Introduction to MongoDB: Part 2

Contents

1 Further MongoDB

This workbook assumes that you have carried out Sections 1 and 2 from the MongoDB Wiki:

https://mi-linux.wlv.ac.uk/wiki/index.php/MongoDB_Workbook

Twitter data contains nested documents so you may also want to carry out Sections 3 and 4 too.

1.1 Importing Tweets

On Google Classroom you can find a JSON file that contains some tweets about the weather. Download this file.

Before starting MongoDB you need to use the *mongoimport* command first to load the data. The format for this command is (type on one line only):

```
mongoimport - - db yourDBname - - collection your_Colleciton_Name - - File Location_of_File
```

1.2 Start MongoDB

Start MongoDB by using the command:

```
mongo
```

First check that the tweets have been imported:

```
show collections
```

The Twitter data is stored in the *weather* collection. We can run *db.weather.count()* to count the number of documents:

```
db.weather.count()
```

1.3 Explore the Documents

We can examine the contents of one of the documents by running:

```
db.weather.findOne()
```

The document is quite complex and has several fields, such as: *name, retweet_count, tweet_ID, etc.*. There are also nested fields under *user,* e.g., *screen_name, friends_count, location*, etc.

We can find the distinct values for a specific field by using the *distinct()* command. For example, let's find the distinct values for *lang (language)*:

```
db.weather.distinct("lang")
```

or find what the text of the tweets say:

```
db.weather.distinct("text")
```

Finding unique values within a nested document needs the use of the dot notation, for example, find a user’s name:

```
db.weather.distinct("user.name")
```

Some documents can have several layers of nesting, which can be accessed using the dot notation:

```
db.weather.distinct("extended_entities.media.type")
```

1.4 Search for specific field value

We can search for fields with a specific value using the *find()* command. For example, pick one of the names seen in the previous command and search *for* their tweets. For example:

```
db.weather.find({"user.name":"BBC Highlands"})
```

- ➢ Choose another user to view their tweets.
- ➢ Try viewing a different nested field, such as their screen_name

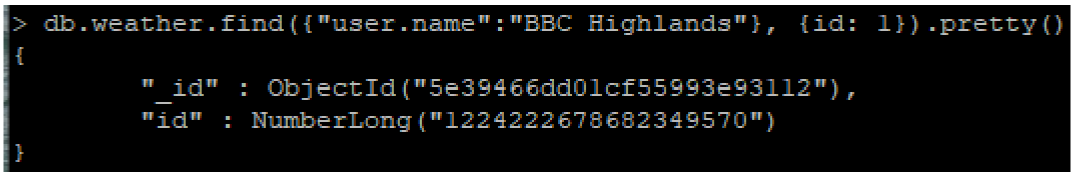
Please note:

- ❖ Apologies in advance if there are any swear or offensive words in the text – do remember this is data produced by Twitter and is outside our control. Steps were taken to remove anything obvious, but offensive data could have slipped through given it is a large dataset. If you do find anything offensive please let the module leader know!

1.5 Filter fields returned by query

We can specify a second argument to the *find()* command to only show specific field(s) in the result. Let's repeat the previous search, but only show the *tweet_ID* field:

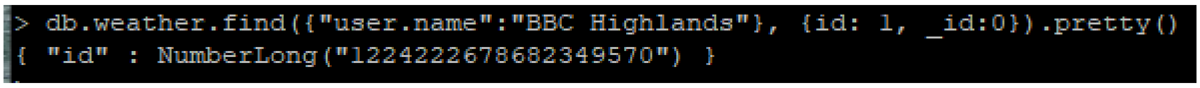
```
db.weather.find({"user.name":"BBC Highlands"}, {id: 1}).pretty()
```



```
> db.weather.find({"user.name":"BBC Highlands"}, {id: 1}).pretty()
{
        "_id" : ObjectId("5e39466dd01cf55993e93112"),
        "id" : NumberLong("1224222678682349570")
}
```

The *_id* field is objectID for every document and will be different for each user. We can remove it from the results with the following filter:

```
db.weather.find({"user.name":"BBC Highlands"}, {id: 1, _id:0}).pretty()
```



```
> db.weather.find({"user.name":"BBC Highlands"}, {id: 1, _id:0}).pretty()
{ "id" : NumberLong("1224222678682349570") }
```

Remember the fields names are case sensitive, so if you get a null return:

```
{ }
```

It could be because you have used the wrong case.

