Una base de datos o conjunto de datos son un conjunto de datos con relación entre ellos los cuales se almacenan en un mismo fichero o en ficheros que tienen relación entre ellos con la finalidad de usarlos en el futuro. Estos datos se suelen gestionar usando **SGBD** (Sistemas Gestores de Bases de Datos). Los SGBD tienen la peculiaridad de que implementan sus propios lenguajes de programación como por ejemplo SQL.

Modelos de Bases de Datos:

* Jerárquicas
* De red
* Transaccionales
* **Relacionales**
* Documentales
* Orientadas a objetos
* Deductivas

Las bases de datos relacionales se basan en la idea de crear relaciones entre conjuntos de datos. Cada relación es también una tabla que esta compuesta por registros (las filas o tuplas) y por sus campos (las columnas).

Dentro de este tipo de modelo de base de dato, existen varios sistemas gestores de datos, de los cuales los compatibles con **python** son los siguientes:

* SQL Server (Microsoft)
* Oracle (Oracle Corporation, multiplataforma)
* MySQL (Open source)
* PostgreSQL (Open source, libre, orientada a objetos)
* **SQLite** (relacional, escrito en **C**, hasta 2 TB de info)

SQL: Structured Query Language, Es un **lenguaje** de programación por si mismo, que tenemos que aprender si lo queremos usar.

Conexión, Puntero, Consultas básicas:

Lo primer que tenemos que hacer para usar sqlite3 es importarlo, y luego crear una “conexion” a la base de datos, si la base de datos no esta creada esta es creada en ese momento, este proceso es igual al de abrir un archivo con el metodo **open**, pero en este case es un metodo del módulo sqlite3:

import **sqlite3**

conexión = sqlite3.**connect**(“ejemplo.db”)

conexión.**close**()

Aquí vemos que la sintaxis es básicamente la misma que el metodo open, la única diferencia es que en vez de pasar solamente el metodo conexión, este se pasa con **dot notation**, esto tiene que ver con la manera de la que estamos importando el modulo sqlite3. También vemos que debajo estamos cerrando la conexión, esto es porque tiene la misma sintaxis que el metodo open, y por tanto, hay que cerrar el archivo una vez terminemos de trabajar con él.

La magia de sqlite3 ocurre entre el momento en el que abrimos la conexión y la cerramos, por su puesto, una vez estemos entre estas líneas podemos escribir en lenguaje SQL pero para hacer esto primero tenemos que crear un objeto de tipo **cursor**, recordemos que la función del cursos es determinar la posición en el archivo en la que estamos actualmente para poder leer o modificar el fichero.

El cursor se crea dentro de conexión, con el metodo cursor, de la siguiente manera:

import sqlite3

conexión = sqlite3.connect(“ejemplo.db”)

cursor = conexion.cursor()

cursor.execute(“CREATE TABLE usuarios (nombre VARCHAR(100), edad INTEGER, email VARCHAR (100))”)

conexión.close()

Aquí podemos ver que debajo del cursor hay una línea de comando en la que al objeto cursor se le aplica un metodo llamado **execute**, este metodo es el que ejecuta el comando de tipo SQL dentro del intérprete de python, básicamente es lo que le dice al interprete que nos estamos refiriendo a un código escrito en lenguaje SQL.

En lenguaje SQL, esta sintaxis se traduce a **crear una tabla** con el nombre **usuarios** que tiene los campos que están contenidos en el paréntesis llamados **nombre**, **edad** y **email**. Luego nos percatamos de que luego de declarar dichos campos debemos de declarar su tipo de variable siendo **VARCHAR** el equivalente al string e **INTEGER** el equivalente a int. Además de esto podemos ver que cada una de estas variables posee sus propios metodos ya que se observa que luego del nombre hay un valor del tipo int, restringiendo la cantidad de caracteres posibles en el campo a **100**. Es importante notar que **SIEMPRE** que vallamos a ejecutar un comando en lenguaje SQL, este debe de estar dentro de **comillas**, y dentro del metodo **execute** del objeto de la clase **cursor**.

Aquí hay que recordar algo muy importante, y es que estamos creando una tabla con campos, si ejecutamos este script una vez, tendremos éxito, si lo ejecutamos por segunda vez, nos levantará un **error** poque la tabla o los campos que estamos intentando crear ya existen.

Una vez que tenemos instalado el SQLite podemos perfectamente abrir las bases de datos dentro del programa e inspeccionar cuantas **tablas**, **índices**, **vistas** y **disparadores** existen

Realmente es mucho mas fácil si los introducimos en el programa SQLite, pero esto solamente es eficiente si vamos a insertar pocos valores, si vamos a insertar muchos, o se vamos a hacer un proceso automatizados, es necesario saber escribirlo en un script.

Y a sabemos como crear una tabla y como crear los campos existentes en esta(recordemos que las cabeceras son simplemente elementos del primer row, y los valores los elementos de los rows siguientes), ahora vamos a llenar los estos campos con valores de la manera siguiente:

import sqlite3

conexión = sqlite3.connect(“ejemplo.db”) cursor = conexion.cursor()

cursor.execute(“INSTERT INTO usuarios VALUES ( ‘Hector’, 27, ‘hector@ejemplo.com’)”)

conexion.commit() conexión.close()

Aqui vemos que debajo de el **execute** hemos corrido un metodo **commit**, si, como es de imaginar, si no ejecutamos commit, aunque los cambios estén hechos, estos no van a ser enviados a la base de datos, y por tanto, aunque refresquemos la interfaz grafica del programa SQLite, no veremos ningún cambio por que los cambios no han sido enviados. Este proceso es el mismo que ocurre a la hora de hacer o modificar con Git en GitHub a un repositorio.

