A. Smith-Diesel",375418,6290521,0.99
6,"Put The Finger On You",1,1,1,"Angus Young, Malcolm Young, Brian Johnson",20566
7,"Let's Get It Up",1,1,1,"Angus Young, Malcolm Young, Brian Johnson",233926,7636
8,"Inject The Venom",1,1,1,"Angus Young, Malcolm Young, Brian Johnson",210834,685
9,Snowballed,1,1,1,"Angus Young, Malcolm Young, Brian Johnson",203102,6599424,0.9
10,"Evil Walks",1,1,1,"Angus Young, Malcolm Young, Brian Johnson",263497,8611245,
11,C.O.D.,1,1,1,"Angus Young, Malcolm Young, Brian Johnson",199836,6566314,0.99
12,"Breaking The Rules",1,1,1,"Angus Young, Malcolm Young, Brian Johnson",263288,
13,"Night Of The Long Knives",1,1,1,"Angus Young, Malcolm Young, Brian Johnson",2
14,Spellbound,1,1,1,"Angus Young, Malcolm Young, Brian Johnson",270863,8817038,0.
15,"Go Down",4,1,1,AC/DC,331180,10847611,0.99
```

If you have a file named query.sql that contains the script to query data, you can execute the statements in the file and export data to a CSV file.

>sqlite3 -header -csv c:/sqlite/chinook.db < query.sql > data.csv

Exercise

Short Question

1. Define .dump command.
2. What is CSV file.

Long Question

1. How to Dump the entire database into a file?
2. How Dump a specific table using the SQLite dump command?
3. Explain how to Import a CSV file into a table.
4. Write Steps to Exports data from the SQLite database to a CSV file

Scanned with OKEN Scanner

Unit - 3

Python Interaction with SQLite

Scanned with OKEN Scanner

Module

Python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.

Modules in Python provides us the flexibility to organize the code in a logical way.

To use the functionality of one module into another, we must have to import the specific module.

Create a Module

To create a module just save the code you want in a file with the file extension .py:

Example

Save this code in a file named mymodule.py

```
def display(name):

print("my name, " + name)
```

Here, we need to include this module into our main module to call the method display () defined in the module named file.

Loading the module

Python provides two types of statements as defined below.

1. The import statement
2. The from-import statement

The import statement

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.

Syntax:

```
import module1,module2, module3 ..... module n
```

Hence, if we need to call the function display () defined in the file mymodule.py, we have to import that file as a module into our module as shown in the example below.

Example:

```
import file;
name = input("Enter the name: ")
file.display(name)
```

Output:

```
Enter the name: mayank
my name , mayank
```

The from-import statement

Python provides the flexibility to import only the specific attributes of a module. This can be done by using from - import statement.

Syntax :

```
from< module-name> import <name 1>, <name 2>..,<name n>
```

Example:

```
from file import display;
name = input("Enter the name: ")
file.display(name)
```

Output:

```
Enter the name: mayank
my name , mayank
```

We can also import all the attributes from a module by using *.

Consider the following syntax.

```
from<module> import *
```

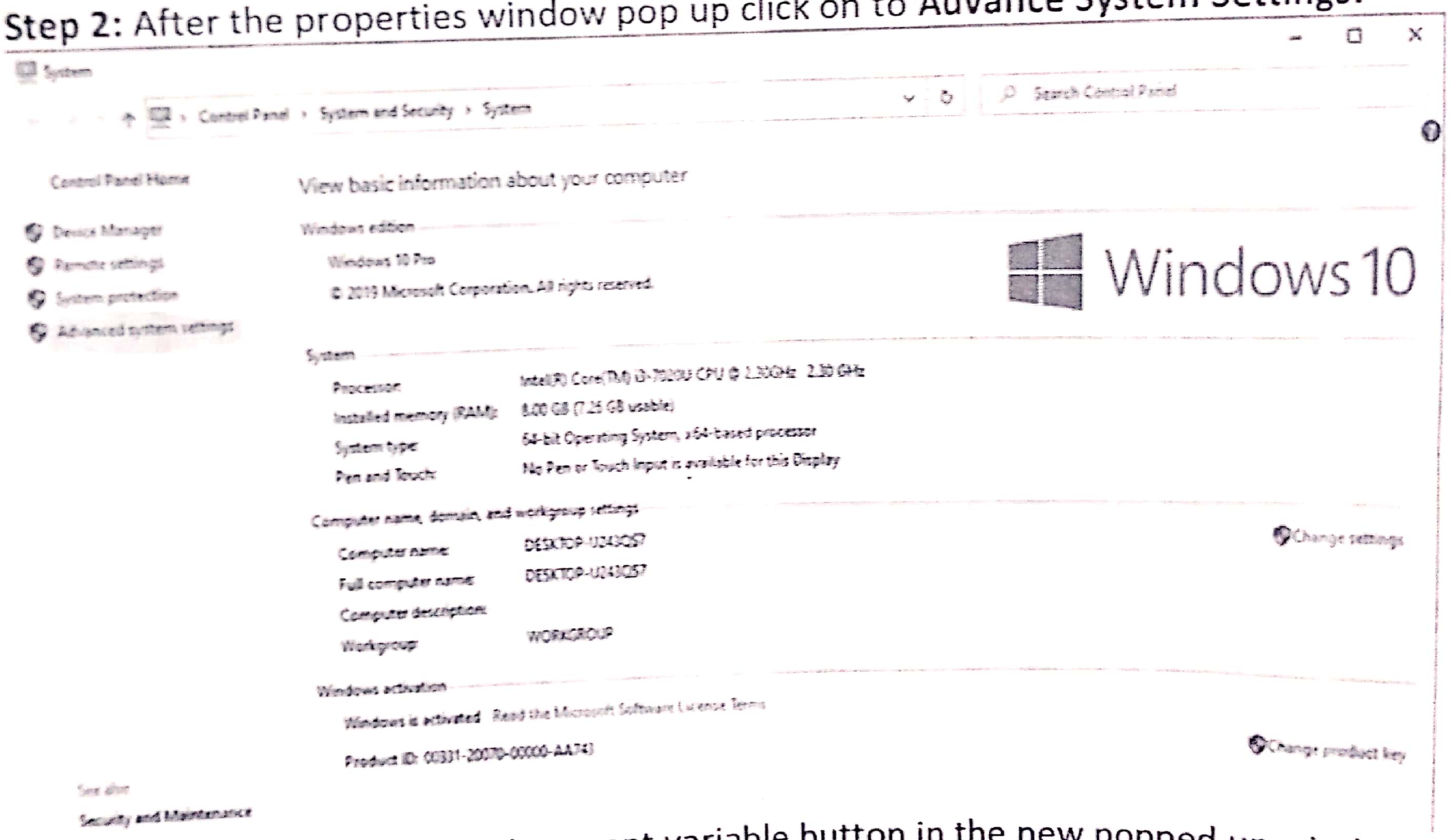
PYTHONPATH

PYTHONPATH is an environment variable which you can set to add additional directories where python will look for modules and packages. For most installations, you should not set these variables since they are not needed for Python to run. Python knows where to find its standard library.

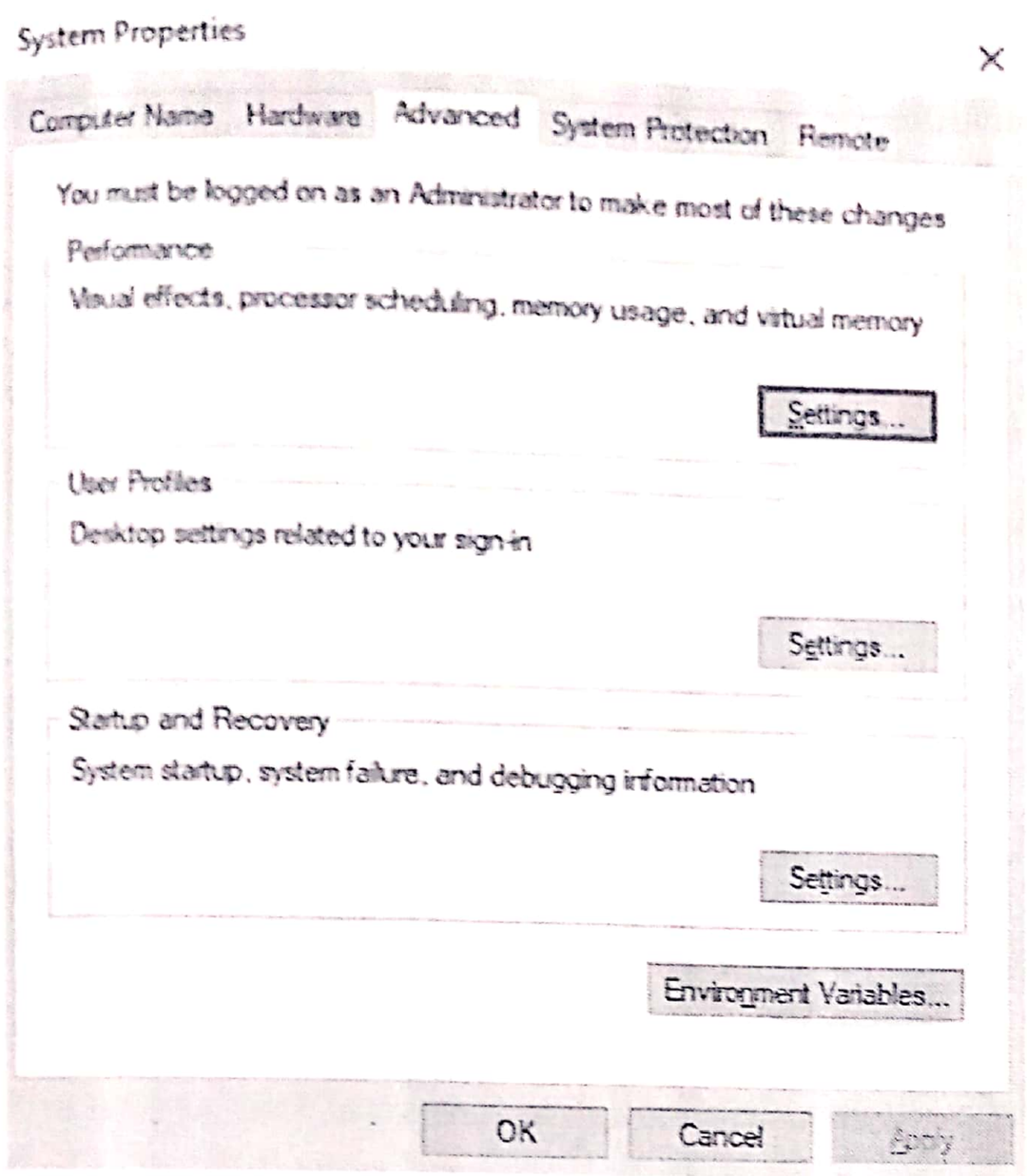
The only reason to set PYTHONPATH is to maintain directories of custom Python libraries that you do not want to install in the global default location (i.e., the site-packages directory).

Setting PYTHONPATH on a **windows** machine follow the below steps:

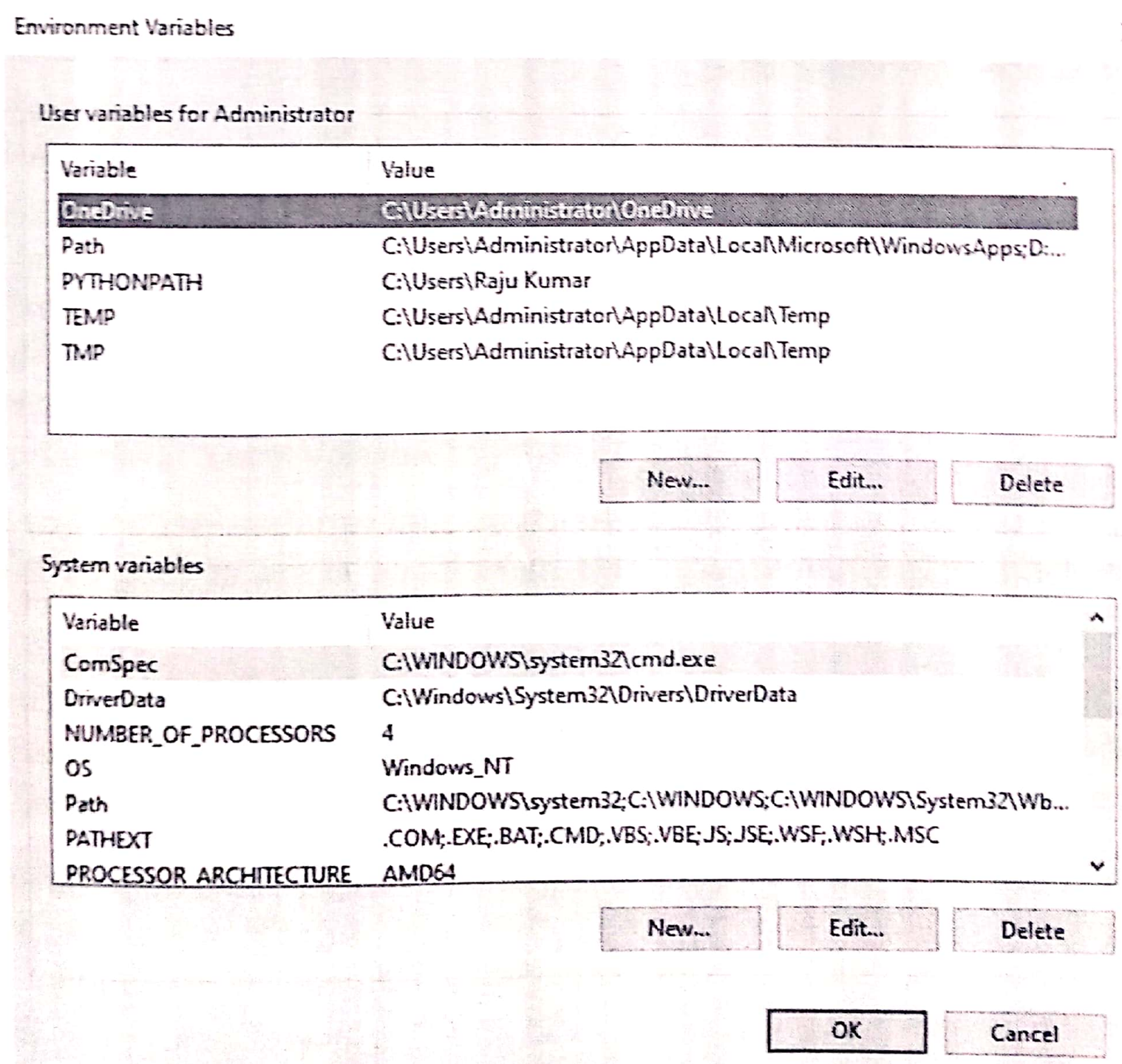
Step 1: Open your **This PC (or My Computer)** and write click and click on **properties.**
Step 2: After the properties window pop up click on to **Advance System Settings:**



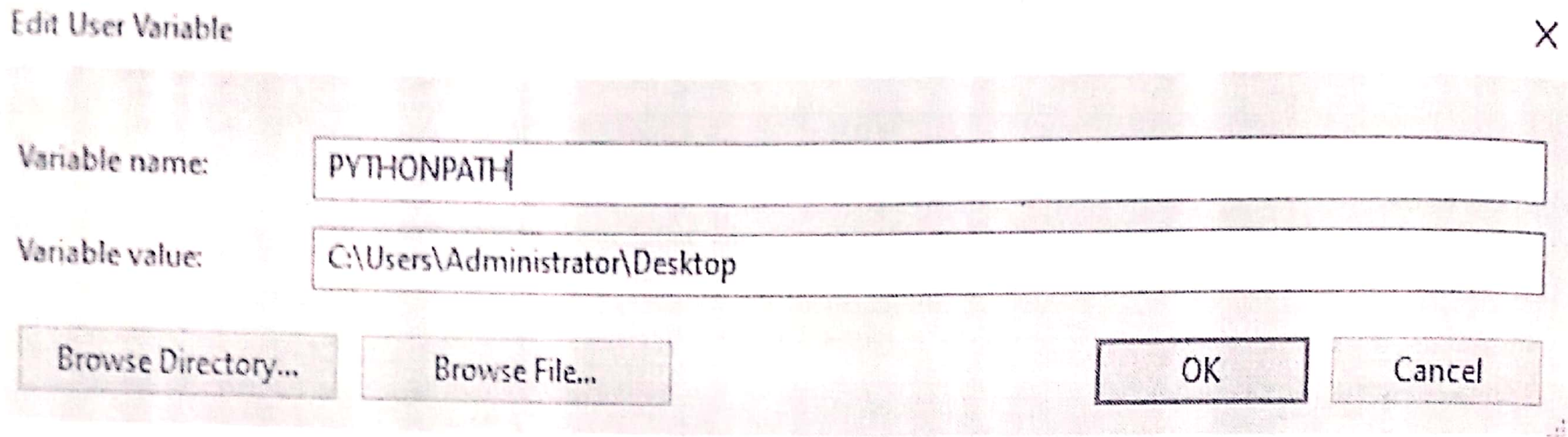
Step 3: Now click on the environment variable button in the new popped up window as shown below:



Step 4: Now in the new Environment Variable dialog box click on New as shown below:



Step 5: Now in the variable dialog box add the name of the variable as PYTHONPATH and in value add the location to the module directory that you want python to check every time as shown below:



Your PYTHONPATH is set.

Concepts of Namespace and Scope

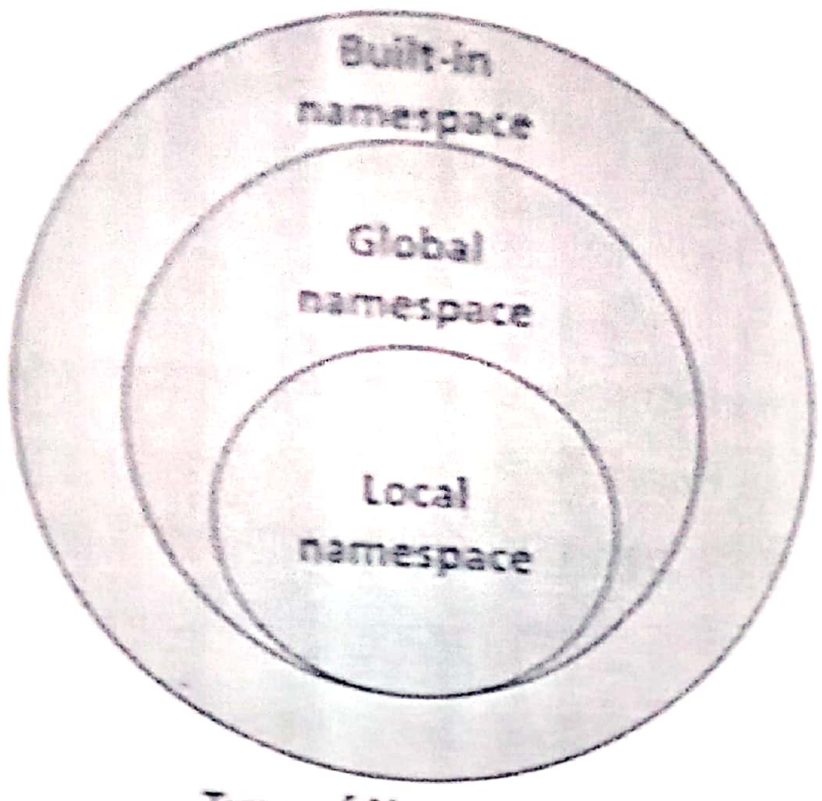
In python we deal with variables, functions, libraries and modules etc. There is a chance the name of the variable you are going to use is already existing as name of another variable or as the name of another function or another method. In such scenario, we need to learn about how all these names are managed by a python program. This is the concept of namespace.

Its Name (which means name, a unique identifier) + Space (which talks something related to scope). Here, a name might be of any Python method or variable and space depends upon the location from where is trying to access a variable or a method.

Following are the three categories of namespace

- **Local Namespace**: All the names of the functions and variables declared by a program are held in this namespace. This namespace exists as long as the program runs. < /p>
- **Global Namespace**: This namespace holds all the names of functions and other variables that are included in the modules being used in the python program. It encompasses all the names that are part of the Local namespace.
- **Built-in Namespace**: This is the highest level of namespace which is available with default names available as part of the python interpreter that is loaded as the programing environment. It encompasses Global Namespace which in turn encompasses the local namespace.

Scanned with OKEN Scanner



Type of Namespaces

Scope of Namespace

The namespace has a lifetime when it is available. That is also called the scope. Also the scope will depend on the coding region where the variable or object is located. You can see in the below program how the variables declared in an inner loop are available to the outer loop but not vice-versa. Also please note how the name of the outer function also becomes part of a global variable.

Example

