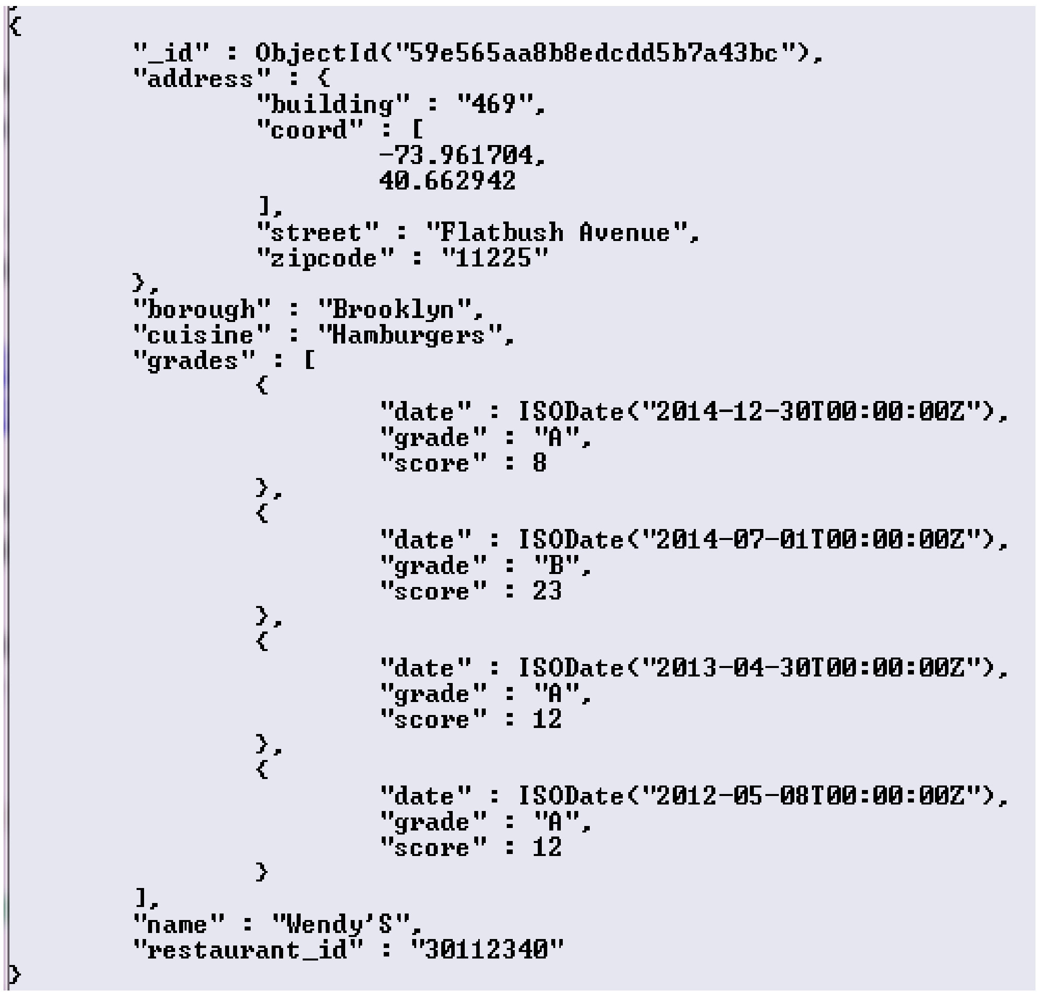
Big Data Lab - Week 5: MongoDB and Python

In this lab you will practice some advanced querying in MongoDB, learn and practice how to use Python to connect to a MongoDB database, and to make use of data stored in the MongoDB database.