1.6 Pattern Matching

You may not always want to search for an exact matching value, instead looking for patterns in the data, similar to the LIKE command in SQL. MongoDB supports pattern matching by using regular expressions.

The format for a regular expression is:

```
{ <field>: /pattern/<options> }
```

For example, if we search for the value *sun* in the tweet's *text* field, there are no results:

```
db.weather.find({text: "sun"})
```

However, if we search using a regular expression, there are many results:

```
db.weather.find({text: /sun/}, {text: 1})
```

Note, the quotes have gone, otherwise it will try to search for "/sun/"

The difference between the queries is that the first searched for where the *text* field value was exactly equal to *sun*, and the second searched for where the field value contained *sun*.

We can append *.count()* to the command to count the number of results:

```
db.weather.find({text: /sun/}).count()
```

How many included *rain* in the text instead:

```
db.weather.find({text: /rain/}).count()
```

1.6.1 $regex

$regex also provides regular expression capabilities for pattern matching *strings* in queries.

$regex supports the following syntax:

```
{ <field>: { $regex: /pattern/, $options: '<options>' } }
{ <field>: { $regex: 'pattern', $options: '<options>' } }
{ <field>: { $regex: /pattern/<options> } }
```

For example, the previous search for *sun* in the text field can also be written using $regex:

```
db.weather.find({ text: { $regex: /sun/ }  }).count()
```

1.6.2 Regular Expression Options

Options can be added to the regular expression.

These include:

`i`	Case insensitivity to match upper and lower cases.
`m`	For patterns that include anchors (i.e. `^` for the start, `$` for the end), match at the beginning or end of each line for strings with multiline values. Without this option, these anchors match at beginning or end of the string. If the pattern contains no anchors or if the string value has no newline characters (e.g. `\n`), the `m` option has no effect.
`x`	"Extended" capability to ignore all white space characters in the `$regex` pattern unless escaped or included in a character class.
`s`	Allows the dot character (i.e. .) to match all characters including newline characters.

The previous query could be missing results if "sun" was actually written as "SUN", "Sun" or other case variations. To make the search case insensitive use the i option too:

```
db.weather.find({ text: { $regex: /sun/i }   }).count()
```

This time the count should be higher.

The above queries will find the value anywhere in the text field. If we want to search for documents that start with a certain value, then use the start anchor, which is a claret (`^`).

For example, search for text values that are retweets (start with *RT*):

```
db.weather.find({ text: { $regex: /^RT/i }   }).count()
```

The same can be done for documents that end with a certain value, this time using the dollar sign (`$`). For example, show all texts ending in *snow*:

```
db.weather.find({ text: { $regex: /snow$/i }   })
```

1.7 $regex vs. /pattern/ Syntax

Depending on what type of query you are asking will determine whether you must use `$regex` or the `/pattern/` syntax. In many cases either can be used, but some certain circumstances will dictate which must be used.

1.7.1 `$in` Expressions

To include a regular expression in an `$in` query expression, you can only use JavaScript regular expression objects (i.e. `/pattern/`). `$in` is similar to the IN clause in SQL, where you can search for a value that matches any value in the list.

The syntax is:

```
{ field: { $in: [<value1>, <value2>, ... <valueN> ] } }
```

For example, find any texts that starts with *sun* in any case or contains rain:

```
db.weather.find({ text: { $in: [/^sun/i, /rain/] } }, {text: 1} )
```

1.7.2 x and s options

To use either the `x` option or `s` options, you must use the `$regex` operator expression with the `$options` operator. In such cases `$options` must also be used for `i` and `m` too.

For example, find any text that contains *cold* and *night*:

```
db.weather.find({ text: { $regex: /cold.*night/, $options: "si" }   }, {_id:0, text:1})
```

To summarise `$regex` cannot be used for `$in` queries, but must be used if using the `i` or `m` options.

2 Indexes

A text index can be created to speed up searches and allows advanced searches with the *$text* query operator. Further details about *$text* can be found here:

https://docs.mongodb.com/manual/reference/operator/query/text/

2.1 Creating an Index

Let's create the index using *createIndex():*

```
db.weather.createIndex({text: "text"})
```

The first argument *text* specifies the field on which to create the index, the second specifies that this is a text field (coincidentally the field we are indexing on is also called text).

