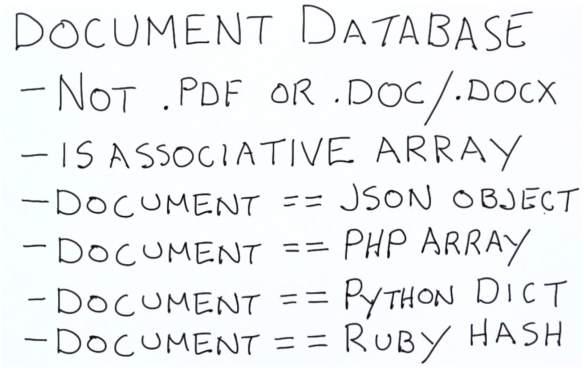
Data Wrangling with MongoDB

# Lesson 4 Notes

## Working With MongoDB

### Intro to MongoDB

The database we'll be working with in this course is: MongoDB. There are a couple of reasons for that. One of which is, since I work for MongoDB it would be bizarre for me to do this using a different database and potentially a career limiting move. But, the primary reason for talking about MongoDB in this particular course is because it makes a lot of sense for data analysis applications. MongoDB is widely used in big data. And, it's the leading no sequel database. Now, while MongoDB is a no sequel database. The term we, we most commonly use is document database. Let me talk a little bit about what we mean by that. We're not talking about a database for storing PDF documents or Microsoft Word documents. What we mean by documents is an associative array. Now, this is the same type of structure that you see in a JSON object. Or a PHP array. A Python dictionary or Ruby hash, and similar data structures in other programming languages. MongoDB allows us to store these types of hierarchical data structures. Directly in the database as individual items or documents. Now the documents stored in mongoDB use a JSON like syntax. I say JSON like because mongoDB is more flexible with the use of quotes and other small syntaxical elements than strict JSON. And yes, it is somewhat oxymoronic to describe anything Javascript related as strict.




### Data Modeling in MongoDB

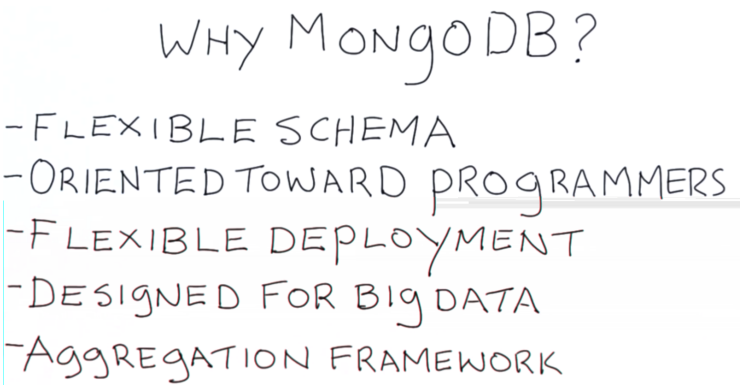
The example we're looking at here, is an automobile info box, in particular for the Tesla Model S. In this lesson, we're going to use info box data throughout our examples, involving Mongo DB. So, let's take a look at the data that's represented in this info box. Manufacturer, we have production years,

model years. Several different assembly plants, a designer, class, body style, with 2 elements, 5 door and liftback, and a number of other fields. So what I'd like to do here, is talk about how we might model this data in MongoDB, as documents using the JSON like syntax I mentioned earlier. So in modeling this data, let's start with some of the scalar valued fields that are here. Here we can see scalar fields for manufacturer, class, and body style. We can also model a couple of array fields here for production, model years, and layout. So here, in our Json like syntax, for MongoDB, we've modeled that same data, using array value fields. Now, we could model the designer field, as just a single string. But to show an alternative, we might also model it as a simple person document, as follows. And here we have a document with a designer field that has a nested document within it, for the designer. And last, putting the idea of array fields and embedded documents together. We might represent the assembly data in this example, like this, with an assembly field that has an array, containing embedded documents for each of the assembly plants. So, in Mongo DB, our entire document for the Tesla Model S might look something like this. With a mixture of fields that are scalars, array valued and embedded documents. One of the many cool things about this, is that Mongo DB natively supports a very common big data format, that being JSON. And by support I mean, that we can store data items structured this way in our database and do queries that would say, for example, identify all documents where the city of assembly is Fremont. Driving down the hierarchy, to actually pull out matches for this piece of data, even though it's nested several layers deep within an individual document.

## Why MongoDB

Okay, so let's talk a little bit more about why we would use MongoDB. One of the most important reasons to use MongoDB is because it has a flexible schema. This enables us to more easily handle data in flat formats, such as CSV, and also hierarchical data, where individual fields Are composed of multiple elements. MongoDB is also oriented toward programmers. One way we see this is that the type of JSON documents stored in MongoDB map directly onto familiar data types found in most popular programming languages. Such as the examples we saw; php arrays, Python dictionaries, etcetera. Another place we see this is in the MongoDB Client Libraries, or drivers as we call them. MongoDB supports drivers for most popular programming languages. These drivers handle the job of translating the data to and from native data types within the particular language as needed by the application, and also support a number of other features That make it easy to develop applications using MongoDB. Another feature that makes MongoDB great for data scientists is your ability to deploy MongoDB in a variety of ways.