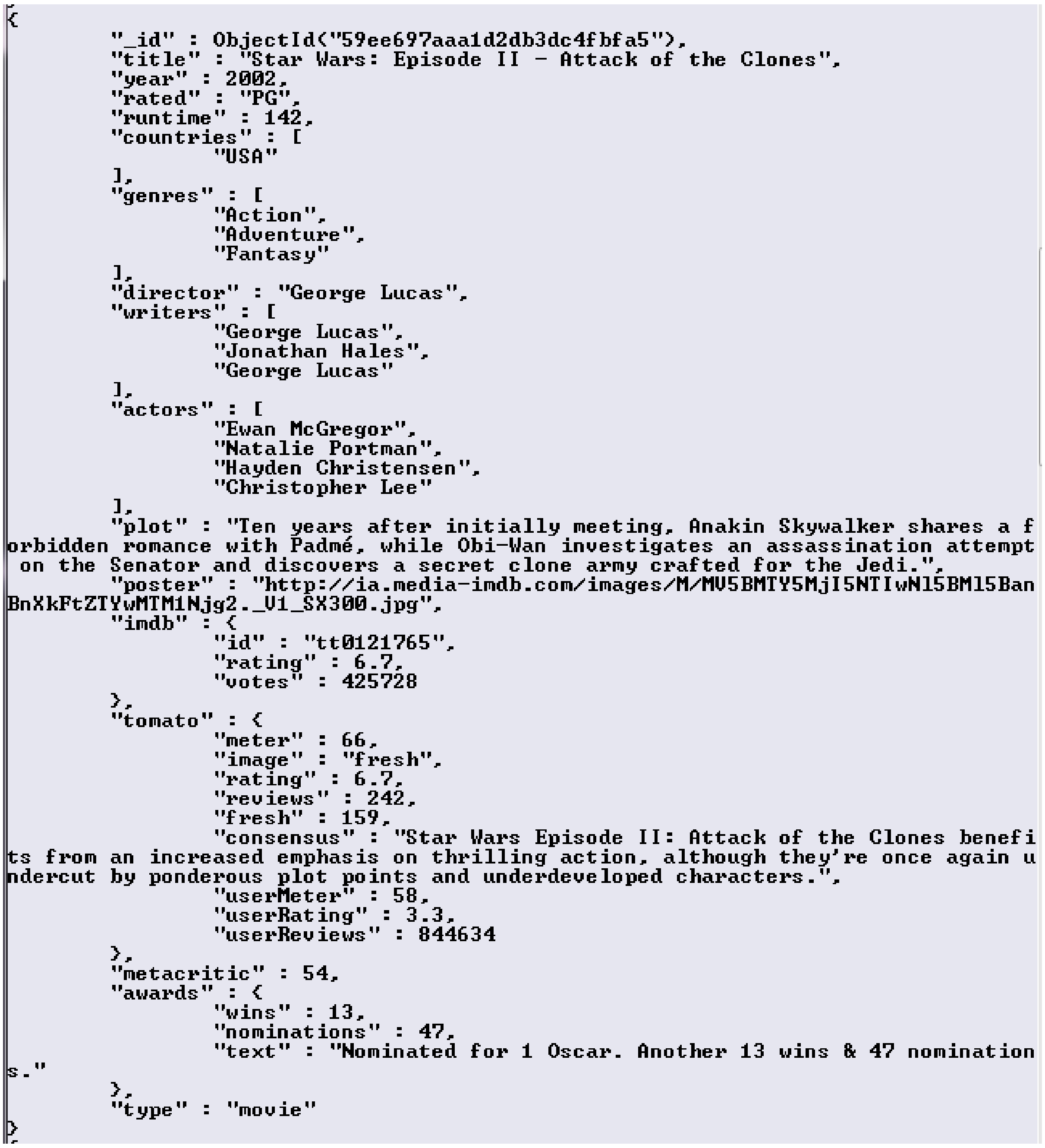
Task 1: MongoDB aggregation

In this task you will devise and run aggregation operations or other advanced queries to retrieve specific data from the ***restaurants*** and ***movieDetails*** data sets (same as the ones we used in week 4's lab). You can refer to the following two snapshots for the data models for these two data sets.



```
{
        "_id" : ObjectId("59e565aa8b8edcdd5b7a43bc"),
        "address" : {
                "building" : "469",
                "coord" : [
                        -73.961704,
                        40.662942
                ],
                "street" : "Flatbush Avenue",
                "zipcode" : "11225"
        },
        "borough" : "Brooklyn",
        "cuisine" : "Hamburgers",
        "grades" : [
                {
                        "date" : ISODate("2014-12-30T00:00:00Z"),
                        "grade" : "A",
                        "score" : 8
                },
                {
                        "date" : ISODate("2014-07-01T00:00:00Z"),
                        "grade" : "B",
                        "score" : 23
                },
                {
                        "date" : ISODate("2013-04-30T00:00:00Z"),
                        "grade" : "A",
                        "score" : 12
                },
                {
                        "date" : ISODate("2012-05-08T00:00:00Z"),
                        "grade" : "A",
                        "score" : 12
                }
        ],
        "name" : "Wendy'S",
        "restaurant_id" : "30112340"
}
```

Figure 1: snapshot of a document in the *restaurants* collection



```
{
        "_id" : ObjectId("59ee697aaa1d2db3dc4fbfa5"),
        "title" : "Star Wars: Episode II - Attack of the Clones",
        "year" : 2002,
        "rated" : "PG",
        "runtime" : 142,
        "countries" : [
                "USA"
        ],
        "genres" : [
                "Action",