2.2 Searching with an Index

Next, we can use the *$text* operator to search the collection. We can perform the previous query to find the documents containing *snow:*

```
db.weather.find({$text : {$search : "snow"}}).count()
```

We can also search for documents not containing a specific value. For example, let's search for documents containing *snow,* but not *ice:*

```
db.weather.find({$text : {$search : "snow -ice"}}).count()
```

2.3 Show Indexes

The function *getIndexes()* can be used to check what indexes exist on a collection:

```
db.weather.getIndexes()
```

Make a note of the index name.

2.4 Dropping an Index

Indexes can also be dropped using the *dropIndex()* function.

For example, to drop the text index:

```
db.weather.dropIndex("text_text")
```

or replace "*text_text*" by the name seen previously.

3 Further Queries

3.1 Operators

MongoDB can also search for field values matching a specific criteria.

For example, we can find where the *retweet_count* is greater than 20:

```
db.weather.find({"retweeted_status.retweet_count": { $gt : 20 }}).count()
```

Operator Summary	**Comment**
`$gt`	Searches for values greater than a specific value
`$lt`	Looks for values less than a given value
`$eq`	Equality search
`$gte`	Searches for values greater than, or equal to a specific value
`$lte`	Looks for values less than or equal to a given value
`$ne`	Not equals

3.2 Null and Boolean data

Missing data is represented by a `null` and Booleans by `true/false` values. For example, count how many tweets have no user location defined:

```
db.weather.find({"user.location": { $eq : null }}).count()
```

Note, null is not in quotes, if you include quotes it will look for the value null, rather than empty data.

How many users are geo_enabled:

```
db.weather.find({"user.geo_enabled": { $eq : true }}).count()
```

How many users have 10 friends or less:

```
db.weather.find({"user.followers_count": { $lte : 10 }}).count()
```

3.3 Nested documents

Note, syntax dictates that nested attributes have to be enclosed in double quotes, such as *retweet_count* which is a nested attribute of *retweeted_status*.

There can be further nesting in a document, the dot notation can be used to traverse the tree:

```
db.weather.find({"retweeted_status.user.followers_count": { $gt : 2000 }} ).count()
```

3.4 Where Clause

We can use the *$where* command to compare between fields in the same document. For example, the following counts how many users have more followers than friends:

```
db.weather.find({$where :"this.user.followers_count > this.user.friends_count" }).count()
```

Note that the field names for *$where* are required to be prefixed with *this,* which represent the current document.

We can combine multiple searches by using *$and*. For example, let's count any tweets containing *ice* where the re*tweet_count* is greater than fifty:

```
db.weather.find({$and : [ {text : /ice/}, { "retweeted_status.retweet_count":
        {$gt: 50}}]}).count()
```

4 Group by Commands and Aggregation

Carrying out statistical analysis of the data involves using the *$group* command and the use of an aggregation pipeline. This is similar to the GROUP BY command in SQL.

For example, to produce the sum and count of all

```
db.weather.aggregate(   [
   {
     $group: {
        _id : null,
        totalAmount: { $sum: "$retweeted_status.retweet_count"  },
        count: { $sum: 1 }
        }
   }
  ] )
```

See the MongoDB manual for further details:

https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/

5 Error Checking

When you start using more complex commands you are more likely to get errors since most will involve a variety of brackets that need to be the right order.

For example, if you typed in this for the last command:

```
db.weather.find({$and : [ {text : /ice/}, { "retweeted_status.retweet_count":
        {$gt: 50}}}]).count()
```

You would get something similar to this error:

```
2020-02-04T17:47:03.031+0000 E QUERY    [thread1] SyntaxError: missing ] after element list
@(shell):1:88
```

Why is this? If you count the brackets there appears to be the right number of opening and closing brackets, however look more carefully at the order they are in. There is a mixture of curly {}, square [] and round () brackets. It is also important to get the order of them correct too!

Note the order of the final set is out of sync:

```
…{$gt: 50}}}]).count()
```

Should in fact be:

```
…{$gt: 50}}]}).count()
```

You can recall any last command in MongoDB by pressing the up arrow, then use the arrow keys to move to where you want to make any corrections. Press return when you have made the necessary changes.

