Table of Contents

The Complete Python Web Developer e-book

Although the name "The Complete Python Web Developer" may sound high-handed, I am extremely proud of the work that has gone and consistently goes into the course, and I encourage my students to tell me if the course does not meet the named expectations. In those cases, I always add more content and clearer explanations to ensure that it is, indeed, "The Complete Python Web Developer Course".

This book is meant to go along the course to help in understanding, and to provide a written reference that students can print out or store in their devices for reading.

If you do not own the course, please consider giving it a go: http://schoolofcode.me/courses/complete-python-web-developer-course-build-8-apps

If at any point the book isn't clear or could be better, let me know at jose@schoolofcode.me, and I will promptly improve on it!

Introduction

In this section we will be looking at the basics of Python. Python is a fantastic programming language which is quick to work with, powerful, runs in nearly every system you can think of, and does not require much work to get the hang of it. I'm sure you will fly through this section with no problems at all!

To complete the lectures and reading for this section will take you approximately 2 hours. The assignment in this section will not take you very long at all to complete, but is definitely recommended as completing it will make sure that you retain the knowledge that we will be covering.

Remember, by programming you will learn a lot more than just by reading or watching! Work on the assignments or on something that interests you.

Introduction to Python

In this section, we're simply going to be introducing Python and making our very first app. In this introduction to Python, we're going to be going over installing Python, and using IDLE, which is a development environment that comes with Python.

We will also look at what variables, strings, and integers are- (just types of data the computer can hold)

Then we're going to be looking at methods and parameters, and then we're going to be going into creating our first app. The app is going to be called "Our Age in Seconds", and this is going to be a simple program that takes our age in years and then gives it back to us in seconds.

As you can imagine, this will be a very simple program, as calculating our age in seconds is just a matter of multiplying our years by a few numbers. There's 365 days in a year (without counting gap years), 24 hours per day, 60 minutes per hour, and 60 seconds per minute.

So our age in seconds would be our age in years multiplied by **365 * 24 * 60 * 60**.

Installing Python

To install Python, all you have to do is go to http://python.org and download Python 3.5. This is the version used in this book and throughout the course, but Python 2.7 will also work just as well, and later versions should also work.

If when you are taking the course, a Python version like Python 4 has come out, it is not clear whether it will work or not, and so it is advisable to use Python 2.7 or Python 3.5.

Installing Python should just be a matter of double-clicking the downloaded package and installing as normal.

Once that is installed, you will have Python in your computer, which lets you create and execute Python programs! You will also have access to IDLE, the Python Shell, which lets you create Python programs interactively, and is a great tool to get started with Python.

Open up the program IDLE, and let's get started with Python!

Python 101

IDLE is an interactive Python shell, which means that whenever you write a Python line of code, it will instantly get evaluated and the result given back to you, if there is a result to give back.

In some instances, a line of code may not give back a result, it may just perform a calculation or assign a value.

Try typing `5+5` into the IDLE program.

You'll see that `10` is shown instantly, as that is what Python would evaluate the expression `5+5` to be.

Now try typing the following code into IDLE:

```
>>> "5" + "5"
```

And you'll see that what comes back is `'55'`. This is because what we added there are not two numbers, but rather two *strings*. Strings--also sometimes known as literals--are just sequences of characters. When we tell Python to add two sequences, the only thing it knows to do is *concatenate* them; i.e. join them together.

Going one step forward and typing the following:

```
>>> "5" + 5
```

You'll see an ugly error. Python does not know how to add a string and a number together, because they are intrinsically different *types*.

> Remember, computers are not intelligent. Computers know how to do things because we, the programmers, told them how to. Python can only do what the creators of Python can do, no more.

Variables in Python

Working with numbers and strings is all good and dandy, but it does not have much use if we can't persist the values we are calculating for them to be used later on.

Thus we can save our calculations into boxes. Each box can hold one value, and each box has a name. We can give a box whatever name we want, as long as it doesn't start with a number and doesn't have any special characters except the underscore (`_`).

In IDLE, we may create some variables like so:

```
>>> x = 5 + 5
>>> z = "123"
>>> my_variable = 15465788
>>> user_name = "Jose"
```

Notice that the following variable names are invalid, and Python will complain if you attempt to name your variables like any of the following:

```
>>> 1var = "A string"
>>> currency$ = "Dollar currency"
>>> email@server.com = "jose@schoolofcode.me"
```

Interacting with Users

Although in IDLE we can just type our expression and it gets evaluated (e.g. typing `5+5` prints out `10`), normally Python would not print anything out for the user to see unless we explicitly tell it to.