                "Adventure",
                "Fantasy"
        ],
        "director" : "George Lucas",
        "writers" : [
                "George Lucas",
                "Jonathan Hales",
                "George Lucas"
        ],
        "actors" : [
                "Ewan McGregor",
                "Natalie Portman",
                "Hayden Christensen",
                "Christopher Lee"
        ],
        "plot" : "Ten years after initially meeting, Anakin Skywalker shares a f
orbidden romance with Padmé, while Obi-Wan investigates an assassination attempt
 on the Senator and discovers a secret clone army crafted for the Jedi.",
        "poster" : "http://ia.media-imdb.com/images/M/MV5BMTY5MjI5NTIwNl5BMl5Ban
BnXkFtZTYwMTM1Njg2._V1_SX300.jpg",
        "imdb" : {
                "id" : "tt0121765",
                "rating" : 6.7,
                "votes" : 425728
        },
        "tomato" : {
                "meter" : 66,
                "image" : "fresh",
                "rating" : 6.7,
                "reviews" : 242,
                "fresh" : 159,
                "consensus" : "Star Wars Episode II: Attack of the Clones benefi
ts from an increased emphasis on thrilling action, although they're once again u
ndercut by ponderous plot points and underdeveloped characters.",
                "userMeter" : 58,
                "userRating" : 3.3,
                "userReviews" : 844634
        },
        "metacritic" : 54,
        "awards" : {
                "wins" : 13,
                "nominations" : 47,
                "text" : "Nominated for 1 Oscar. Another 13 wins & 47 nomination
s."
        },
        "type" : "movie"
}
```

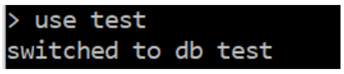
Figure 2: snapshot of a document in the *movieDetails* collection

1. Start MongDB server and client

   As what you did in week 4 lab, start a MongoDB server instance with the **mongod** command; then start a mongo shell client using **mongo** command.