If you find that you get 0's for the counts and you were expecting some results, check that you have:

- Got the name of the attribute correct. Unlike relational databases, there is no schema for the database to check your query against, so it will not flag that the field name is wrong, it will assume it is just not found in the current set of documents.
- Same goes for the collection name and any other user defined names.
- Checking the nesting. Is the attribute you are interested nested within a sub-document? For example, if the earlier query missed the retweet_count nesting, it will just return 0:

  ```
  db.weather.find({$and : [ {text : /box office/}, { "retweet_count":  {$gt: 50}}]}).count()
  ```

6 Exporting Results

NoSQL databases such as MongoDB are relatively less sophisticated from their relational relatives in that they do not offer the same range of functionality such as just capturing the results to use outside the Mongo environment. For example, there is no equivalent of Oracle's SPOOL command, which is useful if you want to use the results outside the database.

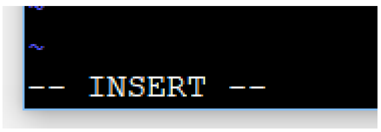
It can be achieved by running a query file against the database and using Linux operating system commands to pipe the results to a text file.

6.1 Query file

Exit MongoDB and create a query file using a text editor such as vi:

```
vi query1.js
```

Press o to put the editor in input mode – you will see – INSERT – at the bottom of the screen:



```
~
~
-- INSERT --
```

and type in the following:

```
db.weather.findOne()
```

Press Esc to exit Input mode, then type:

```
:wq
```

This will exit the vi editor and save the file.

If you want to find out more about vi there are many tutorials online such as:

https://www.tutorialspoint.com/unix/unix-vi-editor.htm

You can also use any other Linux text editor you are familiar with, or use a text editor on Windows, such as NotePad++ and sftp the file to mi-linux.ac.uk.

6.2 Running the Query file

Next type in the following at the Operating System Command prompt:

mongo *dbname* --username *myUsername*--password *mypassword* < query1.js >> result.txt

Where *dbname* is your student number preceded by db, *myUsername* is your student number and *myPassword* is your MongoDB password. For example, if your student number is 0123456, then the command would be:

mongo *db0123456* --username *0123456* --password *myPassword* < query1.js >> result.txt

Do remember that Linux is case sensitive, so *mongo* must be in lower case letters.

This will produce a results file called *result.txt*, which in this case has captured the structure of one tweet. This could be used to work out the structure of the JSON document, since there are no useful data dictionary commands to tell you what the schema is.

7 Cleaning Data

The documents generated by Twitter are quite complex and can be difficult to update. When analysing data you may wish to get a consistent format to some documents, for example, you may only be interested in the country of where the tweets came from, rather than individual towns.

7.1 Updating One Document

By default, the `update()` method updates a **single** document, the `multi` parameter can be set to true to update many, or use the `updateMany()` function.

See the manual for further details:

https://docs.mongodb.com/manual/reference/method/db.collection.update/
https://docs.mongodb.com/manual/reference/method/db.collection.updateOne/
https://docs.mongodb.com/manual/reference/method/db.collection.updateMany/

When using the `update()` method with mu, the `multi` parameter set to false you cannot guarantee which document will be updated. The only way to guarantee updating the document you want is to use the Object Id: "_id".

Note: everyone will have unique identifiers generated, so you will need to replace the ObjectIds below with your own data.

Let's make the language code more user-friendly. See this page for the language associated with each code:

https://developer.twitter.com/en/docs/twitter-for-websites/twitter-for-websites-supported-languages/overview

For example, *en* means *English*, which is the default language.

Let's change one of the languages to the full name, so for example, es would show as Spanish. Check what languages are in your dataset:

```
db.weather.distinct("lang")
```

Pick one of the languages and check the above website to see what the language should be. Then find the documents with this language. Replace languageCode with one of the values seen:

```
db.weather.find({ lang: "languageCode"}, {lang:1})
```

The results will be similar to below, but your Object Ids will be different:

```
{ "_id" : ObjectId("5e39466dd01cf55993e9301f"), "lang" : "languageCode" }
```

Pick one of your ObjectId values, e.g., *5e39466dd01cf55993e9301f* seen above and query just that document:

```
db.weather.find ({ _id: ObjectId("5e39466dd01cf55993e9301f")}, {lang: 1})
```

The results should be for a document for the language code given above.

If you get no results returned, check that you have entered a correct Id.

Using this Id, update the `lang` field so the code shows as the full name instead:

```
db.weather.update({ _id: ObjectId("5e39466dd01cf55993e9301f") }, {$set: {'lang': 'languageName'}})
```

Which should return:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Since we have not set the `multi` option, only one document will be updated, but given we have specified the `ObjectId`, which is unique, it should only update one document anyway.