We can tell Python to print something by using the `print()` command, like so:

```
>>> print("Hello, world!")
Hello, world!
```

When we tell Python to print something--it can be strings, or numbers, or anything else--it will appear in the text-based output for the user to see.

For example, if the user's age was 30 years old, we could print out the user's age in seconds using a simple program:

```
>>> user_age = 30
>>> user_age_in_seconds = user_age * 365 * 24 * 60 * 60
>>> print("Your age in seconds is " + str(user_age_in_seconds))
```

Easy, isn't it!

In the last line, we need to use the `str()` method to temporarily treat the number that is `user_age_in_seconds` as a string, so it can be concatenated with the literal string. Remember that if we try to add a string and a number, we will get an error!

However, in order to do that we have to initialise our program with the user's age. That makes it difficult for the user to get their age in seconds, as they can't tell the program how old they are!

Lets change the program so it asks the user for their age first, and then we'll explain how the program does it:

```
>>> user_age_string = input("Enter your age: ")
>>> user_age_int = int(user_age_string)
>>> user_age_in_seconds = user_age_int * 365 * 24 * 60 * 60
>>> print("Your age in seconds is " + str(user_age_in_seconds))
```

There are a couple changes with respect to the previous version:

- `user_age_string` is no longer the static value `30`, but rather it is whatever the `input()` function returns. This function will print out `Enter your age:` to the user, and then the user can input the number, which the program will save as a string into `user_age_string`.
- `user_age_int` is the converted `user_age_string` into an integer. Integers are just whole numbers, and Python treats them slightly differently to floating-point numbers (numbers with a decimal place).

Then we use `user_age_int` as the user age, and we calculate the age in seconds and print it out to the user.

The `input()` and the `print()` methods both print something to the user, but with the difference that the `input()` method allows the user to type something back, and that can get saved into a variable.

The `format()` method

There is an extremely useful method that we can use instead of having to add strings together:

```
...
>>> print("Your age in seconds is " + str(user_age_in_seconds))
```

That is ugly code, and you need to remember the space at the end of the literal string, so that the literal and the variable don't appear together. You also need to remember the `str()` function, or else you'll get an error.

Instead, we can do this, which is simpler and will allow much greater flexibility later on:

```
...
>>> print("Your age in seconds is {}.".format(user_age_in_seconds))
```

What that tells Python is to swap the set of curly braces (`{}`) by the element that is between the brackets of the `format()` method.

We can do that with more than one element:

```
>>> print("My name is {}, I am {} years old.".format("Jose", -1))
'My name is Jose, I am -1 years old.'
```

Remember that Python is not smart, so me being `-1` year old is absolutely fine!

Advanced usage

We can do more advanced things with the `format()` method, but these are not essential. Feel free to skip them if you want to proceed quickly into the next part of the course.

We can name parameters to make the whole string more readable:

```
>>> print("My name is {name}, I am {age} years old.".format(name="Jose", age=30))
'My name is Jose, I am 30 years old.'
```

Here the value inside the curly braces (e.g. `{name}`) is what Python looks at when swapping, and so it knows to swap `"Jose"` for the `{name}` token.

This means that the parameters to the `format()` method need not be in order:

```
>>> print("My name is {name}, I am {age} years old.".format(age=30, name="Jose"))
'My name is Jose, I am 30 years old.'
```

It also means we can re-use a parameter:

```
>>> print("My name is {name}, is your name {name} too?".format(name="Jose"))
'My name is Jose, is your name Jose too?'
```

These things are a lot more is what you can do with the format method. The course contains a cheatsheet which you can download and optionally print out, and contains a reminder of what was covered here for your convenience.

Methods in Python

Normally when we are working on developing a program, we want it to be readable. This is so that when we go back after weeks or months, we can easily dive in and change things that need to be changed (for example, if we realize that something isn't working quite as we

intended).

It is extremely difficult to do this if our program is just one bit set of single-line expressions, so it is better to give names to groups of expressions, so that we can more easily and quickly understand what the groups do. For example, we could give a name to the group of expressions that asks the user for their age and converts it to a number:

```
>>> def get_user_age():
        user_age_string = input("Enter your age: ")
        user_age_int = int(user_age_string)
        return user_age_int
```

What this accomplishes is that now we can just execute `get_user_age()`, and that will ask the user to input something, wait for the user to write it, then convert that to a number, and give us back the value.