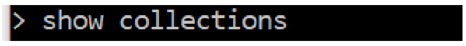
2. Prepare the collections

   Switch to using the *test* database:



```
> use test
switched to db test
```

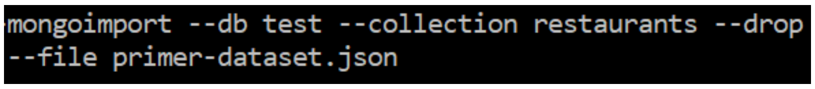
Check if the ***restaurants*** and ***movieDetails*** collections exist in the test database:



```
> show collections
```

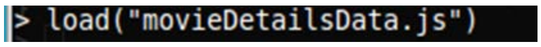
If those collections already exist, move to the next step. Otherwise, you can download *primer-dataset.json* and *movieDetailsData.js* files from GCULearn, then, similar to what you did in week 4 lab:

Run the *mongoimport* utility from the system command line and import the documents from *primer-dataset.json* into the *restaurants* collection in the database *test*:



```
mongoimport --db test --collection restaurants --drop
--file primer-dataset.json
```

Execute the *movieDetailsData.js* file using load in *mongo shell* to import the *movieDetails* documents into the *test* database:



```
> load("movieDetailsData.js")
```

3. Use the aggregation pipeline to list the years with total movie runtime between **2000** and **4000**

```
{ "_id" : 1995, "totalRuntime" : 2188 }
{ "_id" : 1986, "totalRuntime" : 2706 }
{ "_id" : 1997, "totalRuntime" : 2325 }
{ "_id" : 2001, "totalRuntime" : 3826 }
{ "_id" : 1988, "totalRuntime" : 2445 }
{ "_id" : 1998, "totalRuntime" : 2816 }
{ "_id" : 1990, "totalRuntime" : 2073 }
{ "_id" : 1996, "totalRuntime" : 2666 }
{ "_id" : 1989, "totalRuntime" : 2229 }
{ "_id" : 1994, "totalRuntime" : 2636 }
{ "_id" : 1981, "totalRuntime" : 2049 }
{ "_id" : 1993, "totalRuntime" : 2939 }
{ "_id" : 2015, "totalRuntime" : 3583 }
{ "_id" : 1999, "totalRuntime" : 3885 }
```

4. Use the aggregation pipeline to find the average year movie runtime. Hint – you will need two $group stages

```
{ "_id" : null, "avgRuntime" : 1359.0254237288136 }
```

5. Find all restaurants whose name contains at a 3-digit number with spaces before and after the number. Part of the result is shown below, there are 38 matching restaurants. *Hint:* use the *$regex* operator

```
{ "name" : "Cafe 101 16Th Floor Cafeteria" }
{ "name" : "Jerry'S 637 Diner" }
{ "name" : "Tea Shop 168 & Bakery" }
{ "name" : "Grill 149 Plus" }
{ "name" : "West 190 Street Pizza" }
{ "name" : "Lounge 247 I M O K" }
{ "name" : "Stand 142 The Original Cascarino\u001aS Brante" }
{ "name" : "Stand 140 Beer Island" }
{ "name" : "Stand 139 Blue Smoke" }
{ "name" : "Stand 139 Taco Frites" }
{ "name" : "Stand 139 Shake Shack" }
{ "name" : "Stands 303 And 301 Pepsi Porch" }
{ "name" : "Stand 335 Beverages & Snacks" }
{ "name" : "Stand 325 Nathan'S Dogs & Burgers" }
{ "name" : "Stand 321 Caesars Club" }
{ "name" : "Stand 320 - Premio" }
{ "name" : "Stand 127 Food Court" }
{ "name" : "Stand 110 A" }
{ "name" : "Stand 334 Beer Room" }
{ "name" : "Stand 110 B" }
```

6. Use the aggregation pipeline to list the names of all restaurants with a grade dated **2011-03-03**. (same as you did in question 10 of task 3 in week 4 lab) [*Hint*: use ISODate("2011-03-03T00:00:00Z") to specify the date]

```
{ "name" : "The Assembly Bar" }
{ "name" : "Benny'S Famous Pizza" }
{ "name" : "Ray'S Pizza Restuarant" }
{ "name" : "La Piazza Pizzeria & Restaurant" }
{ "name" : "Kennedy Fried Chicken" }
{ "name" : "Barbara Blum Residence / Good Shepherd Services" }
{ "name" : "Le Pain Quotidien" }
{ "name" : "Blimpie" }
```

You could also try to reproduce the result of the *challenge* exercise in that question 10 using the aggregation pipeline:

> use the $*elemMatch* operator in the projection document to include the grade with that date, and no other grades.

