What is SQLite

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 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.

Year	Happenings
2000	SQLite was designed by D. Richard Hipp 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).

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.

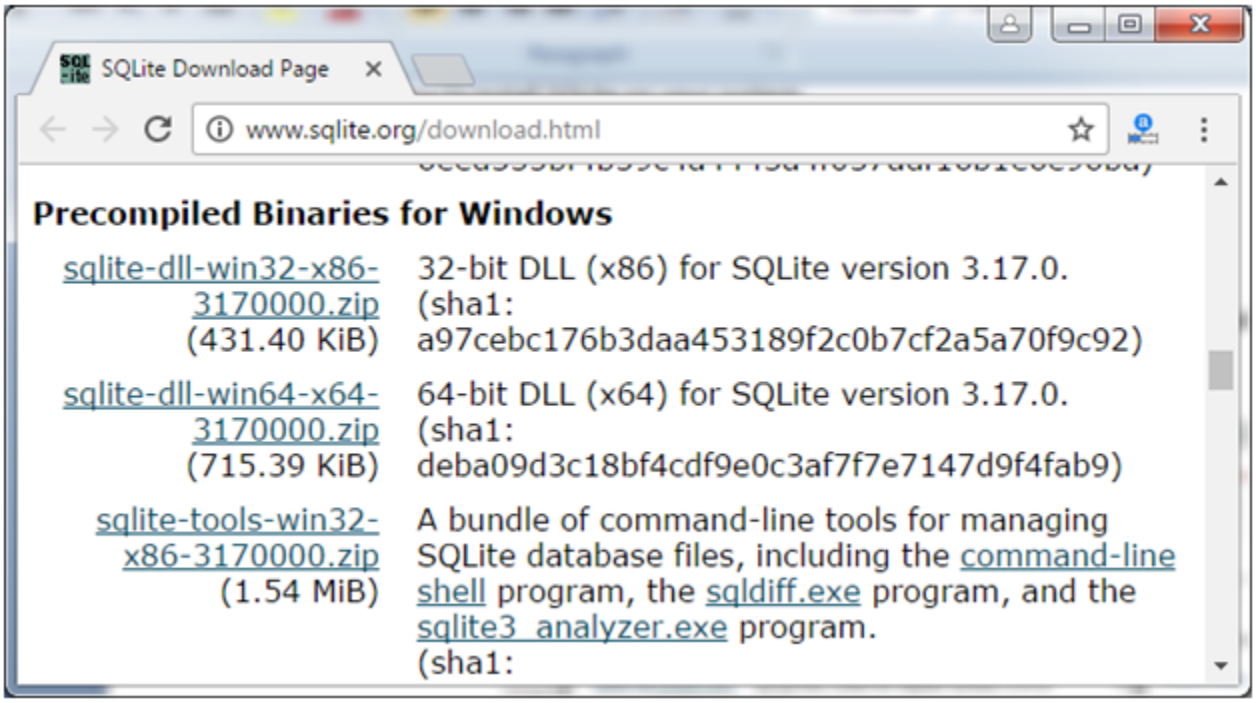
SQLite Installation

SQLite is known for its zero configuration which means no complex setup or administration is required. Let's see how to install SQLite on your system.

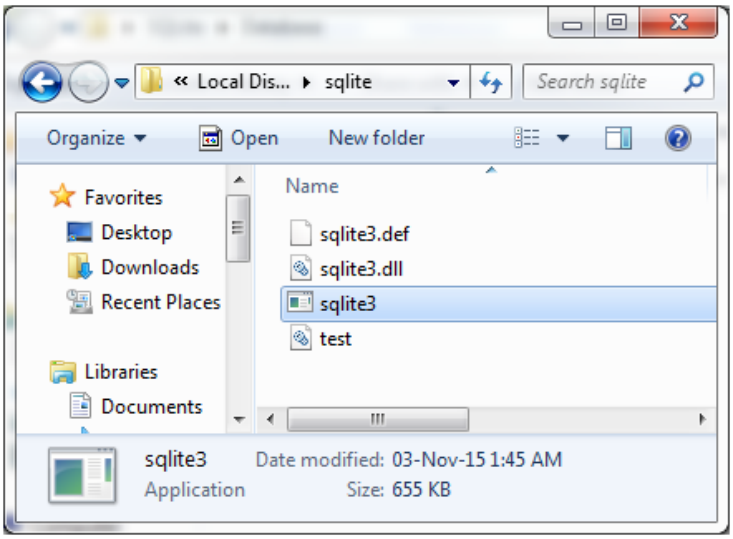
Install SQLite on Windows

Follow the steps given below:

- Go to SQLite official website download page http://www.sqlite.org/download.html And download precompiled binaries from Windows section.



- Download the sqlite-dll and sqlite-shell zip file. Or sqlite-tools-win32-x86-3170000.zip file.
- Create a folder named sqlite in C directory and expand these files.

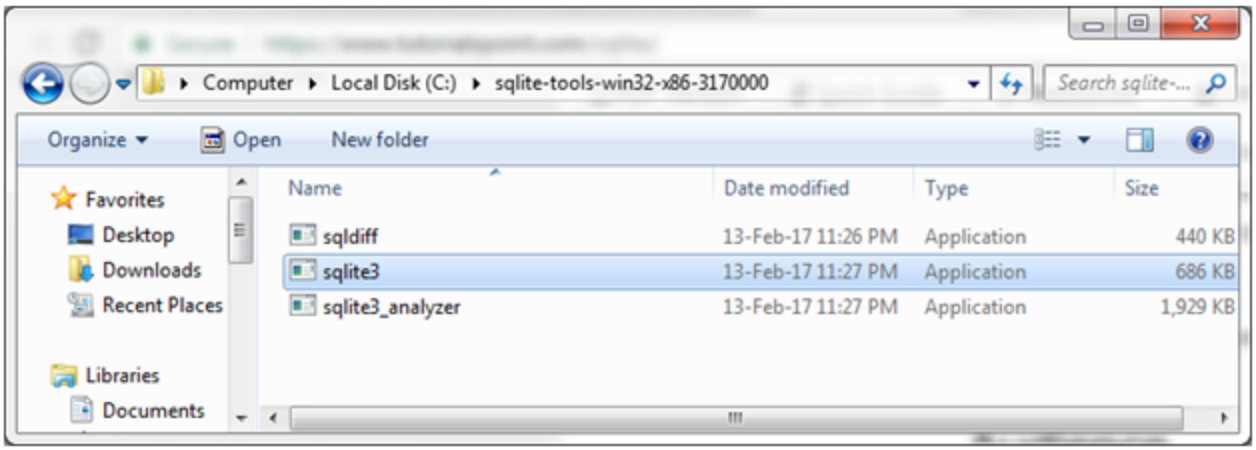


- Open command prompt to set the path. Set your PATH environment variable and open sqlite3 command. It will look like this:

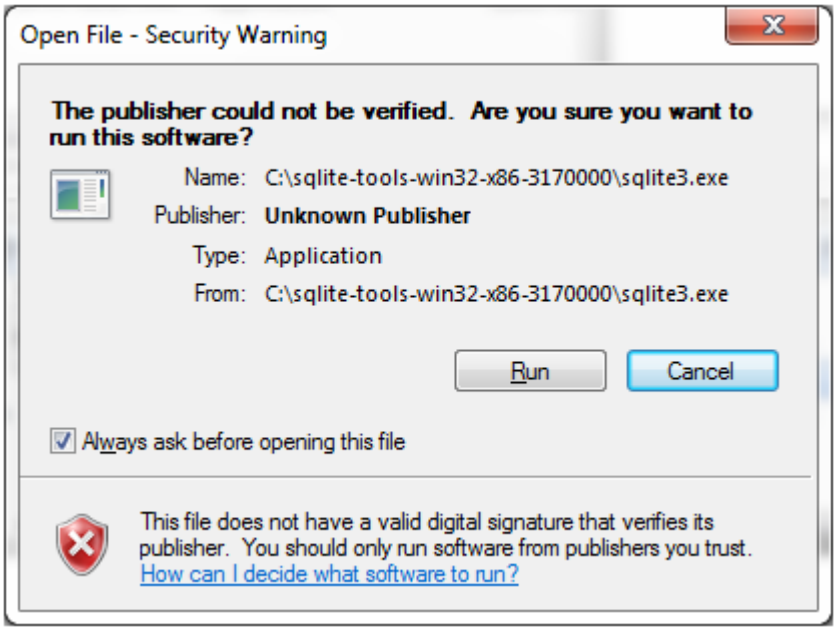
The above method facilitates you a permanent way to create database, attach database and detach database.

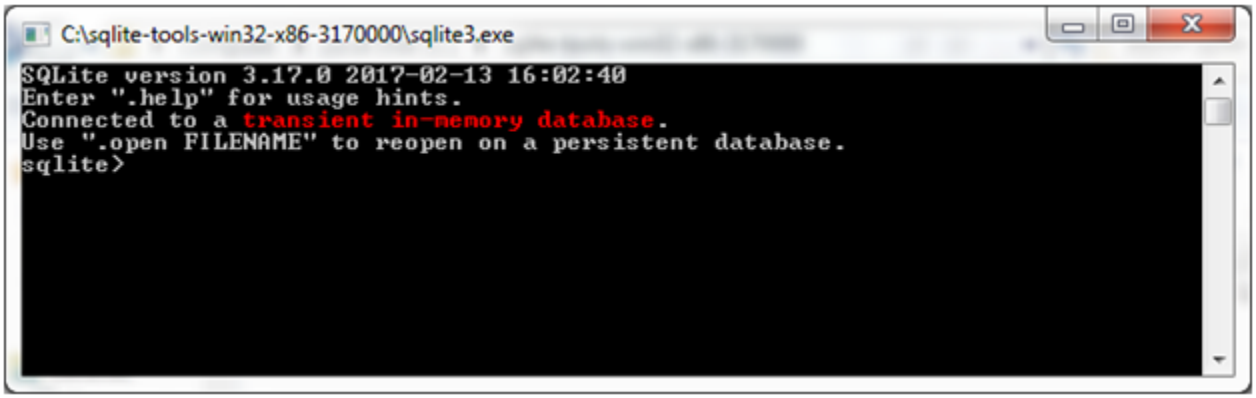
There is another way to execute CRUD operation in SQLite. In this method, there is no need to set a path.

- Just download the SQlite precompiled Binary zip file.
- Expand the zipped file, you will see a page like this:



- Run the selected sqlite3 application:





SQLite Features/ Why to use SQLite

Following is a list of features which makes SQLite popular among other lightweight databases:

- **SQLite is totally free:** SQLite is open-source. So, no license is required to work with it.
- **SQLite is serverless:** 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** 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 Advantages

SQLite is a very popular database which has been successfully used with on disk file format for desktop applications like version control systems, financial analysis tools, media cataloging and editing suites, CAD packages, record keeping programs etc.

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

1) 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.

2) 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.

3) 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.

4) 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.

5) 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.

6) 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.

7) 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 preserve 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

SQLite data types are used to specify type of data of any object. Each column, variable and expression has related data type in SQLite. These data types are used while creating table. SQLite uses a more general dynamic type system. In SQLite, the datatype of a value is associated with the value itself, not with its container.

Types of SQLite data types

SQLite Storage Classes

The stored values in a SQLite database has one of the following storage classes:

Storage Class	Description
NULL	It specifies that the value is a null value.
INTEGER	It specifies the value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
REAL	It specifies the value is a floating point value, stored as an 8-byte IEEE floating point number.
Text	It specifies the value is a text string, stored using the database encoding (utf-8, utf-16be or utf-16le)
BLOB	It specifies the value is a blob of data, stored exactly as it was input.

Note: SQLite storage class is slightly more general than a data type. For example: The INTEGER storage class includes 6 different integer data types of different lengths.

SQLite Afinity Types

SQLite supports type affinity for columns. Any column can still store any type of data but the preferred storage class for a column is called its affinity.

There are following type affinity used to assign in SQLite3 database.

Affinity	Description
TEXT	This column is used to store all data using storage classes NULL, TEXT or BLOB.
NUMERIC	This column may contain values using all five storage classes.
INTEGER	It behaves the same as a column with numeric affinity with an exception in a cast expression.
REAL	It 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 don't persuade data from one storage class into another.

SQLite Affinity and Type Names

Following is a list of various data types names which can be used while creating SQLite tables.

Data Types	Corresponding Affinity
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER
CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB	TEXT

BLOB no datatype specified	NONE
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC

Date and Time Data Type

In SQLite, there is no separate class to store dates and times. But you can store date and times as TEXT, REAL or INTEGER values.

Storage Class	Date Format
TEXT	It specifies a date in a format like "yyyy-mm-dd hh:mm:ss.sss".
REAL	It specifies the number of days since noon in Greenwich on November 24, 4714 B.C.
INTEGER	It specifies the number of seconds since 1970-01-01 00:00:00 utc.

Boolean Data Type

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

SQLite Operators

SQLite operators are reserved words or characters used in SQLite statements when we use WHERE clause to perform operations like comparisons and arithmetic operations.

Operators can be used to specify conditions and as conjunction for multiple conditions in SQLite statements.

There are mainly 4 type of operators in SQLite:

- Arithmetic operators
- Comparison operators
- Logical operators
- Bitwise operators

SQLite Arithmetic Operators

The following table specifies the different arithmetic operators in SQLite. In this table, we have two variables "a" and "b" holding values 50 and 100 respectively.

Operator	Description	Example
+	Addition Operator: It is used to add the values of both side of the operator.	a+b = 150
-	Subtraction Operator: It is used to subtract the right hand operand from left hand operand.	a-b = -50
*	Multiplication Operator: It is used to multiply the values of both sides.	a*b = 5000
/	Division Operator: It is used to divide left hand operand by right hand operand.	a/b = 0.5
%	Modulus Operator: It is used to divide left hand operand by right hand operand and returns remainder.	b/a = 0

SQLite Comparison Operator

The following table specifies the different comparison operators in SQLite. In this table, we have two variables "a" and "b" holding values 50 and 100 respectively.

Operator	Description	Example
==	It is used to check if the values of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
=	It is used to check if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	It is used to check if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	It is used to check if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	It is used to check if the values of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	It is used to check if the values of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	It is used to check if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	It is used to check if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	It is used to check if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	It is used to check if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

SQLite Logical Operator

Following is a list of logical operators in SQLite:

Operator	Description
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
NOT IN	It is the negation of IN operator which is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
GLOB	The GLOB operator is used to compare a value to similar values using wildcard operators. Also, glob is case sensitive, unlike like.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. For example: EXISTS, NOT BETWEEN, NOT IN, etc. These are known as negate operator.
OR	The OR operator is used to combine multiple conditions in an SQL statement's where clause.
IS NULL	The NULL operator is used to compare a value with a null value.
IS	The IS operator work like =

IS NOT	The IS NOT operator work like !=
||	This operator is used to add two different strings and make new one.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

SQLite Bitwise Operators

SQLite Bitwise operators work on bits and perform bit by bit operation.

See the truth table for Binary AND (&) and Binary OR (|):

P	q	p&q	p|q
0	0	0	0
0	1	0	1
1	1	1	1
1	0	0	1

Let's assume two variables "a" and "b", having values 60 and 13 respectively. So Binary values of a and b are:

a= 0011 1100

b= 0000 1101

a&b = 0000 1100

a|b = 0011 1101

~a = 1100 0011

Operator	Description	Example
&	Binary AND operator copies a bit to the result if it exists in both operands.	(a & b) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(a | b) will give 61 which is 0011 1101
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15 which is 0000 1111

SQLite Expressions

SQLite Expressions are the combination of one or more values, operators and SQL functions. These expressions are used to evaluate a value.

SQLite expressions are written in query language and used with SELECT statement.

Syntax:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [CONDITION | EXPRESSION];
```

There are mainly three types of SQLite expressions:

1) SQLite Boolean Expression

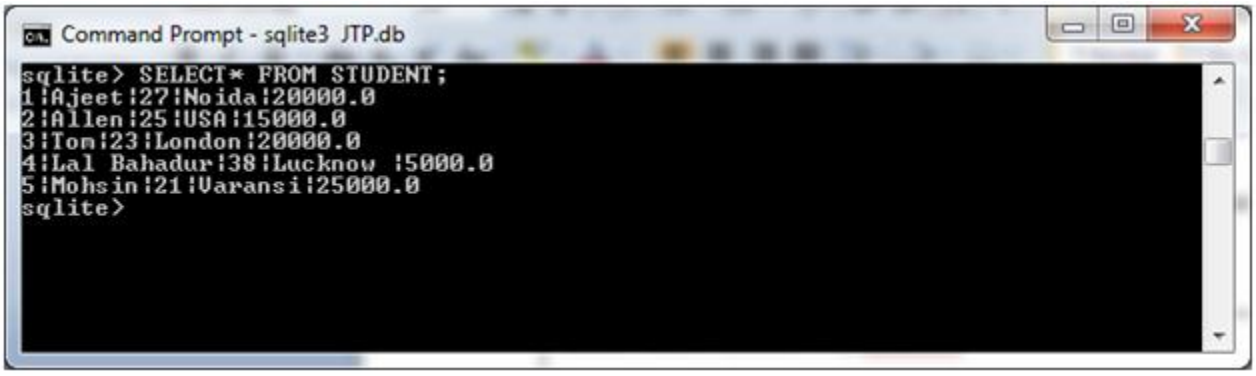
SQLite Boolean expressions are used to fetch the data on the basis of matching single value.

Syntax:

```
SELECT column1, column2, columnN
FROM table_name
WHERE SINGLE VALUE MATCHTING EXPRESSION;
```

Example:

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



See this simple example of SQLite Boolean expression.

```
SELECT * FROM STUDENT WHERE FEES = 20000;
```

Output:

2) SQLite Numeric Expressions

SQLite Numeric expression is used to perform any mathematical operations in the query.

Syntax:

```
SELECT numerical_expression as  OPERATION_NAME
[FROM table_name WHERE CONDITION] ;
```

Example1:

```
SELECT (25 + 15) AS ADDITION;
```

Output:

Numeric expressions contain several built-in functions like avg(), sum(), count(), etc. These functions are known as aggregate data calculation functions.

SELECT COUNT(*) AS "RECORDS" FROM STUDENT;

Output:

3) SQlite Date Expression

SQlite Date expressions are used to fetch the current system date and time values.

Syntax:

```
SELECT CURRENT_TIMESTAMP;
```

SELECT CURRENT_TIMESTAMP;

Output:

SQLite Transaction

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.

A transaction is the propagation of one or more changes to the database. For example, if you are creating, updating, or deleting a record from the table, then you are performing transaction on the table. It is important to control transactions to ensure data integrity and to handle database errors.

Practically, you will club many SQLite queries into a group and you will execute all of them together as part of a transaction.

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.

Transaction Control

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.

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

Transactions can be started using BEGIN TRANSACTION or simply BEGIN command. Such transactions usually persist until the next COMMIT or ROLLBACK command is encountered.

However, a transaction will also ROLLBACK if the database is closed or if an error occurs. Following is the simple syntax to start a transaction.

BEGIN;
or
BEGIN TRANSACTION;

COMMIT Command

COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

Following is the syntax for COMMIT command.

COMMIT;
or
END TRANSACTION;

ROLLBACK Command

ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.

ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

Following is the syntax for ROLLBACK command.

ROLLBACK;

Example

Consider COMPANY table with the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	Paul	32	California	20000.0
2	Allen	25	Texas	15000.0
3	Teddy	23	Norway	20000.0
4	Mark	25	Rich-Mond	65000.0
5	David	27	Texas	85000.0
6	Kim	22	South-Hall	45000.0
7	James	24	Houston	10000.0

Now, let's start a transaction and delete records from the table having age = 25. Then, use ROLLBACK command to undo all the changes.

```
sqlite> BEGIN;
sqlite> DELETE FROM COMPANY WHERE AGE = 25;
sqlite> ROLLBACK;
```

Now, if you check COMPANY table, it still has the following records −

ID	NAME	AGE	ADDRESS	SALARY
1	Paul	32	California	20000.0
2	Allen	25	Texas	15000.0
3	Teddy	23	Norway	20000.0
4	Mark	25	Rich-Mond	65000.0
5	David	27	Texas	85000.0
6	Kim	22	South-Hall	45000.0
7	James	24	Houston	10000.0

Let's start another transaction and delete records from the table having age = 25 and finally we use COMMIT command to commit all the changes.

```
sqlite> BEGIN;
sqlite> DELETE FROM COMPANY WHERE AGE = 25;
sqlite> COMMIT;
```

If you now check COMPANY table is still has the following records −

ID	NAME	AGE	ADDRESS	SALARY
1	Paul	32	California	20000.0
3	Teddy	23	Norway	20000.0
5	David	27	Texas	85000.0
6	Kim	22	South-Hall	45000.0
7	James	24	Houston	10000.0

SQLite Create Table

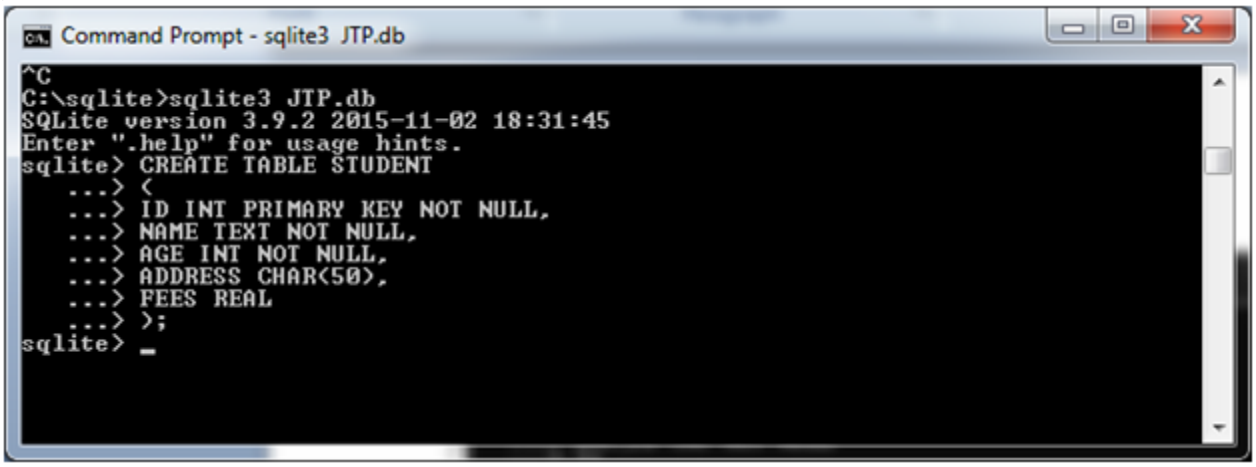
In SQLite, CREATE TABLE statement is used to create a new table. While creating the table, we name that table and define its column and data types of each column.

Syntax:

```
CREATE TABLE database_name.table_name(
   column1 datatype  PRIMARY KEY(one or more columns),
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
);
```

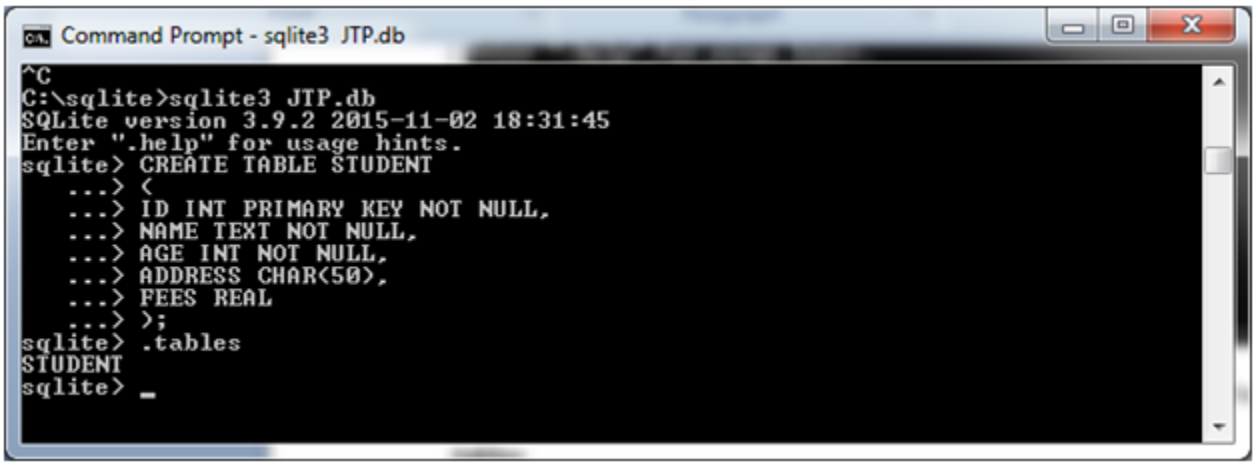
Let's take an example to create table in SQLite database:

```
CREATE TABLE STUDENT(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   FEES           REAL
);
```



Use the SQLite ".tables" command to see if your table has been created successfully.

.tables



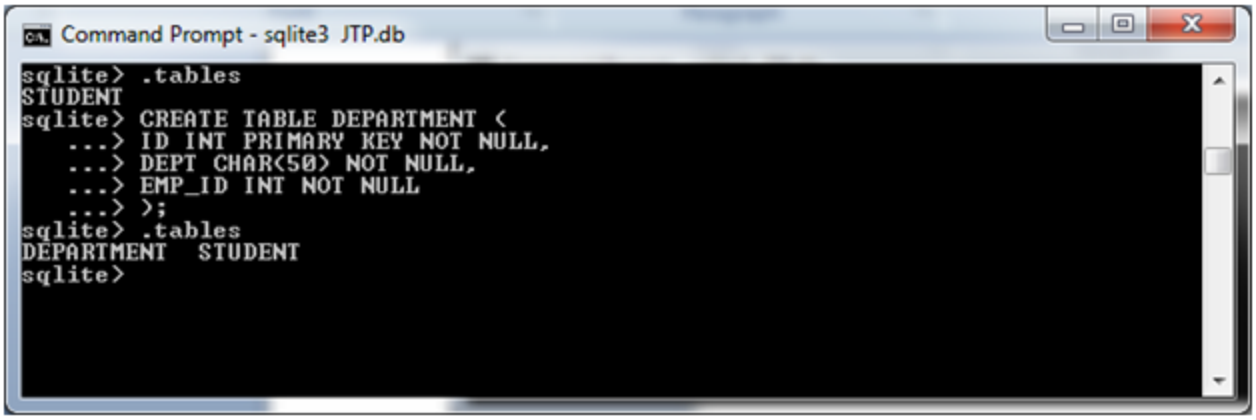
Let's create another table DEPERTMENT for future operations.