OJO. Si este script es ejecutado **varias veces**, simplemente se agregará el mismo campo tantas veces como se ejecute el script ya que no existe ningún control que no permita que existan campos idénticos.

De esta manera queda cubierto como escribir en una base de datos, ahora vamos a ver como podemos **recuperar** los registros que existen en esta.

En este caso, lo que estamos haciendo es que el objeto del tipo Cursor del módulo sqlite3 esta almacenando todos los datos de todos los campos de esta tabla, pero no los podemos obtener simplemente con un print ya que lo que obtendremos será la típica línea de la ubicación del objeto: <sqlite3.Cursor object at 0x0000019F8A6DC810>.

Luego de que nuestro objeto cursor haya sido llenado con toda la información de toda la tabla, podemos asignar una variable, o un objeto el cual será el resultado de aplicar el metodo **fetch o fetchone** del objeto cursor, al objeto cursor como se muestra en la línea 5. Una vez que hemos hecho esto, si imprimimos el contenido de usuario por pantalla, obtendremos la tupla que contiene el row:

Terminal:

<sqlite3.Cursor object at 0x0000019F8A6DC810> (‘Hector’, 27, ‘hector@ejemplo.com’) [Finished in 0.1s]

Cabe destacar que el metodo **fetchone()** tiene el mismo modo de operar que el metodo **readline()**, si se ejecuta por segunda vez, devolverá el segundo row, y así sucesivamente.

La manera de insertar muchos rows es pasándole por ejemplo una lista, con tuplas como elementos, porque SQL lee tuplas y luego las convierte en rows, que se convierten en filas, pero para esto tenemos que usar un metodo diferente al execute del objeto cursor, llamado **executemany**:

import sqlite3

conexión = sqlite3.connect(“ejemplo.db”) cursor = conexion.cursor()

usuarios = [ (‘Mario’,51,’mario@ejemplo.com’) , (‘Mario’,51,’mario@ejemplo.com’) , (‘Mario’,51,’mario@ejemplo.com’),

]

cursor.executemany(“INSTERT INTO usuarios VALUES ( ?,?,?)”), usuarios)

conexion.commit() conexión.close()

Si en lugar de querer recuperar uno a uno los nombres de los usuarios, queremos recuperarlos todos, debemos en vez de usar fetchone, el metodo **fetchall()** del objeto cursor. Esto en vez de devolver solamente una tupla, de la cual ya podíamos acceder a los campos de esta haciendo uso de los índices, va a devolver una lista anidada de tuplas, a la que podríamos acceder de diferentes formas, a sus diferentes campos.

Terminal:

[(‘Hector’, 27, ‘hector@ejemplo.com’), (‘Mario’,51,’mario@ejemplo.com’), (‘Mario’,51,’mario@ejemplo.com’), (‘Mario’,51,’mario@ejemplo.com’),] [Finished in 0.1s]

Claves Primarias, Campos Autoincrementables y Claves Únicas:

Las claves primarias o **PRIMARY KEY** son valores que no se pueden repetir dentro de la tabla, por ejemplo, los DNI que no pueden ser iguales, lo que se hace para hacer que una clave(elementos del primer row) sea primaria es simplemente detrás del tipo de variable, escribir literalmente **PRIMARY KEY**:

cursor.execute(“CREATE TABLE usuarios (DNI, VARCHAR (9) PRIMARY KEY”)

Los campos Autoincrementables con campos que se incrementan cuando se crean, por ejemplo, queremos hace una tabla con productos de los cuales su número de serie, que es único se genere automáticamente e incrementándose y que este sea un valor de uno de los campos de los rows que llamaremos id. Para hacer que el campo sea tanto PARIMARY KEY como AUTOINCREMENT, bueno, simplemente escribimos eso de la siguiente forma:

cursor.execute(“””

CREATE TABLE products (id, INTEGER PRIMARY KEY AUTOINCREMENT,)

“””)

Cuando agregamos un campo Autoincrementables a la tabla y vamos a introducirlos masivamente, lo que se hace es que en ese campo se introduce un **null** en vez de un ‘**?’**

cursor. executemany(“INSTERT INTO usuarios VALUES ( null ,?,?)”), usuarios)

Esto solamente nos sirve para que uno de los calores no se pueda repetir, si queremos que mas valores se reptan, tenemos que hacer uso de la cláusula **UNIQUE**.

CREATE TABLE products (DNI, INTEGER UNIQUE,)

El uso del Where:

Si queremos recuperar de la base de datos un elemento en específico, o elementos que coincidan con criterios de búsqueda específicos debemos usar el **WHERE** cuando estemos haciendo **SELECT** y luego para mostrar todos los resultados, se usa **fetchall()** en el objeto o variable contenedora de dicha información:

cursor.execute(“SELECT \* FROM usuarios WHERE edad = 27”) usuarios = cursor.fetchall() print (usuarios)

Terminal:

[(‘Hector’, 27, ‘hector@ejemplo.com’), (‘Mario’,27,’mario@ejemplo.com’)] [Finished in 0.1s]

Modificar y Borrar:

Aqui no hay mucho más que decir que la sintaxis para modificar (**UDPADET & SET**) y la sintaxis para borrar (**DELETE & FROM**):

UPDATE:

cursor.execute(“UPDATE usuarios SET nombre = ‘Héctor Costa’ WHERE DNI = ‘#########’”) conexión.commit()

DELETE:

cursor.execute(“DELETE FROM usuarios WHERE DNI = ‘#########’”) conexión.commit()

**OJO:**  Es bien importante a la hora de usar estos dos metodos que estemos usando el WHERE porque de no hacerlo, modificaríamos toda la tabla al mismo valor, o borraríamos toda la tabla en caso del metodo DELETE.