This will involve using the *$filter* operator within the *$project* stage, and you may need to do some research to achieve this. As before, only the first document in the result is shown below.

```
{
        "name" : "The Assembly Bar",
        "grades" : [
                {
                        "date" : ISODate("2011-03-03T00:00:00Z"),
                        "grade" : "A",
                        "score" : 6
```

```
            }
        ]
    }
```

7. Use map-reduce to find the number of Chinese restaurants in each borough. You should find that there are 2418 keys emitted reduced to 6 output.

```
{ "_id" : "Bronx", "value" : 323 }
{ "_id" : "Brooklyn", "value" : 763 }
{ "_id" : "Manhattan", "value" : 510 }
{ "_id" : "Missing", "value" : 6 }
{ "_id" : "Queens", "value" : 728 }
{ "_id" : "Staten Island", "value" : 88 }
```

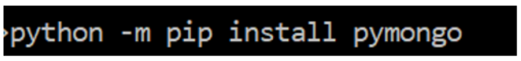
In the following task 2 and task 3, you can use Jupyter Notebook, Spyder IDE, Python shell, or any other locally installed python development environments, but *do **not** use the cloud based Google Colab*. As we are working with a single local MongoDB server (same as in week 4 lab).

Task 2: Getting started with Python and MongoDB

You can get started with MongoDB and your favourite programming language by leveraging one of Mongo Drivers (https://docs.mongodb.com/manual/applications/drivers/). **PyMongo** is the Python driver for MongoDB. It is the recommended way to work with MongoDB in Python.

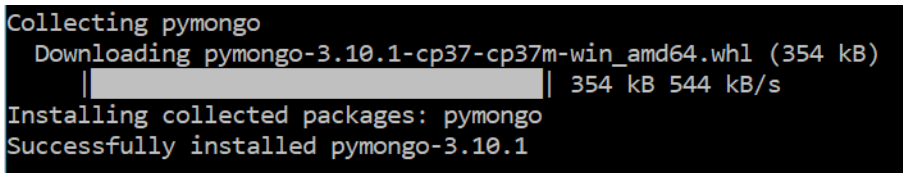
PyMongo has been installed in the Linux Mint Virtual Machine in the lab. You can refer to previous lab materials on how to fire up the virtual machine.

If you have Anaconda on your windows machine, in order to install PyMongo, you can open an Anaconda Prompt window (i.e., Click the Windows Start button and navigate to the Anaconda3 folder, click the arrow on the right end the menu to open the list, then click Anaconda Prompt (Anaconda3)), and run run the following pip command:



```
python -m pip install pymongo
```

You should see information like below:



```
Collecting pymongo
  Downloading pymongo-3.10.1-cp37-cp37m-win_amd64.whl (354 kB)
     |████████████████████████████████| 354 kB 544 kB/s
Installing collected packages: pymongo
Successfully installed pymongo-3.10.1
```

To install PyMongo on another machine, you can find instruction at:

https://docs.mongodb.com/drivers/pymongo/

Additional references for this lab can be find at:

https://pymongo.readthedocs.io/en/stable/

1. Start the MongoDB server, if it is not running. Check if the ***restaurants*** and ***movieDetails*** collections exist.
   (Hint: specify the database and then use "*show collections*" to check if those collections exist)

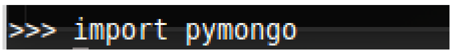
2. Open a new terminal window and start the Python shell by running the command ***python*** (with full path if necessary). When it is ready you should see the following python prompt:

   ```
   >>>
   ```

   If you are using the VM, do not forget to run the following command before starting the Python shell, to make sure you are running Python3