```
CREATE TABLE DEPARTMENT(
```

```
 ID INT PRIMARY KEY     NOT NULL,
 DEPT          CHAR(50) NOT NULL,
 EMP_ID        INT      NOT NULL
);
```

Now, we have two tables "DEPARTMENT" and "STUDENT".

Now check the created tables:



SQLite Drop Table

In SQLite, DROP TABLE statement is used to remove a table definition and all associated data, indexes, triggers, constraints and permission specifications associated with that table.

Syntax:

```
DROP TABLE database_name.table_name;
```

> *Note: You must be very careful while using the DROP TABLE command because once the table is deleted then all the information available in the phone is destroyed and you cannot recover.*

Let's take an example to demonstrate how to delete a table in SQLite.

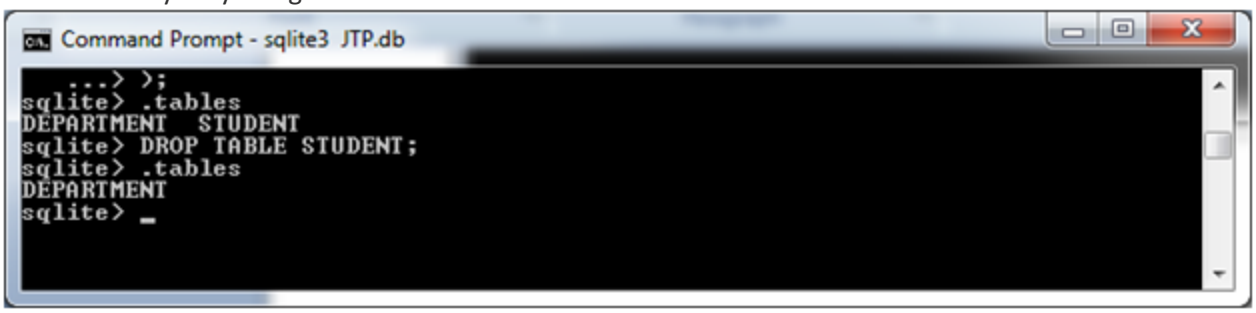
We have already two tables "DEPARTMENT" and "STUDENT". We can verify it by using .tables command.

So you can see that we have two tables.

Let's delete the "STUDENT" table.

DROP TABLE STUDENT;

You can verify it by using .tables command.



You can see only one table is here in the database. It means the other table is dropped.

SQLite Insert Query

In SQLite, INSERT INTO statement is used to add new rows of data into a table. After creating the table, this command is used to insert data into the table.

There are two types of basic syntaxes for INSERT INTO statement:

Syntax1:

INSERT INTO TABLE_NAME [(column1, column2, column3,...columnN)]

VALUES (value1, value2, value3,...valueN);

Here, column1, column2, column3,...columnN specifies the name of the columns in the table into which you have to insert data.

You don't need to specify the columns name in the SQlite query if you are adding values to all the columns in the table. But you should make sure that the order of the values is in the same order of the columns in the table.

Syntax2:

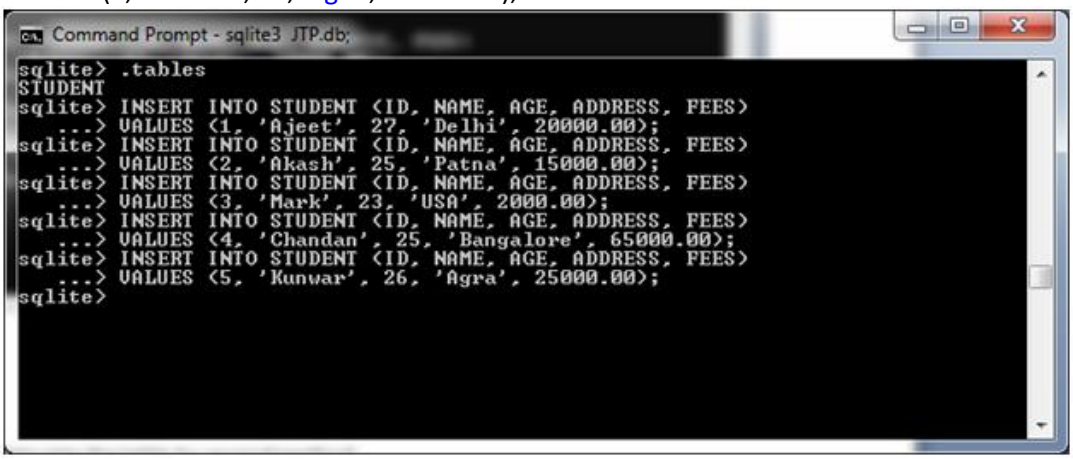
```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

Let's take an example to demonstrate the INSERT query in SQLite database.

We have already created a table named "STUDENT". Now enter some records in that table.

Inserting values by first method:

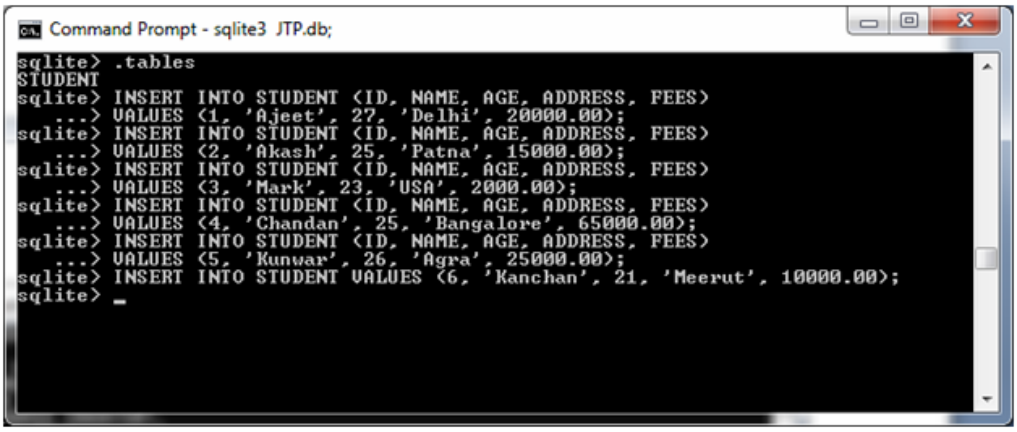
```
INSERT INTO STUDENT (ID,NAME,AGE,ADDRESS,FEES)
VALUES (1, 'Ajeet', 27, 'Delhi', 20000.00);
INSERT INTO STUDENT (ID,NAME,AGE,ADDRESS,FEES)
VALUES (2, 'Akash', 25, 'Patna', 15000.00 );
INSERT INTO STUDENT (ID,NAME,AGE,ADDRESS,FEES)
VALUES (3, 'Mark', 23, 'USA', 2000.00 );
INSERT INTO STUDENT (ID,NAME,AGE,ADDRESS,FEES)
VALUES (4, 'Chandan', 25, 'Banglore', 65000.00 );
INSERT INTO STUDENT (ID,NAME,AGE,ADDRESS,FEES)
VALUES (5, 'Kunwar', 26, 'Agra', 25000.00 );
```



Second Method:

You can also insert the data into the table by second method.

```
INSERT INTO STUDENT VALUES (6, 'Kanchan', 21, 'Meerut', 10000.00 );
```



Output:

```
SELECT * FROM STUDENT;
```

SQLite SELECT Query

In SQLite database, SELECT statement is used to fetch data from a table. When we create a table and insert some data into that, we have to fetch the data whenever we require. That's why select query is used.

Syntax:

```
SELECT column1, column2, columnN FROM table_name;
```

Here, column1, column2...are the fields of a table, which values you have to fetch. If you want to fetch all the fields available in the field then you can use following syntax:

```
SELECT * FROM table_name;
```

Let's see an example:

```
SELECT * FROM STUDENT;
```

SQLite UPDATE Query

In SQLite, UPDATE query is used to modify the existing records in a table. It is used with WHERE clause to select the specific row otherwise all the rows would be updated.

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

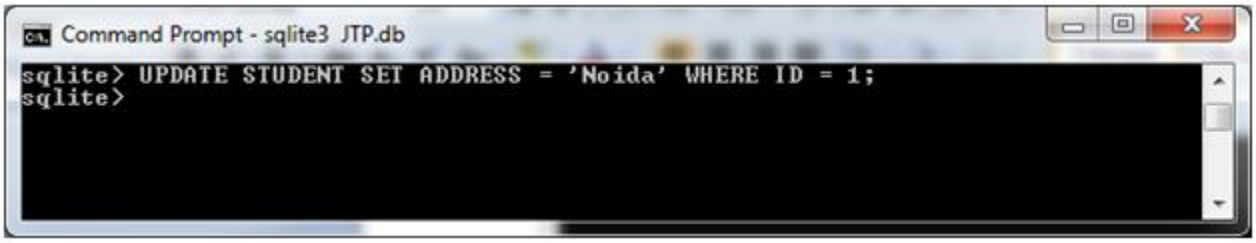
Example:

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

Example1:

Update the ADDRESS of the student where ID is

```
UPDATE STUDENT SET ADDRESS = 'Noida' WHERE ID = 1;
```



Now the address is updated for the id 1. You can check it by using SELECT statement:

```
SELECT * FROM STUDENT;
```

Output:

Example2:

If you don't use WHERE clause, it will modify all address in the STUDENT table:

```
UPDATE STUDENT SET ADDRESS = 'Noida';
```

Output:

SQLite DELETE Query

In SQLite, DELETE query is used to delete the existing records from a table. You can use it with WHERE clause or without WHERE clause. WHERE clause is used to delete the specific records (selected rows), otherwise all the records would be deleted.

Syntax:

```
DELETE FROM table_name
WHERE [conditions.....................];;
```

Note: We can use N number of "AND" or "OR" operators with "WHERE" clause.

Example:

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

Example1:

Delete the records of a student from "STUDENT" table where ID is 4.

```
DELETE FROM STUDENT WHERE ID = 4;
```

The student's record of id 4 is deleted; you can check it by using SELECT statement:

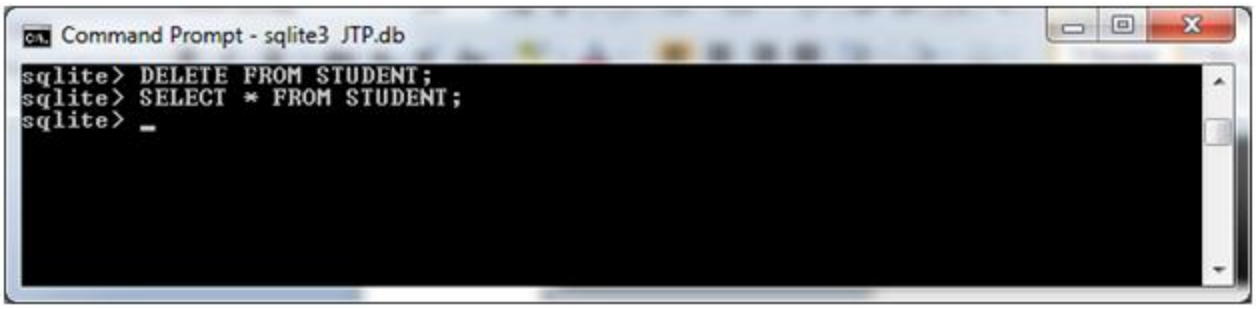
```
SELECT * FROM STUDENT;
```

Output:

Example2:

If you want to delete all records from the table, don't use WHERE clause.

DELETE FROM STUDENT;



You can see that there is no data in the table "STUDENT".

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

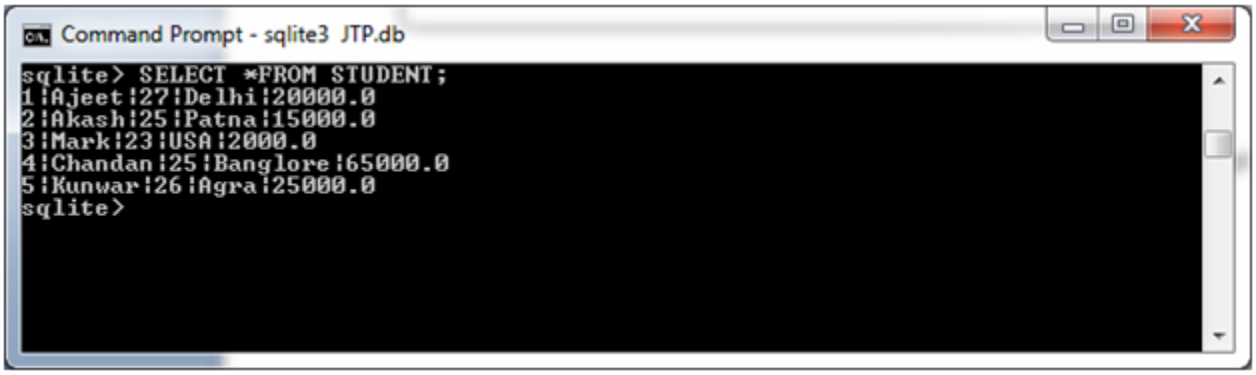
FROM table_name

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:

Example2:

Select students form STUDENT table where name starts with 'A' doesn't matter what come after 'A'.

```
SELECT * FROM STUDENT WHERE NAME LIKE 'A%';
```

Output:

SQLite AND Operator

The SQLite AND Operator is generally used with SELECT, UPDATE and DELETE statement to combine multiple conditions. It is a conjunctive operator which makes multiple comparisons with different operators in the same SQLite statement.

It is always used with WHERE Clause.

Syntax:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

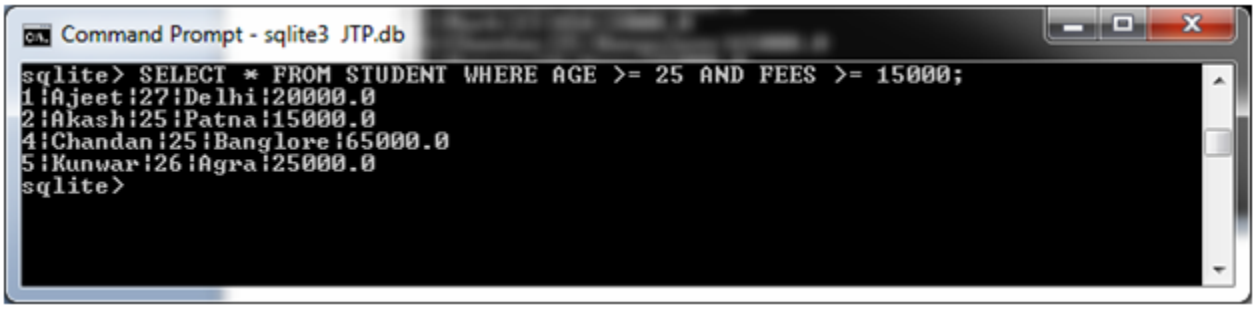
Example:

We have a table named 'STUDENT' having following data:

Select all students from the table 'STUDENT' where AGE is greater than or equal to 25 AND fees is greater than or equal to 20000.00

```
SELECT * FROM STUDENT WHERE AGE >= 25 AND FEES >= 15000;
```

Output:



SQLite OR Operator

The SQLite OR Operator is generally used with SELECT, UPDATE and DELETE statement to combine multiple conditions. OR operator is always used with WHERE clause and the complete condition is assumed true if anyone of the both condition is true.

Syntax:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

You can combine multiple number of conditions using OR operator.

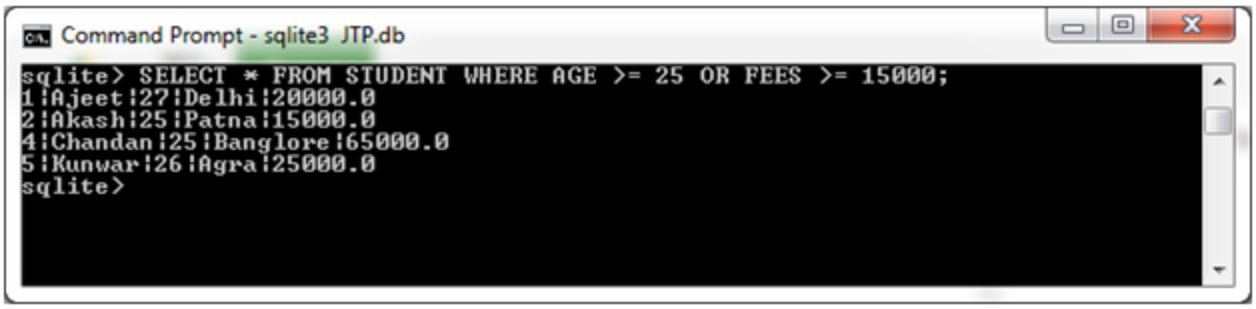
Example:

We have a table named 'STUDENT' having following data:

Select all students from the table 'STUDENT' WHERE age is greater than or equal to 25 OR fees is greater than or equal to 15000.00

```
SELECT * FROM STUDENT WHERE AGE >= 25 OR FEES >= 15000;
```

Output:



SQLite LIKE Clause (Operator)

The SQLite LIKE operator is used to match text values against a pattern using wildcards. In the case search expression is 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.

Syntax:

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'
```

or

```
SELECT FROM table_name
WHERE column LIKE '%XXXX%'
```

or

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

or

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

or

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

Here, XXXX could be any numeric or string value.

Example:

We have a table named 'STUDENT' having following data:

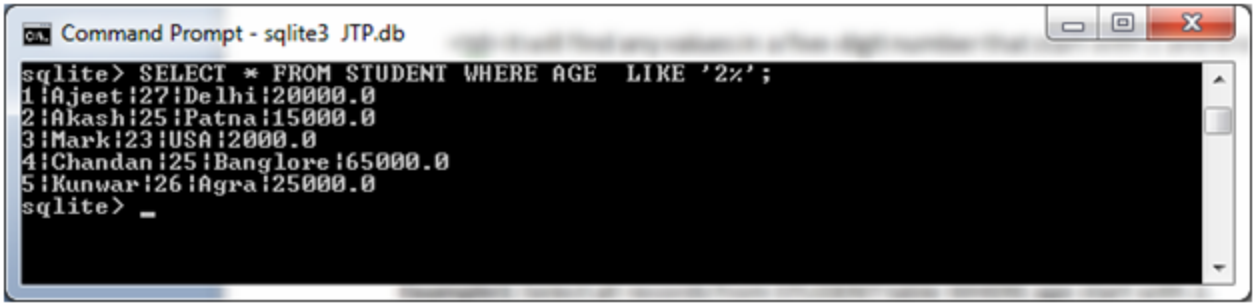
In these examples the WHERE statement having different LIKE clause with '%' and '_' operators and operation is done on 'FEES':

Statement	Description
Where FEES like '200%'	It will find any values that start with 200.
Where FEES like '%200%'	It will find any values that have 200 in any position.
Where FEES like '_00%'	It will find any values that have 00 in the second and third positions.
Where FEES like '2_%_%'	It will find any values that start with 2 and are at least 3 characters in length.
Where FEES like '%2'	It will find any values that end with 2
Where FEES like '_2%3'	It will find any values that have a 2 in the second position and end with a 3
Where FEES like '2___3'	It will find any values in a five-digit number that start with 2 and end with 3

Example1: Select all records from STUDENT table WHERE age start with 2.

```
SELECT * FROM STUDENT WHERE AGE  LIKE '2%';
```

Output:



Example2:

Select all records from the STUDENT table WHERE ADDRESS will have "a" (a) inside the text:

```
SELECT * FROM STUDENT WHERE ADDRESS LIKE '%a%';
```

Output:

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]
```

Example:

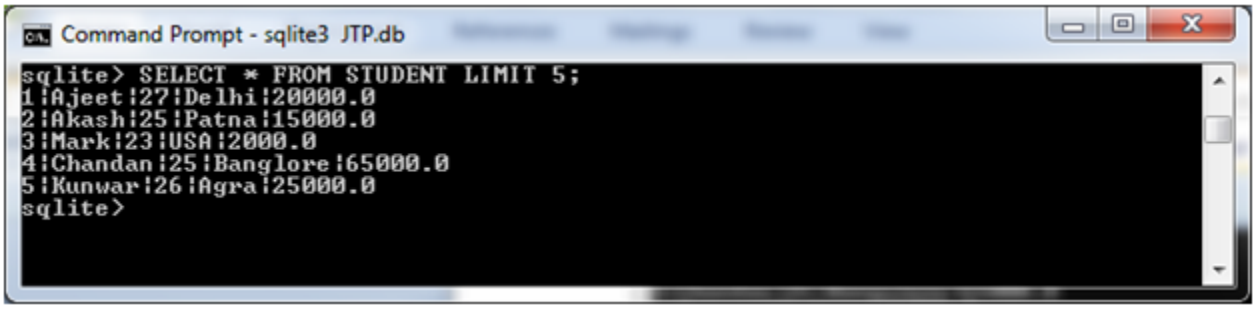
Let's take an example to demonstrate SQLite LIMIT clause. We have a table named 'STUDENT' having following data:

Example1:

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

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

Output:



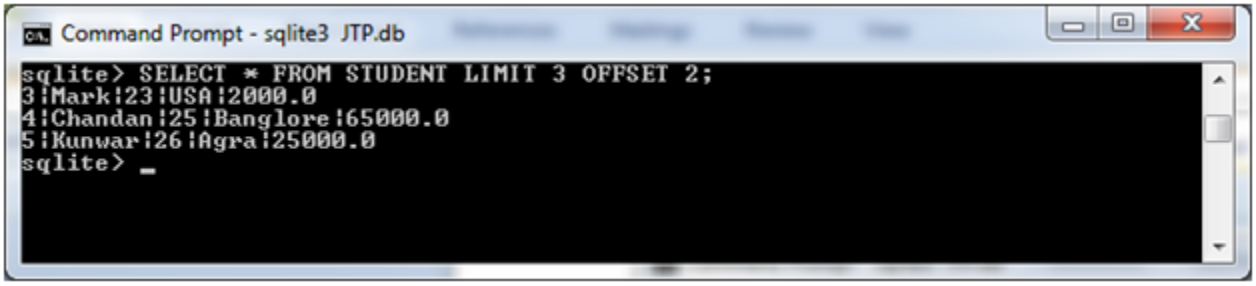
Example2:

OFFSET is used to not retrieve the offset records from the table. It is used in some cases where we have to retrieve the records starting from a certain point:

Select 3 records form table "STUDENT" starting from 3rd position.

```
SELECT * FROM STUDENT LIMIT 3 OFFSET 2;
```

Output:



SQLite ORDER BY Clause

The SQLite ORDER BY clause is used to sort the fetched data in ascending or descending order, based on one or more column.

