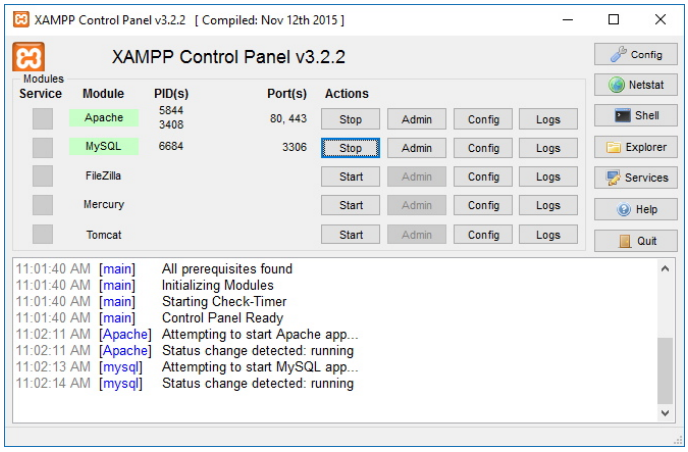
Acceso a bases de datos

En este documento daremos unas nociones básicas de cómo acceder a diferentes sistemas gestores de bases de datos desde Python. En concreto nos centraremos en un ejemplo de base de datos relacional como MySQL, y en otro de base de datos No-SQL, como MongoDB. Como ocurre en muchos otros casos, necesitaremos instalar la librería adecuada para poder trabajar con la base de datos en cuestión, y después utilizar los métodos correspondientes para conectar, insertar, listar, etc.

1. Acceso a bases de datos MySQL

1.1. Instalación del servidor MySQL

Para poder trabajar con bases de datos MySQL, el primer paso será tener instalado un servidor MySQL. Podemos instalar uno de forma manual, y luego configurar los usuarios, bases de datos y tablas desde el terminal, o bien podemos disponer de un entorno algo más visual para poder gestionar el servidor y la base de datos de forma más cómoda. En nuestro caso, ya que no se trata de un curso para administrar servidores de bases de datos, optaremos por esta segunda opción a través de la herramienta XAMPP. Al instalarla dispondremos de un servidor web Apache, y un servidor MySQL. Podremos poner ambos en marcha desde la herramienta *XAMPP Manager* que viene incorporada, y poner en marcha ambos servidores.



La ventaja que tiene utilizar esta herramienta es que también disponemos de una aplicación web llamada *phpMyAdmin*, que se pone en marcha sobre Apache y nos permite gestionar las bases de datos que tengamos instaladas: crearlas, definir las tablas, hacer copia de seguridad, etc.

1.2. Instalación de las librerías de Python

Para poder acceder a MySQL desde Python necesitamos utilizar alguna librería externa que nos facilite los mecanismos de conexión y gestión de esa base de datos. Una de las más utilizadas es `mysql-connector-python`, que instalamos con el correspondiente comando `pip`:

```
pip install mysql-connector-python
```

Después importaremos el elemento `connector` en nuestro programa:

```
import mysql.connector
```

1.3. Operaciones con la base de datos

Veremos a continuación un ejemplo sencillo de cada una de las operaciones habituales con MySQL.

1.3.1. Conexión a una base de datos

La primera operación que tendremos que realizar será conectar con la base de datos que queramos. Supondremos que ya la tendremos previamente creada, a través de *phpMyAdmin* alguna otra herramienta. En cualquier caso, la propia librería *mysql.connector* dispone de mecanismos para crear bases de datos y tablas, pero no los veremos aquí.

La instrucción para conectar es como sigue:

```
import mysql.connector

bd = mysql.connector.connect(
    host="localhost",
    user="usuario",
    password="password",
    database="miBD"
)
```

> **NOTA**: en la instrucción anterior deberemos reemplazar los valores de los parámetros por los reales. Lo normal es que conectemos desde el mismo ordenador donde está la base de datos, con lo que el *host* suele ser *localhost*. Pero el usuario y contraseña pueden cambiar. La instalación por defecto de XAMPP deja un usuario *root* con contraseña vacía. Finalmente, el parámetro *database* apuntará al nombre de la base de datos que queramos utilizar.

1.3.2. Operaciones de inserción, borrado y actualización

Para realizar operaciones INSERT, DELETE o UPDATE, a partir del objeto obtenido al conectar, crearemos un *cursor* que nos permita acceder a la base de datos, y usaremos su instrucción `execute` para ejecutar la instrucción correspondiente. Por ejemplo, si tenemos una tabla *personas* con unos campos *nombre* y *edad*, podemos insertar una nueva persona así:

```
# ... Código anterior para conectar

# Obtenemos el cursor
cursor = bd.cursor()

# Planteamos la instrucción SQL
sql = "INSERT INTO personas (nombre, edad) VALUES(%s, %s)"
valores = ("Nacho", 45)
cursor.execute(sql, valores)
bd.commit()
```

Notar que definimos unos comodines `%s` en la instrucción SQL para luego reemplazarlos por los valores que queramos. La propia librería ya se encarga de reemplazar y guardar cada dato con su tipo adecuado (en este caso, un texto y un entero).

Del mismo modo podemos insertar múltiples datos, especificando un vector de tuplas:

```
sql = "INSERT INTO personas (nombre, edad) VALUES(%s, %s)"
valores = [
    ("Nacho", 45),
    ("Ana", 41),
    ("Juan", 70)
]
cursor.execute(sql, valores)
bd.commit()
```

En inserciones que generen un *id* autonumérico podemos recuperarlo tras la inserción, a través del cursor. También podemos obtener otras propiedades, como el número de filas insertadas:

```
...
bd.commit()
filas_insertadas = cursor.rowcount
ultimo_id = cursor.lastrowid
```

También de forma similar podemos lanzar borrados o actualizaciones, y obtener el número de filas afectadas:

```
sql = "DELETE FROM personas WHERE nombre = %s"
valores = ("Jaime", )
cursor.execute(sql, valores)
bd.commit()
print("Se han borrado", cursor.rowcount, "elementos")

sql = "UPDATE personas SET edad=50 WHERE nombre=%s"
valores = ("Pepe", )
cursor.execute(sql, valores)
bd.commit()
print("Se han actualizado", cursor.rowcount, "elementos")
```

1.3.3. Consultas

Para lanzar consultas (SELECT) construimos la instrucción SQL de forma similar, pero usamos el método `fetchall` para obtener los resultados, en lugar de `commit`. Por ejemplo, así obtenemos las personas mayores que la edad que diga el usuario previamente:

```
edad_usuario = # ... Le pedimos la edad al usuario

sql = "SELECT nombre, edad FROM personas WHERE edad > %s"
valores = (edad_usuario,)

# Ejecutar la consulta
cursor.execute(sql, valores)

# Obtener los resultados
resultados = cursor.fetchall()

# Imprimir los resultados
for r in resultados:
    print("Nombre:", r[0])
    print("Edad:", r[1])
```

Adicionalmente, disponemos del método `fetchone` en lugar de `fetchall` si sólo queremos acceder al primer elemento que cumpla la selección.

1.3.4. Cerrar la conexión

Tras finalizar las operaciones necesarias en la base de datos, conviene cerrar tanto el cursor como la conexión a la base de datos:

```
cursor.close()
bd.close()
```

> **Ejercicio 1:**
>
> Descarga este backup de una base de datos MySQL llamada *contactos* e impórtalo en tu servidor. Creará una base de datos *contactos* con una tabla *contactos* con una serie de registros insertados. Crea un programa `ContactosMySQL.py` y prueba a hacer una consulta que muestre los contactos cuyo nombre contenga el texto indicado por el usuario.
>
> **AYUDA:** vídeo con la solución del ejercicio

2. Acceso a bases de datos MongoDB

2.1. Instalación del servidor MongoDB

Como ocurría en el caso anterior, si queremos trabajar con bases de datos MongoDB necesitamos disponer de un servidor instalado. Nuevamente, al no ser éste el propósito principal del curso, aquí os dejamos algunas nociones de cómo instalarlo y qué herramienta(s) utilizar para gestionarlo.

- Instalación de MongoDB
- Herramientas para gestionar bases de datos MongoDB

2.2. Instalación de las librerías de Python

La librería que usaremos en este caso para acceder a Mongo será `pymongo`, que podemos instalar con el correspondiente comando:

```
pip install pymongo
```

La incluiremos en nuestro código fuente de este modo:

```
import pymongo
```

2.3. Operaciones con la base de datos

Veremos a continuación un ejemplo sencillo de cada una de las operaciones habituales con MongoDB.

2.3.1. Conexión y creación de una base de datos

Para conectar a un servidor Mongo usamos esta expresión:

```
import pymongo

cliente = pymongo.MongoClient("mongodb://localhost:27017")
bd = cliente["nombreBD"]
```

La primera instrucción conecta con el servidor indicado (en nuestro caso, *localhost*), por el puerto por defecto en que suele escuchar Mongo (27017). La segunda instrucción crea una base de datos *nombreBD* si no existe y, en caso contrario, accede a ella. Realmente la base de datos no va a existir hasta que añadamos contenido.

2.3.3. Crear una colección y añadir documentos.

Las bases de datos en MongoDB se componen de colecciones, del mismo modo que una base de datos en MySQL se compone de tablas. Para crear una colección simplemente la nombramos desde la base de datos.

```
...
bd = cliente["nombreBD"]
# Accedemos a la coleccion "personas" de la BD
coleccion = bd["personas"]
```

Nuevamente, la colección no se creará hasta que añadamos contenido en ella (documentos). Para **insertar documentos** en una colección debemos definir los campos que tendrán los documentos, en forma de mapa. Después podemos utilizar el método `insert_one` de la colección para insertar el documento.

```
persona = {'nombre': 'Nacho', 'edad': 45}
coleccion.insert_one(persona)
```

En el caso de bases de datos Mongo, a cada documento se le asigna un identificador alfanumérico automático en un campo interno llamado `_id`. Para poderlo recuperar debemos guardarnos en una variable el objeto insertado, para luego acceder a esa información con la propiedad `inserted_id`:

```
persona = {'nombre': 'Nacho', 'edad': 45}
nueva_persona = coleccion.insert_one(persona)
print("Persona insertada con id", nueva_persona.inserted_id)
```

Si queremos añadir múltiples documentos, los añadimos a un vector y utilizamos el método `insert_many`:

```
personas = [
    {'nombre': 'Nacho', 'edad': 45},
    {'nombre': 'Ana', 'edad': 41},
    {'nombre': 'Juan', 'edad': 70}
]
datos = coleccion.insert_many(personas)
print("Ids insertados:", datos.inserted_ids)
```

2.3.4. Búsquedas

Para buscar documentos en una colección empleamos los métodos `find_one` o `find`, dependiendo de si queremos encontrar uno o varios.

```
resultado1 = coleccion.find_one({'nombre': 'Nacho'})
print(resultado1)

resultado2 = coleccion.find({'nombre': 'Nacho'})
for r in resultado2:
    print(r)
```

También podemos acceder a un dato concreto de cada documento como si fuera un diccionario. Por ejemplo:

```
...
for r in resultado2:
    print(r['nombre'])
```

Como podemos ver, se pueden especificar criterios de filtrado o búsqueda como primer parámetro de los métodos anteriores. También se pueden especificar criterios de búsqueda más complejos, como se puede consultar en la documentación de pymongo.

2.3.5. Actualizaciones y borrados

Para eliminar documentos de una colección emplearemos los métodos `delete_one` o `delete_many`, dependiendo de la cantidad:

```
coleccion.delete_one({'nombre': 'Nacho'})
coleccion.delete_many({'edad': 30})
```

> **NOTA**: si no especificamos criterios de filtrado en `delete_many` se borrarán TODOS los documentos de la colección.

De forma análoga, podemos emplear los métodos `update_one` y `update_many` para actualizar los campos de documentos:

```
coleccion.update_one({'_id': 1}, { '$set': {'nombre': 'Nombre modificado', 'edad': 3
```

En este caso, los métodos reciben dos parámetros: el criterio para buscar el documento (o documentos) a actualizar, y el conjunto de nuevos datos a asignar, que suele venir en un bloque denominado `$set`, donde se asigna a cada campo su valor.

> **Ejercicio 2**:

Crea un programa `BibliotecaMongo.py` que cree una base *biblioteca* en tu servidor MongoDB, con una colección llamada *libros*. Inserta en ella estos dos libros con sus campos:

- *titulo*: La tabla de Flandes, *autor*: Arturo Pérez Reverte, *paginas*: 312
- *titulo*: El juego de Ender, *autor*: Orson Scott Card, *paginas*: 452

Después prueba a buscar los libros que tengan más de 400 páginas, para ver si muestra datos correctos. Busca en Internet cómo especificar un rango en el filtrado (libros cuyo número de páginas sea *mayor que* X).

AYUDA: vídeo con la solución del ejercicio