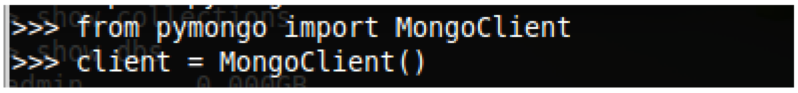
   source activate py37

3. In the python shell, run the following command to check that the PyMongo distribution is installed - this should execute without error:

   ```
   >>> import pymongo
   ```

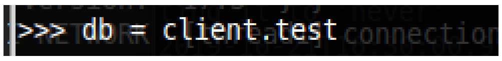
4. When working with PyMongo, the first step is to create a **MongoClient** to the running mongod instance.

   ```
   >>> from pymongo import MongoClient
   >>> client = MongoClient()
   ```

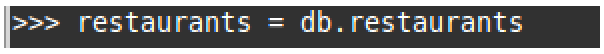
   This will connect to the local mongod instance on the default port.

5. Get a reference to the ***test*** database

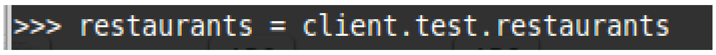
   ```
   >>> db = client.test
   ```

6. Get a reference to the ***restaurants*** collection

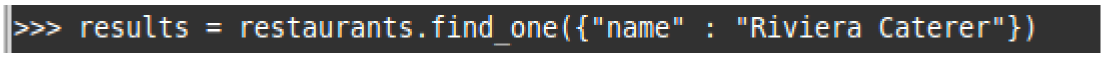
   ```
   >>> restaurants = db.restaurants
   ```

   Or you can omit step 5 above and get a reference to the *restaurants* collection using:



```
>>> restaurants = client.test.restaurants
```

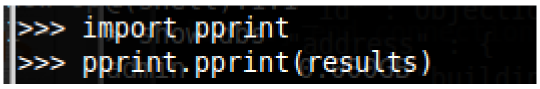
7. Now you can retrieve a document from the collection:



```
>>> results = restaurants.find_one({"name" : "Riviera Caterer"})
```

The ***find_one()*** method returns a single document matching the query. It is similar to using *findOne()* at the Mongo shell. The result is return to Python as a dictionary.

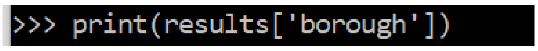
8. You can print the properties in the retrieved document. In order to make the output structured and visually appealing from a console, you need to import ***pprint***:



```
>>> import pprint
>>> pprint.pprint(results)
```

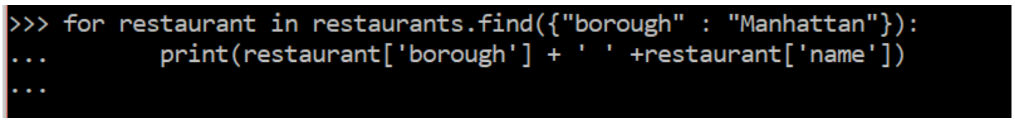
You will see the '*u*' character in front of a string in the printed results, which denotes that it is a Unicode string.

9. The notation for representing python dictionary is very similar to JSON. Enter the following to get the ***borough*** for the document you have retrieved:



```
>>> print(results['borough'])
```

10. To get more than a single document as the result of a query use the ***find()*** method. *find()* returns a Cursor, which allows you to iterate over all matching documents. Enter the following code to find all the restaurants in Manhattan.



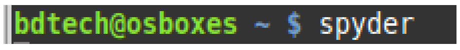
```
>>> for restaurant in restaurants.find({"borough" : "Manhattan"}):
...         print(restaurant['borough'] + ' ' +restaurant['name'])
...
```

Task 3: CRUD operation using PyMongo

In this task, you will practice how to perform CRUD operations from python with PyMongo. You can use Spyder, Jupyter Notebook, Python shell, or any other locally installed python development environment you prefer, although Spyder IDE is used below.