Syntax:

SELECT column-list

FROM table_name

[**WHERE** condition]

[**ORDER BY** column1, column2, .. columnN] [**ASC** | **DESC**];

You can use one or more columns in ORDER BY clause. Your used column must be presented in column-list.

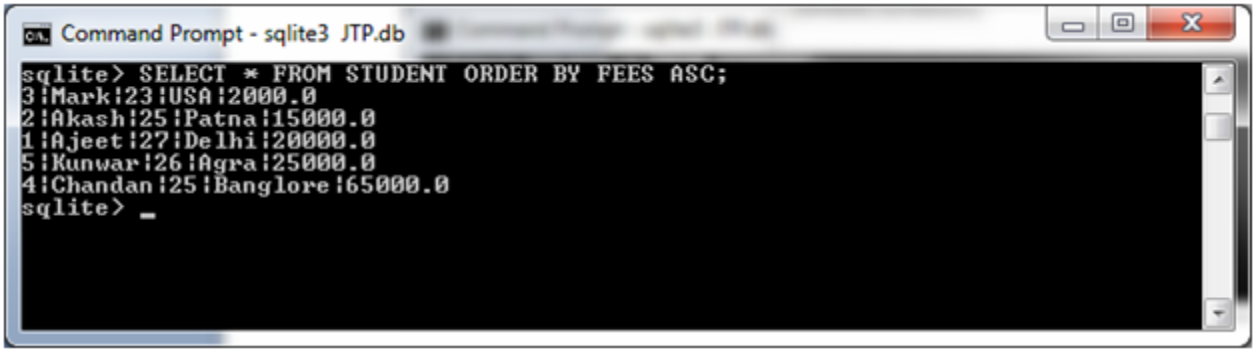
Let's take an example to demonstrate ORDER BY clause. We have a table named "STUDENT" having the following data:

Example1:

Select all records from "STUDENT" where FEES is in ascending order:

SELECT * **FROM** STUDENT **ORDER BY** FEES **ASC**;

Output:

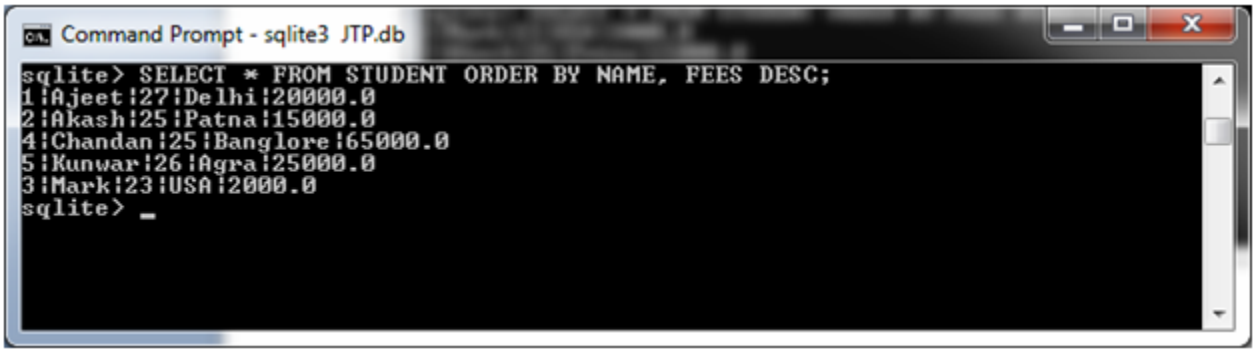


Example2:

Fetch all data from the table "STUDENT" and sort the result in descending order by NAME and FEES:

SELECT * **FROM** STUDENT **ORDER BY NAME**, FEES **DESC**;

Output:



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

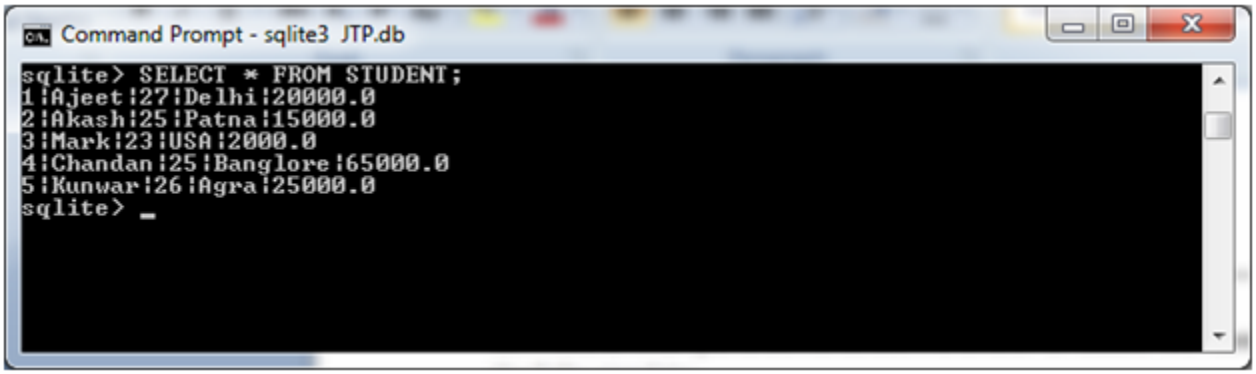
FROM table_name

WHERE [conditions]

GROUP BY column1, column2....columnN

ORDER BY column1, column2....columnN

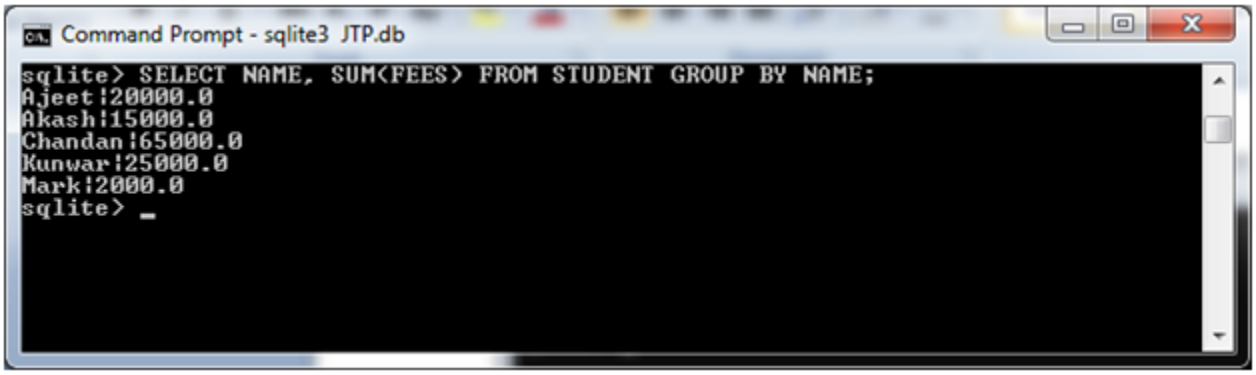
Let's take an example to demonstrate the GROUP BY clause. We have a table named "STUDENT", having the following data:



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

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

Output:

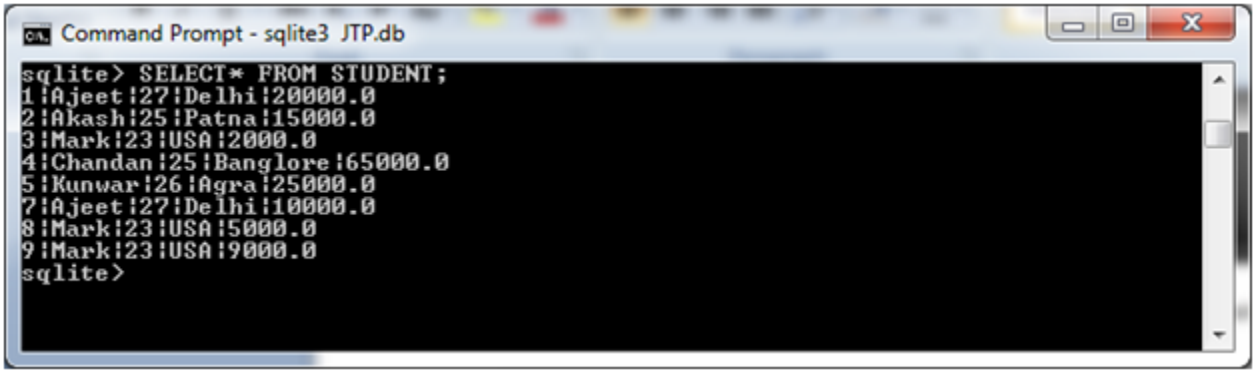


Now, create some more records in "STUDENT" table using the following INSERT statement:

INSERT INTO STUDENT **VALUES** (7, 'Ajeet', 27, 'Delhi', 10000.00);

INSERT INTO STUDENT **VALUES** (8, 'Mark', 23, 'USA', 5000.00);

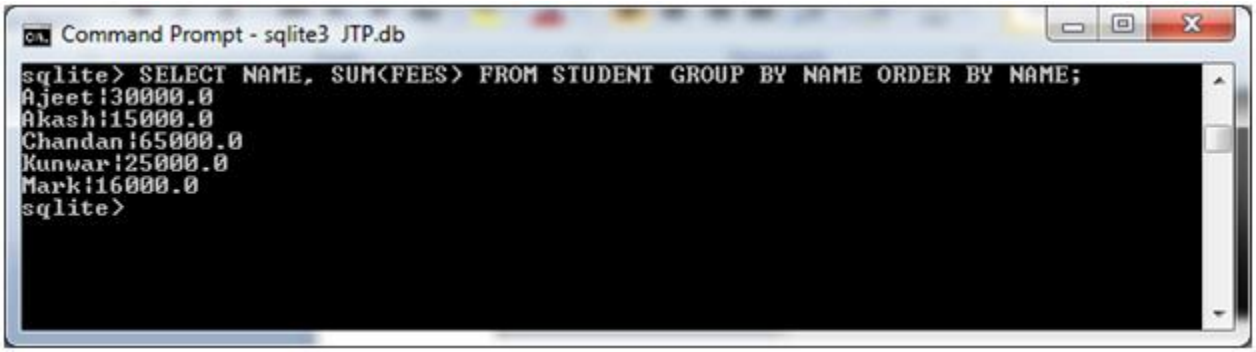
INSERT INTO STUDENT **VALUES** (9, 'Mark', 23, 'USA', 9000.00);



The new updated table has the inserted entries. Now, use the same GROUP BY statement to group-by all the records using NAME column:

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

Output:



You can use ORDER BY clause along with GROUP BY to arrange the data in ascending or descending order.

SELECT NAME, SUM(FEES)

FROM STUDENT **GROUP BY NAME ORDER BY NAME DESC**;

Output:

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:

SELECT

FROM

WHERE

GROUP BY

HAVING

ORDER BY

Syntax:

SELECT column1, column2

FROM table1, table2

WHERE [conditions]

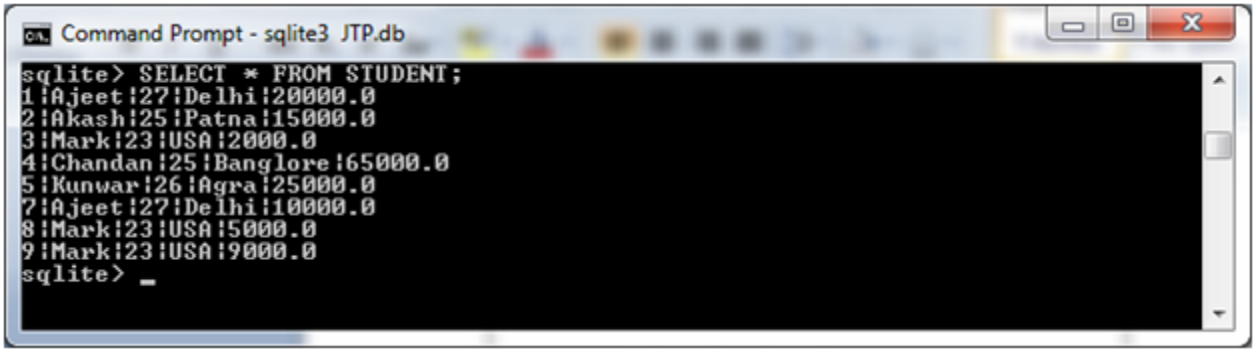
GROUP BY column1, column2

HAVING [conditions]

ORDER BY column1, column2

Example:

Let's take an example to demonstrate HAVING Clause. We have a table named "STUDENT", having the following data:



Example1:

Display all records where name count is less than 2:

```
SELECT * FROM STUDENT GROUP BY NAME HAVING COUNT(NAME) < 2;
```

Output:

Example2:

Display all records where name count is greater than 2:

```
SELECT * FROM STUDENT GROUP BY NAME HAVING COUNT(NAME) > 2;
```

Output:

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:

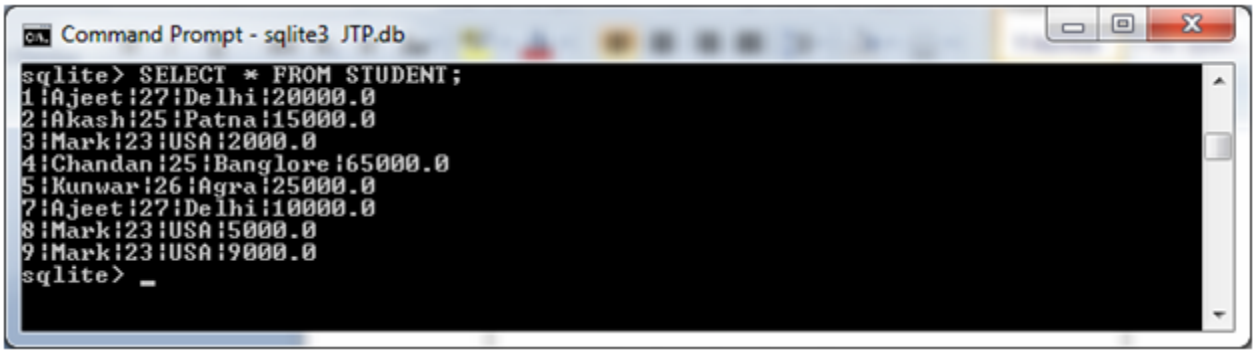
SELECT DISTINCT column1, column2,.....columnN

FROM table_name

WHERE [condition]

Example:

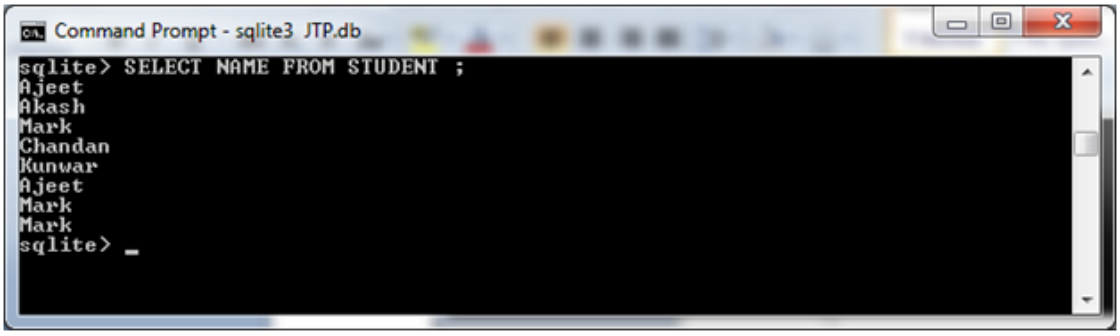
We have a table named "STUDENT", having the following data:



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

SELECT NAME FROM STUDENT ;

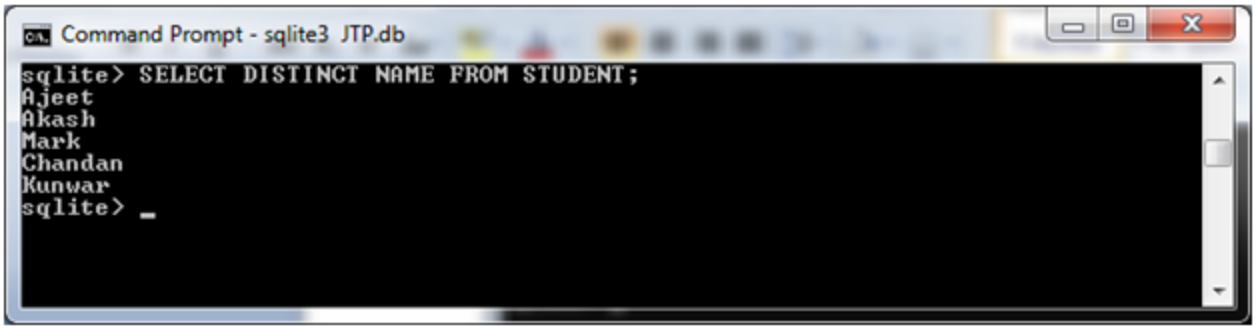
Output:



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

```
SELECT DISTINCT NAME FROM STUDENT;
```

Output:



SQLite Union Operator

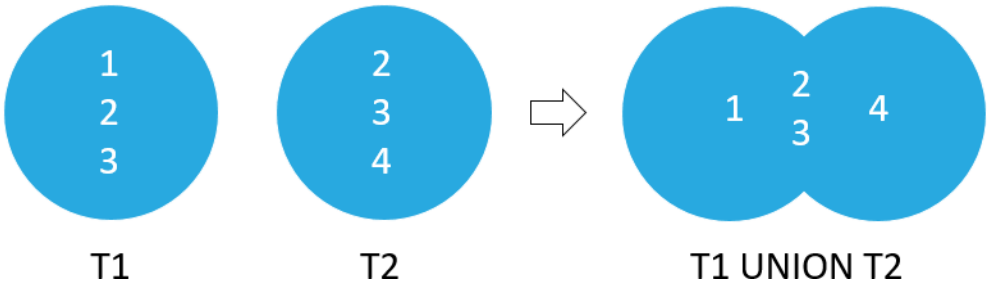
SQLite UNION Operator is used to combine the result set of two or more tables using SELECT statement. The UNION operator shows only the unique rows and removes duplicate rows.

While using UNION operator, each SELECT statement must have the same number of fields in the result set.

Syntax:

```
SELECT expression1, expression2, ... expression_n
FROM tables
[WHERE conditions]
UNION
SELECT expression1, expression2, ... expression_n
FROM tables
[WHERE conditions];
```

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



Example:

We have two tables "STUDENT" and "DEPARTMENT".

The "STUDENT" table is having the following data:

The "DEPARTMENT" table is having the following data:

Example1: Return Single Field

This simple example returns only one field from multiple SELECT statements where the both fields have same data type.

Let's take the above two tables "STUDENT" and "DEPARTMENT" and select id from both table to make **UNION.**

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

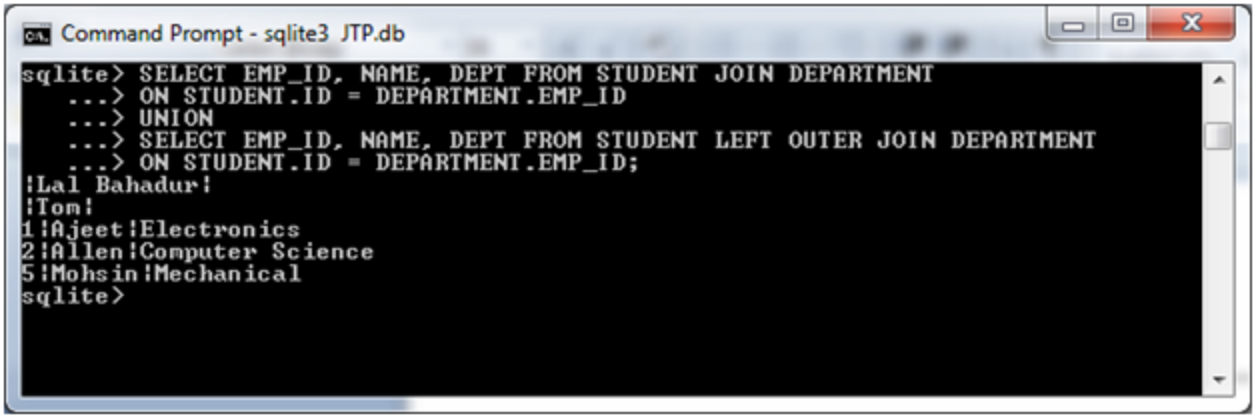
Output:

Example2: Union with Inner and Outer Join

Let's take the above two tables "STUDENT" and "DEPARTMENT" and make an inner join and outer join according to the below conditions along with UNION Clause:

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

Output:



SQLite INTERSECT Operator

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

The following illustrates the syntax of the INTERSECT operator:

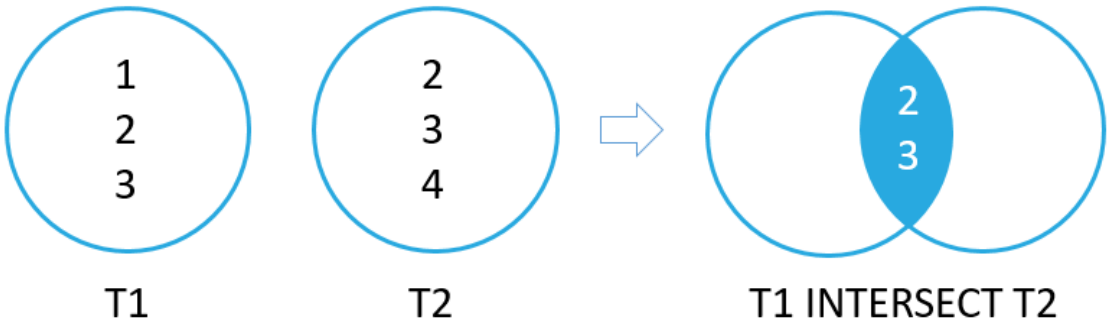
```
SELECT select_list1
FROM table1
INTERSECT
SELECT select_list2
FROM table2
```

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.

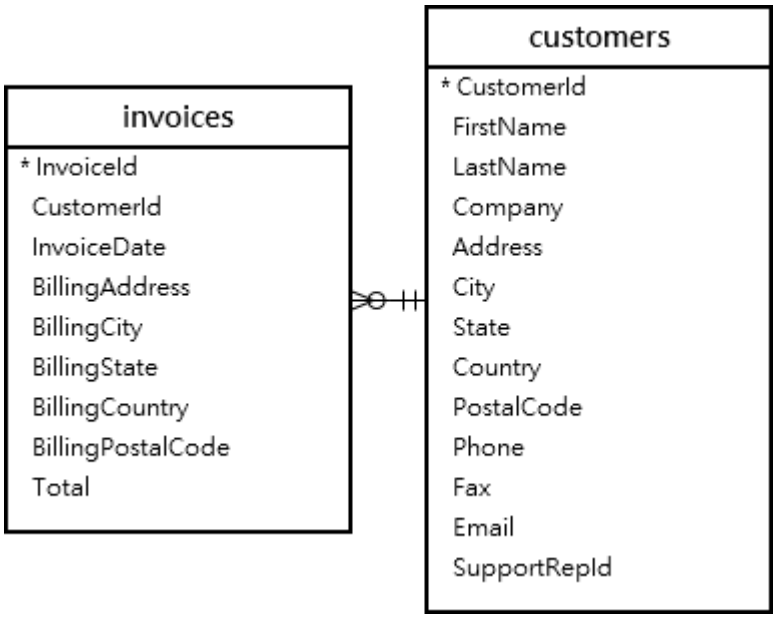
For the demonstration, we will create two tables t1 and t2 and insert some data into both:

The following picture illustrates the INTERSECT operation:



SQLite INTERSECT example

For the demonstration, we will use the customers and invoices tables from the



The following statement finds customers who have invoices:

```
SELECT CustomerId FROM customer
INTERSECT
SELECT CustomerId FROM invoices;
```

The following picture shows the partial output:

CustomerId
1
2
3
4
5
6
7
8
9
10

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.