Check the update has worked:

```
db.weather.find ({ _id: ObjectId("5e39466dd01cf55993e9301f")}, {lang: 1})
```

And the response should now show the new language name:

```
{ "_id" : ObjectId("5e39466dd01cf55993e9301f"), "lang" : "languageName" }
```

7.2 Undefined Language

You may find that you have a language called "*und*" in your list. This is where the language is unknown or underdetermined. To make this clearer, lets change *und* to say *unknown* instead for all documents:

```
db.weather.updateMany({ lang: "und" }, {$set: {'lang': 'unknown'}})
```

This time many documents will be changed:

```
{ "acknowledged" : true, "matchedCount" : 13, "modifiedCount" : 13 }
```

7.3 Nested Documents

Nested documents can also be updated using the dot notation. For example, update the collection so all locations containing England, just say UK, removing any town or county information:

```
db.weather.updateMany({ 'user.location': {$regex:'England', $options: 'i' }} ,
    {$set:  {'user.location': 'UK'}})
```

Remember nested fields must be in quotes. This time `updateMany()` was used, so the response should show several documents have been updated:

```
{ "acknowledged" : true, "matchedCount" : 26, "modifiedCount" : 26 }
```

To do:

- Write some code to check this has updated as expected.
- Check what other locations are in the collection and update any documents where the user's location includes the value `UK` to just say `UK`.

7.4 Updating Tweets

Tweets can contain information you may want to update to be consistent, e.g., if someone has used abbreviations and you want all tweets to say the same time. For example, find all tweets that contain *`pic`*:

```
db.weather.find({'text': {'$regex':'pic', '$options': 'i' }   }, {'text':1})
```

Update this so it says *`picture`* instead, this involves matching substrings:

```
db.weather.find({'text': {'$regex':'pic' }}).forEach(function(e)
        {    e.text=e.text.replace("pic","picture");
        db.weather.save(e); });
```

The "e" represents a copy of each document found. That copy has its text field updated based upon its original value (all those documents where the text contains "pic"). "e" keeps all of its original fields with the one modified field. Saving e back into the collection overwrites the original "e".

Use of the function is based on information from this page:

https://stackoverflow.com/questions/12589792/how-to-replace-substring-in-mongodb-document

Note:

- currently function has no equivalent in Python.
- tweets can contain Unicode characters to represent icons, etc. How they are represented can depend on how you view the data. For example, putty may show the characters as square boxes, whilst a Python notebook might show the Unicode character, or the icon.

7.5 Reshaping the Documents

You might only be interested in certain fields for your analysis and want create a collection with just the required data. An aggregation pipeline using `$project` can be used to pick out the required columns and the results saved to a new collection.

For example, say you just want to keep the information that shows the user's location and the text:

```
db.weather.find( {}, {"user.location":1, text:1})
```

To save these details into a new collection called `projWeather`:

```
db.weather.aggregate([
        {$project:{"user.location":1, text:1}},
        {$out:"projWeather"}
])
```

Check that the new collection exists with the required data:

```
db.projWeather.find().pretty()
```

You can also reduce the number of rows in a collection. This could be done with a query that is based on some criteria, or if you just want to get a subset of the data, not based on any particular criteria you can use an aggregation pipeline using `$limit` to restrict the number of rows.

The syntax is:

```
{ $limit: <positive integer> }
```

For example, just keep 50 rows from the `projWeather` collection:

```
db.projWeather.aggregate([
    {$limit: 50},
    {$out:"newWeather"} ])
```

Check it does indeed only have 50 rows:

```
db.newWeather.count()
```

Compared to projWeather:

```
db.projWeather.count()
```

8 Reloading the Collection

MongoDB is not so good at transaction support compared to relational databases; updates will be made with no option to rollback the changes if you change your mind.

If you want to return to the original data, one option is to drop the collection completely (do not do this unless you want to remove the collection completely!):

```
db.weather.drop()
```

Then reload from the original `weather.json` file, as seen in Section 1.2 above.

9 Further Tweets

To obtain your own Twitter data you need to register for a Developer's account. See this page for further details:

https://developer.twitter.com/en/apply-for-access.html

10 Python Notebook

The examples in this work book use putty and the MongoDB shell, which uses JavaScript. A python notebook version of this workbook called `mongo-Weather-Template.ipynb` is also available on Canvas.