```
prog_var = 'Hello'
defouter_func():
outer_var = 'x'
definner_func():
inner_var = 'y'
print(dir(), ' Local Variable in Inner function')

inner_func()
print(dir(), 'Local variables in outer function')

outer_func()
print(dir(), 'Global variables ')
```

Output

```
['inner_var'] Local Variable in Inner function
['inner_func', 'outer_var'] Local variables in outer function
```

```
['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'outer_func', 'prog_var']
Global variables
```

Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

Consider a file *Display.py* available in *pack1* directory. This file has following line of source code –

```
def disp():
        print "student doing bca"
```

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

- *pack1/a1.py* file having function a1()
- *pack1/b1.py* file having function b1()

Now, create one more file __init__.py in *pack1* directory –

- pack1/__init__.py

To make all of your functions available when you've imported *pack1* , you need to put explicit import statements in __init__.py as follows –

```
# Now import your pack1 Package.
import pack1

pack1.disp()
pack1.a1()
pack1.b1()
```

Output:

```
student doing bca
    this is from a1
   this is from b1
```

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

Importing sqlite3 module

SQLite3 can be integrated with Python using sqlite3 module, which was written by Gerhard Haring. It provides an SQL interface compliant with the DB-API 2.0.
You do not need to install this module separately because it is shipped by default along with Python version 2.5.x onwards.
To use sqlite3 module, you must first create a connection object that represents the database and then optionally you can create a cursor object, which will help you in executing all the SQL statements.
Importing sqlite3 :

```
import sqlite3
```

connect()

This routine opens a connection to the SQLite database file. You can use ":memory:" to open a database connection to a database that resides in RAM instead of on disk. If database is opened successfully, it returns a connection object.
When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The timeout parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds).

If the given database name does not exist then this call will create the database. You can specify filename with the required path as well if you want to create a database anywhere else except in the current directory.
Syntax:

```
sqlite3.connect(database [,timeout ,other optional arguments])
```

Example:

```
import sqlite3
conn = sqlite3.connect(student.db')
print "Opened database successfully";
```

Now connection is created with the student database.

Execute()

This routine executes an SQL statement. The SQL statement may be parameterized (i. e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks and named placeholders (named style).

Syntax:

cursor.execute(sql [, optional parameters])

For example – cursor.execute("insert into student values (?, ?)", (who, age))

fetchone()

fetchone() method returns a single record or None if no more rows are available. To fetch a single row from a result set we can use cursor.fetchone(). This method returns a single tuple.

It can return a none if no rows are available in the resultset. cursor.fetchone() increments the cursor position by one and return the next row.

Syntax:

cursor.fetchone()

Example:

```
import sqlite3

try:
connection = sqlite3.connect(student.db')
cursor = connection.cursor()
print("Connected to database")

query = "SELECT * from student"
cursor.execute(query)
print("Fetching single row")
record = cursor.fetchone()
print(record)

print("Fetching next row")
record = cursor.fetchone()
print(record)

cursor.close()

except sqlite3.Error as error:
print("Failed to read data from table", error)
finally:
if connection:
connection.close()
print("The Sqlite connection is closed")
```

Output:

```
Connected to database
Fetching single row
(1, 'amit', 'BCA',88)

Fetching next row
(2, 'rani', 'MCA',98)
The Sqlite connection is closed
```

fetchall()

fetchall() method fetches all (remaining) rows of a query result, returning a list. An empty list is returned when no rows are available. Get resultSet (all rows) from the cursor object using a cursor.fetchall().

Example:

```
import sqlite3

try:
connection = sqlite3.connect('student.db')
cursor = connection.cursor()
print("Connected to SQLite")

query = "SELECT * from student"
cursor.execute(query)
records = cursor.fetchall()
print("Total rows are:  ", len(records))
print("Printing each row")
for row in records:
print("Id: ", row[0])
print("Name: ", row[1])
print("course: ", row[2])
print("marks: ", row[3])
print("\n")

cursor.close()

except sqlite3.Error as error:
print("Failed to read data from table", error)
```

```
finally:
if connection:
connection.close()
print("The Sqlite connection is closed")
```

Output:

```
Connected to database
Total rows are:  3

Printing each row
Id:  1
Name:  raj
course:  BCA
marks:  88

Id:  2
Name:  rani
course:  MCA
marks:  99

Id:  3
Name:  deep
course:  BCA
marks:  50

The Sqlite connection is closed
```

Create a table using python

Create a table "STUDENT" within the database "demo_db".

Create a python file "createtable.py", having the following code:

```
import sqlite3
conn = sqlite3.connect('demo_db ')
print "Opened database successfully";

conn.execute("CREATE TABLE STUDENT
```

```
(ID INT PRIMARY KEY NOT NULL,
NAME TEXT NOT NULL,
AGE INT NOT NULL,
ADDRESS CHAR(50)
);")

print ("Table created successfully")

conn.close()
```

A table "STUDENT " is created in the "demo_db" database.

Insert Records

Insert some records in " STUDENT " table.

Create a python file "insert.py", having the following code:

```
import sqlite3

conn = sqlite3.connect('javatpoint.db')

print ("Opened database successfully")

conn.execute("INSERT INTO STUDENT (ID,NAME,AGE,ADDRESS) VALUES (1, 'aditya',
27, 'surat' )")

conn.commit()

print ("Records inserted successfully")

conn.close()
```

output:

```
Opened database successfully

Records inserted successfully
```


Scanned with OKEN Scanner

Select Records

To fetch and display your records from the table "STUDENT" by using SELECT statement.

Create a python file "select.py", having the following code:

```
import sqlite3

conn = sqlite3.connect('javatpoint.db')

data = conn.execute("select * from Employees")

for row in data:
   print ("ID = ", row[0]  )
   print ("NAME = ", row[1]  )
   print ("AGE = ", row[2]  )
   print ("ADDRESS = ", row[3] , "\n")

conn.close();
```

Output:

```
ID =1
NAME =aditya
AGE=27
ADDRESS =surat

ID =2
NAME =raj
AGE=32
ADDRESS =vapi
```

Delete Records

Delete some records in " STUDENT " table.

Create a python file "delete.py", having the following code:

```
import sqlite3

conn = sqlite3.connect('javatpoint.db')

print ("Opened database successfully")

id  = int(input("Enter id to delete:")
```

```
conn.execute("DELETE FROM STUDENT WHERE ID = ",id)
conn.commit()
print ("Total number of rows deleted :", conn.total_changes)
conn.close()
```

Output:

```
Opened database successfully
Enter id to delete:1
Total number of rows deleted :1
```

total_changes Returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened

commit()

commit() method commits the current transaction. If you don't call this method, anything you did since the last call to commit() is not visible from other database connections. If you wonder why you don't see the data you've written to the database, please check you didn't forget to call this method.

Example:

```
import sqlite3
con = sqlite3.connect('demo_db')
cur = con.cursor()

con.execute("""CREATE TABLE student
        (id integer, value text)""")
con.execute("INSERT INTO student VALUES (1, 'mehul')")
con.commit()
result = con.execute("SELECT * from student")
for row in result:
        print (row)
```

Output:

```
(1, 'mehul')
```

Here we create one table in demo_db database and insert one record using execute() method and then commit() method call to commits the current transaction and then data was display from database using select query using execute () method.

Exercise

Short Question

1. Define Module.
2. Explain PYTHONPATH.
3. Define Packages.
4. Difference between fetchone() and fetchall() methods.
5. Explain commit () method

Long Question

1. How to load module in code?
2. Explain Concepts of Namespace and Scope.
3. Explain Packages in Python.
4. Explain connect () and execute() methods of sqlite3 module.
5. Write a code to Insert, update, delete using execute () method in below table Student_tbl(id, name, dob, class, gender, city)

Unit - 4

Python Interaction with Text and CSV

Scanned with OKEN Scanner

Python Interaction with text and CSV

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

- Open a file
- Read or write - Performing operation
- Close the file

Opening a file

Python provides an open() function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax:

file object = open(<file-name>, <access-mode>, <buffering>)

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

Access mode	Description
R	It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
W	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
A	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.

Scanned with OKEN Scanner

r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
w+	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
a+	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
Rb	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
Wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file.
Ab	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

Example

```
#opens the file file.txt in read mode

fileptr = open("file.txt","r")


if fileptr:

    print("file is opened successfully")
```

Output:

```
file is opened successfully
```

In the above code, we have passed **filename** as a first argument and opened file in read mode as we mentioned **r** as the second argument. The **fileptr** holds the file object and if the file is opened successfully, it will execute the print statement.

close() method

Once all the operations are done on the file, we must close it through our Python script using the **close()** method. Any unwritten information gets destroyed once the **close()** method is called on a file object.

Syntax:

```
fileobject.close()
```

Example:

```
try:
  fileptr = open("file.txt")
  # perform file operations
finally:
  fileptr.close()
```

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed.

Reading and Writing Files

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

write() Method

The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string –

Syntax:

```
fileObject.write(string)
```

Example:

```
# Open a file
fo = open("demo.txt", "wb")
fo.write( "hello students enjoy python \n")

# Close opened file
fo.close()
```

The above method would create demo.*txt* file and would write given content in that file and finally it would close that file.

read() Method

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

Syntax

```
fileObject.read([count])
```

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

Example

```
# Open a file
fo = open("deno.txt", "r+")
str = fo.read(15);
print "Read String is : ", str
# Close opend file
fo.close()
```

Output:

```
Read String is :  hello students
```

readline() Method

The **readline()** method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Example:

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r");

#stores all the data of the file into the variable content
content = fileptr.readline()
content1 = fileptr.readline()

#prints the content of the file
print(content)
print(content1)
```

```
#closes the opened file
fileptr.close()
```

readlines() Method

Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

Example:

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r");

#stores all the data of the file into the variable content
content = fileptr.readlines()

#prints the content of the file
print(content)

#closes the opened file
fileptr.close()
```

The with statement

The **with** statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

Syntax :

```
with open(<file name>, <access mode>) as <file-pointer>:
        #statement suite
```

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the **with** statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file.

Example:

```
with open("file.txt",'r') as f:
   content = f.read();
   print(content)
```

What is a CSV File?

A CSV file (Comma Separated Values file) is a type of plain text file that uses specific structuring to arrange tabular data. Because it's a plain text file, it can contain only actual text data—in other words, printable ASCII or Unicode characters.
The structure of a CSV file is given away by its name. Normally, CSV files use a comma to separate each specific data value. Here's what that structure looks like:

CSV File

```
column 1 name,column 2 name, column 3 name
first row data 1,first row data 2,first row data 3
second row data 1,second row data 2,second row data 3
...
```

Notice how each piece of data is separated by a comma. Normally, the first line identifies each piece of data—in other words, the name of a data column. Every subsequent line after that is actual data and is limited only by file size constraints.

In general, the separator character is called a delimiter, and the comma is not the only one used. Other popular delimiters include the tab (\t), colon (:) and semi-colon (;) characters. Properly parsing a CSV file requires us to know which delimiter is being used.

CSV module

The csv library provides functionality to both read from and write to CSV files. Designed to work out of the box with Excel-generated CSV files, it is easily adapted to work with a variety of CSV formats. The csv library contains objects and other code to read, write, and process data from and to CSV files.
The csv module implements classes to read and write tabular data in CSV format. It allows programmers to say, "write this data in the format preferred by Excel," or "read data from this file which was generated by Excel," without knowing the precise details of the CSV format used by Excel.
The csv module's reader and writer objects read and write sequences. Programmers can also read and write data in dictionary form using the DictReader and DictWriter classes.

Open a CSV File:

When you want to work with a CSV file, the first thing to do is to open it. You can open a file using open() built-in function specifying its name (same as a text file).