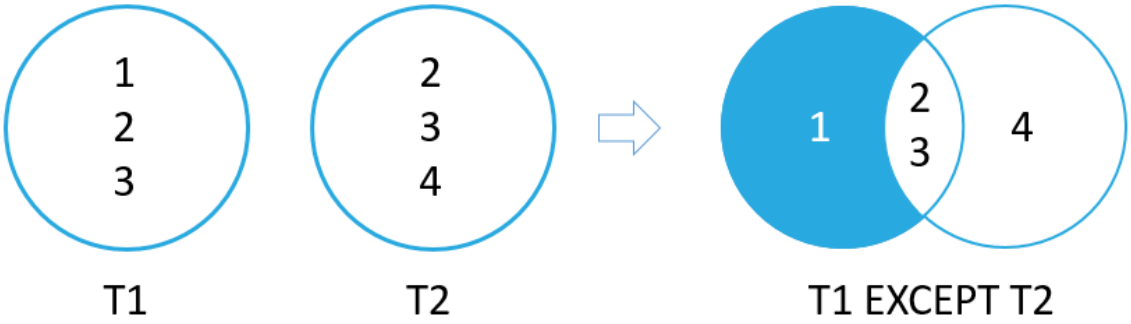
The following shows the syntax of the EXCEPT operator:

```
SELECT select_list1
FROM table1
EXCEPT
SELECT select_list2
FROM table2
```

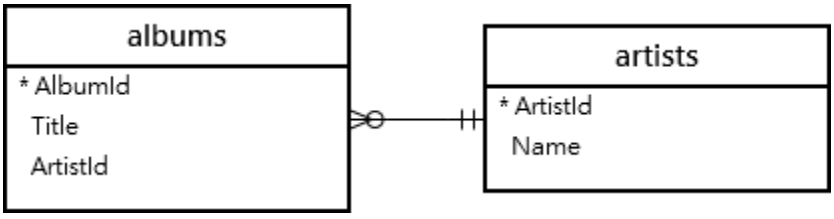
This query must conform to the following rules:

- 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:



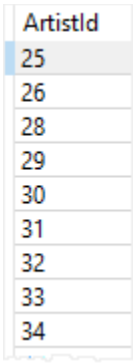
We will use the artists and albums tables from the sample database for the demonstration.



The following statement finds artist ids of artists who do not have any album in the albums table:

```
SELECT ArtistId
FROM artists
EXCEPT
SELECT ArtistId
FROM albums;
```

The output is as follows:



ArtistId
25
26
28
29
30
31
32
33
34

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

Example:

We have two tables "STUDENT" and "DEPARTMENT".

The "STUDENT" table is having the following data:

The "DEPARTMENT" table is having the following data:

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 ...

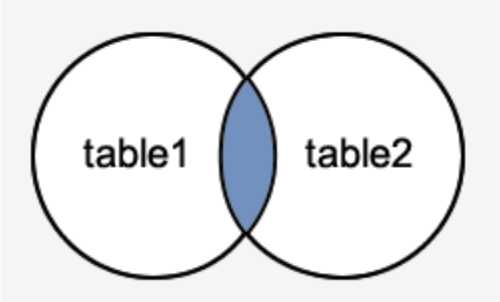
or

SELECT ... **FROM** table1 JOIN table2 USING (column1 ,...) ...

or

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

Image representation:



We have two tables "STUDENT" and "DEPARTMENT".

The "STUDENT" table is having the following data:

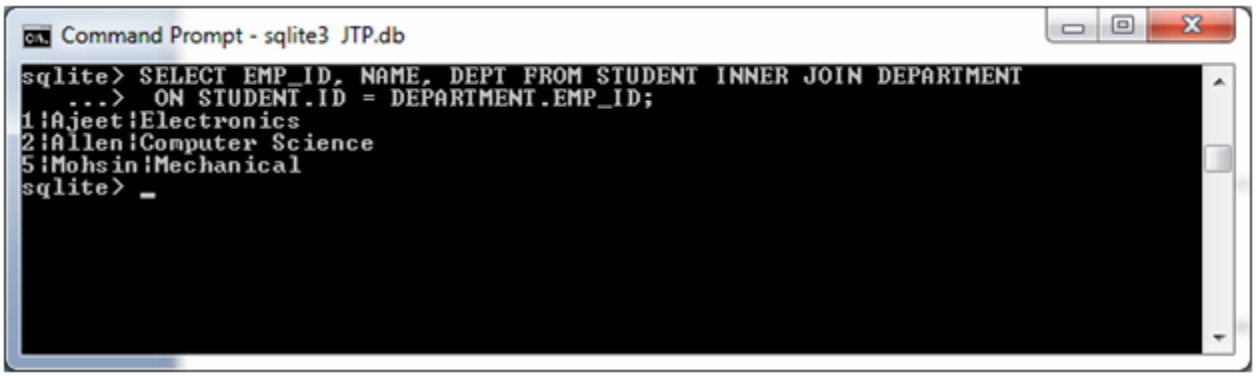
The "DEPARTMENT" table is having the following data:

Let's take the above two tables "STUDENT" and "DEPARTMENT" and make an inner join according to the below conditions:

Example:

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

Output:



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.

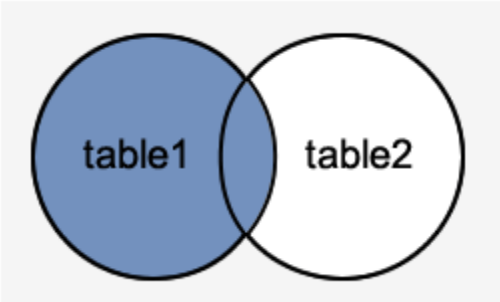
Syntax:

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

Or

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

Image representation:



We have two tables "STUDENT" and "DEPARTMENT".

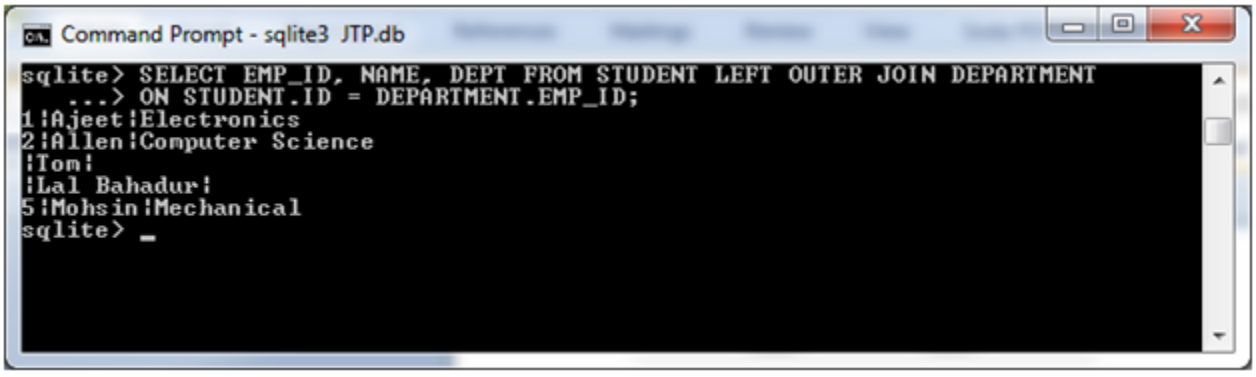
The "STUDENT" table is having the following data:

The "DEPARTMENT" table is having the following data:

Let's take the above two tables "STUDENT" and "DEPARTMENT" and make an inner join according to the below conditions:

Example:

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



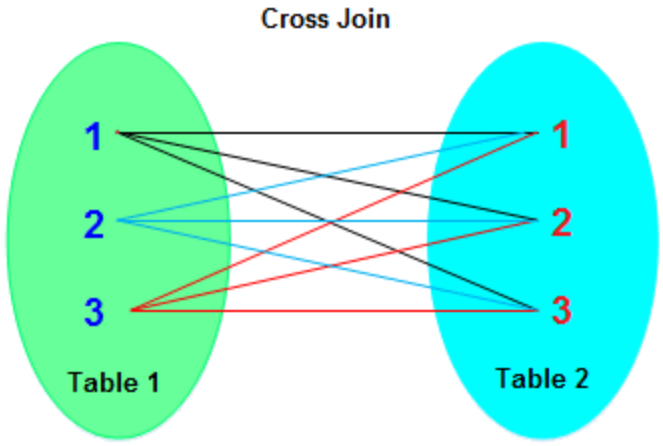
SQLite Cross Join

The SQLite Cross join is used to match every rows of the first table with every rows of the second table. If the first table contains x columns and second table contains y columns then the resultant Cross join table will contain the x*y columns.

Syntax:

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

Image Representation:



We have two tables "STUDENT" and "DEPARTMENT".

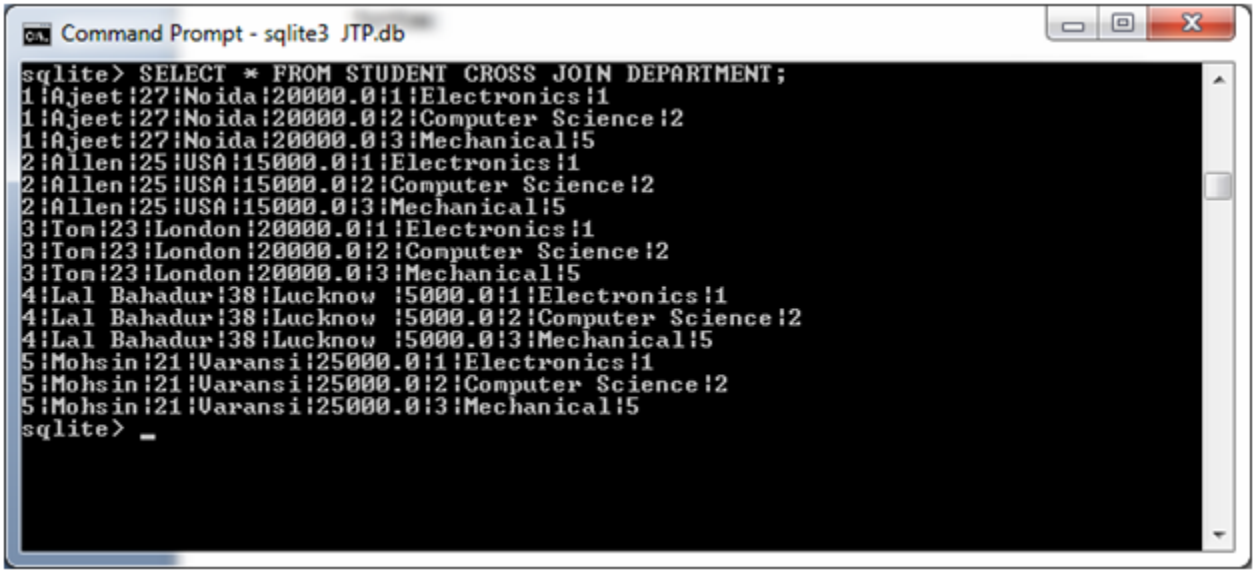
The "STUDENT" table is having the following data:

The "DEPARTMENT" table is having the following data:

Example:

Select all records from tables STUDENT and DEPARTBMENT after cross join:

```
SELECT * FROM COMPANY CROSS JOIN DEPARTMENT;
```



SQLite Triggers

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.

How to create trigger

The CREATE TRIGGER statement is used to create a new trigger in SQLite. This statement is also used to add triggers to the database schema.

Syntax:

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] event_name
ON table_name
BEGIN
 -- Trigger logic goes here....
END;
```

Here, trigger_name is the name of trigger which you want to create.

event_name could be INSERT, DELETE, and UPDATE database operation.

table_name is the table on which you do the operation.

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

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

The following table illustrates the rules:

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

SQLite Triggers:Example AFTER INSERT

In the following example, we have two tables : emp_details and emp_log. To insert some information into emp_logs table (which have three fields emp_id and salary and edttime) every time, when an INSERT happen into emp_details table we have used the following trigger :

```
CREATE TRIGGER aft_insert AFTER INSERT ON emp_details

BEGIN

INSERT INTO emp_log(emp_id,salary,edittime)

      VALUES(NEW.employee_id,NEW.salary,current_date);

END;
```

SQLite Triggers:Example BEFORE INSERT

In the following example, before inserting a new record in emp_details table, a trigger check the column value of FIRST_NAME, LAST_NAME, JOB_ID and
- If there are any space(s) before or after the FIRST_NAME, LAST_NAME, LTRIM() function will remove those.
- The value of the JOB_ID will be converted to upper cases by UPPER() function.

Here is the trigger befo_insert:

```
CREATE TRIGGER befo_insert BEFORE INSERT ON emp_details

BEGIN

SELECT CASE

WHEN ((SELECT emp_details . employee_id FROM emp_details WHERE
emp_details.employee_id = NEW.employee_id ) ISNULL)

THEN RAISE(ABORT, 'This is an User Define Error Message - This employee_id does not exist.')

END;

END;
```

SQLite Triggers:Example AFTER UPDATE

We have two tables student_mast and stu_log. student_mast have three columns STUDENT_ID, NAME, ST_CLASS. stu_log table has two columns user_id and description.

```
CREATE TRIGGER aft_update AFTER UPDATE ON student_mast

BEGIN

INSERT into stu_log (description) values('Update Student Record '||

    OLD.NAME || '   Previous Class : '||OLD.ST_CLASS ||'   Present Class '||

    NEW.st_class);

END;
```

SQLite Triggers:Example BEFORE UPDATE

We have two tables student_mast and student_marks. Here are the sample tables below. The student_id column of student_mast table is the primary key and in student_marks table, it is a foreign key, the reference to student_id column of student_mast table.

```
CREATE TRIGGER befo_update BEFORE UPDATE ON student_mast

BEGIN

SELECT CASE

WHEN ((SELECT student_id FROM student_marks WHERE student_id = NEW.student_id )
ISNULL)

THEN RAISE(ABORT, 'This is a User Define Error Message - This ID can not be updated.')

END;

END;
```

SQLite Triggers:Example AFTER DELETE

In our 'AFTER UPDATE' example, we had two tables student_mast and stu_log. student_mast have three columns STUDENT_ID, NAME, ST_CLASS and stu_log table has two columns user_id and description. We want to store some information in stu_log table after a delete operation happened on student_mast table. Here is the trigger :

Here is the trigger

```
CREATE TRIGGER aft_delete AFTER DELETE ON student_mast

BEGIN

INSERT into stu_log (description) VALUES ('Update Student Record '||

      OLD.NAME||' Class : '||OLD.ST_CLASS||' -> Deleted on  '||

      date('NOW'));

END;
```

SQLite Triggers:Example BEFOR DELETE

We have two tables student_mast and student_marks. Here are the sample tables below. The student_id column of student_mast table is the primary key and in student_marks table, it is a foreign key, a reference to student_id column of student_mast table.

```
CREATE TRIGGER befo_delete BEFORE DELETE ON student_marks

BEGIN

SELECT CASE

WHEN (SELECT COUNT(student_id) FROM student_mast WHERE student_id=OLD.student_id) >
0

THEN RAISE(ABORT,

'Foreign Key Violation: student_masts rows reference row to be deleted.')

END;

END;
```

How to DROP trigger

To delete or destroy a trigger, use a DROP TRIGGER statement. To execute this command, the current user must be the owner of the table for which the trigger is defined.

Syntax:

```
DROP TRIGGER trigger_name
```

Example:

If you delete or drop the just created trigger delete_stu the following statement can be used:

```
DROP TRIGGER delete_stu on student_mast;
```

2.1 SQLite dump:[How To Use The SQLite Dump Command]

The SQLite dump command to backup and restore a database.

.dump command that give you the ability to dump the entire database or table into text file.

2.1.1 Dump the entire database into a file using the SQLite dump command

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

```
C:\sqlite>sqlite3 c:/sqlite/student.db
SQLite version 3.13.0 2016-05-18 10:57:30
Enter".help"for usage hints.
Sqlite>
```

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 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
```

2.1.2 Dump a specific table using the SQLite dump command

To dump a specific table, you specify the table name after 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
```

The following picture shows the contents 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);
```

2.1.3 Dump tables structure only using schema command

To dump the table structures in a database, you use the .schema command.

The following commands set the output file to student_structure.sql file and save the table structures into the student_structure.sql file:

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

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

```
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
);
CREATE TABLE "artists"
(
    [ArtistId] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    [Name] NVARCHAR(120)
);
CREATE TABLE "customers"
(
```

2.1.4 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 follows:

```
sqlite> .mode insert
```
Code language: CSS (css)

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 data.sql file.

```
sqlite> .output data.sql
```

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

```
sqlite> select * from artists;
```

Check the content of the data.sql file, if everything is fine, you will see the following output:

```
INSERT INTO table VALUES(1,'AC/DC');
INSERT INTO table VALUES(2,'Accept');
INSERT INTO table VALUES(3,'Aerosmith');
INSERT INTO table VALUES(4,'Alanis Morissette');
INSERT INTO table VALUES(5,'Alice In Chains');
INSERT INTO table VALUES(6,'Antônio Carlos Jobim');
INSERT INTO table VALUES(7,'Apocalyptica');
INSERT INTO table VALUES(8,'Audioslave');
INSERT INTO table VALUES(9,'BackBeat');
INSERT INTO table VALUES(10,'Billy Cobham');
```

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

2.2 CSV file handling:

2.2.1 Import a CSV file into a table

In the first scenario, you want to import data from CSV file into a table that does not exist in the SQLite database.

1. 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.
2. Second, the sqlite3 tool import data from the second row of the CSV file into the table.

We will import a CSV file named student.csv with following format.

	A	B	C	D	E
1	Roll No	Name	Class	Marks	Grade
2	1	Rakesh	4	67	C
3	2	Rishav	5	56	D
4	3	Harshit	6	90	A
5	4	Diksha	3	56	D
6	5	Rajesh	8	98	A
7	6	Tarun	7	87	A
8	7	Binita	6	76	B
9	8	Seema	5	89	A
10	9	Mohit	4	65	C
11	10	Sumit	5	76	B
12	11	Ramesh	7	69	C

To import the c:\sqlite\student.csv file into the stud 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 student.csv file into the stud table.

```
sqlite>.import c:/sqlite/student.csv stud
```

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

```
sqlite> .schema stud
CREATE TABLE IF NOT EXISTS "stud"(
"Roll No" TEXT, "Name" TEXT, "Class" TEXT, "Marks" TEXT,
 "Grade" TEXT);
sqlite>
```

To view the data of the student table, you use the following SELECT statement.

```
SELECT * FROM STUD;
```

Then the output is:

```
sqlite> select * from stud;
1,Rakesh,4,67,C
2,Rishav,5,56,D
3,Harshit,6,90,A
4,Diksha,3,56,D
5,Rajesh,8,98,A
6,Tarun,7,87,A
7,Binita,6,76,B
8,Seema,5,89,A
9,Mohit,4,65,C
10,Sumit,5,76,B
11,Ramesh,7,69,C
```

Here Record are separate in comma beacase we give mode in csv and not there header.

If we want to header and table format we write following command.

Sqlite>.header on

Sqlite>.mode box

OR

Sqlite>.mode column

```
sqlite> .header on
sqlite> .mode box
sqlite> select * from stud;
```

Roll No	Name	Class	Marks	Grade
1	Rakesh	4	67	C
2	Rishav	5	56	D
3	Harshit	6	90	A
4	Diksha	3	56	D
5	Rajesh	8	98	A
6	Tarun	7	87	A
7	Binita	6	76	B
8	Seema	5	89	A
9	Mohit	4	65	C
10	Sumit	5	76	B
11	Ramesh	7	69	C

2.2.3 Export a CSV file from a table

SQLite project provides you with a command-line program called sqlite3 or sqlite3.exe on Windows. 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.

First you create table employee in sqlite like following formate.

```
sqlite> select * from employee;
```

eno	ename	desg	salary
1	Rakesh	Executive manager	80000.0
2	Rahul	Executive manager	80000.0
3	Vikash	Asst manager	80000.0
4	Varun	Manager	80000.0
5	Harshit	Supervisor	80000.0
6	Tarun	senior manager	80000.0
7	Rishav	Supervisor	80000.0
8	Rakesh	Executive manager	80000.0

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

```
sqlite> .header on
sqlite> .mode csv
sqlite> .output C:\sqlite\emp.csv
sqlite> select * from employee;
sqlite> .quit
```

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

	A	B	C	D
1	eno	ename	desg	salary
2	1	Rakesh	Executive manager	80000
3	2	Rahul	Executive manager	80000
4	3	Vikash	Asst manager	80000
5	4	Varun	Manager	80000
6	5	Harshit	Supervisor	80000
7	6	Tarun	senior manager	80000
8	7	Rishav	Supervisor	80000
9	8	Rakesh	Executive manager	80000

Python Modules

A 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.

Example

In this example, we will create a module named as file.py which contains a function func that contains a code to print some message on the console.

Let's create the module named as **file.py.**

```
#displayMsg prints a message to the name being passed.
def displayMsg(name)
   print("Hi "+name);
```

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

Loading the module in our python code

We need to load the module in our python code to use its functionality. 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.

The syntax to use the import statement is given below.

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

Hence, if we need to call the function displayMsg() defined in the file file.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.displayMsg(name)
```

Output:

```
Enter the name?John
Hi John
```

The from-import statement

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement. The syntax to use the from-import statement is given below.

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

Consider the following module named as calculation which contains three functions as summation, multiplication, and divide.

calculation.py:

```
#place the code in the calculation.py
def summation(a,b):
   return a+b
def multiplication(a,b):
   return a*b;
def divide(a,b):
   return a/b;
```

Main.py:

```
from calculation import summation
#it will import only the summation() from calculation.py
a = int(input("Enter the first number"))
b = int(input("Enter the second number"))
print("Sum = ",summation(a,b)) #we do not need to specify the module name while accessing s
ummation()
```

Output:

```
Enter the first number10
Enter the second number20
Sum =  30
```

The from...import statement is always better to use if we know the attributes to be imported from the module in advance. It doesn't let our code to be heavier. We can also import all the attributes from a module by using *.

Consider the following syntax.

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

Renaming a module

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

The syntax to rename a module is given below.