This part is optional and for information only, though you may find using the Notebooks easier to understand the data.

The notebook can used be on various Python Notebooks:

Anaconda 5.3.0 (Py… Anaconda, Inc.	Anaconda (via Apps Anywhere). Pick the 5.3.0 version
	If you have a Google account you can use Google Colaboratory found here: https://colab.research.google.com/

Important:

You will need to update the username and password details in the Notebook before it can be run successfully.

10.1 PyMongo

The python workbook makes use of the PyMongo API, which contains tools for working with MongoDB. Further details about this API can be found here:

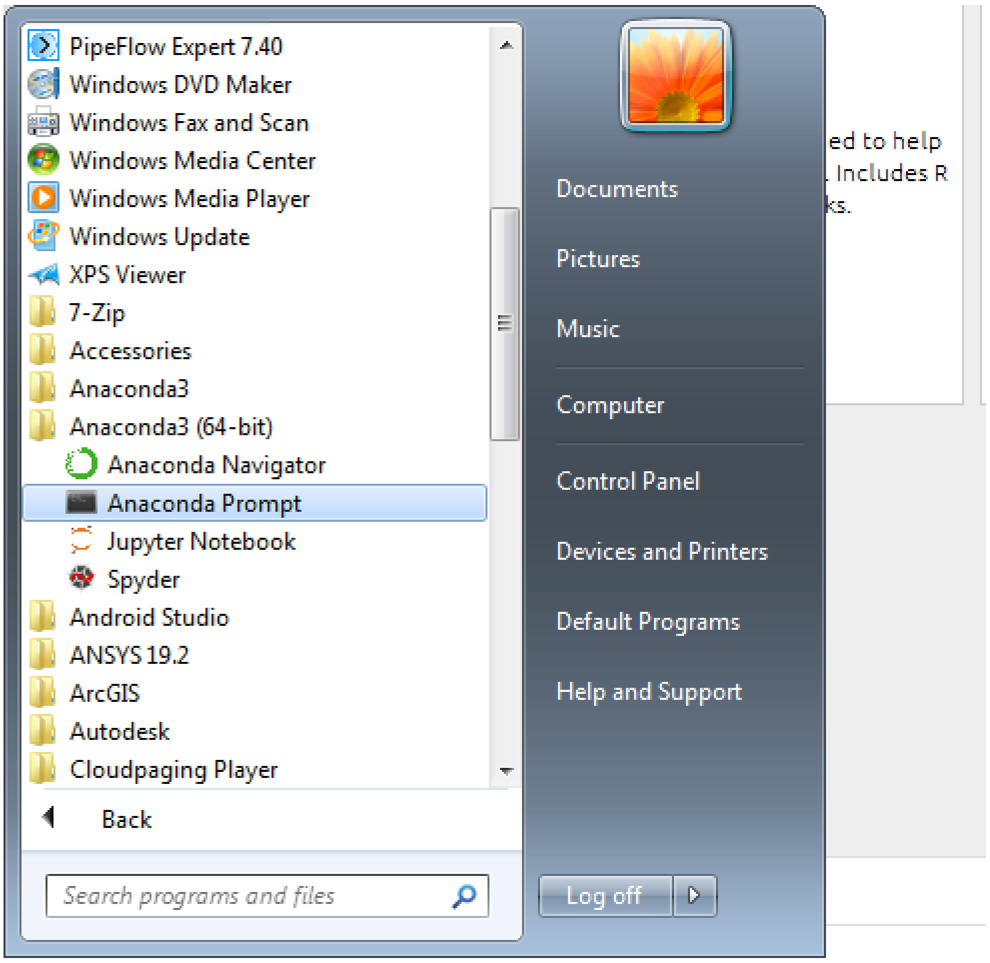
https://api.mongodb.com/python/current/

Google Colaboratory has this module already imported, so no further installation work is needed.

10.2 Anaconda

If using Anaconda, you will need to start the App from Apps Anywhere.