```
f = open('myfile.csv')
```

When you specify the filename only, it is assumed that the file is located in the same folder as Python. If it is somewhere else, you can specify the exact path where the file is located.

```
f = open(r'C:\Python37\Scripts\myfile.csv')
```

NOTE: While specifying the exact path, characters prefaced by \ (like \n \r \t etc.) are interpreted as special characters. You can escape them using:

- raw strings like r'C:\new\text.txt'
- double backslashes like 'C:\\new\\text.txt'

File Mode

Character	Mode	Description
'r'	Read (default)	Open a file for read only
'w'	Write	Open a file for write only (overwrite)
'a'	Append	Open a file for write only (append)
'r+'	Read+Write	open a file for both reading and writing
'x'	Create	Create a new file

Example:

```
# Open a file for reading
f = open('myfile.csv')

# Open a file for writing
f = open('myfile.csv', 'w')

# Open a file for reading and writing
f = open('myfile.csv', 'r+')
```

csv .reader()

Reading from a CSV file is done using the reader object. The CSV file is opened as a text file with Python's built-in open() function, which returns a file object. This is then passed to the reader, which does the heavy lifting.

Syntax:

```
csv.reader(csvfile, dialect='excel', **fmtparams)
```

Return a reader object which will iterate over lines in the given csvfile.

Here's the employee.txt file:

CSV file

```
name,department,birthday month
ram jain,Sales,april
rohit shah,IT,may
```

```
import csv

with open('employee.txt') as csv_file:
```

```
csv_reader = csv.reader(csv_file, delimiter=',')
line_count = 0
for row in csv_reader:
    if line_count == 0:
        print(f'Column names are {", ".join(row)}')
        line_count += 1
    else:
        print(f'\t{row[0]} works in the {row[1]} department, and was born in {row[2]}.')
        line_count += 1
print(f'Processed {line_count} lines.')
```

Output:

```
Column names are name, department, birthday month
ram jain works in the Accounting department, and was born in april.
rohit shah works in the IT department, and was born in may.
Processed 3 lines.
```

Each row returned by the reader is a list of String elements containing the data found by removing the delimiters. The first row returned contains the column names, which is handled in a special way.

csv.writer()

To write to a CSV file in Python, we can use the csv.writer() function.

Syntax:

```
csv.writer(csvfile, dialect='excel', **optional_parameters)
```

The csv.writer() function returns a writer object that converts the user's data into a delimited string. This string can later be used to write into CSV files using the writerow() function.

Example:

```
import csv
with open(demo.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["ID", "Name", "Class"])
    writer.writerow([1, "Rohit", "Bca"])
    writer.writerow([2, "Pankaj", "BBA"])
```

When we run the above program, a demo.csv file is created with the following content:

```
ID, Name, Class
1, Rohit, Bca
2, Pankaj, BBA
```

In the above program, we have opened the file in writing mode.

Then, we have passed each row as a list. These lists are converted to a delimited string and written into the CSV file.

Writerows() method

We use **writerows()** method to write multiple rows at once.

Example:

```
# importing the csv module
import csv

# field names
fields = ['Name', 'Branch', 'Year', 'CGPA']

# data rows of csv file
rows = [ ['Nikhil', 'COE', '2', '9.0'],
                ['Sanchit', 'COE', '2', '9.1'],
                ['Aditya', 'IT', '2', '9.3'],
                ['Sagar', 'SE', '1', '9.5'],
                ['Prateek', 'MCE', '3', '7.8'],
                ['Sahil', 'EP', '2', '9.1']]

# name of csv file
filename = "university_records.csv"

# writing to csv file
with open(filename, 'w') as csvfile:
        # creating a csv writer object
        csvwriter = csv.writer(csvfile)

        # writing the fields
        csvwriter.writerow(fields)

        # writing the data rows
        csvwriter.writerows(rows)
```

Let us try to understand the above code:

- fields and rows have been already defined. fields is a list containing all the field names. rows is a list of lists. Each row is a list containing the field values of that row.

```
with open(filename, 'w') as csvfile:
        csvwriter = csv.writer(csvfile)
```

- Here, we first open the CSV file in WRITE mode. The file object is named as csvfile. The file object is converted to csv.writer object. We save the csv.writer object as csvwriter.

```
csvwriter.writerow(fields)
```

- Now we use writerow method to write the first row which is nothing but the field names.

```
csvwriter.writerows(rows)
```

- We use writerows method to write multiple rows at once.

DictReader()

The objects of a csv.DictReader() class can be used to read a CSV file as a dictionary.

Syntax:

```
csv.DictReader(f, fieldnames=None, restkey=None, restval=None,
dialect='excel', *args, **kwds)
```

Create an object that operates like a regular reader but maps the information in each row to a dict whose keys are given by the optional fieldnames parameter.

Here's the employee.txt file:

CSV file

```
name,department,birthday month
ram jain,Sales,april
rohit shah,IT,may
```

Example:

```
import csv

with open('employee.txt', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        print(f'\t{row["name"]} works in the {row["department"]} department,
and was born in {row["birthday month"]}.')
        line_count += 1
```

```
print(f'Processed {line_count} lines.')
```

Output:

```
Column names are name, department, birthday month
    ram jain works in the Accounting department, and was born in april.
    rohit shah works in the IT department, and was born in may.
Processed 3 lines.
```

Where did the dictionary keys come from? The first line of the CSV file is assumed to contain the keys to use to build the dictionary. If you don't have these in your CSV file, you should specify your own keys by setting the fieldnames optional parameter to a list containing them.

DictWriter()

The objects of csv.DictWriter() class can be used to write to a CSV file from a Python dictionary.

Syntax:

csv.DictWriter(f, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwds)

file - CSV file where we want to write to
fieldnames - a list object which should contain the column headers specifying the order in which data should be written in the CSV file.

Create an object which operates like a regular writer but maps dictionaries onto output rows.

Example:

```
# importing the csv module
import csv

# my data rows as dictionary objects
mydict =[{'branch': 'COE', 'cgpa': '9.0', 'name': 'Nikhil', 'year': '2'},
         {'branch': 'COE', 'cgpa': '9.1', 'name': 'Sanchit', 'year': '2'},
         {'branch': 'IT', 'cgpa': '9.3', 'name': 'Aditya', 'year': '2'},
         {'branch': 'SE', 'cgpa': '9.5', 'name': 'Sagar', 'year': '1'},
         {'branch': 'MCE', 'cgpa': '7.8', 'name': 'Prateek', 'year': '3'},
         {'branch': 'EP', 'cgpa': '9.1', 'name': 'Sahil', 'year': '2'}]
# field names
fields = ['name', 'branch', 'year', 'cgpa']
```

```
# name of csv file
filename = "university_records.csv"

# writing to csv file
with open(filename, 'w') as csvfile:
        # creating a csv dict writer object
        writer = csv.DictWriter(csvfile, fieldnames = fields)

        # writing headers (field names)
        writer.writeheader()

        # writing data rows
        writer.writerows(mydict)
```

In this example, we write a dictionary mydict to a CSV file.

- with open(filename, 'w') as csvfile:
  writer = csv.DictWriter(csvfile, fieldnames = fields)
  Here, the file object (csvfile) is converted to a DictWriter object. Here, we specify the fieldnames as an argument.

- writer.writeheader()
  writeheader method simply writes the first row of your csv file using the pre-specified fieldnames.

- writer.writerows(mydict)
  writerows method simply writes all the rows but in each row, it writes only the values(not keys).

Important Points:

In csv modules, an optional dialect parameter can be given which is used to define a set of parameters specific to a particular CSV format. By default, csv module uses excel dialect which makes them compatible with excel spreadsheets. You can define your own dialect using register_dialect method.

Example:

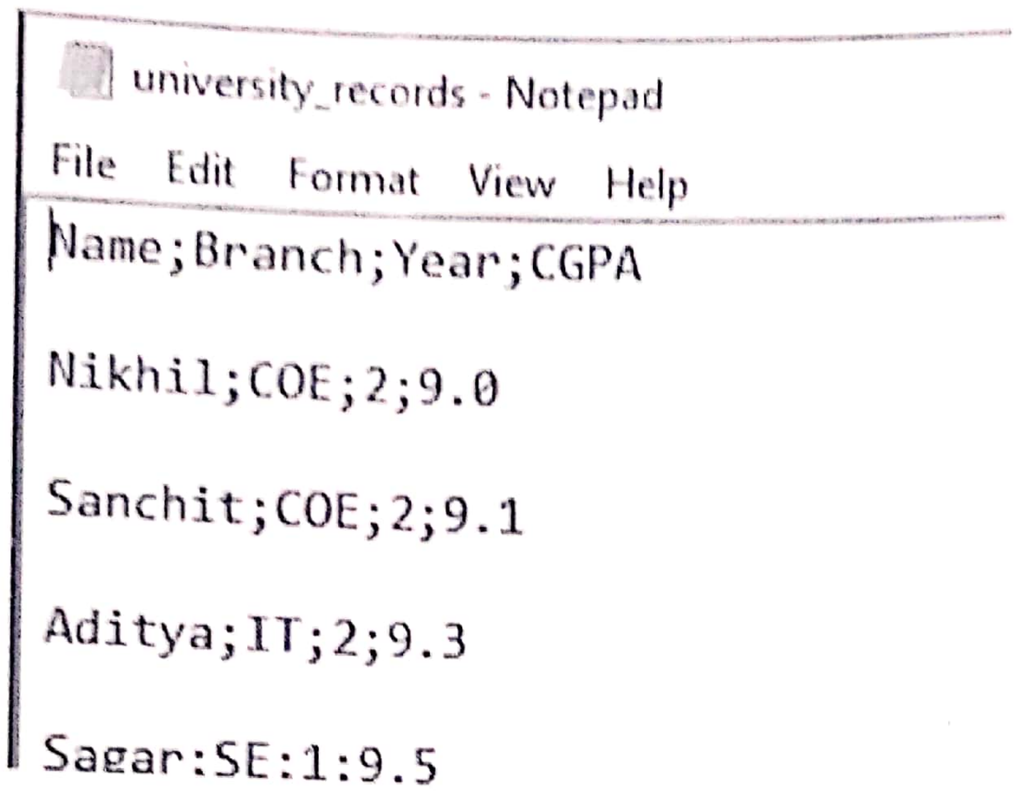
```
csv.register_dialect(
'mydialect',
delimiter = ',',
quotechar = '"',
doublequote = True,
skipinitialspace = True,
lineterminator = '\r\n',
```

```
quoting = csv.QUOTE_MINIMAL)
```

Now, while defining a csv.reader or csv.writer object, we can specify the dialect like this:

```
csvreader = csv.reader(csvfile, dialect='mydialect')
```

Now, consider that a CSV file looks like this in plain-text:



We notice that the delimiter is not a comma but a semi-colon. Also, the rows are separated by two newlines instead of one. In such cases, we can specify the delimiter and line terminator as follows:

```
csvreader = csv.reader(csvfile, delimiter = ';', lineterminator = '\n\n')
```

Close a CSV File

Use the **close()** function to close an open file. It's a good practice to close the file once you are finished with it.

There are two approaches to ensure that a file is closed properly, even in cases of error.

The first approach is to use the **with** keyword, which Python recommends, as it automatically takes care of closing the file once it leaves the **with** block (even in cases of error).

Example:

```
with open('myfile.csv') as f:
    print(f.read())
```

The second approach is to use the try-finally block:

Example:

```
f = open('myfile.csv')
try:
    # File operations goes here
```

```
finally:
    f.close()
```

Dataframe Handling using Panda and Numpy

Python Pandas library

Python Pandas is defined as an open-source library that provides high-performance data manipulation in Python. The name of Pandas is derived from the word **Panel Data**, which means **an Econometrics from Multidimensional data**. Pandas library Developed by **Wes McKinney** in **2008.**

Key Features of Pandas

- It has a fast and efficient DataFrame object with the default and customized indexing.
- Used for reshaping and pivoting of the data sets.
- Group by data for aggregations and transformations.
- It is used for data alignment and integration of the missing data.
- Provide the functionality of Time Series.
- Process a variety of data sets in different formats like matrix data, tabular heterogeneous, time series.
- Handle multiple operations of the data sets such as subsetting, slicing, filtering, groupBy, re-ordering, and re-shaping.
- It integrates with the other libraries such as SciPy, and scikit-learn.
- Provides fast performance, and If you want to speed it, even more, you can use the **Cython**.

Pandas deal with the following data structures –

- Series
- DataFrame

Series

It is defined as a one-dimensional array that is capable of storing various data types. The row labels of series are called the **index**. We can easily convert the list, tuple, and dictionary into series using "series' method. A Series cannot contain multiple columns. It has one parameter.

A pandas Series can be created using the following constructor –

pandas.Series(data, index, dtype, copy)

Example:

Here we create a series from a Numpy Array.

```
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print(s)
```

Output:

```
0  a
1  b
2  c
3  d
dtype: object
```

DataFrame

It is a widely used data structure of pandas and works with a two-dimensional array with labeled axes (rows and columns). DataFrame is defined as a standard way to store data and has two different indexes, i.e., row index and column index. It consists of the following properties:

- The columns can be heterogeneous types like int, bool, and so on.
- It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.

A pandas DataFrame can be created using the following constructor –

pandas.DataFrame(data, index, columns, dtype, copy)

Example:

Let us now create an indexed DataFrame using arrays.

```
import pandas as pd
data = {  'Name':['raj', 'Jay', 'saanvi','pooja'], 'Age':[28,3,2,4]  }
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print(df)
```

Its output is as follows –

```
        Age   Name
rank1    28   raj
rank2    3    jay
```

rank3 2 saanvi

rank4 4 pooja

Create DataFrame using List

The DataFrame can be created using a single list or a list of lists.