```
import <module-name> as <specific-name>
```

Example

```
#the module calculation of previous example is imported in this example as cal.
import calculation as cal;
a = int(input("Enter a?"));
b = int(input("Enter b?"));
print("Sum = ",cal.summation(a,b))
```

Output:

```
Enter a?10
Enter b?20
Sum =  30
```

PYTHONPATH

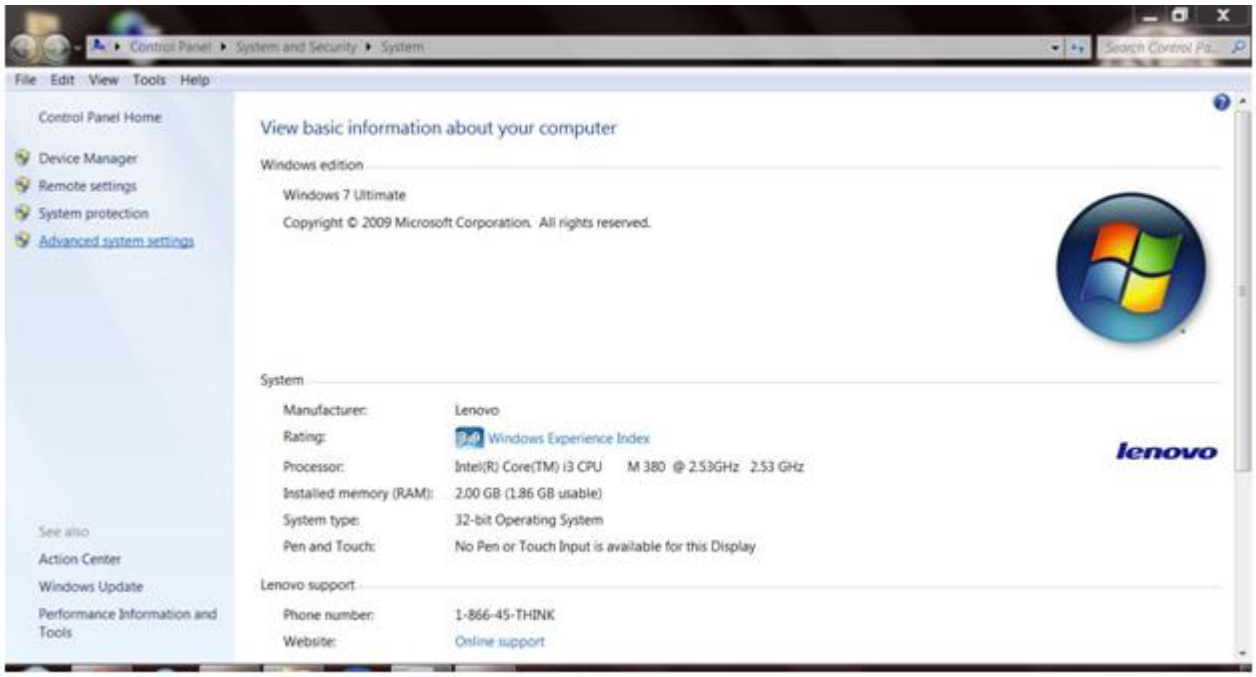
PYTHONPATH is an environment variable which you can set to add additional directories where python will look for modules and packages. For most installations, you should not set these variables since they are not needed for Python to run. Python knows where to find its standard library.

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 step:

1. Right click on My Computer and click on properties.

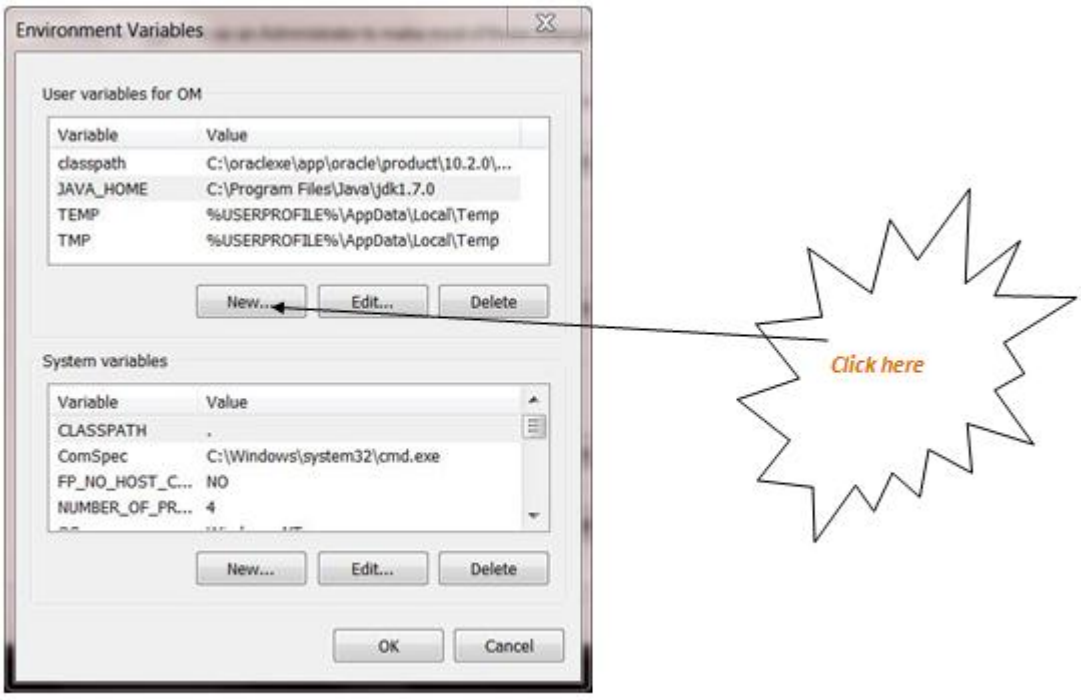
2. Click on Advanced System settings



javatpoint.com

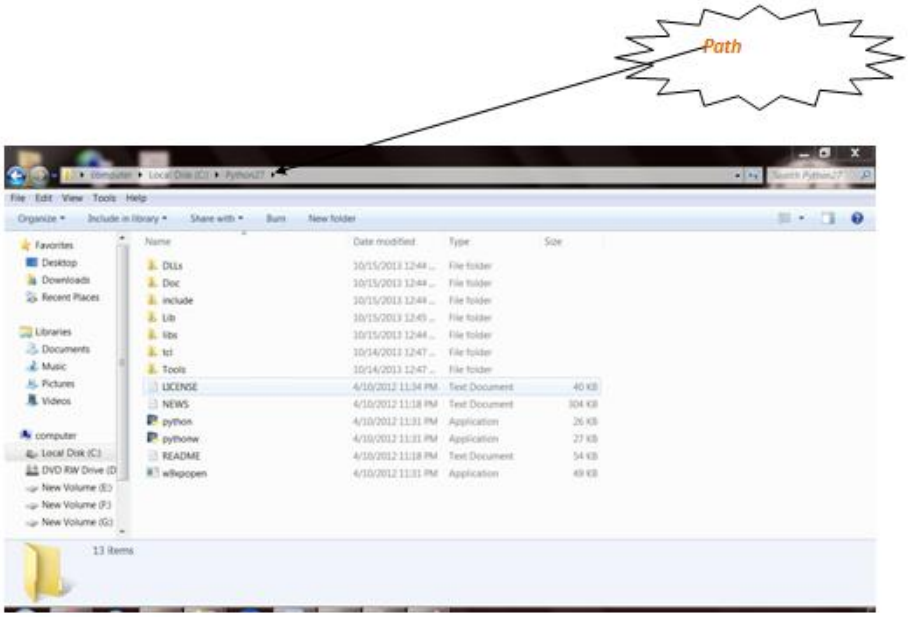
3. Click on Environment Variable tab.

4. Click on new tab of user variables.

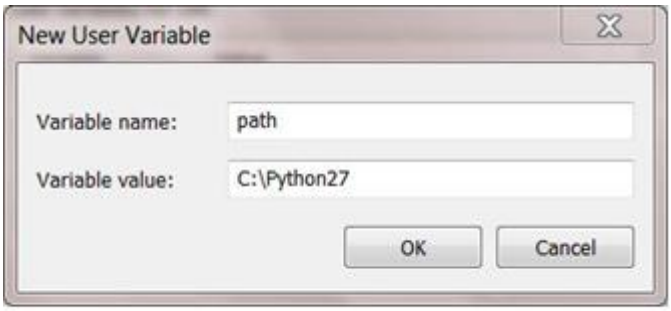


5. Write path in variable name

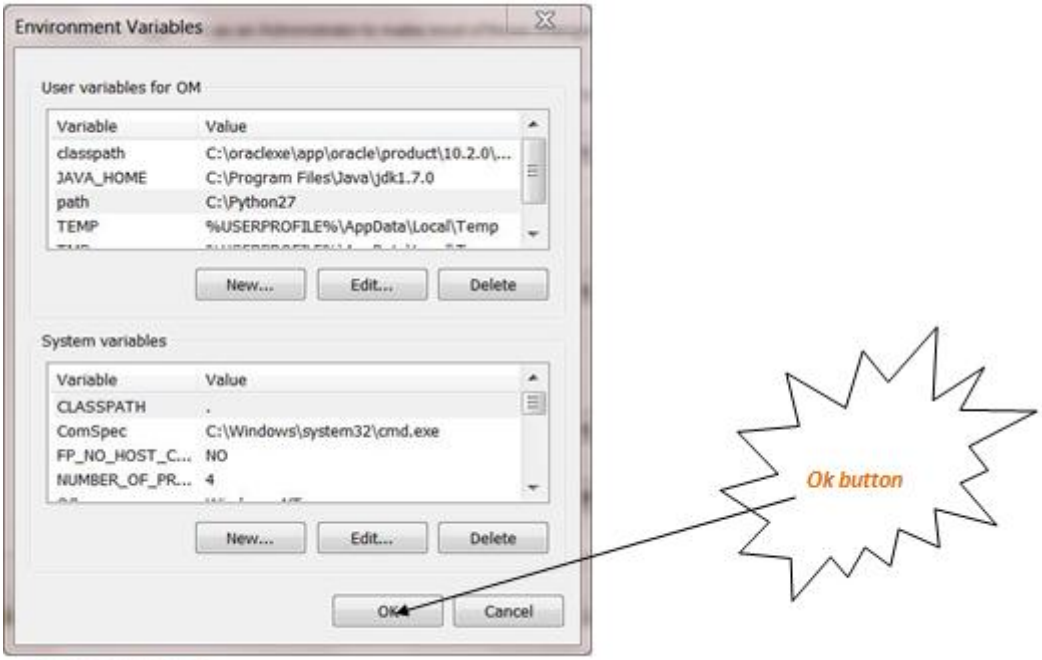
6. Copy the path of Python folder



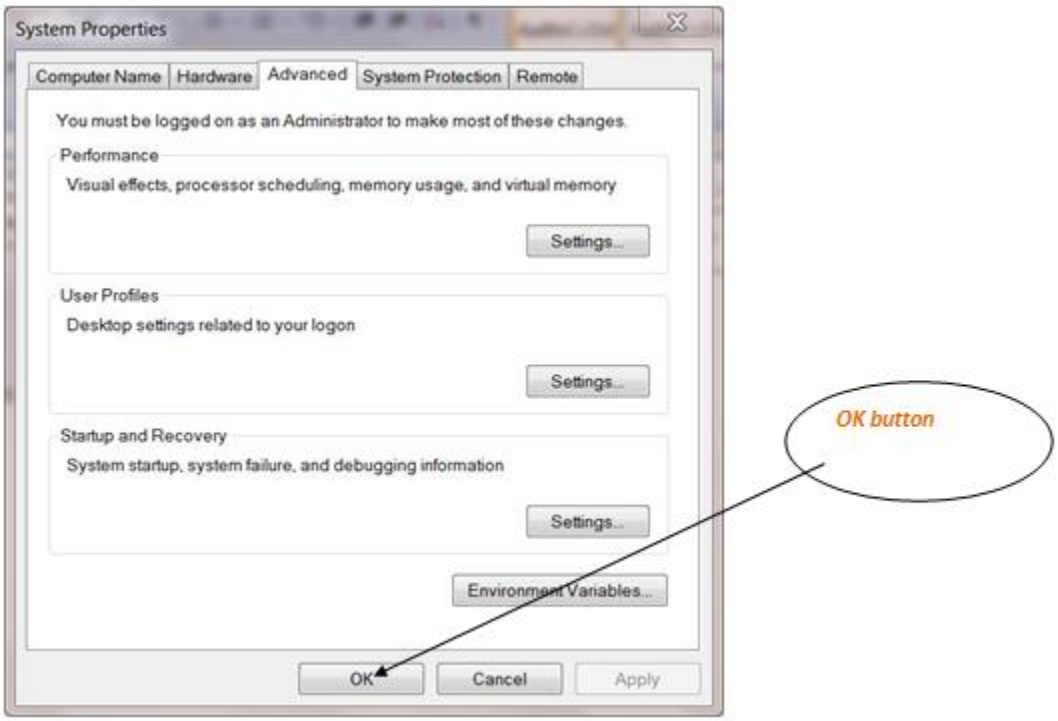
7. Paste path of Python in variable value.



8. Click on Ok button:



9. Click on Ok button:

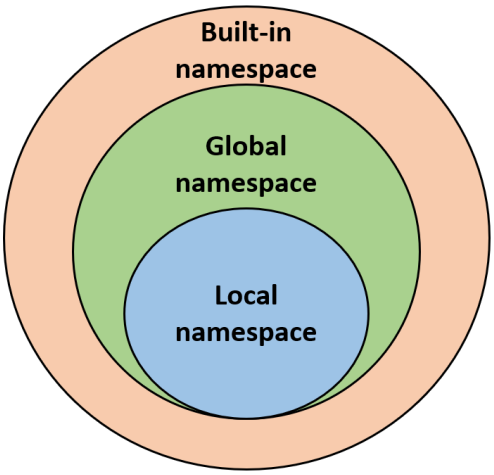


Concepts of Namespace and scope

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.

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.



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'
def outer_func():
  outer_var = 'x'

  def inner_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 ')
```

Running the above code gives us the following result –

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

Package in Python

We organize a large number of files in different folders and subfolders based on some criteria, so that we can find and manage them easily. In the same way, a package in Python takes the concept of the modular approach to next logical level. As you know, a module can contain multiple objects, such as classes, functions, etc. A package can contain one or more relevant modules. Physically, a package is actually a folder containing one or more module files.

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

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

greet.py

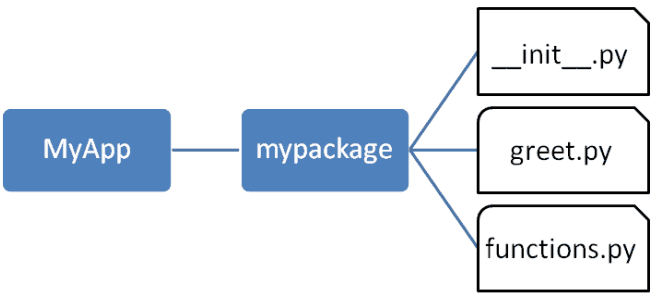
```
def SayHello(name):
   print("Hello ", name)
```

functions.py

Copy

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

def average(x,y):
   return (x+y)/2

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

That's it. We have created our package called mypackage. The following is a folder structure:



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 specification described by PEP 249. 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 to Database

Here to connect with database we use connect()

connect()

This routine opens a connection to the SQLite database file.You can use ".memory:" to open a database connection to database that resides in RAM instead of on disk.If database is opened successfully,it return a connection object.

When a database is accessed by multiple connections,and one of the processes modify the database,the sqlite database is locked until that transaction is committed.

Syntax:

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

Example:

Following Python code shows how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned.

```
import sqlite3

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

print('Opened database successfully')
```

Now, let's run the above program to create our database **test.db** in the current directory. You can change your path as per your requirement. Keep the above code in sqlite.py file and

execute it as shown below. If the database is successfully created, then it will display the following message.

Output:

Open database successfully

Execute()

This routine execute an SQL statement may be parameterize.The sqlite3 module supports two kinds of placeholders:question marks and named placeholder.

Syntax:

Cursor.execute(sql,[,optional parameters])

Example:

Cursor.execute("insert into student(sno,sname) values(1,'Ram')")

Create a Table

Following Python program will be used to create a table in the previously created database.

```
import sqlite3

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

print('Opened database successfully')

cursor = conn.cursor()

cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

sql ='''CREATE TABLE EMPLOYEE(

FIRST_NAME  CHAR(20) NOT NULL, LAST_NAME  CHAR(20),

AGE INT,SEX CHAR(1), INCOME FLOAT)'''

cursor.execute(sql)

print("Table created successfully.....")
```

Output:

Opened database successfully

Table created successfully.....

Insert Operation

Following Python program shows how to create records in the COMPANY table created in the above example.

```
import sqlite3

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

print('Opened database successfully')

cursor = conn.cursor()

cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

sql ='''CREATE TABLE EMPLOYEE(

FIRST_NAME  CHAR(20) NOT NULL, LAST_NAME  CHAR(20),

AGE INT,SEX CHAR(1), INCOME FLOAT)'''

cursor.execute(sql)

print("Table created successfully.....")

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES ('Ramya', 'Rama Priya', 27, 'F', 9000)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES ('Vinay', 'Battacharya', 20, 'M', 6000)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES ('Sharukh', 'Sheik', 25, 'M', 8300)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME)
VALUES ('Sarmista', 'Sharma', 26, 'F', 10000)''')

cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,INCOME)
VALUES ('Tripthi', 'Mishra', 24, 'F', 6000)''')
```

```
print("Records inserted.....")
```

When the above program is executed, it will create the given records in the COMPANY table and it will display the following two lines –

Output:

```
Opened database successfully

Table created successfully.....

Records inserted.....
```

Select Operation

For fetch and display record from table we use two methods fetchone() and fetchall().

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 return a single tuple.

Syntax:

```
Cursor.fetchone()
```

Fetchone():

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

Syntax:

```
Cursor.fetchall()
```

Example of fetchone() and fetchall():

Following Python program shows how to fetch and display records from the COMPANY table created in the above example.

```
import sqlite3
```

```
conn = sqlite3.connect('example.db')

cursor = conn.cursor()

cursor.execute("SELECT * from EMPLOYEE")

result = cursor.fetchone();

print(result)

result = cursor.fetchall();

print(result)

conn.commit()

conn.close()
```

When the above program is executed, it will produce the following result.

Output:

```
('Ramya', 'Rama Priya', 27, 'F', 9000.0)

[('Vinay', 'Battacharya', 20, 'M', 6000.0), ('Sharukh', 'Sheik', 25, 'M', 8300.0), ('Sarmista', 'Sharma', 26, 'F', 10000.0), ('Tripthi', 'Mishra', 24, 'F', 6000.0)]
```

Update operation

Following Python code shows how to use UPDATE statement to update any record and then fetch and display the updated records from the COMPANY table.

```
import sqlite3

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

conn.execute("update EMPLOYEE set FIRST_NAME='Kavya' where LAST_NAME='Sharma'")

print("record updated.....")

cursor = conn.cursor()

cursor.execute("SELECT * from EMPLOYEE")

result = cursor.fetchall();
```

```
print(result)

conn.commit()

conn.close()
```

When the above program is executed, it will produce the following result.

Output:

```
record updated.....

[('Ramya', 'Rama Priya', 27, 'F', 9000.0), ('Sharukh', 'Sheik', 25, 'M', 8300.0), ('Kavya', 'Sharma', 2
6, 'F', 10000.0), ('Tripthi', 'Mishra', 24, 'F', 6000.0)]
```

Delete Operation

Following Python code shows how to use DELETE statement to delete any record and then fetch and display the remaining records from the COMPANY table.

```
import sqlite3

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

conn.execute("delete from EMPLOYEE where AGE=20")

print("record deleted.....")

cursor = conn.cursor()

cursor.execute("SELECT * from EMPLOYEE")

result = cursor.fetchall();

print(result)

conn.commit()

conn.close()
```

Output:

```
record deleted.....

[('Ramya', 'Rama Priya', 27, 'F', 9000.0), ('Sharukh', 'Sheik', 25, 'M', 8300.0), ('Sarmista', 'Sharma'
, 26, 'F', 10000.0), ('Tripthi', 'Mishra', 24, 'F', 6000.0)]
```

Python File Handling

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.

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

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.

SN	Access mode	Description
1	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.
2	rb	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
3	r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
4	rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
5	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.
6	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.
7	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.
8	wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	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.
10	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.
11	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.
12	ab+	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

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

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)

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

Example

```
fo = open("demo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n")
fo.close()
```

The above method would create foo.txt file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

Output:

Python is a great language.

Yeah its great!!

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

Let's take a file foo.txt, which we created above.

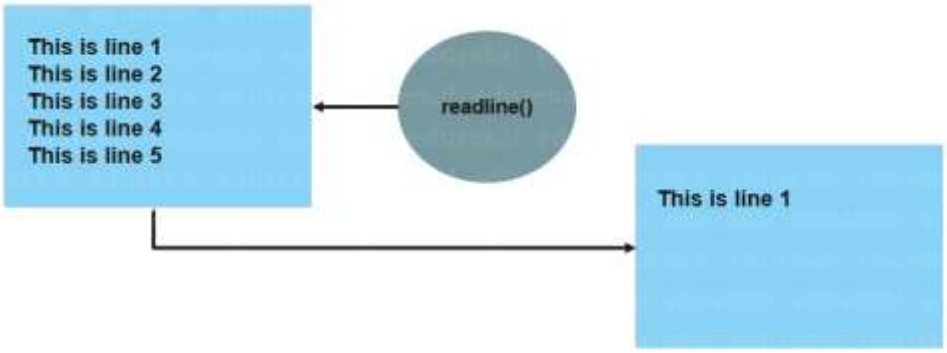
```
fo = open("demo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
fo.close()
```

Output:

Read String is : Python is

Readline() Method

readline() method will return a line from the file when called.



Syntax

file.readline()

Example

Let us suppose we have a text file with the name **examples.txt** with the following content.

examples.txt

Python is the best programming language in the world in 2020
Edureka is the biggest Ed-tech platform to learn python
Python programming is as easy as writing a program in simple English language

Now, to use the readline, we will have to open the example.txt file.

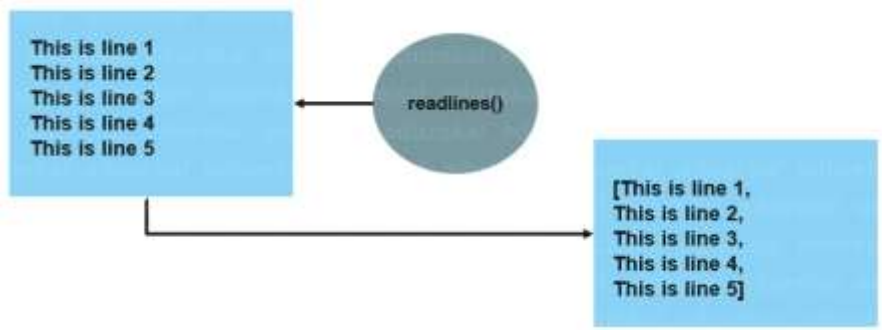
```
file = open("example.txt", "r")
example1 = file.readline()
example2 = file.readline(14)
print(example1)
print(example2)
```

Output:

```
Python is the best programming language in the world in 2020
Edureka is the
```

Readlines() Method

readlines() method will return all the lines in a file in the format of a list where each element is a line in the file.



Syntax

file.readlines()

Example

Let us suppose we have a text file with the name **examples.txt** with the following content.

examples.txt

Python is the best programming language in the world in 2020
Edureka is the biggest Ed-tech platform to learn python
Python programming is as easy as writing a program in simple English language

Now, to use the readline, we will have to open the example.txt file.

```
file = open("example.txt", "r")
example1 = file.readlines()
example2 = file.readlines(80)
print(example1)
print(example2)
```

Output: ['Python is the best programming language in the world in 2020',
'Edureka is the biggest Ed-tech platform to learn python',
'Python programming is as easy as writing a program in simple English language']

['Python is the best programming language in the world in 2020',
'Edureka is the biggest Ed-tech platform to learn python']

The 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.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.

The syntax to use the **close()** method is given below.

Syntax

```
fileobject.close()
```

Consider the following example.

```
fileptr = open("file.txt","r")
if fileptr:
   print("file is opened successfully")
fileptr.close()
```

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

We should use the following method to overcome such type of problem.

```
try:
```

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

The syntax to open a file using with the statement is given below.

```
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, we don't need to write the **close()** function. It doesn't let the file to corrupt.

Consider the following example.

Example

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

What is the 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:

```
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.

Reading a csv file in Python

To read a CSV file in Python, you follow these steps:

First, import the csv module:

```
import csv
```

Second, open the CSV file using the built-in open() function in the read mode:

```
f = open('path/to/csv_file')
```

If the CSV contains UTF8 characters, you need to specify the encoding like this:

```
f = open('path/to/csv_file', encoding='UTF8')
```

Third, pass the file object (f) to the reader() function of the csv module. The reader() function returns a csv reader object:

```
csv_reader = csv.reader(f)
```

The csv_reader is an iterable object of lines from the CSV file. Therefore, you can iterate over the lines of the CSV file using a for loop:

```
for line in csv_reader:
    print(line)
```

Each line is a list of values. To access each value, you use the square bracket notation []. The first value has an index of 0. The second value has an index of 1, and so on.

For example, the following accesses the first value of a particular line:

```
line[0]
```

Finally, always close the file once you're no longer access it by calling the close() method of the file object:

```
f.close()
```

It'll be easier to use the with statement so that you don't need to explicitly call the close() method.