When the `Anaconda Navigator` has started, it will also have added some options to the Start menu:

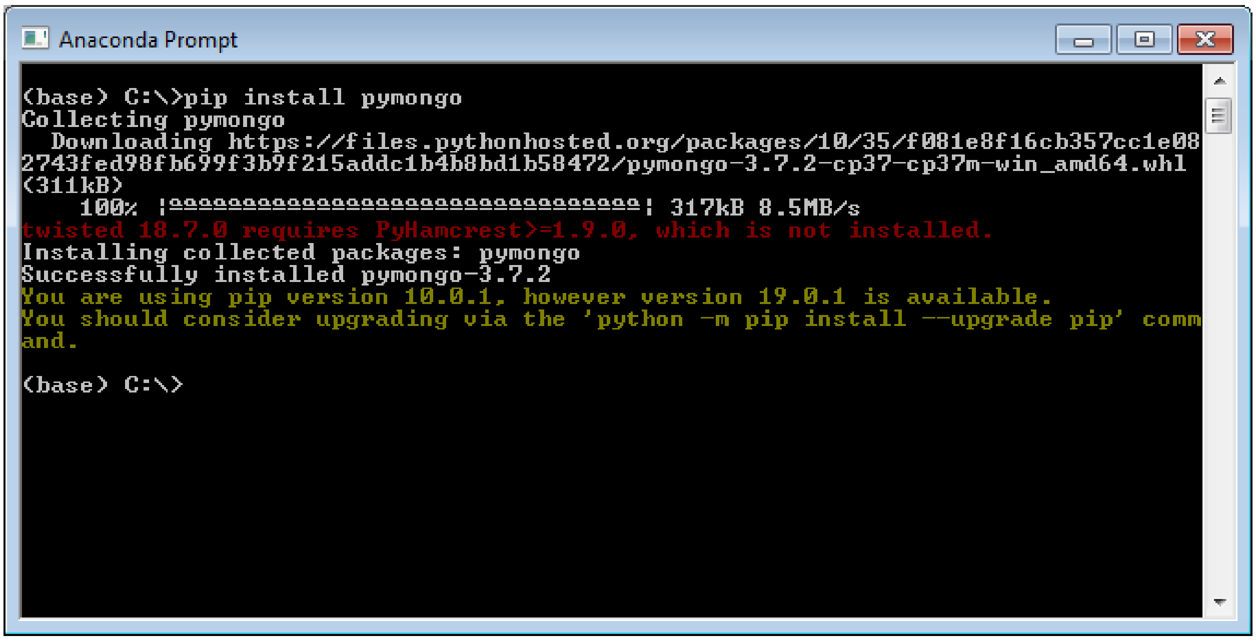


Select the `Anaconda Prompt` option and type in:

pip install pymongo

Note, it may already be setup in some labs.

Which should respond with:

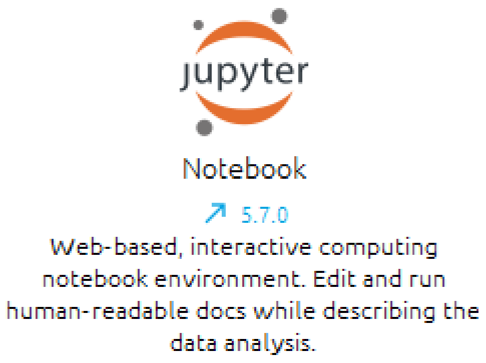


```
(base) C:\>pip install pymongo
Collecting pymongo
  Downloading https://files.pythonhosted.org/packages/10/35/f081e8f16cb357cc1e08
2743fed98fb699f3b9f215addc1b4b8bd1b58472/pymongo-3.7.2-cp37-cp37m-win_amd64.whl
(311kB)
    100% |################################| 317kB 8.5MB/s
twisted 18.7.0 requires PyHamcrest>=1.9.0, which is not installed.
Installing collected packages: pymongo
Successfully installed pymongo-3.7.2
You are using pip version 10.0.1, however version 19.0.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' comm
and.

(base) C:\>
```

Ignore any messages about updating the package.

You can now start a Python Notebook by selecting a `jupyter Notebook` from the the `Anaconda Navigator`:


jupyter
Notebook
5.7.0
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

10.3 Importing the Weather Template

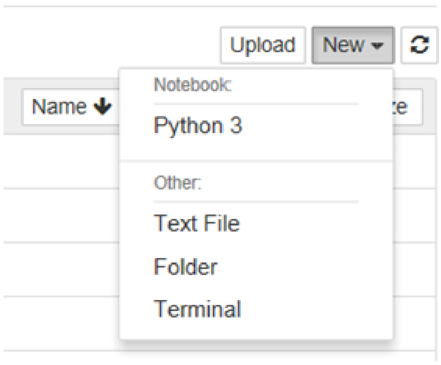
First download the `mongo_Weather_Template.ipynb` file from Canvas and save somewhere accessible.