So for example in just a couple of minutes you can download install and run MongoDB on your laptop and quickly begin developing an application on your laptop. Or you can deploy MongoDB on multiple servers. With several demons running to support a big data application. And, of course, there are lots of options in between these two. MongoDB is designed for big data. It's highly scalable horizontally, on commodity hardware, and includes native support for Map Reduce. And it includes the aggregation framework. That enables efficient analytics applications. Okay so enough chatter about MongoDB, let's dive in and actually start using it.




## Flexible Schema

As so often the case with our data, some entries or documents will have fields that other do not. In any project, a data model usually goes through several iterations. MongoDB was designed to address both of these issues by providing a flexible schema that deals well with both individual documents that vary in the fields they contain as well as the schema for our entire collection needing to change. Let's take a look at person info box data, as an example.

| | |
|---|---|
| **Born** | Rolihlahla Mandela<br>18 July 1918<br>Mvezo, Cape Province, Union of South Africa |
| **Died** | 5 December 2013 (aged 95)<br>Johannesburg, South Africa |

Now for nearly everyone, it probably makes sense to include fields for birth and death dates. Maybe nationality and even profession, but not everyone will have held office, and not everyone is associated with a political party. And even if we're talking about people who are not famous, some people will have spouses. Some will have more than one. And others won't. Some will have children and some will not. Leaving aside the question of what is the right data model for person data. In MongoDB we can represent each person using the fields that are appropriate to them even if some person documents contain fields that others don't have. MongoDB's indexing system and query execution system take this into account. So we can query a people collection for people with two or more children, and it will work as expected retrieving only data for people that have two or more entries in the array that serves as the value for the children field. Ignoring documents with fewer As well as documents that don't

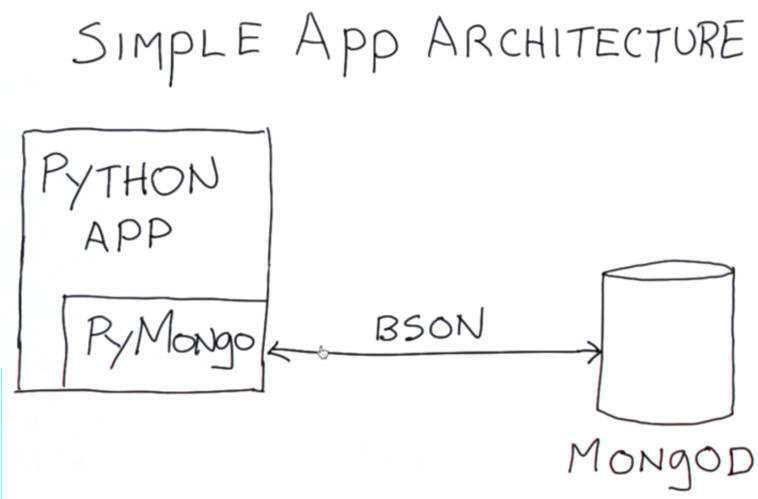
have a children field at all. What this also means is that it's easy to evolve your scheme as new needs emerge or more data becomes available. It's a simple matter to begin adding documents to a collection that have new fields you now want to track or to change the way you model existing fields. As an example of this, let's take a look at the dbpedia page, that describes the data sets that are available. Now, if I scroll down this page. And, I've already done that here, we have an example for city data. So, so far we've looked at automobile data, personal data and now, city data, from the info-box data set. And, I'm showing this example to illustrate the fact that the scheme for city info boxes has evolved. And, we can see that in this old example for Innsbruck. And comparing it to the new example for Innsbruck and if you look through this data there are a couple of places where the data has changed. There's no mayor listed here as there is here and while there's no time zone listed in the old form of the data. The time zone for central and daylight savings time are listed here. Small differences, but these are the type of subtle changes that we would expect to see in a schema that is evolving.

| | |
|---|---|
| **Spouse(s)** | Evelyn Ntoko Mase<br>(m. 1944–1957; divorced)<br>Winnie Madikizela<br>(m. 1958–1996; divorced)<br>Graça Machel<br>(m. 1998–2013; his death) |
| **Children** | Thembekile Mandela<br>Makaziwe Mandela<br>Makgatho Mandela<br>Makaziwe Mandela<br>Zenani Mandela |