1. Launch the Spyder IDE

   If you are using the VM, Open a terminal window, enter the following command:

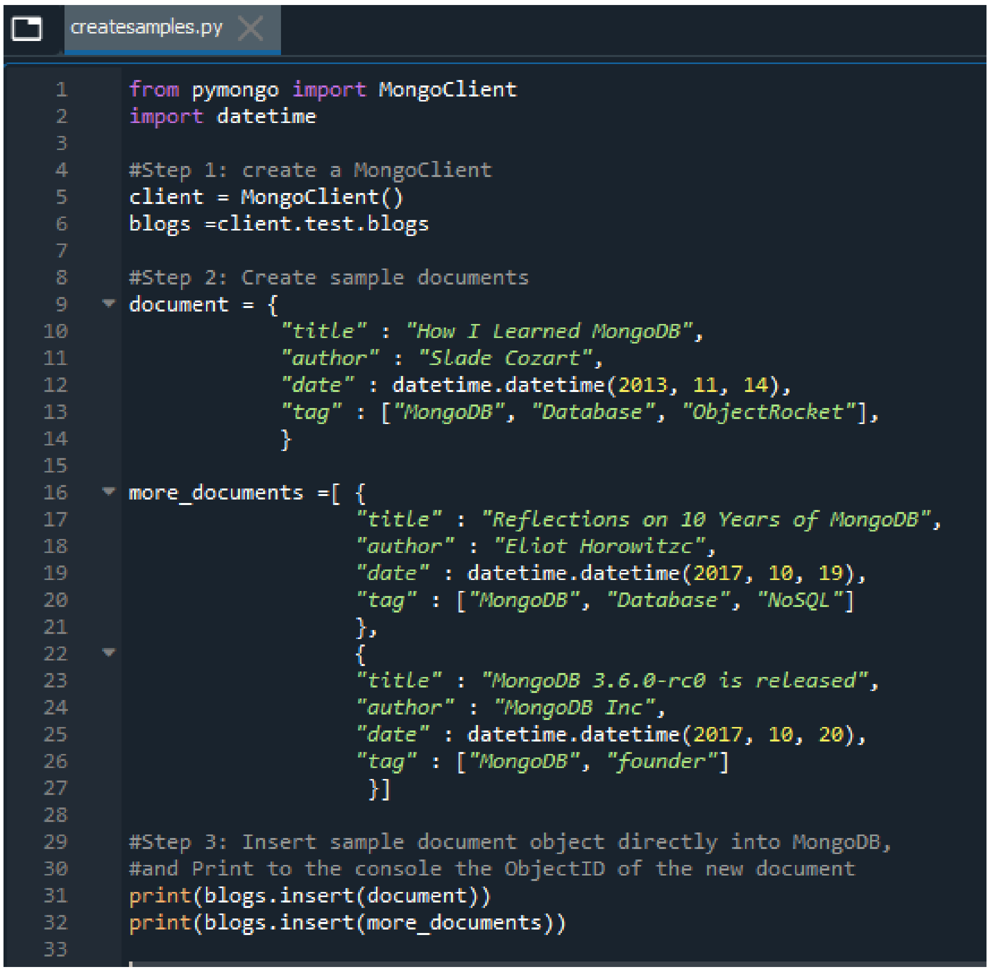


```
bdtech@osboxes ~ $ spyder
```

You will see the *Spyder* window. Note, try *spyder --new-instance* if the above command does not work.

If you are using Anaconda on a windows machine, click the Windows Start button and navigate to the Anaconda3 folder, click the arrow on the right end of the menu to open the list, then click **Spyder (Anaconda3)** to launch the *Spyder* IDE.