The following illustrates all the steps for reading a CSV file:

```
import csv

with open('path/to/csv_file', 'r') as f:
    csv_reader = csv.reader(f)
    for line in csv_reader:
        print(line)
```

Reading a CSV file examples

We'll use the country.csv file that contains country information including name, area, 2-letter country code, 3-letter country code:

country.csv

```
1  "name","area","country_code2","country_code3"
2  "Afghanistan","652090.00","AF","AFG"
3  "Albania","28748.00","AL","ALB"
4  "Algeria","2381741.00","DZ","DZA"
5  "American Samoa","199.00","AS","ASM"
6  "Andorra","468.00","AD","AND"
7  "Angola","1246700.00","AO","AGO"
```

country.csv file

	A	B	C	D
1	name	area	country_code2	country_code3
2	Afghanistan	652090	AF	AFG
3	Albania	28748	AL	ALB
4	Algeria	2381741	DZ	DZA
5	American Samoa	199	AS	ASM
6	Andorra	468	AD	AND
7	Angola	1246700	AO	AGO
8	Anguilla	96	AI	AIA

The following shows how to read the country.csv file and display each line to the screen:

```
import csv

with open('country.csv', encoding="utf8") as f:
    csv_reader = csv.reader(f)
    for line in csv_reader:
        print(line)
```

Output:

```
['name', 'area', 'country_code2', 'country_code3']
['Afghanistan', '652090.00', 'AF', 'AFG']
['Albania', '28748.00', 'AL', 'ALB']
['Algeria', '2381741.00', 'DZ', 'DZA']
['American Samoa', '199.00', 'AS', 'ASM']
...
```

Reading a CSV file using the DictReader class

When you use the csv.reader() function, you can access values of the CSV file using the bracket notation such as line[0], line[1], and so on. However, using the csv.reader() function has two main limitations:

- First, the way to access the values from the CSV file is not so obvious. For example, the line[0] implicitly means the country name. It would be more expressive if you can access the country name like line['country_name'].

- Second, when the order of columns from the CSV file is changed or new columns are added, you need to modify the code to get the right data.

This is where the DictReader class comes into play. The DictReader class also comes from the csv module.

The DictReader class allows you to create an object like a regular CSV reader. But it maps the information of each line to a dictionary (dict) whose keys are specified by the values of the first line.

By using the DictReader class, you can access values in the country.csv file like line['name'], line['area'], line['country_code2'], and line['country_code3'].

The following example uses the DictReader class to read the country.csv file:

```
import csv

with open('country.csv', encoding="utf8") as f:
    csv_reader = csv.DictReader(f)
        for line in csv_reader:
        print(f"The area of {line['name']} is {line['area']} km2")
```

Output:

```
The area of Afghanistan is 652090.00 km2
The area of Albania is 28748.00 km2
The area of Algeria is 2381741.00 km2
...
```

If you want to have different field names other than the ones specified in the first line, you can explicitly specify them by passing a list of field names to the DictReader() constructor like this:

```
import csv

fieldnames = ['country_name', 'area', 'code2', 'code3']

with open('country.csv', encoding="utf8") as f:
    csv_reader = csv.DictReader(f, fieldnames)
    next(csv_reader)
    for line in csv_reader:
        print(f"The area of {line['country_name']} is {line['area']} km2")
```

In this example, instead of using values from the first line as the field names, we explicitly pass a list of field names to the DictReader constructor.

Steps for writing a CSV file

To write data into a CSV file, you follow these steps:

- First, open the CSV file for writing (w mode) by using the open() function.
- Second, create a CSV writer object by calling the writer() function of the csv module.
- Third, write data to CSV file by calling the writerow() or writerows() method of the CSV writer object.
- Finally, close the file once you complete writing data to it.

The following code illustrates the above steps:

```
import csv
f = open('path/to/csv_file', 'w')
writer = csv.writer(f)
writer.writerow(row)
f.close()
```

It'll be shorter if you use the with statement so that you don't need to call the close() method to explicitly close the file:

```
import csv
with open('path/to/csv_file', 'w') as f:
    writer = csv.writer(f)
    writer.writerow(row)
```

If you're dealing with non-ASCII characters, you need to specify the character encoding in the open() function.

The following illustrates how to write UTF-8 characters to a CSV file:

```
import csv
with open('path/to/csv_file', 'w', encoding='UTF8') as f:
    writer = csv.writer(f)
    writer.writerow(row)
```

Writing to CSV files example

The following example shows how to write data to the CSV file:

```
import csv
```

```
header = ['name', 'area', 'country_code2', 'country_code3']
data = ['Afghanistan', 652090, 'AF', 'AFG']

with open('countries.csv', 'w', encoding='UTF8') as f:
    writer = csv.writer(f)
    writer.writerow(header)
    writer.writerow(data)
```

If you open the countries.csv, you'll see one issue that the file contents have an additional blank line between two subsequent rows:

```
1   name,area,country_code2,country_code3
2
3   Afghanistan,652090,AF,AFG
```

To remove the blank line, you pass the keyword argument newline='' to the open() function as follows:

```
import csv

header = ['name', 'area', 'country_code2', 'country_code3']
data = ['Afghanistan', 652090, 'AF', 'AFG']


with open('countries.csv', 'w', encoding='UTF8', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(header)
    writer.writerow(data)
```

Output:

```
1   name,area,country_code2,country_code3
2   Afghanistan,652090,AF,AFG
```

Writing multiple rows to CSV files

To write multiple rows to a CSV file at once, you use the writerows() method of the CSV writer object.

The following uses the writerows() method to write multiple rows into the countries.csv file:

```
import csv
```

```
header = ['name', 'area', 'country_code2', 'country_code3']
data = [
   ['Albania', 28748, 'AL', 'ALB'],
   ['Algeria', 2381741, 'DZ', 'DZA'],
   ['American Samoa', 199, 'AS', 'ASM'],
   ['Andorra', 468, 'AD', 'AND'],
   ['Angola', 1246700, 'AO', 'AGO']
]

with open('countries.csv', 'w', encoding='UTF8', newline='') as f:
   writer = csv.writer(f)
   writer.writerow(header)
   writer.writerows(data)
```

Writing to CSV files using the DictWriter class

If each row of the CSV file is a dictionary, you can use the DictWriter class of the csv module to write the dictionary to the CSV file.

The example illustrates how to use the DictWriter class to write data to a CSV file:

```
import csv

# csv header
fieldnames = ['name', 'area', 'country_code2', 'country_code3']

# csv data
rows = [
   {'name': 'Albania',
   'area': 28748,
   'country_code2': 'AL',
   'country_code3': 'ALB'},
   {'name': 'Algeria',
   'area': 2381741,
   'country_code2': 'DZ',
   'country_code3': 'DZA'},
   {'name': 'American Samoa',
   'area': 199,
   'country_code2': 'AS',
   'country_code3': 'ASM'}
]
```

```
with open('countries.csv', 'w', encoding='UTF8', newline='') as f:
    writer = csv.DictWriter(f, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerows(rows)
```

How it works.

- First, define variables that hold the field names and data rows of the CSV file.
- Next, open the CSV file for writing by calling the open() function.
- Then, create a new instance of the DictWriter class by passing the file object (f) and fieldnames argument to it.
- After that, write the header for the CSV file by calling the writeheader() method.
- Finally, write data rows to the CSV file using the writerows() method.

DataFrame Handling using Panda and Numpy

Python Pandas Introduction

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**. It is used for data analysis in Python and developed by **Wes McKinney** in **2008**.

Data analysis requires lots of processing, such as **restructuring, cleaning** or **merging**, etc. There are different tools are available for fast data processing, such as **Numpy, Scipy, Cython**, and **Panda**. But we prefer Pandas because working with Pandas is fast, simple and more expressive than other tools.

Pandas is built on top of the **Numpy** package, means **Numpy** is required for operating the Pandas.

Before Pandas, Python was capable for data preparation, but it only provided limited support for data analysis. So, Pandas came into the picture and enhanced the capabilities of data analysis. It can perform five significant steps required for processing and analysis of data irrespective of the origin of the data, i.e., **load, manipulate, prepare, model, and analyze**.

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**.

Benefits of Pandas

The benefits of pandas over using other language are as follows:

- **Data Representation:** It represents the data in a form that is suited for data analysis through its DataFrame and Series.
- **Clear code:** The clear API of the Pandas allows you to focus on the core part of the code. So, it provides clear and concise code for the user.

Python Pandas Data Structure

The Pandas provides two data structures for processing the data, i.e., **Series** and **DataFrame**, which are discussed below:

1) 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:

Data: It can be any list, dictionary, or scalar value.

Creating Series from Array:

Before creating a Series, Firstly, we have to import the numpy module and then use array() function in the program.

```
import pandas as pd
import numpy as np
info = np.array(['P','a','n','d','a','s'])
a = pd.Series(info)
print(a)
```

Output

```
0 P
1 a
2 n
3 d
4 a
5 s
```

Explanation: In this code, firstly, we have imported the **pandas** and **numpy** library with the **pd** and **np** alias. Then, we have taken a variable named "info" that consist of an array of some values. We have called the **info** variable through a **Series** method and defined it in an "**a**" variable. The Series has printed by calling the **print(a)** method.

2) 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.

Create a DataFrame using List:

We can easily create a DataFrame in Pandas using list.

```
import pandas as pd
x = ['Python', 'Pandas']
df = pd.DataFrame(x)
print(df)
```

Output

```
    0
0  Python
1  Pandas
```

Explanation: In this code, we have defined a variable named "x" 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

Dict={'Name':['ram','deep','pankaj','jay'],'marks':[44,65,75,48]}

df=pd.DataFrame(Dict)

print(df)
```

Output:

```
      Name  marks
0      ram     44
1     deep     65
2   pankaj     75
3      jay     48
```

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.

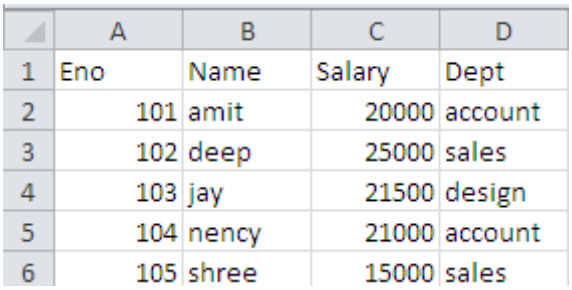
```
Import pandas as pd

Data=[{‘x’:10,’y’:20}.{‘x’:55,’y’:15,’z’:88}]

Df=pd.DtaFrame(data)
```

Output

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

Create Data Frame from Excel Spreadsheet

The Pandas library provides features using which we can read the Exel file in full as well in parts for only a selected group of data,we can also read an Exel 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 Exel program and save the file as demo.xlsx.



	A	B	C	D
1	Eno	Name	Salary	Dept
2	101	amit	20000	account
3	102	deep	25000	sales
4	103	jay	21500	design
5	104	nency	21000	account
6	105	shree	15000	sales

```
import pandas as pd

data=pd.read_excel('demo.xlsx')

print(data)
```

Output

```
   Eno    Name  Salary     Dept
0  101    amit   20000  account
1  102    deep   25000    sales
2  103     jay   21500   design
3  104   nency   21000  account
4  105   shree   15000    sales
```

Create 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.

```
import pandas as pd

data=pd.read_csv('demo.csv')
```

print(data)

Output

```
   Eno    Name  Salary     Dept
0  101    amit   20000  account
1  102    deep   25000    sales
2  103     jay   21500   design
3  104   nency   21000  account
4  105   shree   15000    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.

Matplotlib Pyplot

Pyplot

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

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

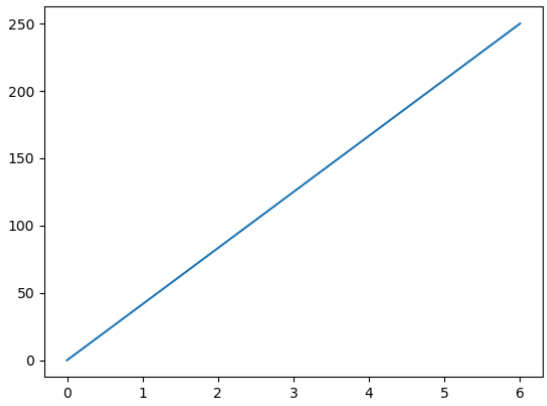
Now the Pyplot package can be referred to as plt.

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()
```

Result:



Matplotlib Plotting

Plotting x and y points

The plot() function is used to draw points (markers) in a diagram.

By default, the plot() function draws a line from point to point.

The function takes parameters for specifying points in the diagram.

Parameter 1 is an array containing the points on the **x-axis**.

Parameter 2 is an array containing the points on the **y-axis**.

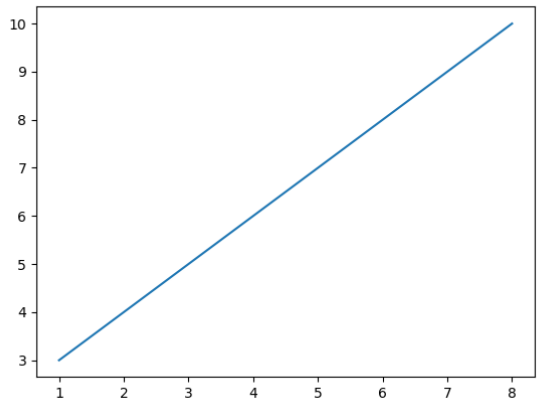
If we need to plot a line from (1, 3) to (8, 10), we have to pass two arrays [1, 8] and [3, 10] to the plot function.

Example

Draw a line in a diagram from position (1, 3) to position (8, 10):

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

xpoints = np.array([1, 8])
ypoints = np.array([3, 10])

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

Result:



Matplotlib **Markers**

Markers

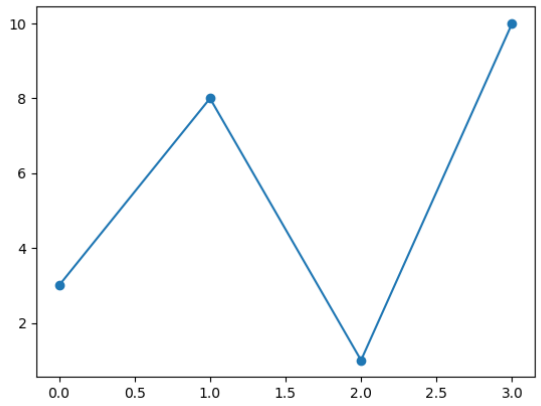
You can use the keyword argument marker to emphasize each point with a specified marker:

Example

Mark each point with a circle:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o')
plt.show()
```

Result:

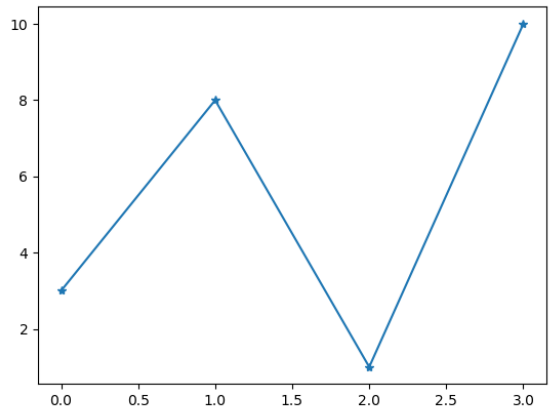


Example

Mark each point with a star:

```
...
plt.plot(ypoints, marker = '*')
...
```

Result:



Marker Reference

You can choose any of these markers:

Marker	Description
'o'	Circle
'*'	Star
'.'	Point
','	Pixel
'x'	X

'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up

'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
'|'	Vline
'_'	Hline

Format Strings **fmt**

You can use also use the *shortcut string notation* parameter to specify the marker.

This parameter is also called fmt, and is written with this syntax:

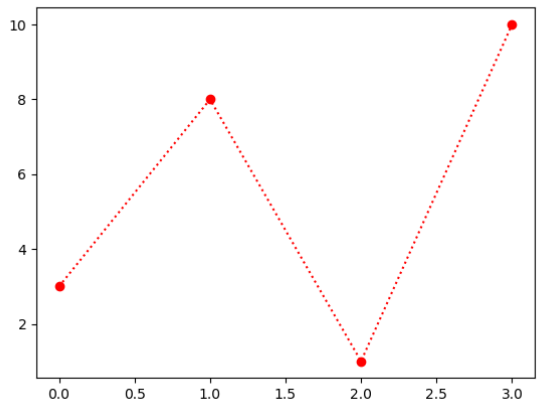
marker|*line*|*color*

Example

Mark each point with a circle:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, 'o:r')
plt.show()
```

Result:



The marker value can be anything from the Marker Reference above.

The line value can be one of the following:

Line Reference

Line Syntax	Description
'-'	Solid line
':'	Dotted line

'--'	Dashed line
'-.'	Dashed/dotted line

Note: If you leave out the *line* value in the fmt parameter, no line will be plotted.

The short color value can be one of the following:

Color Reference

Color Syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow

'k'	Black
'w'	White

Marker Size

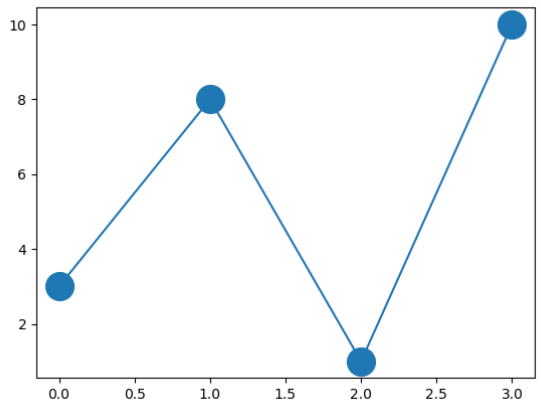
You can use the keyword argument markersize or the shorter version, ms to set the size of the markers:

Example

Set the size of the markers to 20:

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

ypoints = np.array([3, 8, 1, 10])
plt.plot(ypoints, marker = 'o', ms = 20)
plt.show()
```

Result:



Marker Color

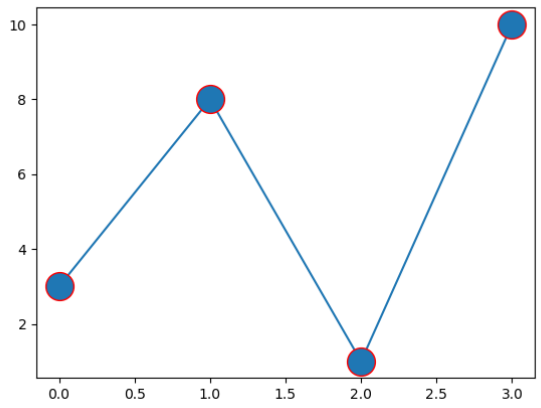
You can use the keyword argument markeredgecolor or the shorter mec to set the color of the *edge* of the markers:

Example

Set the EDGE color to red:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mec = 'r')
plt.show()
```

Result:



You can use the keyword argument markerfacecolor or the shorter mfc to set the color inside the edge of the markers:

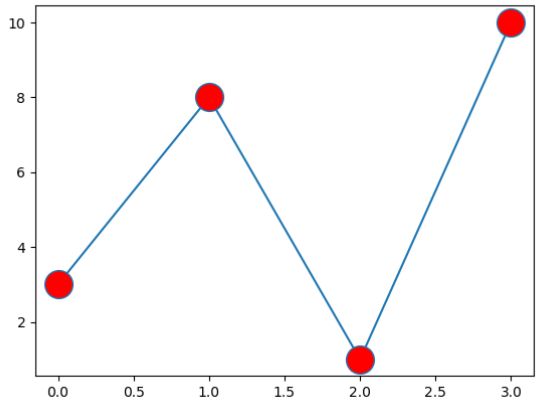
Example

Set the FACE color to red:

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

```
ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mfc = 'r')
plt.show()
```

Result:



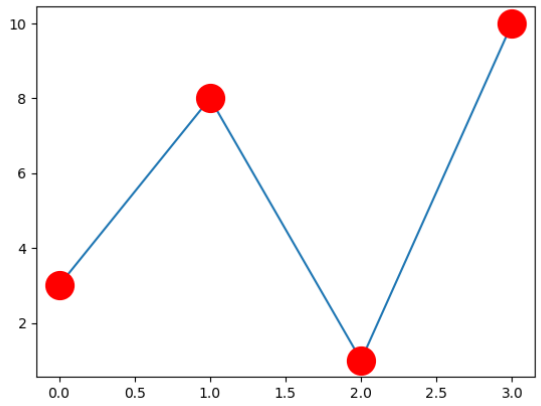
Use *both* the mec and mfc arguments to color of the entire marker:

Example

Set the color of both the *edge* and the *face* to red:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mec = 'r', mfc = 'r')
plt.show()
```

Result:



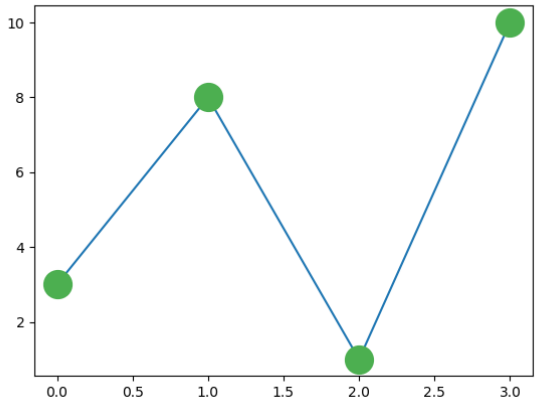
You can also use Hexadecimal color values:

Example

Mark each point with a beautiful green color:

```
...
plt.plot(ypoints, marker = 'o', ms = 20, mec = '#4CAF50', mfc = '#4CAF50')
...
```

Result:



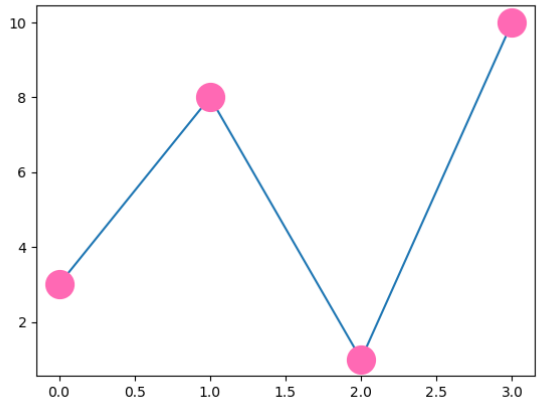
Or any of the 140 supported color names.