```
import pandas as pd
# a list of strings
list = ['Python', 'data','bca','mca']

# Calling DataFrame constructor on list
df = pd.DataFrame(list)
print(df)
```

Output:

```
   0
0  Python
1  data
2  bca
3  mca
```

In this code, we have defined a variable named "list" that consist of string values. The DataFrame constructor is being called on a list to print the values.

Create a DataFrame from Dict of equal length of Lists

All the list in dict must be of same length. If index is passed, then the length of the index should equal to the length of the arrays. If no index is passed, then by default, index will be range(n), where n is the array length.

Example:

```
import pandas as pd
dic = {'Name':['ram', 'deep', 'pankaj', 'jay'],'Marks':[44,65,75,48]}
df = pd.DataFrame(dic)
print( df)
```

Output:

```
   Marks  Name
0     44   ram
1     65  deep
2     75  pankaj
3     44   jay
```

Observe the values 0,1,2,3. They are the default index assigned to each using the function range(n).

Example : Create a DataFrame from List of Dicts

List of Dictionaries can be passed as input data to create a DataFrame. The dictionary keys are by default taken as column names.

```
import pandas as pd
data = [{'x': 10, 'y': 20},{'x': 55, 'y': 15, 'z': 88}]
df = pd.DataFrame(data)
print df
```

Output:

```
   x   y    z
0 10  20  NaN
1 55  15  88.0
```

Observe, NaN (Not a Number) is appended in missing areas.

Create Data Frame from Excel Spreadsheet

The Pandas library provides features using which we can read the Excel file in full as well as in parts for only a selected group of Data. We can also read an Excel file with multiple sheets in it. We use the read_excel function to read the data from it. You can create this file using the Excel Program and save the file as demo.xlsx.

	A	B	C	D
1	Eno	Name	Salary	dept
2	101	amit	2000	account
3	102	deep	2500	sales
4	103	jay	2150	design
5	104	nency	2100	account
6	105	shree	1500	sales
7				
8				

The read_excel function of the pandas library is used read the content of an Excel file into the python environment as a pandas DataFrame.

```
import pandas as pd
data = pd.read_excel('demo.xlsx')
print (data)
```

Output:

```
   Eno   Name  Salary  dept
0  101  amit    2000  account
1  102  deep    2500  sales
```

```
2 103  jay    2150  design
3 104  nency  2100  account
4 105  shree  1500  sales
```

When we execute the above code, it produces the following result. Please note how an additional column starting with zero as a index has been created by the function.

Creating Data Frame from .csv file

A CSV is a comma-separated values file, which allows data to be saved in a tabular format. CSVs look like a garden-variety spreadsheet but with a .csv extension.read_csv() function is used to load data from .csv file in python data frame. save the file as demo.csv.

Example:

```
import pandas as pd
data = pd.read_csv('demo.csv')
print (data)
```

Output:

```
Eno    Name   Salary   dept
0 101  amit   2000     account
1 102  deep   2500     sales
2 103  jay    2150     design
3 104  nency  2100     account
4 105  shree  1500     sales
```

Operation on DataFrame

Once we create a Data Frame we can do various operation on data frame .This operation help to analyzing the data and manipulating the data.

Knowing number of Rows and Columns

To know the number of row and columns available in the data frame, we can use shape attribute. It returns that contain number of rows and columns.

```
import pandas as pd
data = pd.read_excel('demo.xlsx')
print(data.shape)
```

output:

```
(5, 4)
```

If we want to print only number of rows and columns then we can do below code:

```
import pandas as pd
```

```
data = pd.read_excel('demo.xlsx')
r,c =data.shape
print(r)  # print only number of row
```

Output:

```
5
```

- **Retrieving Rows from data frame:**

head() method

head() method gives first n (by default value 5) record from data frame .

Syntax:

```
DataFrame.head(n=5)
```

Parameters

n: It refers to an integer value that returns the number of rows.

Return

It returns the DataFrame with top n rows.

Example:

```
import pandas as pd
data = pd.read_excel('demo.xlsx')
da = data.head()   # only 5 row
print(da)
```

To get only first two rows so head() method can have pass parameter 2 while calling.

Example:

```
da = data.head(2)   # only first 2 row
print(da)
```

Output:

```
   Eno Name Salary    dept
0  101  amit   2000 account
1  102 deep   2500   sales
```

tail () method

tail() method give last n (by default value 5) record from data frame .

Syntax

DataFrame.tail(n=5)

Parameters

n: It refers to an integer value that returns the number of rows.

Return

It returns the DataFrame with last n rows.

Example:

```
import pandas as pd
data = pd.read_excel('demo.xlsx')
da = data.tail()  # only 5 row
print(da)
```

Retrieving a Range of Rows

Using slicing we can retrieve the rows from the data frame objects .if we write data[3:6] then we get 2nd row to 5th row .

Example:

```
import pandas as pd
data = pd.read_excel('demo.xlsx')
da = data[2:6]        # 2nd row to 5th row
print(da)
```

As to display alternative rows, we can use data[0::2] or data[::0] as shown below.

Example:

```
import pandas as pd
data = pd.read_excel('demo.xlsx')
da = data[0::2] # alternate rows
print(da)
```

Output:

```
Eno  Name  Salary   dept
0  101  amit   2000   account
2  103   jay   2150  design
4  105  shree   1500  sales
```

To display data in the reverse order so we have to use negative steps in slicing as data[5:0:-1].

Example:

```
import pandas as pd
data = pd.read_excel('demo.xlsx')
da = data[5:0:-1] # reverse order
print(da)
```

Output:

```
   Eno  Name  Salary   dept
4  105  shree  1500    sales
3  104  nency  2100  account
2  103  jay    2150   design
1  102  deep   2500    sales
```

To retrieve Column Names

To retrieve the column names from the data frame we can use columns attribute as below.

Example:

```
import pandas as pd
data = pd.read_excel('demo.xlsx')
print(data.columns)
```

Output:

```
Index(['Eno', 'Name', 'Salary', 'dept'], dtype='object')
```

To Retrieve Selected column data

To retrieve the column data , we have to mention the column name as subscript .For Example data['Eno'] this will display all employee numbers. This can also be done using data.Eno.

Example:

```
>>>data.Eno
```

Output:

```
0    101
1    102
2    103
3    104
4    105
Name: Eno, dtype: int64
```

Same way to retrieve multiple columns we can provide list of column names as subscripts to data frame objects as data[[list of column name]].

Example:

```
>>>data[['Eno','Name']]
```

Output:

```
Eno    Name
0   101    amit
1   102    deep
2   103    jay
3   104    nency
4   105    shree
```

to_numpy()

The **DataFrame.to_numpy()** function is applied on the DataFrame that returns the numpy ndarray. For performing some high-level mathematical functions, we can convert Pandas DataFrame to numpy arrays. It uses the DataFrame.to_numpy() function.

Syntax:

```
DataFrame.to_numpy(dtype=None, copy=False)
```

Parameters

- **dtype**: It is an optional parameter that pass the dtype to numpy.asarray().
- **copy**: It returns the boolean value that has the default value False.

It ensures that the returned value is not a view on another array.

Returns

It returns the numpy.ndarray as an output.

Example:

```
import pandas as pd
#initializing the dataframe
info = pd.DataFrame([[17, 62, 35],[25, 36, 54],[42, 20, 15],[48, 62, 76]],
columns=['x', 'y', 'z'])
print('DataFrame\n----------\n', info)
#convert the dataframe to a numpy array
arr = info.to_numpy()
```

```
print('Numpy Array\n----------\n', arr)
```

Output:

```
DataFrame
----------
  x  y  z
0 17 62 35
1 25 36 54
2 42 20 15
3 48 62 76
Numpy Array
----------
 [[17 62 35]
 [25 36 54]
 [42 20 15]
 [48 62 76]]
```

- **Finding Maximum and Minimum Values**

The column containing numerical data can be apply max() method to find maximum value and min()method to find minimum value from that column.

Example: To get maximum and minimum salary

```
>>>data['Salary'].max()
```

Output:

```
2500
```

```
>>>data['Salary'].min()
```

Output:

```
1500
```

- **DataFrame.loc[]**

The DataFrame.loc[] is used to retrieve the group of rows and columns by labels or a boolean array in the DataFrame. It takes only index labels, and if it exists in the caller DataFrame, it returns the rows, columns, or DataFrame.

The **DataFrame.loc[]** is a label based but may use with the boolean array.

The allowed inputs for **.loc[]** are:

- Single label, e.g.,**7** or **a**. Here, **7** is interpreted as the label of the index.
- List or array of labels, e.g. ['x', 'y', 'z'].
- Slice object with labels, e.g. 'x':'f'.
- A boolean array of the same length. e.g. [True, True, False].
- **callable** function with one argument.

Syntax:

```
pandas.DataFrame.loc[]
```

Example:

```
import pandas as pd
# Creating the DataFrame
info = pd.DataFrame({'Age':[32, 41, 44, 38, 33],
                     'Name':['chetan', 'mayank', 'deep', 'raj', 'rahul']})
# Create the index
index_ = ['Row_1', 'Row_2', 'Row_3', 'Row_4', 'Row_5']

# Set the index
info.index = index_

# return the value
final = info.loc['Row_2', 'Name']

# Print the result
print(final)
```

Output:

```
mayank
```

DataFrame.iloc[]

The DataFrame.iloc[] is used when the index label of the DataFrame is other than numeric series of 0,1,2,....,n or in the case when the user does not know the index label.

We can extract the rows by using an imaginary index position which is not visible in the DataFrame. It is an integer- based position(from 0 to length-1 of the axis), but may also be used with the boolean array.

Syntax:

pandas.DataFrame.iloc[]

It can raise the **IndexError** if we request the index is out-of-bounds, except slice indexers, which allow the out-of-bounds indexing.

Example:

```
import numpy as np
import pandas as pd
mydict = [{'p': 2, 'q': 3, 'r': 4, 's': 5},
        {'p': 200, 'q': 300, 'r': 400, 's': 500},
        {'p': 2000, 'q': 3000, 'r': 4000, 's': 5000 }]
df = pd.DataFrame(mydict)
print(type(df.iloc[0]))
print(df.iloc[0])
```

Output:

```
pandas.core.series.Series
p  2
q  3
r  4
s  5
Name: 0, dtype: int64
```

Retrieving Statistical information

The **describe()** method is used for calculating some statistical data like percentile, mean and std of the numerical values of the Series or DataFrame. It analyzes both numeric and object series and also the DataFrame column sets of mixed data types.

Example:

```
>>>data.describe()
```

Output:

```
EnoSalary
Count  5.000000        5.000000
mean   103.000000      2050.000000
```

std	1.581139	360.555128
min	101.000000	1500.000000
25%	102.000000	2000.000000
50%	103.000000	2100.000000
75%	104.000000	2150.000000
Max	105.000000	2500.000000

Performing Queries on Data

We can retrieve rows based on a query. The query should be given as subscript in the data frame object. For Example to retrieve all employee whose salary is greater than 2000 rs :

```
>>>data[data.Salary>2000]
```

Output:

	Eno	Name	Salary	dept
1	102	deep	2500	sales
2	103	jay	2150	design
3	104	nency	2100	account

Sorting the Data

Pandas sort_values() function sorts a data frame in Ascending or Descending order of passed Column. It's different than the sorted Python function since it cannot sort a data frame and particular column cannot be selected. Returns a sorted Data Frame with Same dimensions as of the function caller Data Frame.

Syntax:

DataFrame.sort_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')

Parameters:

by: Single/List of column names to sort Data Frame by.

axis: 0 or 'index' for rows and 1 or 'columns' for Column.

ascending: Boolean value which sorts Data frame in ascending order if True.

inplace: Boolean value. Makes the changes in passed data frame itself if True.

kind: String which can have three inputs('quicksort', 'mergesort' or 'heapsort') of algorithm used to sort data frame.

na_position: Takes two string input 'last' or 'first' to set position of Null values. Default is 'last'.

Example:

sorting data frame by name

```
data.sort_values("Name", axis = 0, ascending = True, inplace = True,
na_position ='last')
```

Handling Missing Data

The most time consuming part of a data science project is data cleaning and preparation. However, there are many powerful tools to expedite this process. One of them is Pandas which is a widely used data analysis library for Python. Handling missing values is an essential part of data cleaning and preparation process because almost all data in real life comes with some missing values.

	A	B	C	D
1	Eno	Name	Salary	dept
2	101	amit	2000	account
3	102		2500	sales
4	103	jay	2150	
5	104	nency		account
6	105	shree	1500	sales
7				

Now we convert data into dataframe , the missing data in above file is represented by NaN(Not a Number), NaN is default marker for the missing value.

Example:

```
import pandas as pd
data = pd.read_excel('demo1.xlsx')
print(data)
```

Output:

	Eno	Name	Salary	dept
1	102	deep	2500	sales
2	103	jay	2150	design
3	104	nency	2100	account

We can use **fillna()** method to replace the Na or NaN values by a specified value.

```
>>>data.fillna(0)
```

Output:

```
   Eno   Name  Salary     dept
0  101   amit  2000.0  account
1  102      0  2500.0    sales
2  103    jay  2150.0        0
3  104  nency     0.0  account
4  105  shree  1500.0    sales
```

But this is not so useful as it is filling any type of column with zero. We can fill each column with a different value by passing column names and the value to be used to fill in the Column.

Example:

```
import pandas as pd
data = pd.read_excel('demo1.xlsx')/
print(data.fillna({'Name':'missing' , 'Salary':0 ,'dept' : 'admin'}))
```

Output:

```
   Eno     Name  Salary     dept
0  101     amit  2000.0  account
1  102  missing  2500.0    sales
2  103      jay  2150.0    admin
3  104    nency     0.0  account
4  105    shree  1500.0    sales
```

In above code we fill name missing value by missing word salary by 0 and dept by admin.

If we do not want missing data then we can remove all those missing data row by using **dropna()** method as:

```
import pandas as pd
data = pd.read_excel('demo1.xlsx')
print(data.dropna())
```

Output:

```
   Eno    Name  Salary     dept
0  101    amit  2000.0  account
4  105   shree  1500.0    sales
```

In this way , filling the necessary data or eliminating the missing data is called "***data cleansing***".

Python NumPy

NumPy stands for numeric python which is a python package for the computation and processing of the multidimensional and single dimensional array elements. **Travis Oliphant** created NumPy package in 2005 by injecting the features of the ancestor module Numeric into another module Numarray. It is an extension module of Python

which is mostly written in C. It provides various functions which are capable of performing the numeric computations with a high speed.
NumPy provides various powerful data structures, implementing multi-dimensional arrays and matrices. These data structures are used for the optimal computations regarding arrays and matrices. Using NumPy, mathematical and logical operations on arrays can be performed.

There are the following advantages of using NumPy for data analysis.

- NumPy performs array-oriented computing.
- It efficiently implements the multidimensional arrays.
- It performs scientific computations.
- It is capable of performing Fourier Transform and reshaping the data stored in multidimensional arrays.
- NumPy provides the in-built functions for linear algebra and random number generation.

Nowadays, NumPy in combination with SciPy and Mat-plotlib is used as the replacement to MATLAB as Python is more complete and easier programming language than MATLAB.

NumPy Environment Setup

NumPy doesn't come bundled with Python. We have to install it using the python pip installer. Execute the following command.

