Lesson 1: Getting Started

Connecting to the server

The first thing we need is a connection to the database. In in order to get that, we need to import pymongo and grab a `Connection()` object:

In [1]:
```
import pymongo
conn = pymongo.Connection()
conn
```

Out[1]:
```
Connection('localhost', 27017)
```

If you are running your MongoDB server on a non-default port, or on a different host, you'll need to specify a connection string.

In [2]:
```
conn = pymongo.Connection('mongodb://localhost:27018')
```

If you are using a password to authenticate to your database, you can pass it along in the connection string as well. Note, however, that the MongoDB wire protocol is *not* encrypted (though the password is), so your traffic is vulnerable to snooping unless you are behind a firewall.

In [3]:
```
pymongo.Connection(
    'mongodb://myuser:mypassword@ds031277.mongolab.com:31277/tutorial-test')
```

Out[3]:
```
Connection('ds031277.mongolab.com', 31277)
```

Getting a database

Once you have a connection reference, you can access the database(s) on that server either by attribute access...

In [4]:
```
db = conn.tutorial
db
```

Out[4]:
```
Database(Connection('localhost', 27017), u'tutorial')
```

or by using square brackets (this is useful when your database name is not valid as a Python identifier):

In [5]:
```
conn['my-tutorial-test']
```

Out[5]:
```
Database(Connection('localhost', 27017), u'my-tutorial-test')
```

Once you have the database, you can see what collections are defined in the database quite simply:

In [6]:
```
db.collection_names()
```

Out[6]:
```
[]
```

Getting a collection

All the data in MongoDB is stored in collections, so obtaining a connection reference is our next step. There's no need for the collection to have been pre-defined in order to be used, as MongoDB will simply create a new, empty collection the first time your reference a name:

```
In [7]: db.my_collection
```

```
Out[7]: Collection(Database(Connection('localhost', 27017), u'tutorial'), u'my_collection')
```

```
In [8]: db['my.dotted.collection.name']
```

```
Out[8]: Collection(Database(Connection('localhost', 27017), u'tutorial'),
        u'my.dotted.collection.name')
```

You can also access collections whose names are not valid python identifiers using the square bracket lookup just like databases:

```
In [9]: db['my collection-name']
```

```
Out[9]: Collection(Database(Connection('localhost', 27017), u'tutorial'), u'my collection-
        name')
```

Creating some documents

MongoDB stores data as *documents* rather than the *rows* you may be used to from relational databases. Documents are stored in a data format called BSON, similar to JSON. All you really need to know when you're using Python, however, is that documents are Python dictionaries that can have strings as keys and can contain various primitive types (`int`, `float`, `unicode`, `datetime`) as well as other documents (Python `dict`s) and arrays (Python `lists`).

To insert some data into MongoDB, all we need to do is create a `dict` and call `.insert()` on the collection object:

```
In [10]: from datetime import datetime
         post = {"author": "Bernie",
                 "text": "My first blog post!",
                 "tags": ["mongodb", "python", "pymongo"],
                 "date": datetime.utcnow()}
         db.posts.insert(post)
```

```
Out[10]: ObjectId('4f56cfaffba5224c71000000')
```

Aside: The _id field and bson.ObjectId

Every document in a regular MongoDB collection contains a unique key called `_id`. Any primitive BSON type can be used for the `_id` field, but most commonly, we will use a `bson.objectid.ObjectId` as our _id. (You can think of `bson.objectid.ObjectId` as filling the same role as an integer primary key with auto-increment that you might use in another database).

When we insert a document that does not specify an `_id` field, pymongo helpfully generates a default `ObjectId` based on the client machine, current timestamp, and a few other factors. What you really need to know about `ObjectId`s so generated is that

- they can be assumed to be globally unique identifiers
- they are generated in (generally) increasing order (the most significant bits of an ObjectId are the current timestamp.

The return value of our `insert()` above is the `_id` value of the newly-created document.

Retrieving our data

Inserting data is all well and good, but how about retrieving some of the data we've inserted? For this, `Collection`s provide the `find()` and `find_one()` methods:

```
In [11]: db.posts.find()
```

```
Out[11]: <pymongo.cursor.Cursor at 0x203b910>
```

```
In [12]: list(db.posts.find())
```

```
Out[12]: [{u'_id': ObjectId('4f56cfaffba5224c71000000'),
           u'author': u'Bernie',
           u'date': datetime.datetime(2012, 3, 7, 3, 2, 7, 595000),
           u'tags': [u'mongodb', u'python', u'pymongo'],
           u'text': u'My first blog post!'}]
```

```
In [13]: db.posts.find_one()
```

```
Out[13]: {u'_id': ObjectId('4f56cfaffba5224c71000000'),
          u'author': u'Bernie',
          u'date': datetime.datetime(2012, 3, 7, 3, 2, 7, 595000),
          u'tags': [u'mongodb', u'python', u'pymongo'],
          u'text': u'My first blog post!'}
```

Exercises

I. From the ipython shell, connect to your local database. Explore some of the methods available on the connection object (listing databases, etc).
II. Get a reference to the `tutorial` database on your local MongoDB. List the collections that exist in this database.
III. Insert a few documents into a collection on your tutorial database. Now look at the collections that exist in your `tutorial` database. Try inserting documents into other collections. Keep listing the collections in your database at each step.
IV. Use `find()` to get a cursor over all the documents in a collection. Iterate over each of them, printing them out. Use `find_one()` to retrieve a single document.