10.3.1 Anaconda

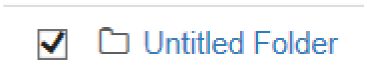
Anacond stores the python notebooks in your User area, for example, if your username is 0123456 they will be stored in:

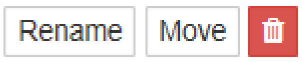
`C:\Users\0123456`

In the Dashboard view of the Notebooks, click on the `New` button and select `Folder` from the options:



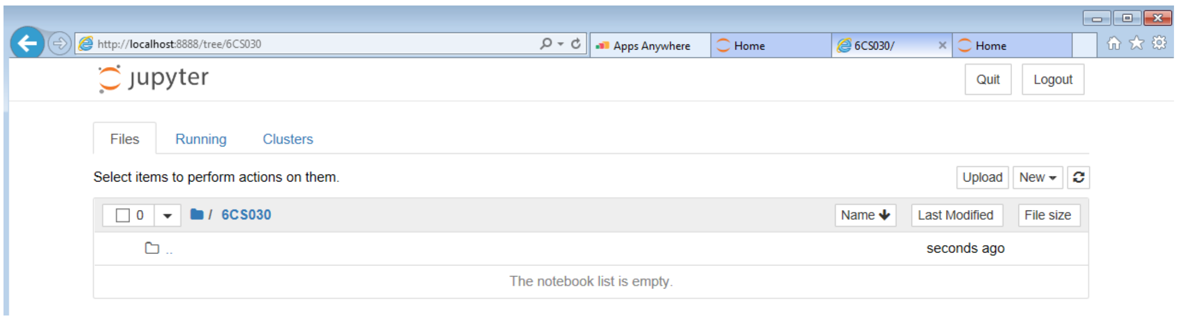
This will create a folder called `Untitled Folder`. Click the box beside:



Then click on the Rename button at the top of the screen: 

and rename the folder to `6CS030`

Click on the folder and it will show an empty directory:



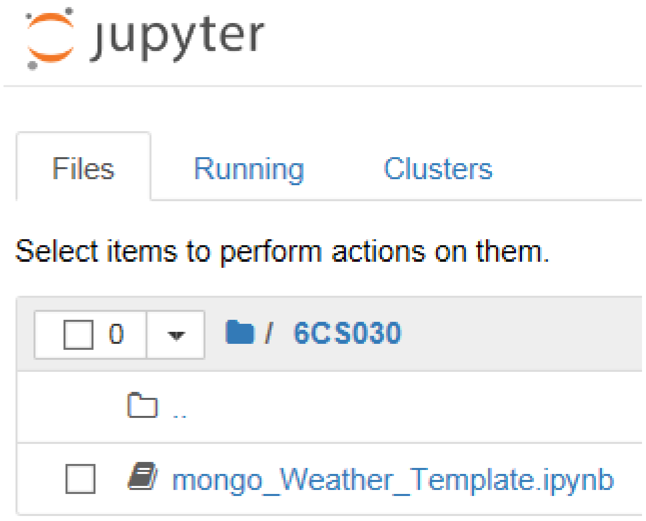
Select items to perform actions on them.

The notebook list is empty.

Use Windows Explorer to copy the `mongo_Weather_Template.ipynb` file into this folder:

C:\Users\myStudentNumber\6CS030

Replace myStudentNumber with your own student number. Your Notebook will now be ready to use:



Select items to perform actions on them.

Rename it to something more appropriate.

It is advisable to take a copy of the Notebook when you have finished and save outside the `c:\Users` directory. For example, keep a copy in your `Documents` folder or on a memory stick.

10.4 Google Colaboratory

You can use the `File>Upload Notebook…` option to load the notebook.

10.5 Python Differences

The syntax is slightly different in Python – see the Notebook for comments.

Some Mongo Shell commands cannot be run in the Notebook, such as:

- the `mongoimport` command seen in Section 1.2,
- the Linux editor (`vi`) seen in Section 6.1.
- updating substrings seen in Section 7.3

Use of this Notebook is optional, though the benefit is viewing and processing the text from the tweets can be easier to understand in the Notebooks and does allow you to add some comments for your own future use.