Example

Mark each point with the color named "hotpink":

```
...
plt.plot(ypoints, marker = 'o', ms = 20, mec = 'hotpink', mfc = 'hotpink')
...
```

Result:



Matplotlib Line

Linestyle

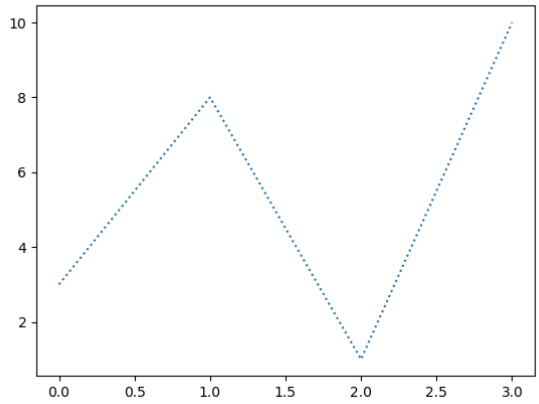
You can use the keyword argument linestyle, or shorter ls, to change the style of the plotted line:

Example

Use a dotted line:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linestyle = 'dotted')
plt.show()
```

Result:

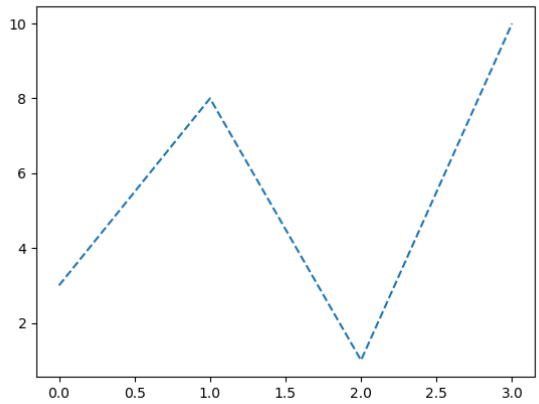


Example

Use a dashed line:

plt.plot(ypoints, linestyle = 'dashed')

Result:



Shorter Syntax

The line style can be written in a shorter syntax:

linestyle can be written as ls.

dotted can be written as :.

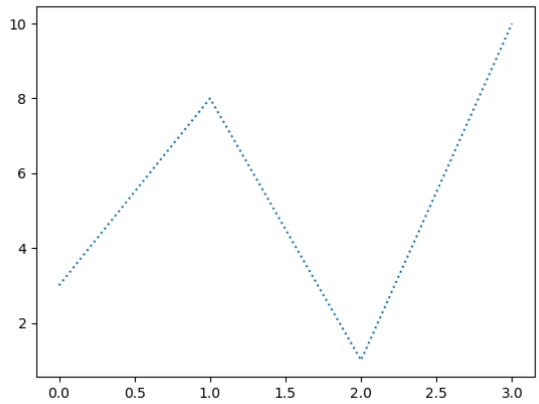
dashed can be written as --.

Example

Shorter syntax:

```
plt.plot(ypoints, ls = ':')
```

Result:



Line Styles

You can choose any of these styles:

Style	Or
'solid' (default)	'-'
'dotted'	':'
'dashed'	'--'

'dashdot'	'-.'
'None'	'' or ' '

Line Color

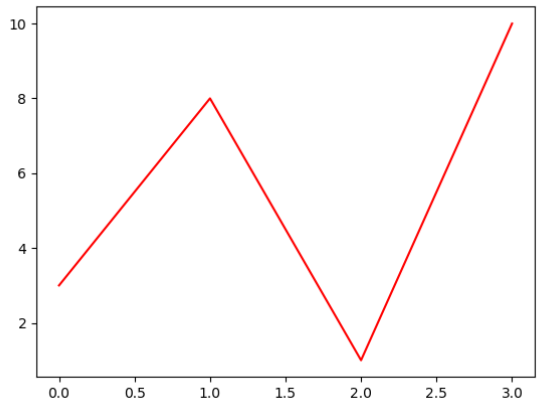
You can use the keyword argument color or the shorter c to set the color of the line:

Example

Set the line color to red:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, color = 'r')
plt.show()
```

Result:



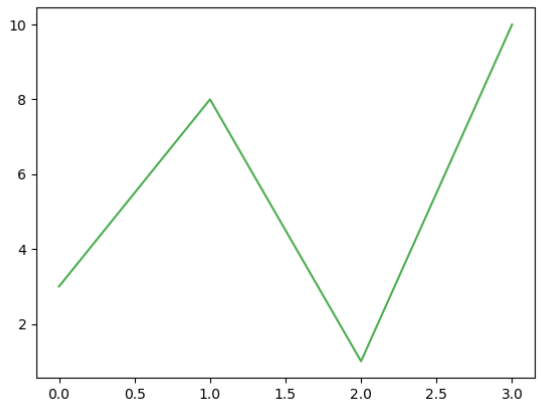
You can also use Hexadecimal color values:

Example

Plot with a beautiful green line:

```
...
plt.plot(ypoints, c = '#4CAF50')
...
```

Result:



Or any of the 140 supported color names.

Line Width

You can use the keyword argument linewidth or the shorter lw to change the width of the line.

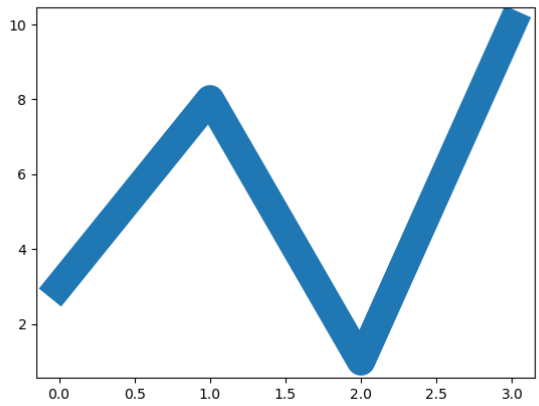
The value is a floating number, in points:

Example

Plot with a 20.5pt wide line:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linewidth = '20.5')
plt.show()
```

Result:

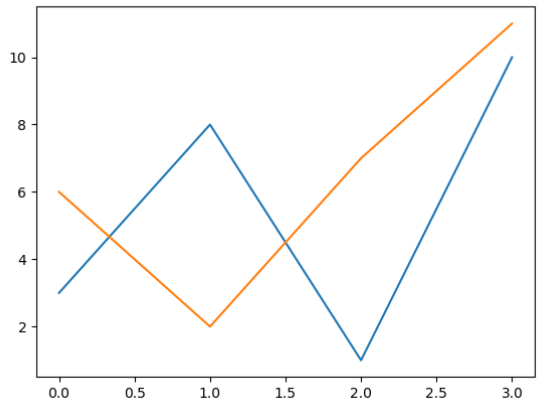


Multiple Lines

Example

```
import matplotlib.pyplot as plt
import numpy as np
y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])
plt.plot(y1)
plt.plot(y2)
plt.show()
```

Result:



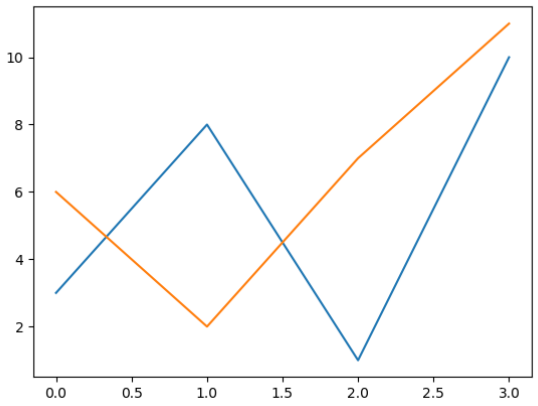
You can also plot many lines by adding the points for the x- and y-axis for each line in the same plt.plot() function. The x- and y- values come in pairs:

Example

Draw two lines by specifiyng the x- and y-point values for both lines:

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

x1 = np.array([0, 1, 2, 3])
y1 = np.array([3, 8, 1, 10])
x2 = np.array([0, 1, 2, 3])
y2 = np.array([6, 2, 7, 11])

plt.plot(x1, y1, x2, y2)
plt.show()
```

Result:



Create Labels for a Plot

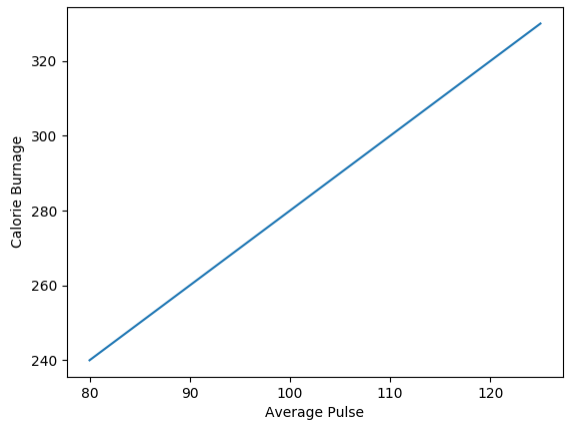
With Pyplot, you can use the xlabel() and ylabel() functions to set a label for the x- and y-axis.

Example

Add labels to the x- and y-axis:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```

Result:



Create a Title for a Plot

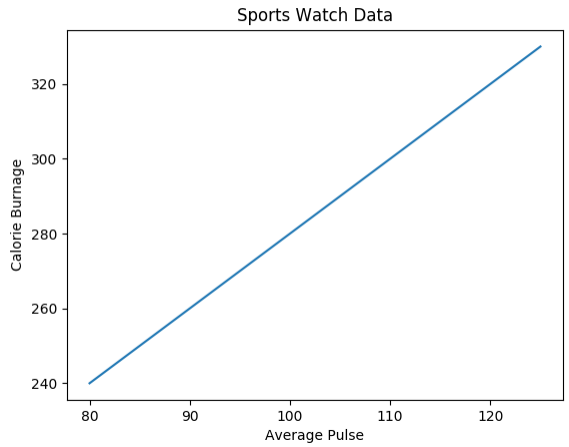
With Pyplot, you can use the title() function to set a title for the plot.

Example

Add a plot title and labels for the x- and y-axis:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```

Result:



Set Font Properties for Title and Labels

You can use the fontdict parameter in xlabel(), ylabel(), and title() to set font properties for the title and labels.

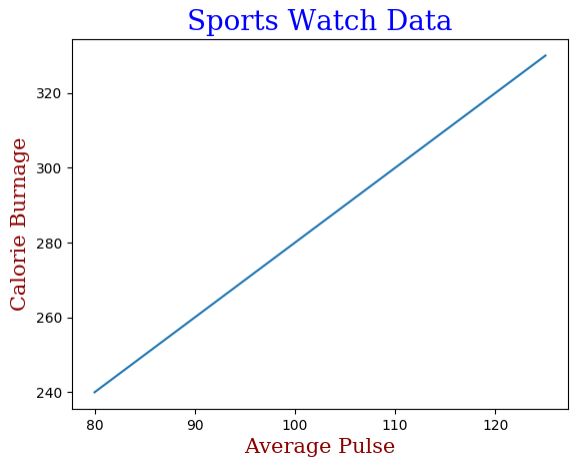
Example

Set font properties for the title and labels:

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

```
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

font1 = {'family':'serif','color':'blue','size':20}
font2 = {'family':'serif','color':'darkred','size':15}

plt.title("Sports Watch Data", fontdict = font1)
plt.xlabel("Average Pulse", fontdict = font2)
plt.ylabel("Calorie Burnage", fontdict = font2)

plt.plot(x, y)
plt.show()
```

Result:



Position the Title

You can use the loc parameter in title() to position the title.

Legal values are: 'left', 'right', and 'center'. Default value is 'center'.

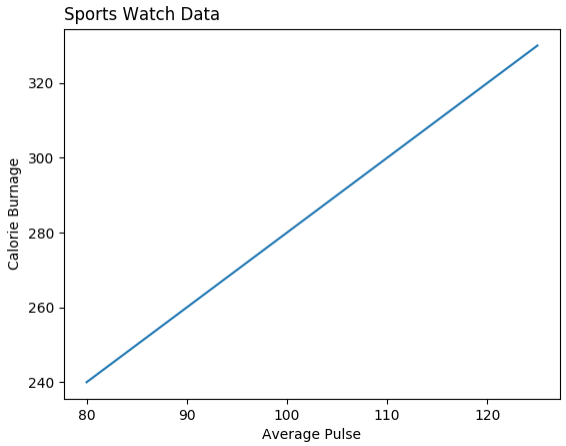
Example

Position the title to the left:

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

```
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data", loc = 'left')
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)
plt.show()
```

Result:



Matplotlib Adding Grid Lines

Add Grid Lines to a Plot

With Pyplot, you can use the grid() function to add grid lines to the plot.

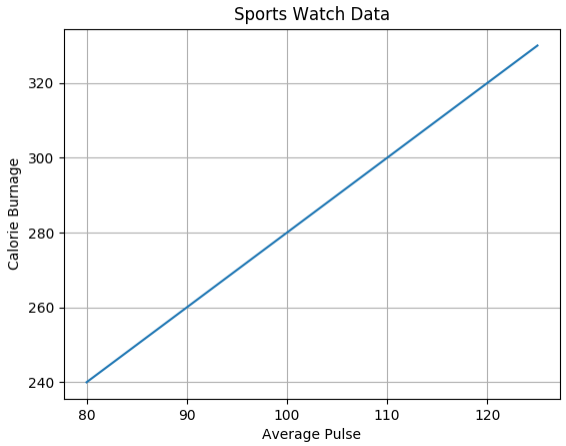
Example

Add grid lines to the plot:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])
```

```
plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)

plt.grid()

plt.show()
```

Result:



Specify Which Grid Lines to Display

You can use the axis parameter in the grid() function to specify which grid lines to display.

Legal values are: 'x', 'y', and 'both'. Default value is 'both'.

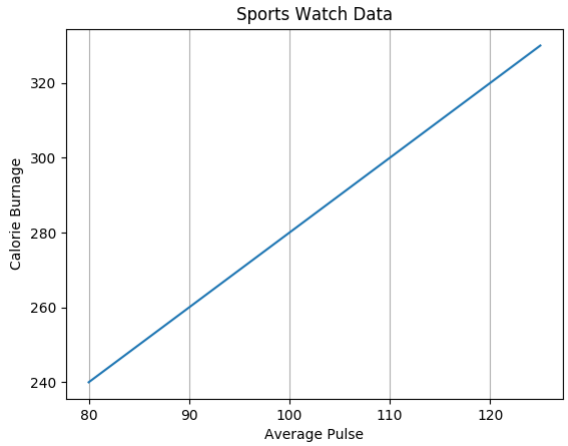
Example

Display only grid lines for the x-axis:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data")
```

```
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)

plt.grid(axis = 'x')

plt.show()
```

Result:



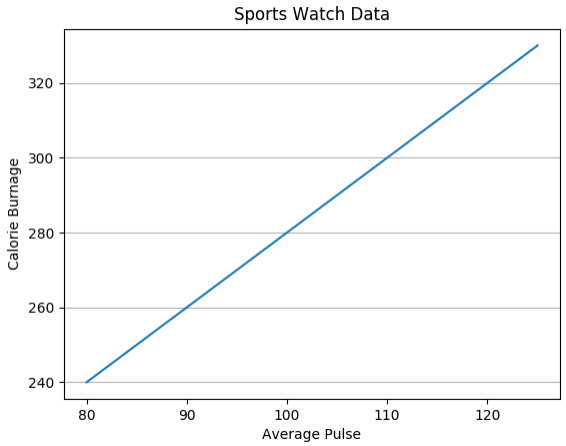
Example

Display only grid lines for the y-axis:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)

plt.grid(axis = 'y')
```

```
plt.show()
```

Result:



Set Line Properties for the Grid

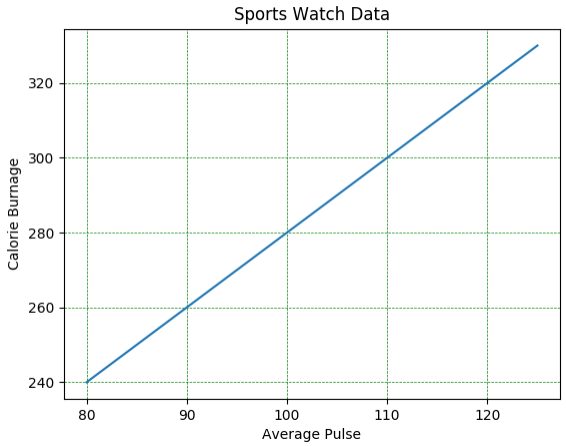
You can also set the line properties of the grid, like this: grid(color = '*color*', linestyle = '*linestyle*', linewidth = *number*).

Example

Set the line properties of the grid:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)

plt.grid(color = 'green', linestyle = '--', linewidth = 0.5)

plt.show()
```

Result:



Matplotlib Subplot

Display Multiple Plots

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

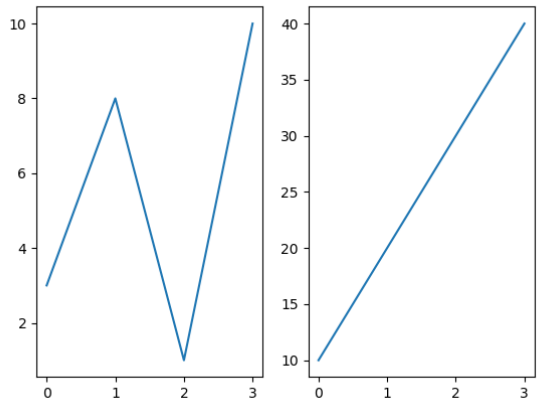
Example

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()
```

Result:



The subplot() Function

The subplot() function takes three arguments that describes the layout of the figure.

The layout is organized in rows and columns, which are represented by the *first* and *second* argument.

The third argument represents the index of the current plot.

```
plt.subplot(1, 2, 1)
#the figure has 1 row, 2 columns, and this plot is the first plot.
```

```
plt.subplot(1, 2, 2)
#the figure has 1 row, 2 columns, and this plot is the second plot.
```

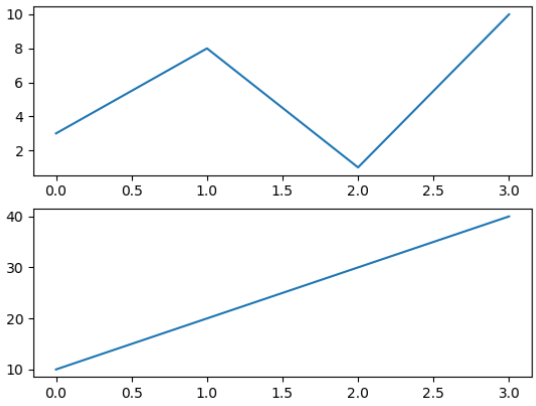
So, if we want a figure with 2 rows an 1 column (meaning that the two plots will be displayed on top of each other instead of side-by-side), we can write the syntax like this:

Example

Draw 2 plots on top of each other:

```
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(2, 1, 1)
plt.plot(x,y)

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

plt.subplot(2, 1, 2)
plt.plot(x,y)

plt.show()
```

Result:



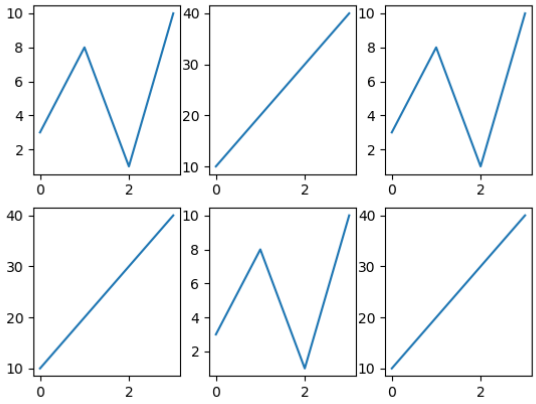
You can draw as many plots you like on one figure, just descibe the number of rows, columns, and the index of the plot.

Example

Draw 6 plots:

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

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 1)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 2)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 3)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 4)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 5)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 6)
plt.plot(x,y)

plt.show()
```

Result:



Title

You can add a title to each plot with the title() function:

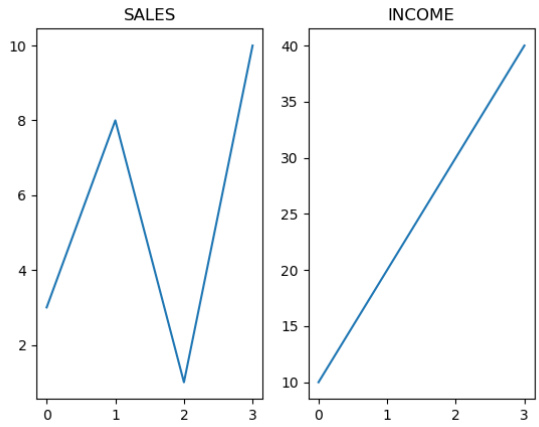
Example

2 plots, with titles:

```
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)
plt.title("SALES")

#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.title("INCOME")
```

```
plt.show()
```

Result:



Super Title

You can add a title to the entire figure with the suptitle() function:

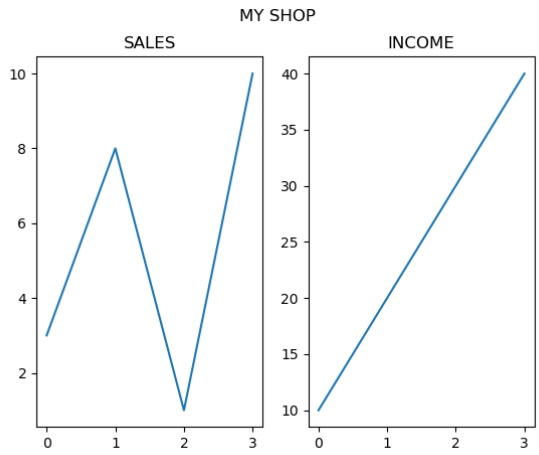
Example

Add a title for the entire figure:

```
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)
plt.title("SALES")

#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.title("INCOME")

plt.suptitle("MY SHOP")
plt.show()
```

Result:



Matplotlib Scatter

Creating Scatter Plots

With Pyplot, you can use the scatter() function to draw a scatter plot.

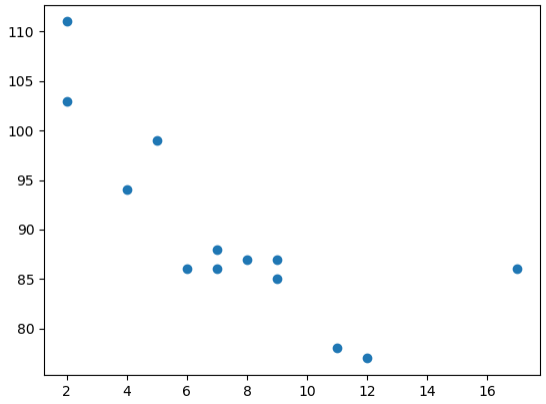
The scatter() function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis:

Example

A simple scatter plot:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])

plt.scatter(x, y)
plt.show()
```

Result:



The observation in the example above is the result of 13 cars passing by.

The X-axis shows how old the car is.

The Y-axis shows the speed of the car when it passes.

Are there any relationships between the observations?

It seems that the newer the car, the faster it drives, but that could be a coincidence, after all we only registered 13 cars.

Compare Plots

In the example above, there seems to be a relationship between speed and age, but what if we plot the observations from another day as well? Will the scatter plot tell us something else?

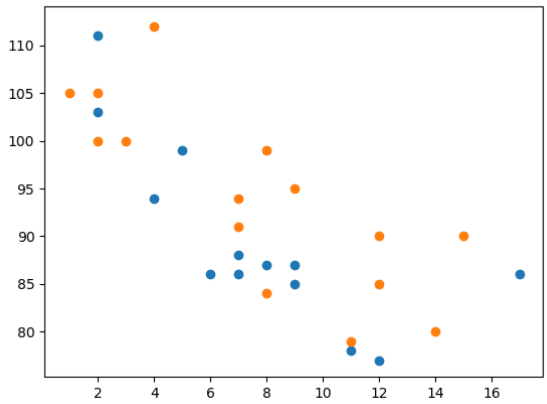
Example

Draw two plots on the same figure:

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

#day one, the age and speed of 13 cars:
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y)
```

```
#day two, the age and speed of 15 cars:
x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y)

plt.show()
```

Result:



Note: The two plots are plotted with two different colors, by default blue and orange, you will learn how to change colors later in this chapter.

By comparing the two plots, I think it is safe to say that they both gives us the same conclusion: the newer the car, the faster it drives.

Colors

You can set your own color for each scatter plot with the color or the c argument:

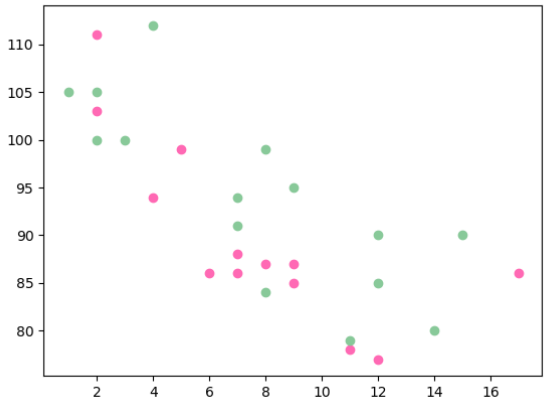
Example

Set your own color of the markers:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
```

```
plt.scatter(x, y, color = 'hotpink')

x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y, color = '#88c999')

plt.show()
```

Result:



Color Each Dot

You can even set a specific color for each dot by using an array of colors as value for the c argument:

Note: You *cannot* use the color argument for this, only the c argument.

Example

Set your own color of the markers:

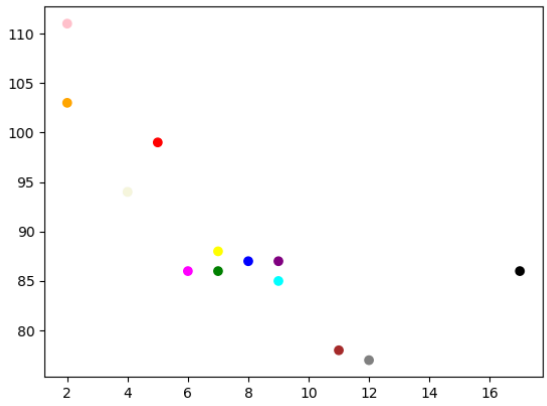
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
colors =
np.array(["red","green","blue","yellow","pink","black","orange","purple","beige","brown","gray","c
```

yan","magenta"])

plt.scatter(x, y, c=colors)

plt.show()

Result:



ColorMap

The Matplotlib module has a number of available colormaps.

