Tutorial

This tutorial is intended as an introduction to working with **MongoDB** and **PyMongo**.

Prerequisites

Before we start, make sure that you have the **PyMongo** distribution installed. In the Python shell, the following should run without raising an exception:

```
>>> import pymongo
```

This tutorial also assumes that a MongoDB instance is running on the default host and port. Assuming you have downloaded and installed MongoDB, you can start it like so:

```
$ mongod
```

Making a Connection with MongoClient

The first step when working with **PyMongo** is to create a `MongoClient` to the running **mongod** instance. Doing so is easy:

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

The above code will connect on the default host and port. We can also specify the host and port explicitly, as follows:

```
>>> client = MongoClient("localhost", 27017)
```

Or use the MongoDB URI format:

```
>>> client = MongoClient("mongodb://localhost:27017/")
```

Getting a Database

A single instance of MongoDB can support multiple independent databases. When working with PyMongo you access databases using attribute style access on `MongoClient` instances:

```
>>> db = client.test_database
```

If your database name is such that using attribute style access won't work (like `test-database`), you can use dictionary style access instead:

```
>>> db = client["test-database"]
```

Getting a Collection

A collection is a group of documents stored in MongoDB, and can be thought of as roughly the equivalent of a table in a relational database. Getting a collection in PyMongo works the same as getting a database:

```
>>> collection = db.test_collection
```

or (using dictionary style access):

```
>>> collection = db["test-collection"]
```

An important note about collections (and databases) in MongoDB is that they are created lazily - none of the above commands have actually performed any operations on the MongoDB server. Collections and databases are created when the first document is inserted into them.

Documents

Data in MongoDB is represented (and stored) using JSON-style documents. In PyMongo we use dictionaries to represent documents. As an example, the following dictionary might be used to represent a blog post:

```
>>> import datetime
>>> post = {
...     "author": "Mike",
...     "text": "My first blog post!",
...     "tags": ["mongodb", "python", "pymongo"],
...     "date": datetime.datetime.now(tz=datetime.timezone.utc),
... }
```

Note that documents can contain native Python types (like `datetime.datetime` instances) which will be automatically converted to and from the appropriate BSON types.

Inserting a Document

To insert a document into a collection we can use the `insert_one()` method:

```
>>> posts = db.posts
>>> post_id = posts.insert_one(post).inserted_id
>>> post_id
ObjectId('...')
```

When a document is inserted a special key, `"_id"`, is automatically added if the document doesn't already contain an `"_id"` key. The value of `"_id"` must be unique across the collection. `insert_one()` returns an instance of `InsertOneResult`. For more information on `"_id"`, see the documentation on _id.

After inserting the first document, the *posts* collection has actually been created on the server. We can verify this by listing all of the collections in our database:

```
>>> db.list_collection_names()
['posts']
```

Getting a Single Document With `find_one()`

The most basic type of query that can be performed in MongoDB is `find_one()`. This method returns a single document matching a query (or `None` if there are no matches). It is useful when you know there is only one matching document, or are only interested in the first match. Here we use `find_one()` to get the first document from the posts collection:

```
>>> import pprint
>>> pprint.pprint(posts.find_one())
{'_id': ObjectId('...'),
 'author': 'Mike',
 'date': datetime.datetime(...),
 'tags': ['mongodb', 'python', 'pymongo'],
 'text': 'My first blog post!'}
```

The result is a dictionary matching the one that we inserted previously.

Note

The returned document contains an `"_id"`, which was automatically added on insert.

`find_one()` also supports querying on specific elements that the resulting document must match. To limit our results to a document with author “Mike” we do:

```
>>> pprint.pprint(posts.find_one({"author": "Mike"}))
{'_id': ObjectId('...'),
 'author': 'Mike',
 'date': datetime.datetime(...),
 'tags': ['mongodb', 'python', 'pymongo'],
 'text': 'My first blog post!'}
```

If we try with a different author, like “Eliot”, we’ll get no result:

```
>>> posts.find_one({"author": "Eliot"})
>>>
```

Querying By ObjectId

We can also find a post by its `_id`, which in our example is an ObjectId:

```
>>> post_id
ObjectId(...)
>>> pprint.pprint(posts.find_one({"_id": post_id}))
{'_id': ObjectId('...'),
 'author': 'Mike',
 'date': datetime.datetime(...),
 'tags': ['mongodb', 'python', 'pymongo'],
 'text': 'My first blog post!'}
```

Note that an ObjectId is not the same as its string representation:

```
>>> post_id_as_str = str(post_id)
>>> posts.find_one({"_id": post_id_as_str})  # No result
>>>
```

A common task in web applications is to get an ObjectId from the request URL and find the matching document. It’s necessary in this case to **convert the ObjectId from a string** before passing it to `find_one`:

```
from bson.objectid import ObjectId

# The web framework gets post_id from the URL and passes it as a string
def get(post_id):
    # Convert from string to ObjectId:
    document = client.db.collection.find_one({'_id': ObjectId(post_id)})
```

See also

When I query for a document by ObjectId in my web application I get no result

Bulk Inserts

In order to make querying a little more interesting, let's insert a few more documents. In addition to inserting a single document, we can also perform *bulk insert* operations, by passing a list as the first argument to `insert_many()`. This will insert each document in the list, sending only a single command to the server:

```
>>> new_posts = [
...     {
...         "author": "Mike",
...         "text": "Another post!",
...         "tags": ["bulk", "insert"],
...         "date": datetime.datetime(2009, 11, 12, 11, 14),
...     },
...     {
...         "author": "Eliot",
...         "title": "MongoDB is fun",
...         "text": "and pretty easy too!",
...         "date": datetime.datetime(2009, 11, 10, 10, 45),
...     },
... ]
>>> result = posts.insert_many(new_posts)
>>> result.inserted_ids
[ObjectId('...'), ObjectId('...')]
```

There are a couple of interesting things to note about this example:

> - The result from `insert_many()` now returns two `ObjectId` instances, one for each inserted document.
> - `new_posts[1]` has a different “shape” than the other posts - there is no `"tags"` field and we’ve added a new field, `"title"`. This is what we mean when we say that MongoDB is *schema-free*.

Querying for More Than One Document

To get more than a single document as the result of a query we use the `find()` method. `find()` returns a `Cursor` instance, which allows us to iterate over all matching documents. For example, we can iterate over every document in the `posts` collection:

```
>>> for post in posts.find():
...     pprint.pprint(post)
...
{'_id': ObjectId('...'),
 'author': 'Mike',
 'date': datetime.datetime(...),
 'tags': ['mongodb', 'python', 'pymongo'],
 'text': 'My first blog post!'}
{'_id': ObjectId('...'),
 'author': 'Mike',
 'date': datetime.datetime(...),
 'tags': ['bulk', 'insert'],
 'text': 'Another post!'}
{'_id': ObjectId('...'),
 'author': 'Eliot',
 'date': datetime.datetime(...),
 'text': 'and pretty easy too!',
 'title': 'MongoDB is fun'}
```

Just like we did with `find_one()`, we can pass a document to `find()` to limit the returned results. Here, we get only those documents whose author is “Mike”:

```
>>> for post in posts.find({"author": "Mike"}):
...     pprint.pprint(post)
...
{'_id': ObjectId('...'),
 'author': 'Mike',
 'date': datetime.datetime(...),
 'tags': ['mongodb', 'python', 'pymongo'],
 'text': 'My first blog post!'}
{'_id': ObjectId('...'),
 'author': 'Mike',
 'date': datetime.datetime(...),
 'tags': ['bulk', 'insert'],
 'text': 'Another post!'}
```

Counting

If we just want to know how many documents match a query we can perform a `count_documents()` operation instead of a full query. We can get a count of all of the documents in a collection:

```
>>> posts.count_documents({})
3
```

or just of those documents that match a specific query:

```
>>> posts.count_documents({"author": "Mike"})
2
```

Range Queries

MongoDB supports many different types of advanced queries. As an example, lets perform a query where we limit results to posts older than a certain date, but also sort the results by author:

```
>>> d = datetime.datetime(2009, 11, 12, 12)
>>> for post in posts.find({"date": {"$lt": d}}).sort("author"):
...     pprint.pprint(post)
...
{'_id': ObjectId('...'),
 'author': 'Eliot',
 'date': datetime.datetime(...),
 'text': 'and pretty easy too!',
 'title': 'MongoDB is fun'}
{'_id': ObjectId('...'),
 'author': 'Mike',
 'date': datetime.datetime(...),
 'tags': ['bulk', 'insert'],
 'text': 'Another post!'}
```

Here we use the special `"$lt"` operator to do a range query, and also call `sort()` to sort the results by author.

Indexing

Adding indexes can help accelerate certain queries and can also add additional functionality to querying and storing documents. In this example, we'll demonstrate how to create a unique index on a key that rejects documents whose value for that key already exists in the index.

First, we'll need to create the index:

```
>>> result = db.profiles.create_index([("user_id", pymongo.ASCENDING)], unique=True)
>>> sorted(list(db.profiles.index_information()))
['_id_', 'user_id_1']
```

Notice that we have two indexes now: one is the index on `_id` that MongoDB creates automatically, and the other is the index on `user_id` we just created.

Now let's set up some user profiles:

```
>>> user_profiles = [{"user_id": 211, "name": "Luke"}, {"user_id": 212, "name": "Ziltoid
>>> result = db.profiles.insert_many(user_profiles)
```

The index prevents us from inserting a document whose `user_id` is already in the collection:

```
>>> new_profile = {"user_id": 213, "name": "Drew"}
>>> duplicate_profile = {"user_id": 212, "name": "Tommy"}
>>> result = db.profiles.insert_one(new_profile)  # This is fine.
>>> result = db.profiles.insert_one(duplicate_profile)
Traceback (most recent call last):
DuplicateKeyError: E11000 duplicate key error index: test_database.profiles.$user_id_1 du
```

See also

The MongoDB documentation on indexes