## Intro to PyMongo

Okay, let's talk about PyMongo. Now PyMongo is one of the MongoDB drivers. Or client libraries. And because Python is the programming language for this course, we'll be using the Python driver. You'll find that you have access to PyMongo in the IDE. Now one thing to keep in mind about PyMongo or any other MongoDB driver is that, their basic job is to maintain a connection to the database and to allow you to work in your language of choice and interact with your data in a very natural way. Now if you'd like to know about drivers for other languages I'd encourage you to visit api.mongodb.org where you can read about all supported drivers. Now before we dive in and look at an example I want to talk a little bit about the application architecture, just so you have that picture in your mind as we're working through the rest of the material in this lesson. Okay, so if you download MongoDB and run it on your local computer This picture basically represents the architecture. You'll start Mongo DB, and the demon that's running is Mongo D. And then you'll create a Python application that includes the PyMongo module. And using the PyMongo module, you can send requests, and receive responses from Mongo DB. PyMongo

communicates with the database using what we call the wire protocol, and the data that's exchanged is in a format we call BSON. BSON is a binary encoding for JSON. If you Google for BSON, you'll easily find the spec. Okay, so let's look at an app. Okay. Here I'm going to use that example that we say before for the Tesla S but let's talk about the PyMongo components of this particular application. Here we're inputting the MongoClient class from the pymongo module. And then we're going to create a client object and note that here, we specify the connection string. We won't go into too much detail into this but essentially this will allow us to connect to Any MongoD instance that we have access to. Okay, here's the encoding of the Tesla as a Python dictionary, and then here what we're doing is specifying that we want to use the examples database, and what we want to do here is insert this document here. Finally, we're going to do a find query, and then for every document that we get back We're simply going to print that document out. Okay, now before we run this, I want to point out two things, one is that we're using the insert command in order to store this document in the autos collection for the examples database. Here we're using the find command, and find, when run this way, will simply give us back a cursor for all of the documents in, in this case, the autos collection. Whatever collection we specify here. Okay? And we'll go into more detail on these, later on in this lesson. So, let's run this. Okay, and we can see that what we got back and printed out was the one and only document that we had inserted into this collection. Brand new collection. When we inserted the document, that became the first document in the collection. And one thing I want to point out is that in addition to all of the fields we specified in our code, there is a new field, underscore ID that has this object ID value. Okay? That's a value that was added by the database and we'll talk a little bit later about exactly what this value represents. The short story is that MongoDB ensures that any document we insert can be uniquely identified by it's underscore id field. And if we don't specify a value for underscore id, mongoDB will create one for us.




## Querying Using Field Selection

Now queries in Mongo DB are based on the document model as well. In order to find items of interest, we want to specify a document that resembles those we're looking for. So let's take a look at an example with the autos data set. If we want to find all autos that have a manufacturer of Toyota We construct a query document like this. The general principle here is that we construct the query document, having the fields and values for those fields that we'd like to see in every document in our results set. One additional thing I'll point out here is, that we're simply looping through our results

and printing out each one of them. Okay let's run this. Okay, so we can see that in the documents we got back, we have a manufacturer value of Toyota. And if we were to scroll through this entire result set, we'd see that all of them in fact do.

## Multiple Field Queries

So we just looked at how we do simple queries in MongoDB. If we to more precisely specify the form of documents we're looking for, we can simply add additional fields to our query document. Here's an example building on the last one. So, again,manufacturer Toyota. In this case though, we're also going to specify that we want to see documents where the class field has been set to mid-size car. And let's run this. Okay, now in this case, we can see manufacturer and mid-size car have the correct values for the documents that have been retrieved. And again, if we were to scroll through, we'd see that for every document.

## Projection Queries

Okay, one last thing I'd like to look at before we move on to some more advanced query topics. Is the ability to specify a projection document in addition to our query document for calls to find. Now, what projection does is it essentially describes the shape we would like documents to take in the result set. I'm also going to break this out a little bit here with the query and projection variable, and pass those to find, instead of directly passing the dictionary that we constructed to find. I point this out merely because we'll see this later on in the lesson, and I don't want anyone to be confused. Now, let's imagine that instead of getting back all of the documents for our query specifying constraints on manufacture and class, we actually are only interested in getting the name of the document back. Well, we could simply print out the name down here. But another way we can do that is by specifying a projection as a second parameter to find. And what that's going to do is say, all of the result documents that come back, instead of having their full contents, I only want to see name. No, by default, unless we explicitly say, I don't want to see the ID field, we'll also get ID. So that's why I've got ID set to zero here. Okay? So the way we do this, is, essentially, we specify all

the fields for the documents in this collection. And then specify whether or not we want them to occur in a result set. Alright? Let's run this. Alright and here we can see that the documents we get back simply have the name field in them.

```
from pymongo import MongoClient
import pprint

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

db = client.examples

def find():
    query = { "manufacturer" : "Toyota" , "class": "mid-size ca
    projection = {"_id" : 0, "name" : 1}
    autos = db.autos.find(query, projection)

    for a in autos:
        pprint.pprint(a)

if __name__ == '__main__':
    find()
```

## Getting Data into MongoDB

Now, after we've gone through and cleaned our data, we're at a place where we ne queries to MongoDB and you've done some exercis collections. Now, the database we've been working looked at a couple of different collections here. We done some queries against the cities collection. No the script that I wrote to clean the automobiles dat would simply output JSON documents that I could then import into MongoDB. And that's actually the, the strategy tha for you to have an understanding of inserting documents Into MongoDB. And so, we'll take this opportunity to show you a little bit about how to do that. Now, there's a lot about the insert statement that I'm not going to cover right here. But, this is the simplest form of using insert. Now, what we're going to do is simply loop through all of the autos that I've created up here in the code. Where I've cleaned the data and essentially created collection. Then we're going the insert statement for each into a JSON encoding and se couple of print statements. C do this series of inserts and c using a command we haven'

```
$ python projection_query.py
{u'name': u'Toyota Prius'}
{u'name': u'Toyota Camry'}
{u'name': u'Toyota Cressida'}
{u'name': [u'Toyota Camry Solara', u'Toyota Solara']}
{u'name': [u'Toyota Highlander/Toyota Kluger', u'Toyota Kluger']}
{u'name': u'Toyota Crown'}
{u'name': [u'Reiz', u'Toyota Mark X (X120)']}
{u'name': [u'TRD Aurion', u'Toyota Aurion (XV40)', u'Toyota Camry']}
{u'name': [u'Land Cruiser Prado', u'Lexus GX (USA)']}
{u'name': u'Toyota Venza'}
{u'name': u'Mark X ZiO (ANA10)'}
{u'name': u'Prius Plug-In Hybrid Concept'}
```

```
    if model_years:
        auto['modelYears'] = model_years
    autos.append(auto)

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

db = client.examples

num_autos = db.myautos.find().count()
print "num_autos before:", num_autos

for a in autos:
    db.myautos.insert(a)

num_autos = db.myautos.find().count()
print "num_autos after", num_autos
```

amry']}

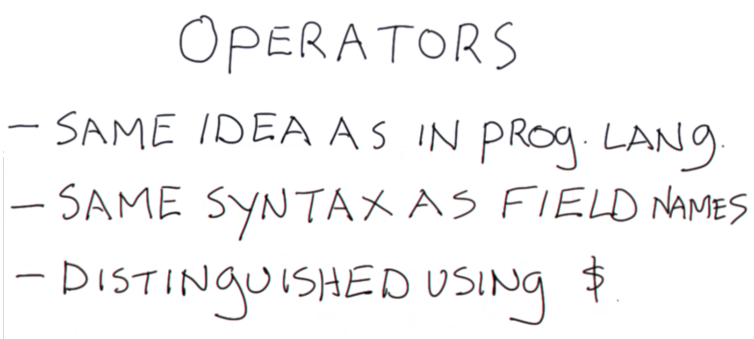
tells us how many documents were returned for this find command. Okay nov went ahead and inserted all of those document collection, from the Pyeadio for autos. Then what I di collection, before I did the inserts and got a cou

## Using mongoimport

Now we just looked at how to use the insert command from the PyMongo client in order to get documents into our MongoDB database. A better way to do this in my opinion, is to output all of our documents as a Json file. And that's what I did here. So, instead of our Python cleaning program, actually putting data into MongoDB directly. We simply separate the two concerns, the task of cleaning the data from the task of getting it into mongoDB. And the intermediary we use is a file that has one Json document per line. Now we can bulk import this data into mongoDB using the mongo import system command. I'll show you a very common use for mongo import here. See the instructor notes for the link to the mongo import documentation. Okay, mongoimport has a number of command-line parameters associated with it, and we can see those by typing --help. Okay, another way to get documentation on mongoimport. Okay, so let's use mongoimport to get our auto's data into a collection. First, I'm going to specify the database, and we'll continue using the examples database that we've been using all along. And then I'm going to specify the collection into which this data should be stored. In this case I'm going to store it in another collection, myautos2. And then finally, I need to specify the file that I actually want to import. That file is in the same directory, and it's called autos.json. Now if I run this, we get an, a nice output message telling us that we imported those same documents we saw before when we were using the Insert command in order to get data into MongoDB. So now we've looked at two different ways to get data into MongoDB. And there are others.

## Operators

In many situations, we need to match based on inexact criteria, such as all people above a certain age, or all cities with populations greater than some number. MongoDB provides a variety of operator to solve these and other types of problems. So the idea with MongoDB operators is very similar to what you find in programming languages, where you have comparison operators, inequality operators, et cetera. MongoDB operators use the same syntax as field names. And we distinguish operators from field names using the dollar sign character. So, all operators have a leading dollar sign.




## Range Queries

So let's take a look at the use of operators to support Range Queries. Let me show you what I mean. MongoDB provides several inequality operators that enable us to perform this type of query. The inequality operators we may use include greater than, less than, greater than or equal to, less than or

equal to, or not equal to. There are also a couple that are designed for querying array fields. We'll talk about those in a later lesson. Now let's look at some queries against our cities collection that provide examples of using inequality operators. Okay. So imagine that we want to query for all cities with the population greater than a quarter million. Instead of specifying an exact number here, as the value for our query document, which is going to be passed to find here, or instead going to use the greater than operator and the value of population in a query document it's going to be a sub document using that operator. We can specify a more constrained range for a query like this using an additional operator. Now this example is very similar to what we just looked at, the only difference being that in the value for population in our query document we're specifying an additional operator of less than or equal to 500,000. Thus defining a more constrained range for this query here. Now this type of range based query works across many different data types. We can do something similar for strings. So here, the semantics of this range are give us all city names that begin with x. Let's go ahead and run this code. And here we can see that in the results we get back. Looks like we're getting back cities that just begin with the letter X. Total match of 22 cities out of the 20,000 or so that are in this collection. Okay, let's look at our code again. We can also do this type of query against dates. Here we're specifying a Python date as the lower and upper bound, and so here we're looking for all cities that have a founding date sometime in the year 1837.

```
11      query = {"foundingDate" : {"$gt" : datetime(1837, 1, 1),
12                                 "$lte" : datetime(1837, 12, 31)}
13      cities = db.cities.find(query)
14
15      num_cities = 0
16      for c in cities:
17          pprint.pprint(c)
18          num_cities += 1
19
20      print "\nNumber of cities matching: %d\n" % num_cities
```

And actually, to be precise here, we should really have this be greater than or equal to. Okay, now let's run this. Okay, and we get three cities that match. We can see that the founding date here seems to be within range, as it is here, and also here, for the city of Chicago. Finally, we can also do things like use the not equals operator to for example in this case, find all cities with a country that's not equal to the United States.

## Exists

Okay, let's look at a couple more operators. I'm going to switch over and use the Mongo shell for some of this. This is a console application that ships with the database, and it provides a great interface for exploring MongoDB's query language. So, the first thing that I need to do is switch into the right database which I've already done here with use examples. Now I'm just going to do a find against the whole collection and we see a whole bunch of documents scroll by. Okay, so as we've discussed, some documents will contain fields that others do not. In our cities' collection, for many cities,

especially those that are less important or simply smaller, much of the detail is missing. For example, a small town in the U.S. state of Iowa where I used to live is going to have much less detail than let's say Berlin, London, or other major cities. Mongo DB allows you to query based on the structure of information that's in the documents, as well as their values. So let's take a look at an example. So I happen to know that relatively few documents have a field for government type. It's useful to know what type of government a particular city has, but it's only the larger cities that tend to have this field. MongoDB provides an operator called Exists that allows us to retrieve documents based on whether or not documents contain a particular field. And the value for Exists is whether we want documents that contain the field, or those that don't. In this case, I want to find documents that do have a government type field. Oop and you see what happened there is I forgot to close the query document. So, let's go back and do that, and there we go. And let's do a count here. And of the 20,000 documents in this collection, a relatively small number contain this field. We can do the inverse query and see we get a much larger number. Let's take a look at one of these documents. Here I'm going to use pretty to make it more human readable. Okay, so let's go back to the documents that actually contain this field. And again I'll make em pretty. Okay, so this happens to be the city of Lansing, Michigan, and you can see there's a quite a bit of detail here. Okay. Now, what we can do is contrast this with documents that don't have this field, and it just happens to be the case that this document is conveniently, for example purposes, much more sparse. So the Exists Operator allows us to do queries and asks some questions about the structure of a document.

## Regex Operator

Earlier we looked at querying string valued fields using the greater than and less than operators. Returning to our consideration of queries matching string fields. There are more complicated sorts of queries that you might want to perform on strings than just equality and inequality. For example, we often want to look for patterns in strings. MongoDB supports querying for string patterns using the regex operator. Regex is based on a regular expression library. In particular, the Perl Compatible Regular Expressions library. For more information, I'd encourage you to just Google for this. So the regex operator allows us to do regular expression queries in MongoDB. Let's look at some examples. Now in this collection, city documents, contain a motto field or at least some of them do. So as an example of doing regex queries in MongoDB, let's take a look at some of the words these mottos contain. This is a very simple regular expression. In fact, we don't even need regex to do this one. So if I do the query this way, I should match only documents where friendship is the entire string of the motto. And here we go, friendship. Okay, so now let's begin to expand this regular expression. What I'm going to do first is introduce a very simple change. Now we're looking for all mottos that contain the word friendship. And friendship can either be capitalized or not. And so here we have four, where before we have simply one document. Okay so, now let's expand this a little bit more. And if your regex skills are a little rusty, please see the instructor notes for documentation and pointers to some tutorials. So let's expand this to include another word. Okay, so this regular expression will identify all documents containing a motto with either the word friendship or the word pride. And either word

can be capitalized. Let's take a look at the actual models themselves. And again, I'm going to use a projection to make it a little bit easier to see the mottos. Okay and we can see that each one of these has either the word pride or the word friendship in it. And capitalization doesn't matter. Okay so, that's about all the farther I want to go with regex queries in this example. The point here is that MongoDB supports Perl Compatible Regular Expressions. So, you have a lot of power in what types of queries you can do involving string value fields.

## Querying Arrays Using Scalars

One of the most powerful features of MongoDB is the ability to query against fields that are not simply scalar values such as strings or integers. But those that are themselves structured data such as a arrays. As we know, MongoBD can have rate values and you see a couple of examples, right here on the screen. We can query within arrays with a couple of different ways. Now I think the best example from the Autos collection actually has to do with model years. So let's take a look at that. Now the simplest example is to provide a single value for model years in our query document. Let's see what happens. So in this case we can see that our model years here, for this particular vehicle are is an array, this value will match. MongoDB searches inside array value fields, for individual values to match. And by building an index on a fields such as this, those queries will be very efficient.

## In Operator

Okay, so we just looked at using a scalar value in our query document for an array value field. Let's look at another way we can query arrays in MongoDB. In this case we're going to introduce the IN operator. Let's take a look at an example, and then we'll talk about it. Okay, so what happened there? IN allows us to specify an array of values. In this query then, we'll retrieve all documents for which the model years field contains any of the values in this array here. We can use IN even if the field we're querying against is not an array valued field. Thus giving us a number of optional values that will match documents in our result set. So we can see as I add additional values to this array of options, that our count gets larger and larger. Because we have a larger and larger target that we can hit. K, so let's take a look at the actual documents that are retrieved her. K, and if we scroll up, we can see that in fact this document matches, as do the rest. All of them containing at least one of the values in their model year that we specified.

## All Operator

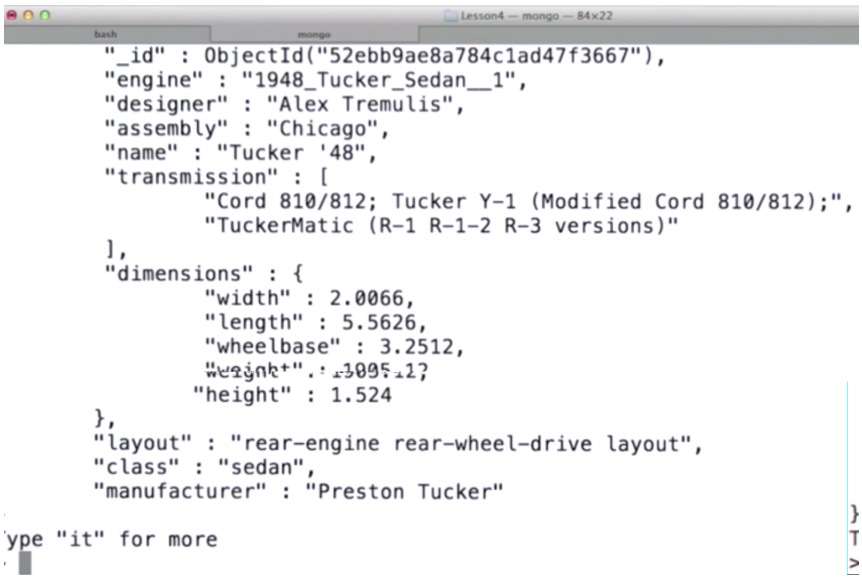
Okay and finally, one last way in which we're going to look at queries involving arrays. So far we've looked at querying array-valued fields. We've also looked at supplying a number of values, using the" in" operator, any of which can be found in a field, and retrieved documents for which the value of this field matches any of the values supplied here. We're going to look at the inverse of" in," and that

is the all operator. So if you remember, with in, the more values we add, the larger our query target gets. Always the inverse. In this case, we're specifying an increasingly narrow target. That is to say, the model uris field, for a document must contain all of these values, in order to be retrieved by this query. Let's take a look at an actual result set. Okay, and we can see that here, all of our values are found in this document. As with this one, and this one, so on. So that's the all operator.

## Programming Quiz: Dot Notation

Okay, now that we've learned about arrays, let's look at some other ways that queries can get inside of documents in MongoDB. You know that our automobile collection contains fields with some nested documents. For example, there's a dimensions field that contains entries for weight, wheelbase, et cetera. In MongoDB, we can query for values inside nested documents. Using a dot notation syntax. Here's an example. Here I've written a query that will retrieve all autos with the weight greater than 5000. Now note that I'm digging into the dimensions field using this dot notation and accessing the weight field of this dimension sub-document. So let's take a look at the results. And what I'm going to do here, is just get a count. Okay. Now these weights are in kilograms. So for my fellow Americans, a kilogram is a little more than two pounds. What we're going to do here is, see what happens as we increase the weight here. Okay.

So we're done to about seven documents. Let's go ahead and take a look at those. I'll just pretty print this. Now note that one here is the Terex Titan. It weighs more than 500,000 kilos. This one caught my eye because, having two boys, we're all about the big dump trucks. Okay, let's take a look at an example from a different data set. Now this is one of my favorite features of MongoDB. It's a powerful way to dig into your data and really get a good idea for what it contains. As data scientists, especially when we're thinking about data wrangling and data cleaning, this is especially useful as you're auditing your data. We haven't talked about it yet, but it's important to audit your data. After you get it into the database, as well. So, looking forward to the next lesson where we'll be using the Twitter data set, let's have a little preview here. Now in order to do this, I need to access the Twitter collection which is in the same example as database. Okay, let's see what documents in this



collection look like. In particular, I want to draw your attention to this entity's field for each Tweet and the hashtags field of this entity's sub-document. Hashtags is where we've pulled out essentially any hashtags that are used in this particular tweet. Okay, so let's do a little mongoDB magic here. And, what I'm going to do is set up a query where we are also going to do a projection. And I'm going to do all this using dot notation. Okay, so for our projection here, what I want to do is pull out just the hashtag itself. And, I'm going to make sure that that underscore id field doesn't appear just so the results are a little easier to read. And, here we go.

So, we'll go back in just a second and take a look at the actual documents. But, what I did here with this query was find all tweets where the hashtags value for the entities sub-document was not equal to an empty array. Because the way this data set is set up, if there are no hashtags, the value is set to the empty array. Then what I've done is I've said, all right, instead of seeing the entire document for each document in the result set, I want to see just the text value for each of the hashtags. In a hashtags array. Okay and note here that for those tweets that actually have multiple hashtags in them, I'm going to get the text value for all of those. Okay now, let's get rid of the projection and see what this type of document looks like as a whole. And now you can see that for hashtags, we actually have an array. And so for every hashtag that's mentioned will be enclosed as a sub-document like this. With a field value for text and one for indices. These being the character position, where this begins and ends in the tweet. Okay? So what I've done with this query Is use dot notation, not just to find documents of interest to me, but also to project out exactly what values from those documents I want to see. And one more time, that's accessing the entities field, then the hashtags subfield and finally text subfield for all documents in this hashtags array.

## Updates

Okay, so we've looked at a number of different ways of interacting with MongoDB collections. We've looked at, how we go about inserting documents into the collection and a number of different ways of querying a collection to find documents of interest. We're now going to turn our attention to updating. And by that, I mean making modifications to existing documents in a collection. There are several ways to do this in MongoDB. The one that we're going to look at here, is the use of the save command. Now in PyMongo, this is actually a method on collections objects. Save does one or two things. If the object we pass to it already has an underscore ID and a document with that underscore ID is already in the collection. Then it simply replaces the document with that ID with this one. If there is no document with that underscore ID or the object that we're passing does not have an underscore ID field, then MongoDB will create a new document for us. Okay, so what we're doing here is, querying the cities collection for the city Munich or Munchen. And in this case, we'll get back a single document. And the reason why we're guaranteed to get back a single document, is because instead of using find, we're using the find_one command. Instead of returning a cursor as find does, find_one simply returns towards the first document it finds, that matches our query. So what we're doing in this code, is modifying this city document, so that it has a field for iso Country Code and value

for the appropriate three-character iso Country Code for Germany, which is DEU. Okay? When we call save here, this document will be updated to include this field. Let's go ahead and run this. Okay, and now what I'm going to do is, I'm going to check that the document was updated appropriately. Here we can see the document before we did the update. I did a find on it earlier. And now we see this iso Country Code field, showing up in the document. So what I'm going to do here is look at a few different examples, run them and then test the results of having run them, in the MongoDB shell.

## Set Unset

OK, so we just looked at a way of modifying a document using the save method. Now let's look at an alternative way of updating documents. What I want to introduce you to here, are two things. One, the update command or method in pymongo it's a method of collection objects. And the set operator. Let's talk about update first. Update expects a query document as its first parameter, and then as its second parameter, it expects an update document. And that update document specifies what operations MongoDB should perform on the document matching this query. Now, by default, update operates on just one document. The first one it finds matching the query document. The semantics of set are, if this document does not already contain the fields specified here, then what should happen is that field should be added with this value. If the document already contains this field, then that field will be updated to the value supplied. Okay, now before I run this code, what I'd like to do is introduce you to the inverse operator for set, which is unset. Okay, so these two pieces of code are almost exactly the same. The only differences are here and here. The semantics of unset are, for whatever document matches this query, if it has the field specified here, remove it. This value is ignored. Now, if this document does not already have this field, then this call has no effect. Okay, so now we've looked at the set operator and the unset operator, and their use with the update command. Now, let's go and run these two programs. And see what their effect is. But first, I want to query for this document in the Mongo shell to look at its current state, and then we can see how running those two programs changes its current state. So I'm going to run a find one command here. Now, this is exactly the same command as we saw in the Python programs we were looking at. The only difference is in the name. And the names are different merely to match the conventions of the two different programming languages represented. In this case, JavaScript because the language of the Mongo Shell is JavaScript. And, in the other case, Python, because we're using the Pymongo

driver. Okay?

So, going to issue this query. And, I want to point out that the Iso Code is in fact present. Now let's run the unset example and then go back to the Mongo shell and issue our query again. Okay? And note that the iso country code has disappeared because we just ran this command. Okay, now let's run our set example. And again remember this simply uses the set operator. Let's do our query again in the Mongo shell and note that the iso country code is now in our document again. Okay, so now we've looked at two different commands that allow us to modify documents. The save command, and the update command. And we've looked at two different operators being used in conjunction with the update command. Okay, now I feel like it's necessary before we move on to just give you a heads up. It's really easy to forget to put an operator here. Okay? If you issue your command this way, what's going to happen is this entire document, the document matching this query, will be replaced so that it contains the underscore ID field and this field only. So, essentially, the document will be composed entirely of a single field. So, always take care to use update correctly.


## Multi Update

Okay. The last thing I want to talk about with regard to update, is how to update multiple documents at once. There are many situations in which we want to do some sort of global modification to all documents matching a certain set of criteria. So, here we selected a single document, and updated its country code to DEU. And updated it so that it included this country code. But this is a country code, not a city code. And we could go into why we would want to do this. This is definitely a denormalization of this data, if you're familiar with that term. Essentially where we headed here with this example, is that the country code will be replicated in every single document that represents the city in Germany. In Mongo DB, we design our document schemas so that they match the access patterns. So that ideally, in order to satisfy any one query, you only have to hit the database once. Therefore the normalization makes sense. It's a way of getting really hyper formats and being able to scale really well, your database. Okay, pardon the slight digression, but I assume that for at least a few of you, the normalized versus denormalized data question was coming to mind. Okay, back to

the topic at hand. In this case, what I'd like to do, is use update to query for documents whose country is Germany. And what I'd like to do is set the iso Country Code field for all such documents to this value. The way we do that using update, is by specifying a 3rd parameter, multi equals true. Now by default again, update will modify just the first document that it finds. So, in order to get it to modify all documents matching this query, we need to specify this parameter. Okay, so let's run this. Okay, and then, we can go back to the Mongo shell and do a query. Now, this time, I need to modify my query so that we're looking for all cities with the country field of Germany. And because we want them all, I need to change this back to a Find, instead FindOne. Right? And then I want to pretty print these, and here we go. Here is Chemnitz, hope I'm pronouncing that properly. And it has the correct I.S.O. code, and here we have another city, with the correct country code and so on as we move through this data. Okay, so that's multiple update and of course we could use other operators besides just set in this same way.

```
        "country" : "Germany",
        "elevation" : 519,
        "homepage" : [
                "http://www.muenchen.de/"
        ],
        "isoCountryCode" : "DEU",
        "lat" : 48.1333,
        "leaderName" : "Christian Ude",
        "leaderTitle" : "Oberbürgermeister",
        "lon" : 11.5667,
        "name" : "München",
        "population" : 1420000,
        "postalCode" : "80331–81929"
}
> db.cities.find({ "country" : "Germany" })
```

## Removing Documents

Okay so we know how to get data into a MongoDB collection. And do queries against that collection. Now I would like to talk about briefly, is how to remove documents from a collection where that's necessary. So the syntax for removing documents is actually quite similar to the syntax for finding documents. So with find if we want to return all the documents in the collection we simply don't express any parameters to the find function. The same is true with remove if I were to execute this query it would remove one by one all of the cities from this collection. A more efficient way of doing that would simply be to call drop which would remove the entire collection and any meta data associated with it. Such as indexes. Now, we can also remove individual documents or documents matching particular criteria. So, for example, I could remove an individual document with the name matching the value "Chicago." So, let me first do a find and there we see that single city. And now, a remove. And then to find again, and sure enough, that city's gone from this collection. Now let's look at a query that touches more than one document. So in this case, what I want to do is identify all documents that don't actually have a name. Okay, you can

```
1  # -*- coding: utf-8 -*-
2
3  from pymongo import MongoClient
4  import pprint
5
6  client = MongoClient("mongodb://localhost:27017")
7
8  db = client.examples
9
10 def main():
11     db.cities.update({ "country" : "Germany" },
12                      { "$set" : {
13                            "isoCountryCode" : "DEU"
14                        }
15                      },
16                      multi=True)
17
```

see there are several returned, in fact 210 documents in this city's collection may seem odd. Don't actually have a name. This is a good example here. We've got a country, but no name for the individual city. Same is true with the rest of these returned. Now, as part of a cleaning pass, for example, we might want to remove all cities from this collection after they've been put into the database, all cities that don't actually have a value for the name field. I can do that by sending this same query document to the remove function. So now, I've removed all of those cities that have no name and if I do a find again I can see that there are now in fact no cites that are missing a name field, so the takeaway here is that remove works in a way very similar to find, you can specify the document to be removed using a query document that has the same syntax and operators available as does the find function.

## Conclusion

Okay so in this lesson, we talked about the mongoDB query language. We looked at putting documents into mongoDB and getting them out using a variety of different operations and operators. In the next lesson, we'll begin to look at doing some analysis in mongoDB using the aggregation framework.