A colormap is like a list of colors, where each color has a value that ranges from 0 to 100.

Here is an example of a colormap:

This colormap is called 'viridis' and as you can see it ranges from 0, which is a purple color, and up to 100, which is a yellow color.

How to Use the ColorMap

You can specify the colormap with the keyword argument cmap with the value of the colormap, in this case 'viridis' which is one of the built-in colormaps available in Matplotlib.

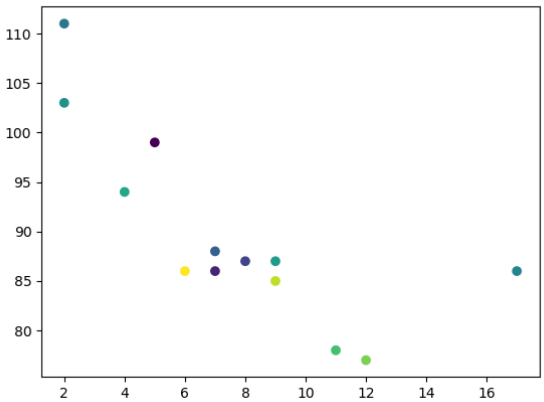
In addition you have to create an array with values (from 0 to 100), one value for each of the point in the scatter plot:

Example

Create a color array, and specify a colormap in the scatter plot:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
colors = np.array([0, 10, 20, 30, 40, 45, 50, 55, 60, 70, 80, 90, 100])

plt.scatter(x, y, c=colors, cmap='viridis')

plt.show()
```

Result:



Size

You can change the size of the dots with the s argument.

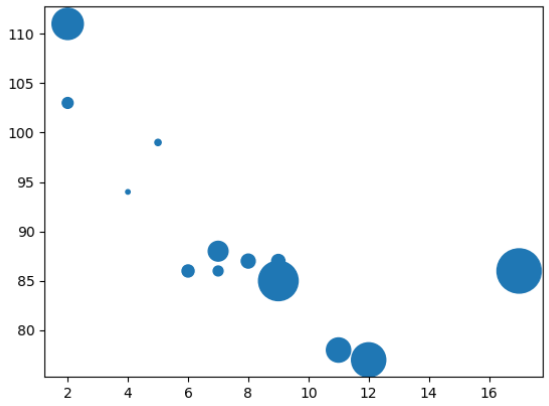
Just like colors, make sure the array for sizes has the same length as the arrays for the x- and y-axis:

Example

Set your own size for the markers:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
sizes = np.array([20,50,100,200,500,1000,60,90,10,300,600,800,75])

plt.scatter(x, y, s=sizes)

plt.show()
```

Result:



Alpha

You can adjust the transparency of the dots with the alpha argument.

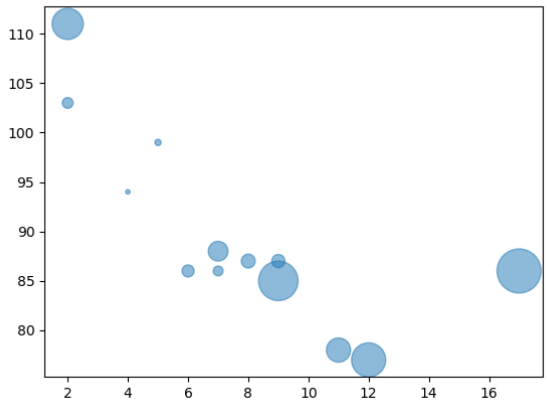
Just like colors, make sure the array for sizes has the same length as the arrays for the x- and y-axis:

Example

Set your own size for the markers:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
sizes = np.array([20,50,100,200,500,1000,60,90,10,300,600,800,75])

plt.scatter(x, y, s=sizes, alpha=0.5)

plt.show()
```

Result:



Combine Color Size and Alpha

You can combine a colormap with different sizes on the dots. This is best visualized if the dots are transparent:

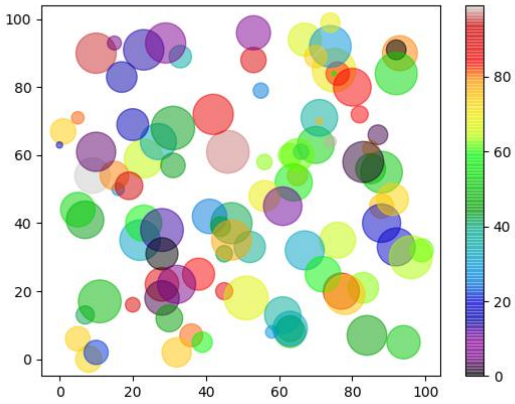
Example

Create random arrays with 100 values for x-points, y-points, colors and sizes:

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

```
x = np.random.randint(100, size=(100))
y = np.random.randint(100, size=(100))
colors = np.random.randint(100, size=(100))
sizes = 10 * np.random.randint(100, size=(100))

plt.scatter(x, y, c=colors, s=sizes, alpha=0.5, cmap='nipy_spectral')

plt.colorbar()

plt.show()
```

Result:



Matplotlib Bars

Creating Bars

With Pyplot, you can use the bar() function to draw bar graphs:

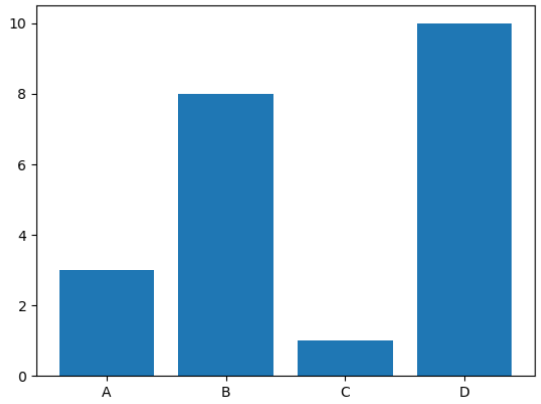
Example

Draw 4 bars:

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

x = np.array(["A", "B", "C", "D"])
```

```
y = np.array([3, 8, 1, 10])

plt.bar(x,y)
plt.show()
```

Result:



The bar() function takes arguments that describes the layout of the bars.

The categories and their values represented by the *first* and *second* argument as arrays.

Example

```
x = ["APPLES", "BANANAS"]
y = [400, 350]
plt.bar(x, y)
```

Horizontal Bars

If you want the bars to be displayed horizontally instead of vertically, use the barh() function:

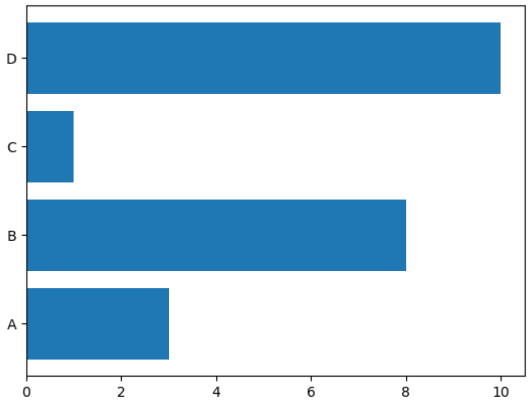
Example

Draw 4 horizontal bars:

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

x = np.array(["A", "B", "C", "D"])
```

```
y = np.array([3, 8, 1, 10])

plt.barh(x, y)
plt.show()
```

Result:



Bar Color

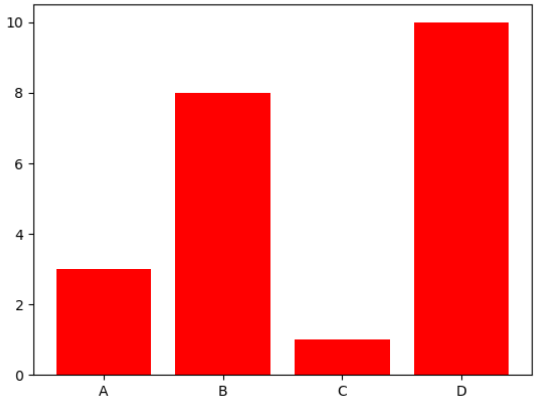
The bar() and barh() takes the keyword argument color to set the color of the bars:

Example

Draw 4 red bars:

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

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x, y, color = "red")
plt.show()
```

Result:



Color Names

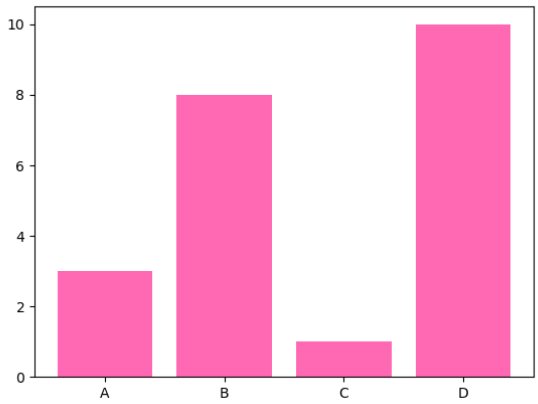
You can use any of the 140 supported color names.

Example

Draw 4 "hot pink" bars:

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

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])
plt.bar(x, y, color = "hotpink")
plt.show()
```

Result:



Color Hex

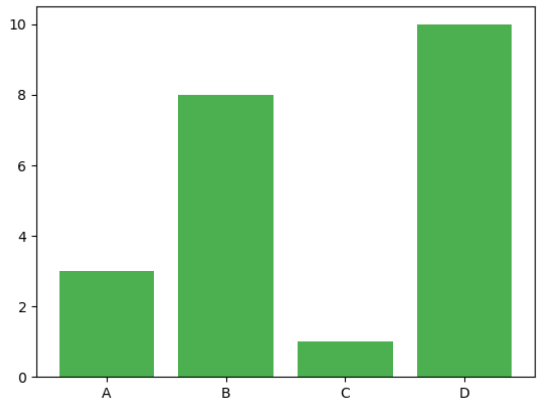
Or you can use Hexadecimal color values:

Example

Draw 4 bars with a beautiful green color:

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

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x, y, color = "#4CAF50")
plt.show()
```

Result:



Bar Width

The bar() takes the keyword argument width to set the width of the bars:

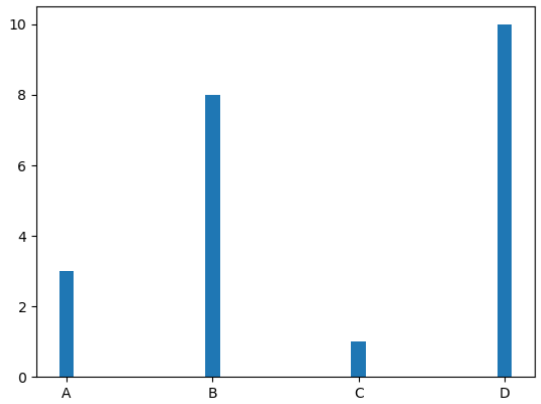
Example

Draw 4 very thin bars:

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

```
x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x, y, width = 0.1)
plt.show()
```

Result:



The default width value is 0.8

Note: For horizontal bars, use height instead of width.

Bar Height

The barh() takes the keyword argument height to set the height of the bars:

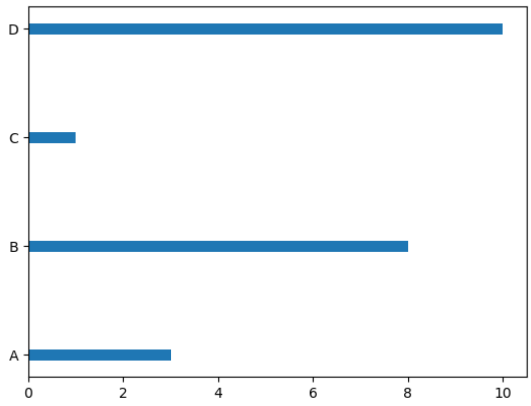
Example

Draw 4 very thin bars:

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

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])
```

```
plt.barh(x, y, height = 0.1)
plt.show()
```

Result:



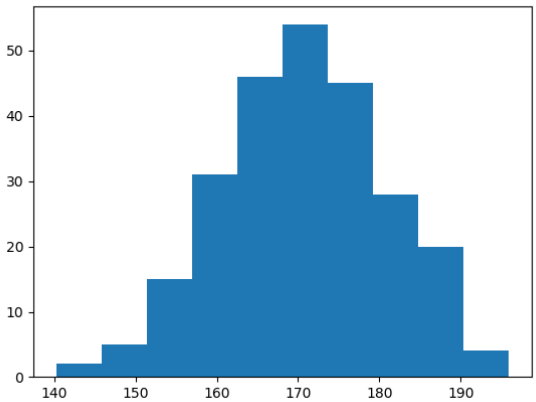
Matplotlib Histograms

Histogram

A histogram is a graph showing *frequency* distributions.

It is a graph showing the number of observations within each given interval.

Example: Say you ask for the height of 250 people, you might end up with a histogram like this:



You can read from the histogram that there are approximately:

2 people from 140 to 145cm
5 people from 145 to 150cm
15 people from 151 to 156cm
31 people from 157 to 162cm
46 people from 163 to 168cm
53 people from 168 to 173cm
45 people from 173 to 178cm
28 people from 179 to 184cm
21 people from 185 to 190cm
4 people from 190 to 195cm

Create Histogram

In Matplotlib, we use the hist() function to create histograms.

The hist() function will use an array of numbers to create a histogram, the array is sent into the function as an argument.

For simplicity we use NumPy to randomly generate an array with 250 values, where the values will concentrate around 170, and the standard deviation is 10. Learn more about Normal Data Distribution in our Machine Learning Tutorial.

Example

A Normal Data Distribution by NumPy:

```
import numpy as np

x = np.random.normal(170, 10, 250)

print(x)
```

Result:

This will generate a *random* result, and could look like this:

```
[167.62255766 175.32495609 152.84661337 165.50264047 163.17457988
 162.29867872 172.83638413 168.67303667 164.57361342 180.81120541
 170.57782187 167.53075749 176.15356275 176.95378312 158.4125473
 187.8842668  159.03730075 166.69284332 160.73882029 152.22378865
 164.01255164 163.95288674 176.58146832 173.19849526 169.40206527
```

```
166.88861903 149.90348576 148.39039643 177.90349066 166.72462233
177.44776004 170.93335636 173.26312881 174.76534435 162.28791953
166.77301551 160.53785202 170.67972019 159.11594186 165.36992993
178.38979253 171.52158489 173.32636678 159.63894401 151.95735707
175.71274153 165.00458544 164.80607211 177.50988211 149.28106703
179.43586267 181.98365273 170.98196794 179.1093176  176.91855744
168.32092784 162.33939782 165.18364866 160.52300507 174.14316386
163.01947601 172.01767945 173.33491959 169.75842718 198.04834503
192.82490521 164.54557943 206.36247244 165.47748898 195.26377975
164.37569092 156.15175531 162.15564208 179.34100362 167.22138242
147.23667125 162.86940215 167.84986671 172.99302505 166.77279814
196.6137667  159.79012341 166.5840824  170.68645637 165.62204521
174.5559345  165.0079216  187.92545129 166.86186393 179.78383824
161.0973573  167.44890343 157.38075812 151.35412246 171.3107829
162.57149341 182.49985133 163.24700057 168.72639903 169.05309467
167.19232875 161.06405208 176.87667712 165.48750185 179.68799986
158.7913483  170.22465411 182.66432721 173.5675715  176.85646836
157.31299754 174.88959677 183.78323508 174.36814558 182.55474697
180.03359793 180.53094948 161.09560099 172.29179934 161.22665588
171.88382477 159.04626132 169.43886536 163.75793589 157.73710983
174.68921523 176.19843414 167.39315397 181.17128255 174.2674597
186.05053154 177.06516302 171.78523683 166.14875436 163.31607668
174.01429569 194.98819875 169.75129209 164.25748789 180.25773528
170.44784934 157.81966006 171.33315907 174.71390637 160.55423274
163.92896899 177.29159542 168.30674234 165.42853878 176.46256226
162.61719142 166.60810831 165.83648812 184.83238352 188.99833856
161.3054697  175.30396693 175.28109026 171.54765201 162.08762813
164.53011089 189.86213299 170.83784593 163.25869004 198.68079225
166.95154328 152.03381334 152.25444225 149.75522816 161.79200594
162.13535052 183.37298831 165.40405341 155.59224806 172.68678385
179.35359654 174.19668349 163.46176882 168.26621173 162.97527574
192.80170974 151.29673582 178.65251432 163.17266558 165.11172588
183.11107905 169.69556831 166.35149789 178.74419135 166.28562032
169.96465166 178.24368042 175.3035525  170.16496554 158.80682882
187.10006553 178.90542991 171.65790645 183.19289193 168.17446717
155.84544031 177.96091745 186.28887898 187.89867406 163.26716924
169.71242393 152.9410412  158.68101969 171.12655559 178.1482624
187.45272185 173.02872935 163.8047623  169.95676819 179.36887054
157.01955088 185.58143864 170.19037101 157.221245   168.90639755
178.7045601  168.64074373 172.37416382 165.61890535 163.40873027
168.98683006 149.48186389 172.20815568 172.82947206 173.71584064
189.42642762 172.79575803 177.00005573 169.24498561 171.55576698
161.36400372 176.47928342 163.02642822 165.09656415 186.70951892
153.27990317 165.59289527 180.34566865 189.19506385 183.10723435
173.48070474 170.28701875 157.24642079 157.9096498  176.4248199 ]
```

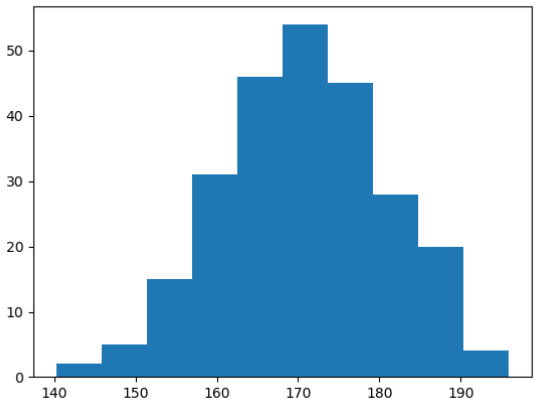
The hist() function will read the array and produce a histogram:

Example

A simple histogram:

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

x = np.random.normal(170, 10, 250)

plt.hist(x)
plt.show()
```

Result:



Matplotlib Pie Charts

Creating Pie Charts

With Pyplot, you can use the pie() function to draw pie charts:

Example

A simple pie chart:

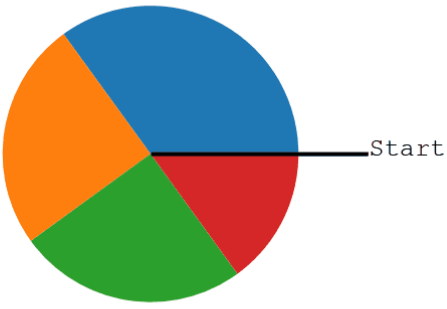
```
import matplotlib.pyplot as plt
import numpy as np
```

```
y = np.array([35, 25, 25, 15])

plt.pie(y)
plt.show()
```

Result:

As you can see the pie chart draws one piece (called a wedge) for each value in the array (in this case [35, 25, 25, 15]).

By default the plotting of the first wedge starts from the x-axis and move *counterclockwise*:



Note: The size of each wedge is determined by comparing the value with all the other values, by using this formula:

The value divided by the sum of all values: x/sum(x)

Labels

Add labels to the pie chart with the label parameter.

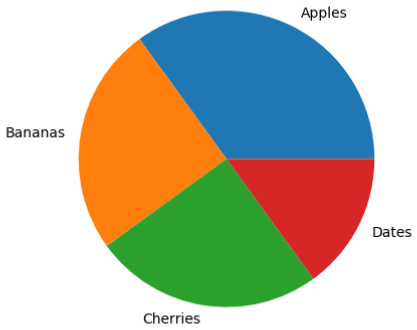
The label parameter must be an array with one label for each wedge:

Example

A simple pie chart:

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

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.show()
```

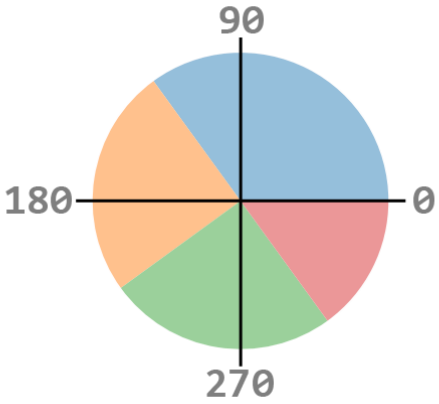
Result:



Start Angle

As mentioned the default start angle is at the x-axis, but you can change the start angle by specifying a startangle parameter.

The startangle parameter is defined with an angle in degrees, default angle is 0:

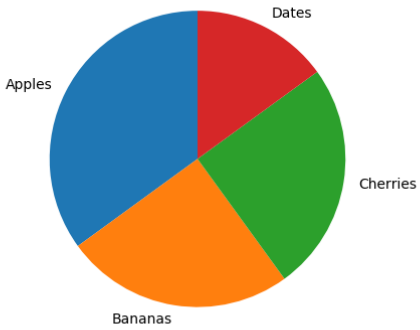


Example

Start the first wedge at 90 degrees:

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

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels, startangle = 90)
plt.show()
```

Result:



Explode

Maybe you want one of the wedges to stand out? The explode parameter allows you to do that.

The explode parameter, if specified, and not None, must be an array with one value for each wedge.

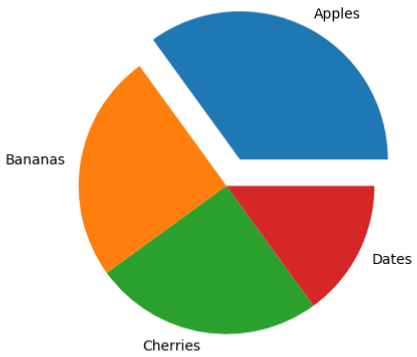
Each value represents how far from the center each wedge is displayed:

Example

Pull the "Apples" wedge 0.2 from the center of the pie:

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

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
myexplode = [0.2, 0, 0, 0]

plt.pie(y, labels = mylabels, explode = myexplode)
plt.show()
```

Result:



Shadow

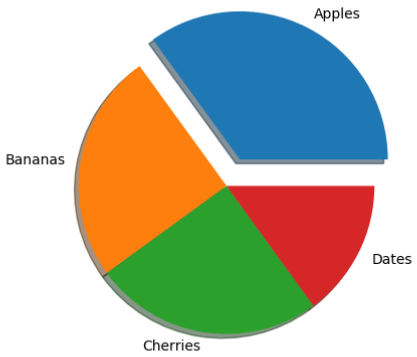
Add a shadow to the pie chart by setting the shadows parameter to True:

Example

Add a shadow:

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

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
myexplode = [0.2, 0, 0, 0]

plt.pie(y, labels = mylabels, explode = myexplode, shadow = True)
plt.show()
```

Result:



Colors

You can set the color of each wedge with the colors parameter.

The colors parameter, if specified, must be an array with one value for each wedge:

Example

Specify a new color for each wedge:

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

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
mycolors = ["black", "hotpink", "b", "#4CAF50"]

plt.pie(y, labels = mylabels, colors = mycolors)
plt.show()
```

Result

You can use Hexadecimal color values, any of the 140 supported color names, or one of these shortcuts:

'r' - Red
'g' - Green
'b' - Blue
'c' - Cyan
'm' - Magenta
'y' - Yellow
'k' - Black
'w' - White

Legend

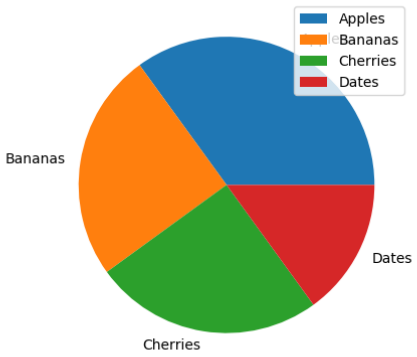
To add a list of explanation for each wedge, use the legend() function:

Example

Add a legend:

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

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.legend()
plt.show()
```

Result:



Legend With Header

To add a header to the legend, add the title parameter to the legend function.

Example

Add a legend with a header:

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

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.legend(title = "Four Fruits:")
plt.show()
```

Result: