Temasek Junior College

2024 JC2 H2 Computing

NoSQL Databases 2 – PyMongo

Section	3	Data and Information
Unit	3.3	Databases and Data Management
Objectives	3.3.6	Understand how NoSQL database management system addresses the shortcomings of relational database management system (SQL).
	3.3.7	Explain the applications of SQL and NoSQL.

1 What is PyMongo?

MongoDB databases can be accessed using different programming languages like C, Java and Python. To access MongoDB databases using Python, we use the Python driver for MongoDB, **PyMongo**.

Step 1

Start the MongoDB server. The MongoDB server window **MUST REMAIN OPENED** while you want to access the MongoDB database.

Step 2

Start your Python program by importing **PyMongo** on your IDLE or Jupyter Notebook.

Example 1

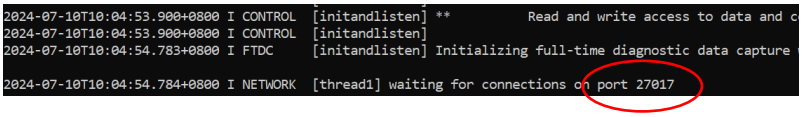
The program below connects to the MongoDB server and outputs the databases currently in the MongoDB server.

Program 1: access.py

```
1  import pymongo
2
3  client = pymongo.MongoClient("127.0.0.1", 27017) # Connect to server
4
5  databases = client.list_database_names() # Check available databases
6
7  print("The databases in the MongoDB server are:")
8  print(databases)
9
10 client.close() # Always remember to close connection
```

Establishing Connection

Line **3** creates a connection to the MongoDB server via port **27017**. (You can obtain the port number when you start the MongoDB server.)



Checking Available Databases

In Line **5**, **list_database_names()** can be used to retrieved the names of the MongoDB databases available as a Python list.

Terminating Connection

Connection to the MongoDB Server can be terminated using **close()**, as shown on Line **10**.

2 Create Operations

Example 2
The program below demonstrates two ways of inserting documents into the **movies** collection of the **entertainment** database.

Note that MongoDB waits until you have inserted at least one document before it actually creates the database and collection.

Program 2: movie.py

```
1  import pymongo
2
3  client = pymongo.MongoClient("127.0.0.1", 27017)  # Connect to server
4  db = client.get_database("entertainment")
5  coll = db.get_collection("movies")
6
7  # Insert one document at a time
8  coll.insert_one({"_id":1, "title":"Johnny Maths", "genre":"comedy"})
9  coll.insert_one({"title":"Star Walls", "genre":"science fiction"})
10 coll.insert_one({"title":"Detection"}) # no genre
11
12 # Create and insert list of documents
13 docs = []
14 docs.append({"title":"Badman", "genre":"adventure", "year":2015})
15 docs.append({"title":"Mean", "genre":["sci-fi","adventure"], "year":2017})
16 docs.append({"title":"Octopus Man", "genre":"adventure", "year":2017})
17 docs.append({"title":"Fantastic Bees", "genre":"adventure", "year":2018})
18
19 coll.insert_many(docs)
20
20 collections = db.list_collection_names()  # Get collections in database
21
22 print(f'Collections in entertainment database: {collections}')
23
24 client.close()  # Always remember to close connection
```

Insert one document
Use **insert_one()**as shown in Lines **8** to **10**. Note that not all fields are required for insertion.

Insert multiple documents
Use the **insert_many()** method to insert a **list of documents** as shown in Line **20**.

Unique document ID assignment
The MongoDB server automatically assigns a unique **_id** to each document. You can customise the **_id** by stating it during insertion as shown in Line **8**. However, you cannot run the program using the same customised **_id** again until you remove this document, otherwise the program will produce an error as the customised **_id** is already in use..

If you run the program again with Line **8** commented out, duplicates of other documents will be created as new **_id** values will be automatically assigned.

Checking Available Collections in a Database
The **list_collection_names()** method in Line **21** can be used to gather a list of collections in the database.

Exercise 1

Write a Python program to ask for one movie title and the year (as an integer) of the movie, then insert the document into the movie collection. Assume no genre is given.

Program 3: insert_from_input.py

```
1  import pymongo
2
3  client = pymongo.MongoClient("127.0.0.1", 27017)
4  db = client.get_database("entertainment")
5  coll = db.get_collection("movie")
6
7  title =                                          # ask for title
8  year =                                           # ask for year
9  year = int(year)
10
11                                                  # insert the document
12
13 client.close()
```

Can you extend the above program to include genres (where movies can have none or multiple genres)?

In fact, to do so in the manner of the above program will be tedious. For large amounts of data, it is more efficient to import from a file.

Example 3

The delimited text file **input.txt** contains comma separated data in the following format:

Text File: input.txt

```
Amanda,45
Bala,28
Charlie,33
Devi,29
...
```

The program below reads from **input.txt** and inserts the documents into the database.

Program 4: insert_from_file.py

```
1  import pymongo, csv
2
3  client = pymongo.MongoClient("127.0.0.1", 27017)
4  db = client.get_database("entertainment")
5  coll = db.get_collection("users")
6
7  # Use of csv is possible for text file with comma separated values
8  with open('input.txt', "r") as csv_file:
9      csv_reader = csv.reader(csv_file)
10     for row in csv_reader:
11         coll.insert_one({"name":row[0], "age":row[1]})
12
13 client.close()
```

Example 4

The JSON file **input.json** contains data stored as a list of dictionaries.

JSON File: input.json

```
[
      {
      "name": "Amanda",
      "age": "45"
      },
      {
      "name": "Bala",
      "age": "28"
      },
      {
      "name": "Charlie",
      "age": "33"
      },
      {
      "name": "Devi",
      "age": "29"
      }
]
```

The **json.load()** function can be used to import the data from **input.json**. The program below demonstrates how this can done.

Program 5: insert_from_json.py

```
1   import pymongo, json
2
3   client = pymongo.MongoClient("127.0.0.1", 27017)
4
5   with open('input.json', "r") as file:
6       data = json.load(file)
7
8   client['entertainment']['moreusers'].insert_many(data)
9
10  client.close()
```

The syntax **client['entertainment']['moreusers']** in Line **8** refers to the database **entertainment** and the collection **moreusers**. This is an alternative method to access the database and collection.

For example, the syntax **db = client.get_database("entertainment")** can be written as **db = client['entertainment']**

and

coll = db.get_collection("users") can be written as **coll = db['moreusers']**.

It follows that the syntax **coll = client['entertainment']['moreusers']** is equivalent to **coll = client.get_database("entertainment").get_collection("users")**

3 Read Operations

Example 5

The program below demonstrates how data can be obtained from the database.

Program 6: view.py

```
1  import pymongo
2
3  client = pymongo.MongoClient("127.0.0.1", 27017)
4  db = client.get_database("entertainment")
5  coll = db.get_collection("movies")
6
7  result = coll.find()
8  docs_in_coll = coll.estimated_document_count()  # Count no. of documents
9
10 print("All documents in movie collection:")
11 for document in result:
12     print(document)
13 print(f'Number of items in movie collection: {docs_in_coll}')
14
15 result = coll.find({'genre': 'adventure'})
16 docs_in_genre = coll.count_documents({'genre': 'adventure'})
17
18 print("All movies with adventure genre:")
19 for document in result:
20     print(document)
21 print(f'Number of adventure movies: {docs_in_genre}')
22
23 criteria = {'genre': 'adventure', 'year': {'$gt': 2016}}
24 result = coll.find(criteria)
25 docs_in_criteria = coll.count_documents(criteria)
26
27 print("All titles of movies with adventure genre after 2016:")
28 for document in result:
29     print(document.get("title"))
30 print(f'There are {docs_in_criteria} movies in the list above.')
31
32 client.close()
```

Retrieving Documents

The **find()** method when applied without arguments returns a cursor of all the documents in the movie collection, as seen in Line **7**.

To retrieve only documents that satisfy a set of criteria, apply the criteria as the argument of the **find()** method. For example, in Line **15**, **find({'genre': 'adventure'})** returns a cursor of all the documents in the collection that belong to the **"adventure"** **"genre"**.

The criteria may also be pre-defined, then used as the argument of the **find()** method. For example, in Line **23**, the criteria **{'genre': 'adventure', 'year': {'$gt': 2016}}** is first defined as **criteria**, then used as the argument in Line **24** as **find(criteria)**. It will return a cursor of all the documents in the collection that belong to the **"adventure"** **"genre"** where the production year is after **"2016"**.

Counting Total Documents in a Collection

The **`estimated_document_count()`** method in Line **`8`** gives the number of documents in the entire movie collection. No arguments need to be provided when using this method.

Counting Number of Documents based on Criteria Specified in Query

When the count is needed for documents that satisfy a set of criteria, the **`count_documents()`** method should be used with the criteria specified as the argument. For example, in line **`16`**, **`count_documents({'genre': 'adventure'})`** will return the count of the documents in the collection that belong to the **`"adventure"`** **`"genre"`**.

The criteria may be pre-defined for use as the argument of the **`count_documents()`** method. For example, in Line **`23`**, **`{'genre': 'adventure', 'year': {'$gt': 2016}}`** is defined first as **`criteria`**, then used as the argument in Line **`25`** as **`count_documents(criteria)`**. It will return the count of the documents in the collection that belong to the **`"adventure"`** **`"genre"`** where the production year is after **`"2016"`**.

Retrieving Specific Fields

The value of a specific field of a document can be retrieved using the **`get()`** method with the field name as the argument. For example, in Line **`29`**, **`get("title")`** retrieves the value of the title field of a document.

3.1 Common Query Operators

The code in Lines **23** and **24** of **Example 5** forms the query to find the documents belonging to the "**adventure**" "**genre**" where the production year is after "**2016**". It can be rewritten using the **$and** operator as follows:

```
{'$and':[{'genre': 'adventure'}, {'year': {'$gt': 2016}}]}
```

The table below shows a list of common MongoDB query operators:

$eq	Equals to	**$in**	In a specified **list**
$gt	Greater than	**$nin**	Not in a specified **list**
$gte	Greater than or equal to	**$or**	Logical **OR**
$lt	Less than	**$and**	Logical **AND**
$lte	Less than or equal to	**$not**	Logical **NOT**
$ne	Not equal to	**$exists**	Matches documents having the named field

Example 6

The program below demonstrates the use of some of the common query operators in Lines **7**, **14** and **21**. Observe how the queries are written and try interpreting them.

Program 7: view2.py

```
1  import pymongo
2
3  client = pymongo.MongoClient("127.0.0.1", 27017)
4  db = client.get_database("entertainment")
5  coll = db.get_collection("movies")
6
7  query1 = {'genre':{'$in':['adventure', 'comedy']}}
8  result = coll.find(query1)
9
10 print("All movies with adventure or comedy genre inside:")
11 for document in result:
12     print(document)
13
14 query2 = {'genre':{'$exists':False}}
15 result = coll.find(query2)
16
17 print("All movies without genre:")
18 for document in result:
19     print(document.get('title'))
20
21 query3 = {'year':{'$eq':2017}}
22 result = coll.find_one(query3)
23
24 print(f'One movie that was released in 2017 is {result}')
25
26 client.close()
```

Retrieving a Single Document

The use of **find_one()** Line **22** will return only one document (the first one) that matches the condition.

Exercise 2

Modify the program with different query operators and options to perform the following:

(a) Find all movies that do not belong to the adventure and comedy genres.

```
query4 = _____________________________________________
result = coll.find(query4)

for document in result:
    print(document)
```

(b) For all movies with year specified in the data. Then print out the movie title and how many years ago the movie was released.

```
from datetime import datetime

current_year = datetime.now().year

query5 = _____________________________
result = coll.find(query5)

for document in result:
    title = ________________________
    years_released = current_year - _____________________
    print(f"Title: {title}, Released {years_released} years ago")
```

(c) Print out all movies released before 2017.

```
query6 =
result = coll.find(query6)

for document in result:
    print(document)
```

4 Update Operations

To modify the content in the database, use

- **`update_one()`** method to modify the first document that matches the query,
- **`update_many()`** method to modify all documents that matches the query.

Example 7

The program below demonstrates the update process. Line **13** uses **`$set`** to set all the **`year`** values greater than **`2016`** to be **`2015`**.

There is also the **`$unset`** operator to remove given fields (see line **29**). Note that even though **`$unset`** operator removes the given fields, there is still a requirement to have a second argument, thus **`0`** is placed even though it won't be updated.

Program 8: update.py

```
1  import pymongo
2
3  client = pymongo.MongoClient("127.0.0.1", 27017)
4  db = client.get_database("entertainment")
5  coll = db.get_collection("movies")
6
7  result = coll.find()
8  print("All documents in movies collection:")
9  for document in result:
10     print(document)
11
12 search = {'year':{'$gt':2016}}
13 update = {'$set':{'year':2015}}
14
15 coll.update_one(search, update)
16
17 result = coll.find()
18 print("All documents in movies collection after update one:")
19 for document in result:
20     print(document)
21
22 coll.update_many(search, update)
23
24 result = coll.find()
25 print("All documents in movies collection after updating all:")
26 for document in result:
27     print(document)
28
29 search = {'year':{'$eq':2014}}
30 remove = {'$unset':{'year':0}}
31 coll.update_many(search, remove)
32
33 result = coll.find()
34 print("All documents in movies collection after unset:")
35 for document in result:
36     print(document)
37
38 client.close()
```

Exercise 3

(a) Modify Lines **12** and **13** to add **'Comedy'** as the **'genre'** to all movies that currently have no value for **'genre'**.

(b) Modify Lines **28** and **29** to remove the **'genre'** field to all movies that currently have **'adventure'** as its **'genre'** or one of its **'genre'**.

5 Delete Operations

Deleting Documents

To delete documents in a collection, you can use

- **`delete_one()`** method to delete the first document that matches the given condition.
- **`delete_many()`** method to delete all the documents that match the condition.

Example 8

The program below demonstrates the use of the above methods.

Program 9: delete.py

```
1  import pymongo
2
3  client = pymongo.MongoClient("127.0.0.1", 27017)
4  db = client.get_database("entertainment")
5  coll = db.get_collection("movies")
6
7  result = coll.find()
8  print("All documents in movies collection:")
9  for document in result:
10       print(document)
11
12 to_del = {'year':2015}
13
14 coll.delete_one(to_del)
15
16 result = coll.find()
17 print("All documents in movies collection after deleting one:")
18 for document in result:
19       print(document)
20
21 coll.delete_many(to_del)
22
23 result = coll.find()
24 print("All documents in movies collection after deleting all:")
25 for document in result:
26       print(document)
27
28 client.close()
```

Exercise 4

Write an appropriate query for deleting all movies with **`'adventure'`** as its **`'genre'`** or one of its **`'genre'`**.

Clearing Collections

To clear the documents in a collection, the **drop()** method can be used

Example 9

The program below demonstrates how to clear a collection using the **drop()** method.

Program 10: remove.py

```
1  import pymongo
2
3  client = pymongo.MongoClient("127.0.0.1", 27017)
4  db = client.get_database("entertainment")
5  coll = db.get_collection("tv")
6
7  coll.insert_one({"title":"X Man", "genre":"science fiction"})
8  coll.insert_one({"title":"Fresh from the boat", "genre":"comedy"})
9  coll.insert_one({"title":"", "genre":"comedy"})
10 coll.insert_one({"genre":"comedy"})
11
12 result = coll.find()
13 no_of_items = coll.estimated_document_count()
14
15 print("All documents in tv collection:")
16 for document in result:
17     print(document)
18 print(f'Number of items in tv collection: {no_of_items}')
19
20 db.drop_collection("tv")
21
22 result = coll.find()
23 no_of_items = coll.estimated_document_count()
24
25 print("After tv collection is dropped:")
26 for document in result:
27     print(document)
28 print(f'Number of items in tv collection: {no_of_items}')
29
30 client.close()
```

Removing a Database

To remove an entire database, you can use the **drop_database()** command with the name of the database to be dropped as the argument. All collections and documents within the database will be removed.

For example, to remove the entertainment database and all its associated collections of documents, the following line can be used:

```
client.drop_database("entertainment")
```