```
$ pip install numpy
```

The best way to enable NumPy is to use an installable binary package specific to your operating system. These binaries contain full SciPy stack (inclusive of NumPy, SciPy, matplotlib, IPython, SymPy and nose packages along with core Python).

Windows

Anaconda (from https://www.anaconda.com/products/individual) is a free Python distribution for SciPy stack. It is also available for Linux and Mac.

Linux

In Linux, the different package managers are used to install the SciPy stack. The package managers are specific to the different distributions of Linux. Let's look at each one of them.
Execute the following command on the terminal.(For Ubuntu)

```
$ sudo apt-get install python-numpy
$ python-scipy python-matplotlibipythonipythonnotebook python-pandas
$ python-sympy python-nose
```

NumPy Ndarray

Ndarray is the n-dimensional array object defined in the numpy which stores the collection of the similar type of elements. In other words, we can define a ndarray as the collection of the data type (dtype) objects. The ndarray object can be accessed by using the 0 based indexing. Each element of the Array object contains the same size in the memory.

Creating a ndarray object

The ndarray object can be created by using the array routine of the numpy module. For this purpose, we need to import the numpy.

Syntax:

numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)

Parameter	Description
Object	It represents the collection object. It can be a list, tuple, dictionary, set, etc.
Dtype	We can change the data type of the array elements by changing this option to the specified type. The default is none.
Copy	It is optional. By default, it is true which means the object is copied.
Order	There can be 3 possible values assigned to this option. It can be C (column order), R (row order), or A (any)
Subok	The returned array will be base class array by default. We can change this to make the subclasses passes through by setting this option to true.
ndmin	It represents the minimum dimensions of the resultant array.

Example:

```
import numpy as np
a = np.array([10,12,13])
print (a)
```

Output:

```
[10, 12, 13]
```

Example: two dimensions array

```
import numpy as np
a = np.array([[11, 22], [43, 54]])
print (a)
```

Output:

```
[[11, 22]
 [43, 54]]
```

Finding the dimensions of the Array

The ndim function can be used to find the dimensions of the array.

Example:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [4, 5, 6, 7], [9, 10, 11, 23]])
print(arr.ndim)
```

Output:

```
2
```

Finding the shape and size of the array

To get the shape and size of the array, the size and shape function associated with the numpy array is used.

Example:

```
import numpy as np
a = np.array([[1,2,3,4,5]])
print("Array Size:",a.size)
print("Shape:",a.shape)
```

Output:

```
Array Size: 5
Shape: (1, 5)
```

Numpy Methods

numpy.mean()

The sum of elements, along with an axis divided by the number of elements, is known as **arithmetic mean.** The numpy.mean() function is used to compute the arithmetic mean along the specified axis. This function returns the average of the array elements. By default, the average is taken on the flattened array. Else on the specified axis, float 64 is intermediate as well as return values are used for integer inputs

Syntax:

numpy.mean(a, axis=None, dtype=None, out=None, keepdims=<no value>)

These are the following parameters in numpy.mean() function:

a: array_like = This parameter defines the source array containing elements whose mean is desired.

axis(optional): This parameter defines the axis along which the means are computed. By default, the mean is computed of the flattened array.

dtype: (optional): This parameter is used to define the data-type used in computing the mean. For integer inputs, the default is float64, and for floating-point inputs, it is the same as the input dtype.

out: (optional)

This parameter defines an alternative output array in which the result will be placed.

keepdims: bool(optional)

When the value is true, the reduced axis is left as dimensions with size one in the output/result.

Example:

```
import numpy
speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]
x = numpy.mean(speed)
print(x)
```

Output:

89.7692307692307 7

Example:

```
import numpy as np
```

```
a = np.array([[1, 2], [3, 4]])
b=np.mean(a)
print(b)
```

Output:

```
2.5
13.0
```

We have imported numpy with alias name np. We have created two arrays 'a' using np.array() function and passed arrays 'a' in the function mean().

numpy.median()

Median is defined as the value separating the higher half of a data sample from the lower half. The function numpy.median() is used to calculate the median of the multi-dimensional or one-dimensional arrays.

Example:

```
import numpy
speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]
x = numpy.median(speed)
print(x)
```

Output:

```
87.0
```

Example:

```
import numpy as np
a = np.array([[10, 7, 4], [3, 2, 1]])
print("\nMedian of array along axis 0:",np.median(a,0))
```

Output:

Median of array along axis 0: [6.5 4.5 2.5]

Mode()

The Mode value is the value that appears the most number of times. The mode is the number that occurs with the greatest frequency within a data set. The SciPy module has a method for mode. scipy.stats.mode() function calculates the mode of the array elements along the specified axis of the array (list in python).

Syntax:

```
scipy.stats.mode(array, axis=0)
```

These are the following parameters in scipy.stats.mode() function:

array : Input array or object having the elements to calculate the mode.
axis : Axis along which the mode is to be computed. By default axis = 0.
Returns : Modal values of the array elements based on the set parameters.

Example:

```
import numpy as np
from scipy import stats
dataset= [1,1,2,3,4,6,18]
#mode value
mode= stats.mode(dataset)
print("Mode: ", mode)
```

Output:

```
Mode:  ModeResult(mode=array([1]), count=array([2]))
```

ModeResult, is returned that has 2 attributes. The first attribute, mode, is the number that is the mode of the data set. The second attribute, count, is the number of times it occurs in the data set.

numpy standard deviation(numpy.std())

Method std() is used to compute the standard deviation along the specified axis. This function returns the standard deviation of the array elements.

Syntax:

numpy.std(a,axis=None,dtype=None,out=None)

These are the following parameters:

arr : [array_like] input array.
axis : [int or tuples of int] axis along which we want to calculate the standard deviation.
out : [ndarray, optional]Different array in which we want to place the result.
dtype : [data-type, optional]Type we desire while computing SD.

Example : one dimensional array

```
import numpy as np
arr = [20, 2, 7, 1, 34]
print("arr : ", arr)
print("std of arr : ", np.std(arr))
```

```
print ("\nMore precision with float32")
print("std of arr : ", np.std(arr, dtype = np.float32))
print ("\nMore accuracy with float64")
print("std of arr : ", np.std(arr, dtype = np.float64))
```

Output:

```
arr :  [20, 2, 7, 1, 34]
std of arr :  12.576167937809991

More precision with float32
std of arr :  12.576168

More accuracy with float64
std of arr :  12.576167937809991
```

Example: two dimensional array

```
import numpy as np
# 2D array
arr = [[2, 2, 2, 2, 2],
       [15, 6, 27, 8, 2],
       [23, 2, 54, 1, 2, ],
       [11, 44, 34, 7, 2]]
# std of the flattened array
print("\nstd of arr, axis = None : ", np.std(arr))
# std along the axis = 0
print("\nstd of arr, axis = 0 : ", np.std(arr, axis = 0))
# std along the axis = 1
print("\nstd of arr, axis = 1 : ", np.std(arr, axis = 1))
```

Output:

```
std of arr, axis = None :  15.3668474320532
std of arr, axis = 0 :  [ 7.56224173 17.68473918 18.59267329  3.04138127  0.
]

std of arr, axis = 1 :  [ 0.          8.7772433  20.53874388 16.40243884]
```

Variance (numpy.var())

The variance is the average of the squared deviations from the mean. numpy.var(arr, axis = None) is used to Compute the variance of the given data (array elements) along the specified axis(if any).

numpy.var(a,axis=None,dtype=None,out=None)

These are the following parameters:

arr : [array_like] input array.

axis : [int or tuples of int] axis along which we want to calculate the standard deviation.

out : [ndarray, optional]Different array in which we want to place the result.

dtype : [data-type, optional]Type we desire while computing SD.

Example : one dimensional array

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
np.var(a)
```

Output:

1.25

Example :

```
import numpy as np

# 2D array
arr = [[2, 2, 2, 2, 2],
   [15, 6, 27, 8, 2],
   [23, 2, 54, 1, 2, ],
   [11, 44, 34, 7, 2]]
# var of the flattened array
print("\nvar of arr, axis = None : ", np.var(arr))
# var along the axis = 0
print("\nvar of arr, axis = 0 : ", np.var(arr, axis = 0))

# var along the axis = 1
print("\nvar of arr, axis = 1 : ", np.var(arr, axis = 1))
```

Output :

```
var of arr, axis = None : 236.14000000000004
var of arr, axis = 0 : [ 57.1875 312.75   345.6875  9.25    0.   ]
var of arr, axis = 1 : [  0.    77.04 421.84 269.04]
```

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the dtype keyword can alleviate this issue.

Exercise

Short Question

1. Explain close() method.
2. List File modes.
3. Define CSV module
4. Define the Python pandas?
5. Explain Dataframe in Pandas.
6. Define Series in Pandas?
7. How can we calculate the standard deviation from the Series?

Long Question

1. Explain open() in python to work with file.
2. Explain reader(), writer(), writerows(), DictReader(), DictWriter() with example.
3. Explain the different types of Data Structures in Pandas?
4. Define the different ways a DataFrame can be created in pandas?
5. How to Extract specific attributes and rows from dataframe?
6. Explain following function of Dataframe: head, tail, loc, iloc, value, to_numpy(), describe()/
7. Explain Numpy Methods mean(),median(),mode(),std(),var()

UNIT - 5

Data Visualization using data frame

Scanned with OKEN Scanner

Data Visualization

Data visualization is the technique to present the data in a pictorial or graphical format. It enables stakeholders and decision makers to analyze data visually. The data in a graphical format allows them to identify new trends and patterns easily.

Matplotlib Python Libraries

matplotlib is a python two-dimensional plotting library for data visualization and creating interactive graphics or plots. Using pythons matplotlib, the data visualization of large and complex data becomes easy.

matplotlib.pyplot is a plotting library used for 2D graphics in python programming language. It can be used in python scripts, shell, web application servers and other graphical user interface toolkits.

matplotlib.pyplot is a collection of command style functions that make Matplotlib work like MATLAB. Each Pyplot function makes some change to a figure. For example, a function creates a figure, a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

install matplotlib

To use matplotlib, we need to install it.

Step 1 – Make sure Python and pip is preinstalled on your system

Type the following commands in the command prompt to check is python and pip is installed on your system.

To check Python

python --version

If python is successfully installed, the version of python installed on your system will be displayed.

To check pip

pip –V

The version of pip will be displayed, if it is successfully installed on your system.

Step 2 – Install Matplotlib

Matplotlib can be installed using pip. The following command is run in the command prompt to install Matplotlib.

pip install matplotlib

This command will start downloading and installing packages related to the matplotlib library. Once done, the message of successful installation will be displayed.

Step 3 – Check if it is installed successfully

To verify that matplotlib is successfully installed on your system, execute the following command in the command prompt. If matplotlib is successfully installed, the version of matplotlib installed will be displayed.

import matplotlib
matplotlib.__version__

importing matplotlib.pyplot

Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias:

```
import matplotlib.pyplot as plt
```

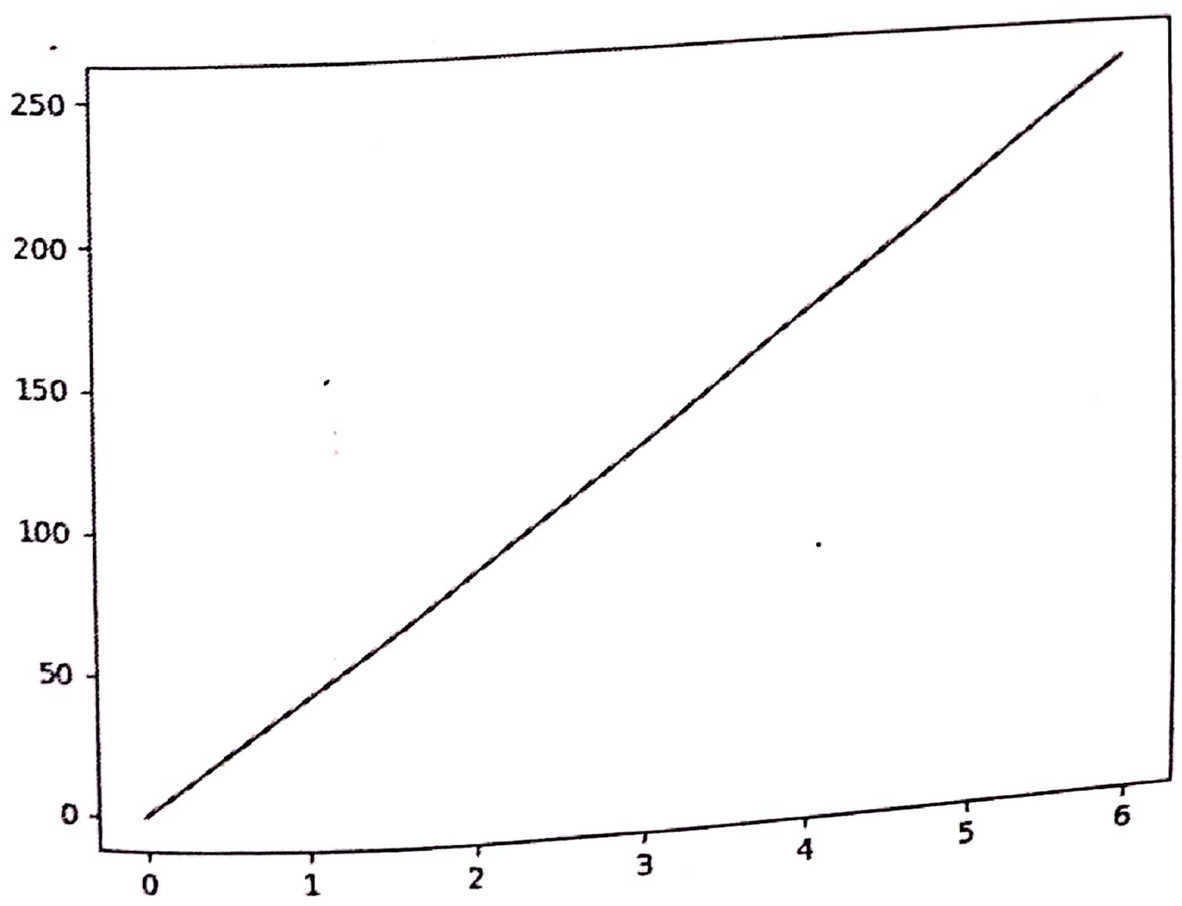
Now the Pyplot package can be referred to as plt.

Example

Draw a line in a diagram from position (0,0) to position (6,250):

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([0, 6])
ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)
plt.show()
```

Output:



Matplotlib.pyplot.legend()

A legend is an area describing the elements of the graph. In the matplotlib library, there's a function called legend() which is used to Place a legend on the axes.

The attribute Loc in legend() is used to specify the location of the legend. Default value of loc is loc="best" (upper left). The strings 'upper left', 'upper right', 'lower left', 'lower right' place the legend at the corresponding corner of the axes/figure.

The attribute bbox_to_anchor=(x, y) of legend() function is used to specify the coordinates of the legend, and the attribute ncol represents the number of columns that the legend has. It's default value is 1.