2. Download the *createsamples.py* file from GCUlearn and open it in Spyder, you will see the following code:



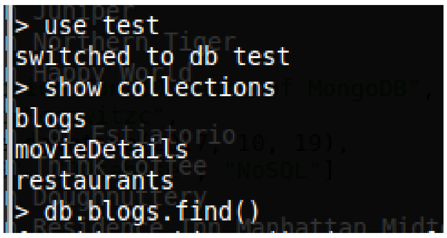
```python
from pymongo import MongoClient
import datetime

#Step 1: create a MongoClient
client = MongoClient()
blogs =client.test.blogs

#Step 2: Create sample documents
document = {
            "title" : "How I Learned MongoDB",
            "author" : "Slade Cozart",
            "date" : datetime.datetime(2013, 11, 14),
            "tag" : ["MongoDB", "Database", "ObjectRocket"],
            }

more_documents =[ {
                  "title" : "Reflections on 10 Years of MongoDB",
                  "author" : "Eliot Horowitzc",
                  "date" : datetime.datetime(2017, 10, 19),
                  "tag" : ["MongoDB", "Database", "NoSQL"]
                  },
                  {
                  "title" : "MongoDB 3.6.0-rc0 is released",
                  "author" : "MongoDB Inc",
                  "date" : datetime.datetime(2017, 10, 20),
                  "tag" : ["MongoDB", "founder"]
                   }]

#Step 3: Insert sample document object directly into MongoDB,
#and Print to the console the ObjectID of the new document
print(blogs.insert(document))
print(blogs.insert(more_documents))
```

There are three steps in this script. In step 1, a ***MongoClient*** is created, and a reference to the ***blogs* collection** is get (the collection will be created lazily if it doesn't exist – i.e. it will be created when you actually insert the first document in it.).

In step 2, a Python dictionary object called ***document*** and a Python list ***more_documents***, with each element of which is a dictionary, are created.

Finally, the ***document*** object and the ***more_documents*** list are inserted into the ***blogs*** collection using ***insert*** command, and the object IDs created on those insertions are printed.

Run this script, and check the output.
You should also be able to find the inserted collection and documents in a Mongo shell:



```
> use test
switched to db test
> show collections
blogs
movieDetails
restaurants
> db.blogs.find()
```

If you are running the script on your own machine with latest MongoDB & PyMongo installed, you may see some warning messages like:

DeprecationWarning: insert is deprecated. Use insert_one or insert_many instead.

You can fix this by replacing step 3 in the script with the following lines, i.e., to insert the *document* object into the *blogs* collection using *insert_one* command, and the *more_documents* list into *blogs* using *insert_many* command.

```
blog_id = blogs.insert_one(document).inserted_id
result = blogs.insert_many(more_documents)

print blog_id
print result.inserted_ids
```

3. Create a python script to print the name and street of all the restaurants in the borough *Queens* with cuisine of type *Delicatessen*.

   Note that the address property for each restaurant is itself represented in Python as a dictionary, so you need to think about how you can access its street property.

4. Create a python script to print the title and awards of movies directed by "Sergio Leone".

   Note that the find function can take one or two parameters – you could:

   - Specify a query document to find all matching *movieDetails*, retrieving the whole document for each, and print only the value of the title, director and awards properties;

- Specify a query document and a projection document, as you have done in the Mongo shell last week, to retrieve documents containing only the title, director and awards properties.

Which do you think would be better for this use case?

5. Create a python script to add a new grade object to the grades array of the restaurant with name '*Line Bagels*', and print out the updated document.

   You will need to create the grade object first:

   - Specify the date as today using datetime.datetime.now().
   - The grade could be "B"
   - The score properties could be 15

   Then you can use ***update()*** operation (https://api.mongodb.com/python/2.7.2/api/pymongo/collection.html) to make the change.

   Note:
   Since you are adding to an array you need to use the *$push* operator instead of *$set*.

6. MongoDB aggregation operations (both Aggregation Pipeline and Map-Reduce function) can be invoked from Python. Read the Aggregation Examples at:

   https://pymongo.readthedocs.io/en/stable/examples/aggregation.html

   and transfer the python statements into a script in Spyder. Check the output of your script.