Syntax:

matplotlib.pyplot.legend(["blue", "green"], bbox_to_anchor=(0.75, 1.15), ncol=2)

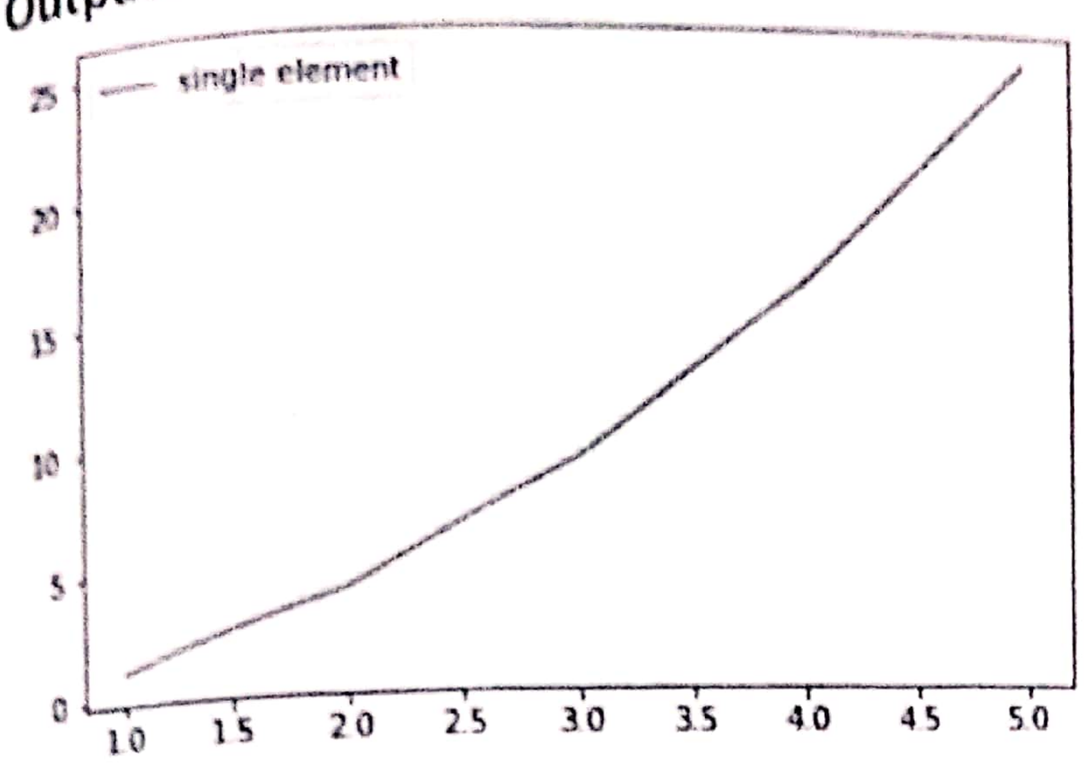
The Following are some more attributes of function legend() :

- **shadow**: [None or bool] Whether to draw a shadow behind the legend. It's Default value is None.
- **markerscale**: [None or int or float] The relative size of legend markers compared with the originally drawn ones. The Default is None.
- **numpoints**: [None or int] The number of marker points in the legend when creating a legend entry for a Line2D (line).The Default is None.
- **fontsize**: The font size of the legend. If the value is numeric the size will be the absolute font size in points.
- **facecolor**: [None or "inherit" or color] The legend's background color.
- **edgecolor**: [None or "inherit" or color] The legend's background patch edge color.

Example:

```
import numpy as np
import matplotlib.pyplot as plt
# X-axis values
x = [1, 2, 3, 4, 5]
# Y-axis values
y = [1, 4, 9, 16, 25]
# Function to plot
plt.plot(x, y)
# Function add a legend
plt.legend(['single element'])
# function to show the plot
plt.show()
```

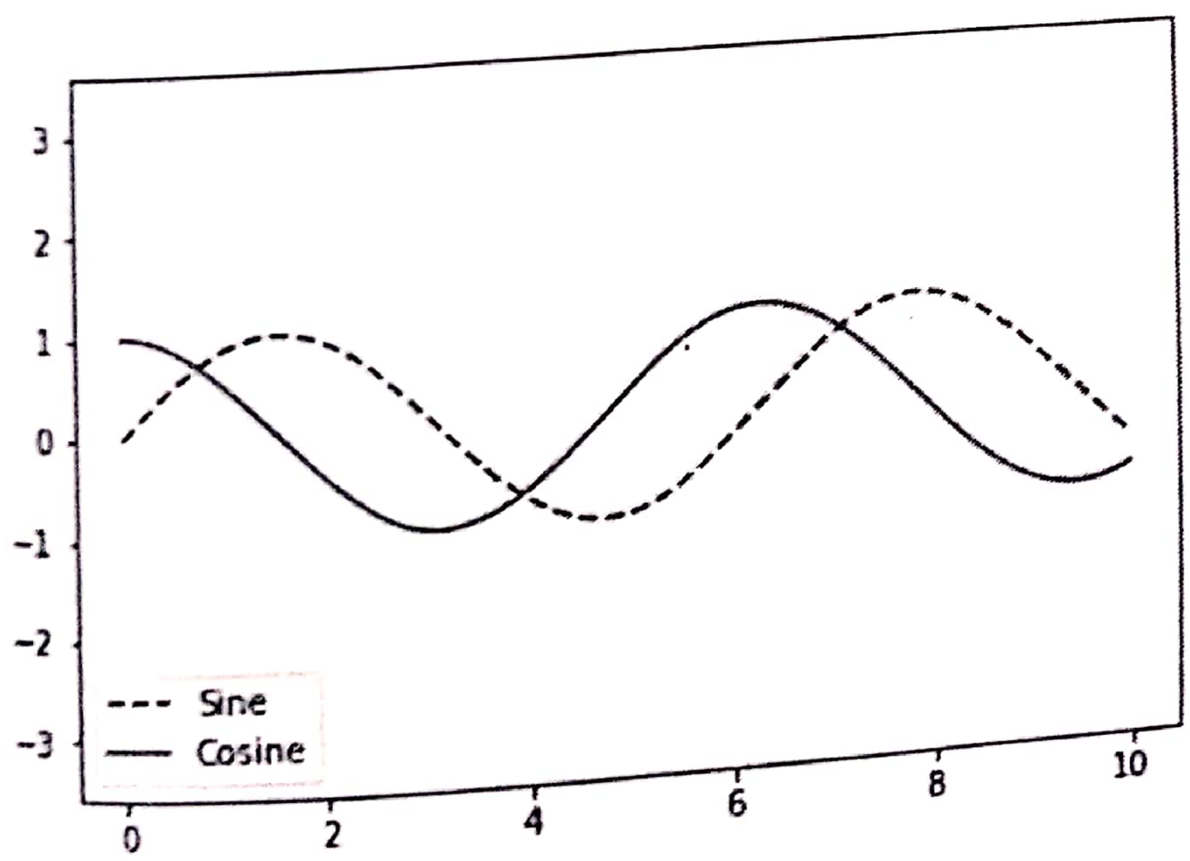
Output:



Example:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 1000)
fig, ax = plt.subplots()

ax.plot(x, np.sin(x), '--b', label ='Sine')
ax.plot(x, np.cos(x), c ='r', label ='Cosine')
ax.axis('equal')

leg = ax.legend(loc ="lower left");
```

Output:



Subplots

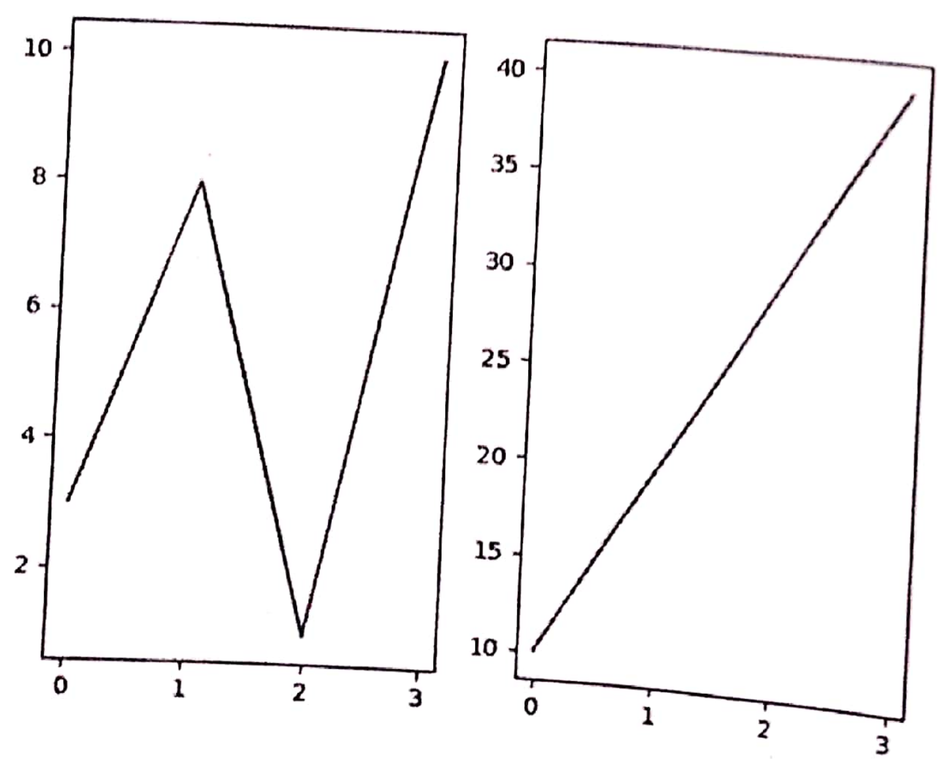
With the subplots() function you can draw multiple plots in one figure:

Example:

Draw 2 plots:

```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(1, 2, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.show()
```

Output:



Scatter Plot in Matplotlib

Scatter plots are great for visualizing data points in two dimensions. They're particularly useful for showing correlations and groupings in data.

In matplotlib, you can create a scatter plot using the pyplot's scatter() function. The following is the **syntax:**

```
plt.scatter(x_values, y_values)
```

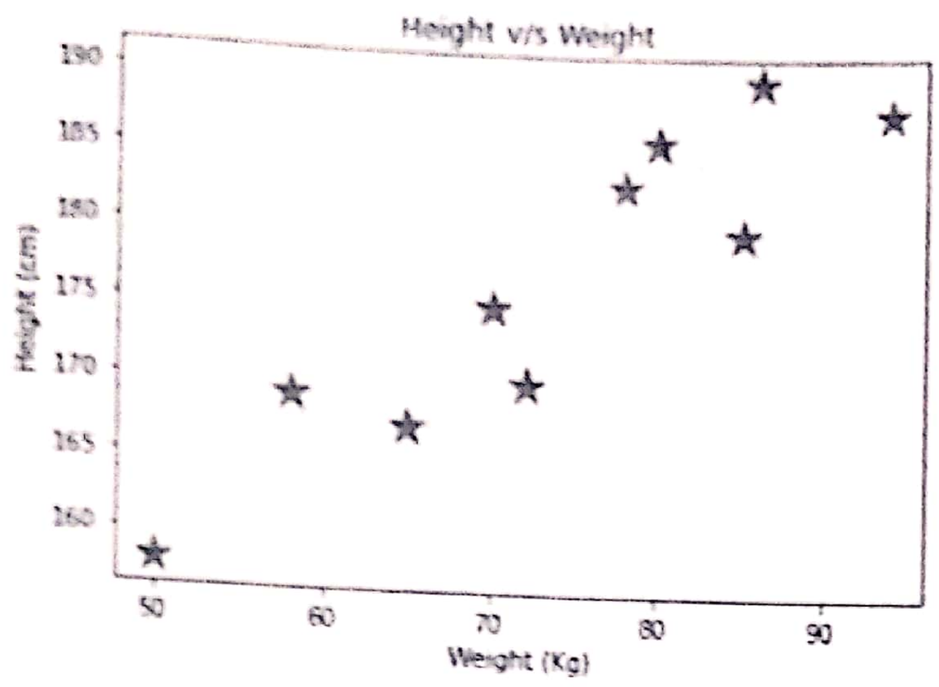
Here, x_values are the values to be plotted on the x-axis and y_values are the values to be plotted on the y-axis.

Examples:

We have the data for heights and weights of 10 students at a university and want to plot a scatter plot of the distribution between them. The data is present in two lists. One having the height and the other having the corresponding weights of each student.

```
import matplotlib.pyplot as plt

# height and weight data

height = [167, 175, 170, 186, 190, 188, 158, 169, 183, 180]

weight = [65, 70, 72, 80, 86, 94, 50, 58, 78, 85]


# plot a scatter plot with star markers

plt.scatter(weight, height, marker='*', s=80)

# set axis lables

plt.xlabel("Weight (Kg)")

plt.ylabel("Height (cm)")

# set chart title

plt.title("Height v/s Weight")

plt.show()
```

Output:



You can alter the shape of the marker with the marker parameter and size of the marker with the s parameter of the scatter() function. Matplotlib's pyplot has handy functions to add axis labels and title to your chart.

Line chart in Matplotlib

A line chart or line graph is a type of chart which displays information as a series of data points called 'markers' connected by straight line segments.

Line graphs are usually used to find relationship between two data sets on different axis; for instance X, Y.

Line charts are used to represent the relation between two data X and Y on a different axis. In matplotlib, you can plot a line chart using pyplot's plot() function. The following is the syntax to plot a line chart:

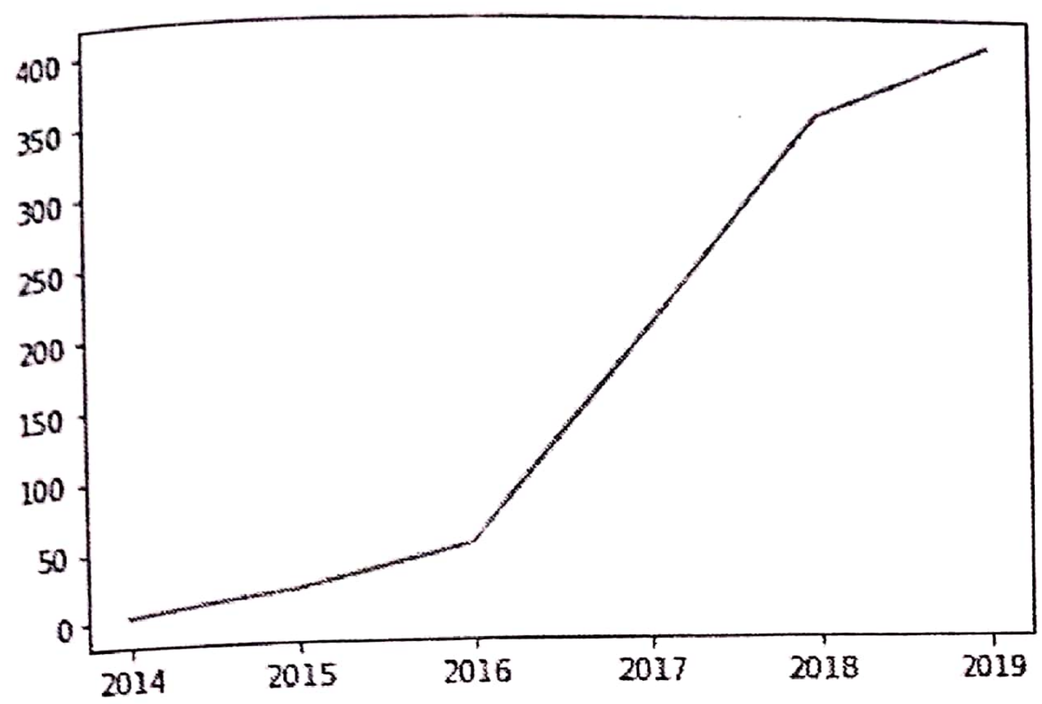
plt.plot(x_values, y_values)

Here, x_values are the values to be plotted on the x-axis and y_values are the values to be plotted on the y-axis.

Example:

```
import matplotlib.pyplot as plt

# number of employees of A
emp_count = [3, 20, 50, 200, 350, 400]
year = [2014, 2015, 2016, 2017, 2018, 2019]

# plot a line chart
plt.plot(year, emp_count)
plt.show()
```

Output:



You can see in the above chart that we have the year on the x-axis and the employee count on the y-axis. The chart shows an upward trend in the employee count at the company A year on year.

Matplotlib's pyplot comes with handy functions to set the axis labels and chart title. You can use pyplot's xlabel() and ylabel() functions to set axis labels and use pyplot's title() function to set the title for your chart.

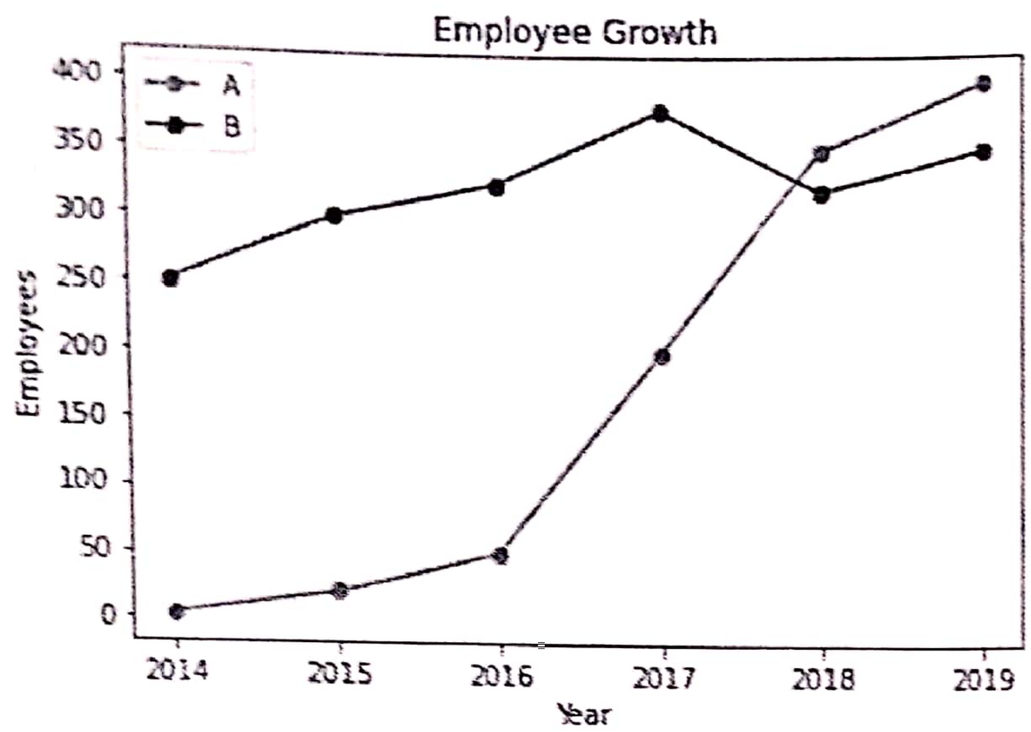
Plot multiple lines in a single chart

Matplotlib also allows you to plot multiple lines in the same chart. Generally used to show lines that share the same axis, for example, lines sharing the x-axis. The y-axis can also be shared if the second series has the same scale, or if the scales are different you can also plot on two different y-axes. Let's look at examples for both cases.

Example:

```
import matplotlib.pyplot as plt
# number of employees
emp_countA = [3, 20, 50, 200, 350, 400]
emp_countB = [250, 300, 325, 380, 320, 350]
year = [2014, 2015, 2016, 2017, 2018, 2019]

# plot two lines
plt.plot(year, emp_countA, 'o-g')
plt.plot(year, emp_countB, 'o-b')
# set axis titles
plt.xlabel("Year")
plt.ylabel("Employees")
# set chart title
plt.title("Employee Growth")
# legend
plt.legend(['A', 'B'])
plt.show()
```

Output:



Both lines share the same axes. Also note, that we added a legend to easily distinguish between the two companies.
With Pyplot, you can use the xlabel() and ylabel() functions to set a label for the x- and y-axis.

Plot Histogram in Matplotlib

Histograms show the frequency distribution of values of a variable across different buckets. They are great for visualizing the distribution of a variable.
A histogram is basically used to represent data provided in a form of some groups. It is accurate method for the graphical representation of numerical data distribution. It is a type of bar plot where X-axis represents the bin ranges while Y-axis gives information about frequency.

Creating a Histogram

To create a histogram the first step is to create bin of the ranges, then distribute the whole range of the values into a series of intervals, and the count the values which fall into each of the intervals. Bins are clearly identified as consecutive, non-overlapping intervals of variables.
The **matplotlib.pyplot.hist()** function is used to compute and create histogram of x. The following table shows the parameters accepted by matplotlib.pyplot.hist() function :

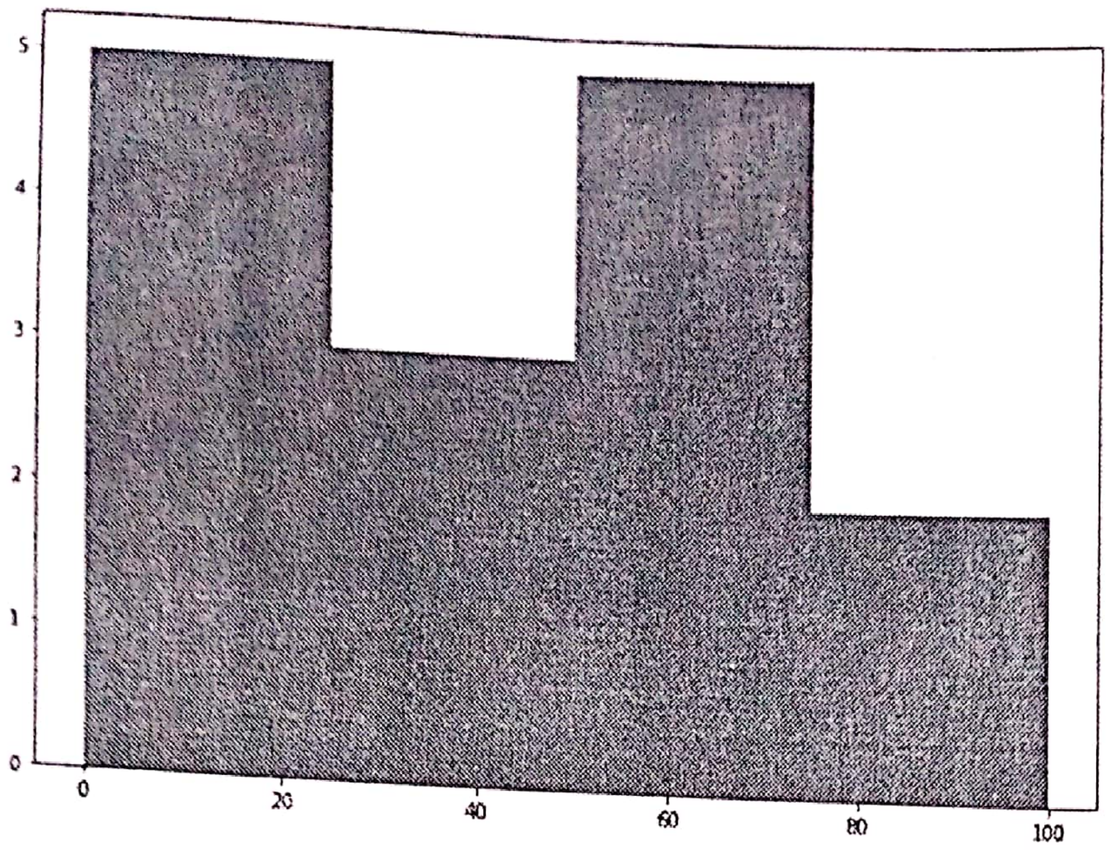
Attribute	parameter
x	array or sequence of array
bins	optional parameter contains integer or sequence or strings
density	optional parameter contains boolean values
range	optional parameter represents upper and lower range of bins

histtype	optional parameter used to create type of histogram [bar, barstacked, step, stepfilled], default is "bar"
align	optional parameter controls the plotting of histogram [left, right, mid]
weights	optional parameter contains array of weights having same dimensions as x
bottom	location of the baseline of each bin
rwidth	optional parameter which is relative width of the bars with respect to bin width
color	optional parameter used to set color or sequence of color specs
label	optional parameter string or sequence of string to match with multiple datasets
log	optional parameter used to set histogram axis on log scale

Let's create a basic histogram of some random values. Below code creates a simple histogram of some random values:

```
from matplotlib import pyplot as plt
import numpy as np

# Creating dataset
a = np.array([22, 87, 5, 43, 56,
              73, 55, 54, 11,
              20, 51, 5, 79, 31,
              27])

# Creating histogram
fig, ax = plt.subplots(figsize =(10, 7))
ax.hist(a, bins = [0, 25, 50, 75, 100])
# Show plot
plt.show()
```

Output:

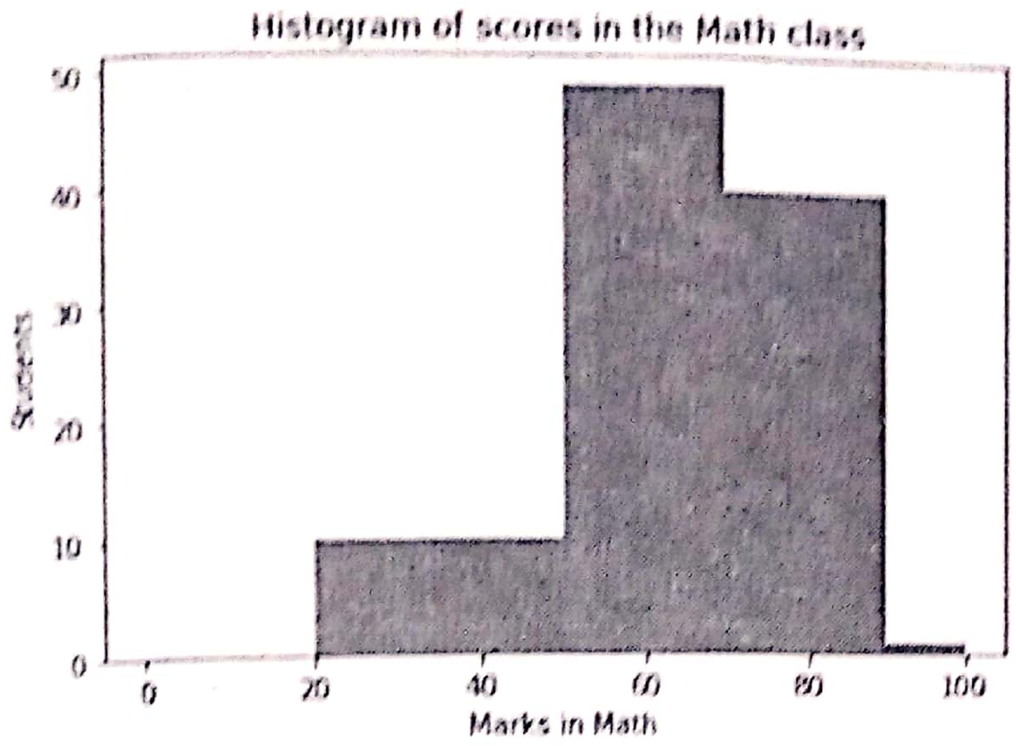


You can also specify your own **bin edges** which can be unequally spaced. For this, instead of passing an integer to the **bins parameter**, pass a sequence with the bin edges. For example, if you want to have bins 0 to 20, 20 to 50, 50 to 70, 70 to 90, and 90 to 100 :

Example:

```
import matplotlib.pyplot as plt

# scores in the Math class
math_scores = [72, 41, 65, 63, 82, 63, 51, 57, 39, 63,62, 68, 52, 76, 62, 73, 72,
73, 71, 62,76, 53, 71, 79, 77, 35, 65, 59, 58, 70,73, 69, 59, 75, 73, 63, 65, 81, 46,
59,53, 71, 79, 80, 60, 60, 64, 40, 73, 75, 68, 58, 81, 65, 55, 62, 82, 47, 85, 62,
39, 77, 82, 78, 57, 58, 72, 75, 65, 68, 86, 49, 39, 64, 54, 68, 85, 77, 62, 53,52,
76, 80, 84, 69, 61, 69, 65, 89, 97, 71, 61, 77, 40, 83, 52, 78, 54, 64, 58]

# specify the bin edges
bin_edges = [0,20,50,70,90,100]

# plot histogram
plt.hist(math_scores, bins=bin_edges)
# add formatting
plt.xlabel("Marks in Math")
plt.ylabel("Students")
plt.title("Histogram of scores in the Math class")
plt.show()
```

Output:



Here, the bins are unequally spaced because of the bin edges specified. Matplotlib's hist() function also has a number of other parameters to customize your plots even further.

Bar Plot in Matplotlib

A bar plot or bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. The bar plots can be plotted horizontally or vertically. A bar chart describes the comparisons between the discrete categories. One of the axis of the plot represents the specific categories being compared, while the other axis represents the measured values corresponding to those categories.

The **syntax of the bar()** function to be used with the axes is as follows:-

```
plt.bar(x, height, width, bottom, align)
```

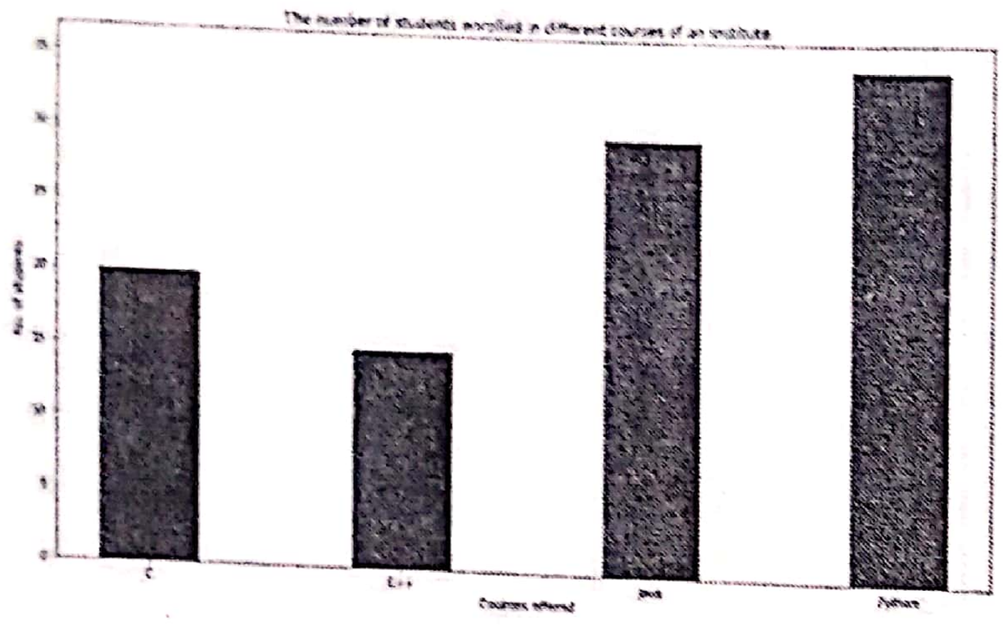
The function creates a bar plot bounded with a rectangle depending on the given parameters. Following is a simple example of the bar plot, which represents the number of students enrolled in different courses of an institute.

Example:

```
import numpy as np
import matplotlib.pyplot as plt


# creating the dataset
data = {'C':20, 'C++':15, 'Java':30,
                'Python':35}
courses = list(data.keys())
values = list(data.values())

fig = plt.figure(figsize = (10, 5))
# creating the bar plot
```

```
plt.bar(courses, values, color ='maroon',
                width = 0.4)

plt.xlabel("Courses offered")
plt.ylabel("No. of students enrolled")
plt.title("Students enrolled in different courses")
plt.show()
```

Output:



Here **plt.bar(courses, values, color='maroon')** is used to specify that the bar chart is to be plotted by using the courses column as the X-axis, and the values as the Y-axis. The **color attribute** is used to set the color of the bars(maroon in this case).
plt.xlabel("Courses offered") and **plt.ylabel("students enrolled")** are used to label the corresponding axes.
plt.title() is used to make a title for the graph.
plt.show() is used to show the graph as output using the previous commands.

Pie chart in Matplotlib

A Pie Chart can only display one series of data. Pie charts show the size of items (called wedge) in one data series, proportional to the sum of the items. The data points in a pie chart are shown as a percentage of the whole pie.

Matplotlib API has a pie() function that generates a pie diagram representing data in an array. The fractional area of each wedge is given by x/sum(x). If sum(x)< 1, then the values of x give the fractional area directly and the array will not be normalized. The resulting pie will have an empty wedge of size 1 - sum(x).

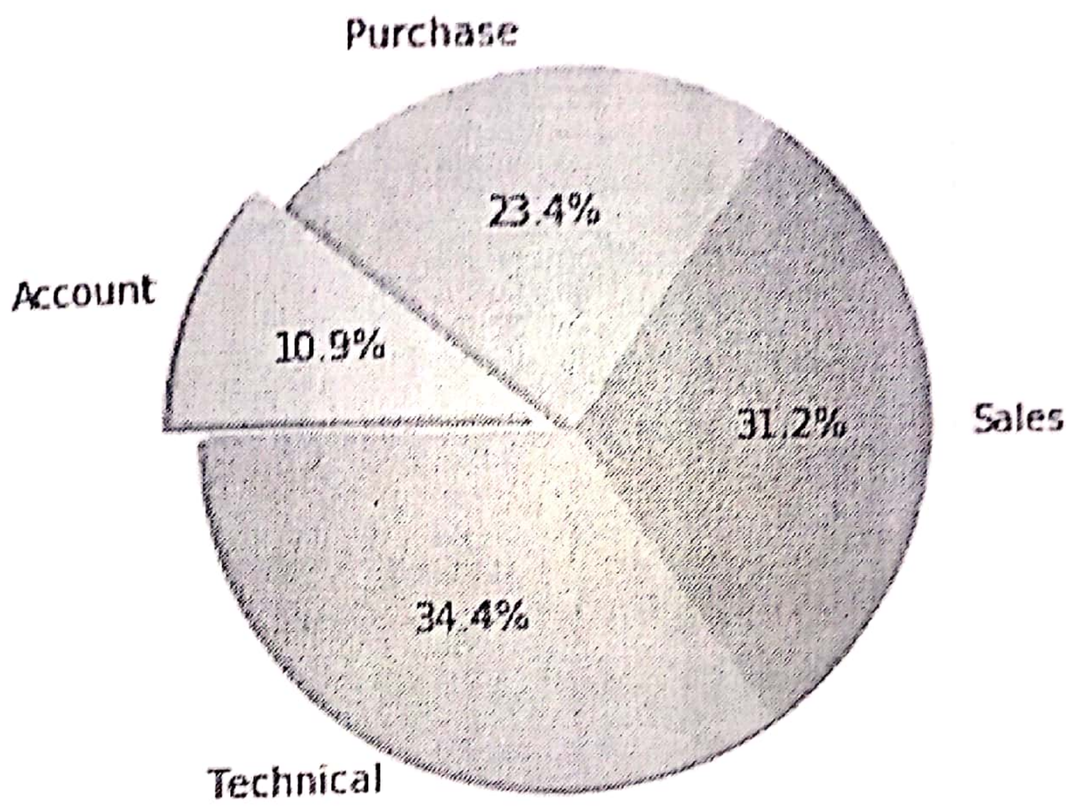
Example:

pie chart showing the percentage of employees in each department of a company.

```
import matplotlib.pyplot as plt

# Data to plot
labels = 'Account', 'Technical', 'Sales', 'Purchase'
noOfEmp = [7, 22, 20, 15]
```

```
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']
explode = (0.1, 0, 0, 0)  # explode 1st slice
# Plot
plt.pie(noOfEmp, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%',
shadow=True, startangle=140)
plt.axis('equal')
plt.show()
```

Output:



Save Plot as a File

Matplotlib is a library in python that offers a number of plotting options to display your data. The plots created get displayed when you use plt.show() but you cannot access them later since they're not saved on disk.
To save a figure created with matplotlib, you can use pyplot's savefig() function. This way, you'll have the plots saved on disk for further use instead of having to plot them all over again.
Syntax:

```
import matplotlib.pyplot as plt
plt.savefig("filename.png")
```

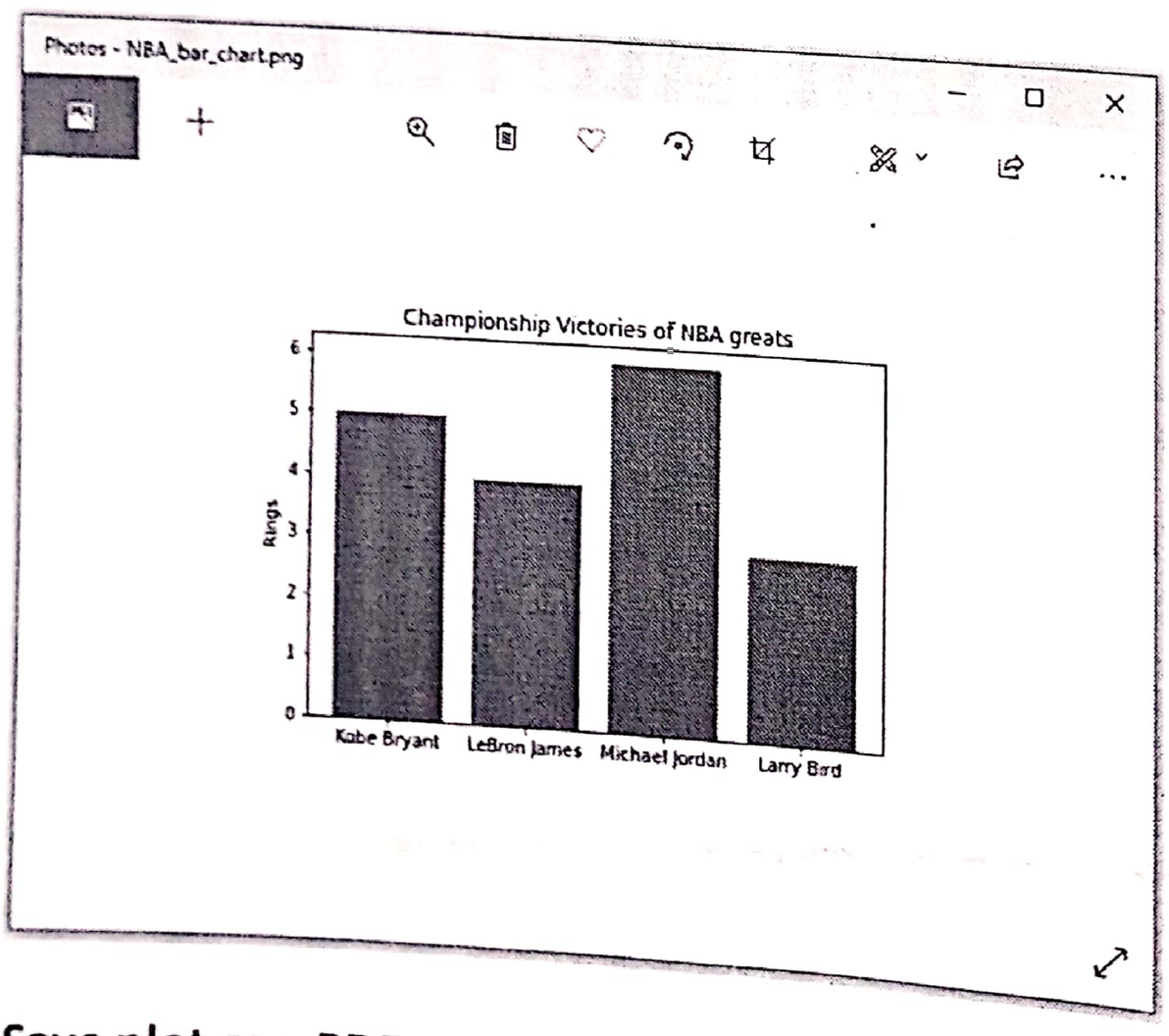
Pass the path where you want the image to be saved. The savefig() function also comes with a number of additional parameters to further customize how your image gets saved.

Save a plot to an image file:

Examples:

```
# NBA championship counts
players = ['Kobe Bryant', 'LeBron James', 'Michael Jordan', 'Larry Bird']
titles = [5,4,6,3]

# plot a bar chart
plt.bar(players, titles)
# add y-axis label
plt.ylabel("Rings")
# add chart title
plt.title("Championship Victories of NBA greats")

# save the plot as a PNG image
plt.savefig("NBA_bar_chart.png")
```

This saves the bar chart as a PNG file with the name *NBA_bar_chart.png* to the current directory. You can specify the path and name of your image as per your needs. This is how the saved plot looks on opening it with an image viewer application:



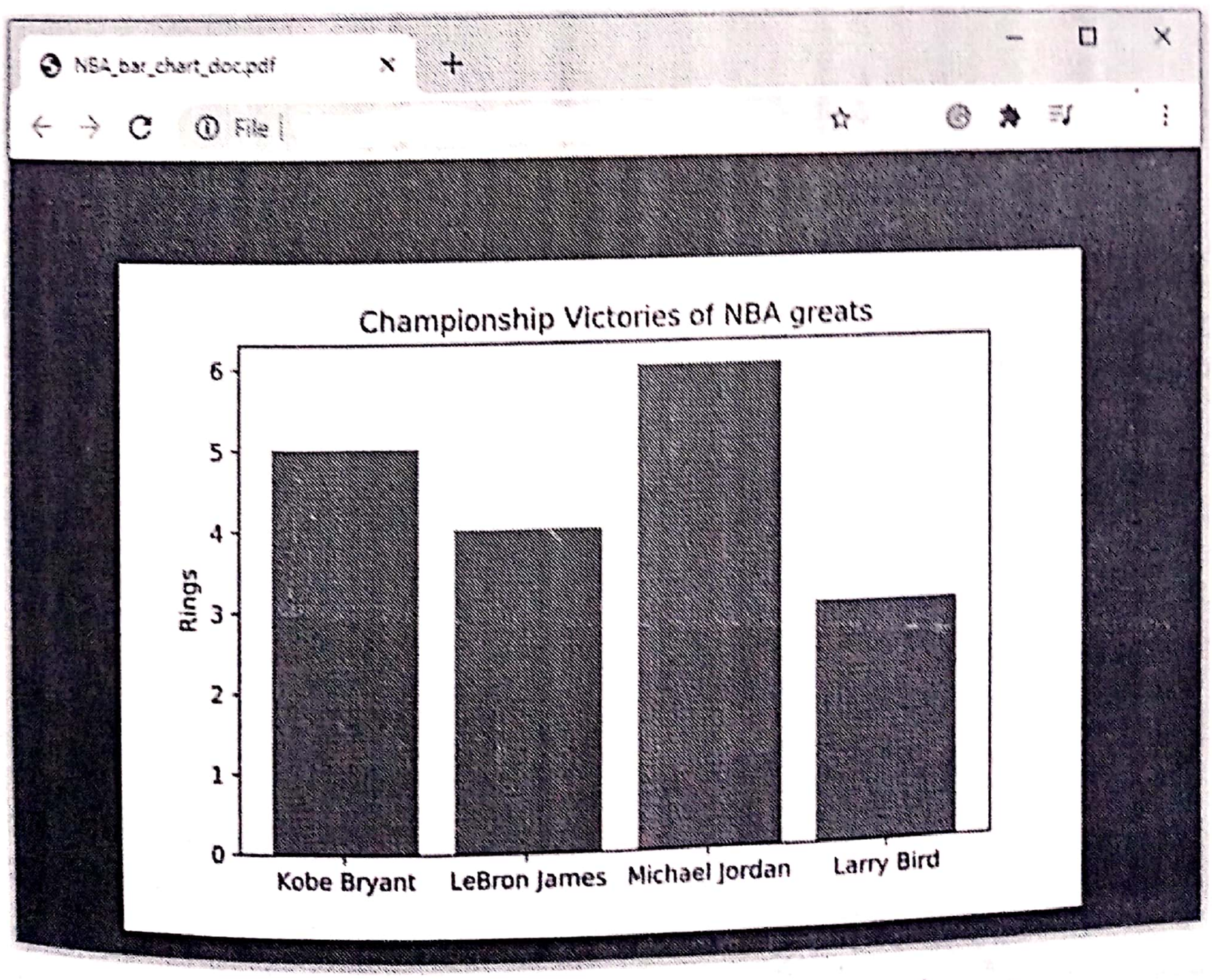
Save plot as a PDF:

Depending on the filename provided plt.savefig() infers the format of the output file. For instance, if you want to save the above image as a PDF file, just use the appropriate file name:

Example:

```
import matplotlib.pyplot as plt

# NBA championship counts
players = ['Kobe Bryant', 'LeBron James', 'Michael Jordan', 'Larry Bird']
titles = [5,4,6,3]

# plot a bar chart
plt.bar(players, titles)
# add y-axis label
plt.ylabel("Rings")
# add chart title
plt.title("Championship Victories of NBA greats")

# save the plot as a PDF
plt.savefig("NBA_bar_chart_doc.pdf")
```

The above code saves the plot as a PDF file with the name NBA_bar_chart_doc.pdf to the current directory. This is how the saved images looks like on opening it in Google Chrome web browser:



Exercise

Short Question

1. Explain Data Visualization.
2. Which is a python package used for 2D graphics?
3. Define Matplotlib Python Libraries.
4. Define set_title() method.
5. Explain xlabel and ylabel.

Long Question

1. How to install matplotlib.
2. Explain range() , subplot() , legend(), columns(), len() functions.
3. Explain scatter plot with example.
4. Explain Concepts of Bar chart .discuss bar(),set title, xlabel and ylabel in detail.
5. Write about the Concepts of histogram chart with code and example.

RAT
3498
ACADEMY EDUCATION TRUST
& SDHG COLLEGE OF BCA & IT
SELF FINANCED BCA LIBRARY
SURAT