Try it yourself!

What we have done is created a **method** called `get_user_age` that will execute all expressions it is associated with. Python knows which expressions it is associated with because those expressions are directly below the method **and** indented consistently.

For example, the following code would be valid:

```
def get_user_age():
  user_age_string = input("Enter your age: ")
  user_age_int = int(user_age_string)
  return user_age_int
```

But the following code would **not** be valid, because the indentation is not consistent:

```
def get_user_age():
  user_age_string = input("Enter your age: ")
user_age_int = int(user_age_string)
  return user_age_int
```

Indeed, Python would assume the expressions associated with the method `get_user_age` finish after the first line below it, where the indentation ends. The line that says `return user_age_int` would then be marked as an error because Python does not understand *why it is indented*. After all, it does not belong inside a method because the line above it breaks the association with the method.

Methods can also use values that are given to them, like so:

```
>>> def calculate_age_in_seconds(age_years):
        return age_years * 365 * 24 * 60 * 60
```

Here, we are telling Python that this method will accept a value, and that that value will be stored in a variable called `age_years`. We can then use this variable inside the method, but **only inside the method**. The variable `age_years` will not exist outside this method.

We can use more methods, and call a method from within another method, and thus we could end up writing our age program with the following methods:

```
>>> def get_user_age():
        user_age_string = input("Enter your age: ")
        user_age_int = int(user_age_string)
        return user_age_int
>>> def calculate_age_in_seconds(age_years):
        return age_years * 365 * 24 * 60 * 60
>>> def run():
        print("Your age in seconds is {}.".format(calculate_age_in_seconds(get_user_ag
e())))
>>> run()
```

Take a few minutes to analyze the code and understand what is happening in it, and what each method does.

Inside the `run()` method, we are calling `format()` using whatever comes out of `calculate_age_in_seconds()`. To that method we are passing the output of `get_user_age`, which as we know returns the user's age as a number.

Simplifying `get_user_age()`

We could simplify `get_user_age()` and make it shorter, if we wanted.

Some might see the simplifications as making it more complex and difficult to read; others may not. Remember in most cases in programming, how you do something doesn't matter much, as long as the outcome is correct and the code is readable.

We could simplify to this:

```
def get_user_age():
  user_age = input("Enter your age: ")
  return int(user_age)
```

Or we could simplify further, to this:

```
def get_user_age():
  return int(input("Enter your age: "))
```

It is **completely up to you** how you want to do this.

> When you are working with a team, you should all come up with common coding style that you all agree with, to simplify working with each other's code.

Conditionals, `if`, `elif`, and `else`

Programs would quickly get useless if they couldn't cater for a range of different circumstances, and that's why all programming languages have some sort of conditional operator. This allows to choose between a range of logical paths.

For example, say our user is using our program to look at prices of chairs. If the price is very high, we want to tell them to not buy the chair. If it is low, we want to tell them to buy the chair:

```
>>> price = 100
>>> if price < 500:
    print("Buy the chair!")
else:
    print("Do not buy the chair!")
```

The first thing to notice is that the `else` is *unindented*. This is so Python can know that the `if` part of this code and the `else` part of this code correspond to one another.

What the code does is check if the conditional clause in the `if` statement is `True`. If it is not `True`, then it executes the code under the `else` part.

In this case, `price < 500` would evaluate to `True`, so the program would print `"Buy the chair!"`.

Try this yourself:

```
>>> def should_i_buy():
        price = int(input("Enter price: "))
        if price < 200:
            print("Buy!")
        else:
            print("Do not buy!")
```

This allows the program to choose between two logical paths: if the price is low or if it isn't. However, sometimes we may want our program to choose between more logical paths.

For example, to do something if the user presses the `'W'` key, or something else if they press `'A'`, `'S'`, or `'D'`.

It could look something like this:

```
...
>>> if key == 'W':
        print("Moving forwards")
elif key == 'A':
        print("Moving left")
elif key == 'S':
        print("Moving down")
elif key == 'D':
        print("Moving right")
else:
        print("Unknown command")
```

In this case, we have four conditional comparisons, checking whether a variable `key` (which would've gotten a value assigned to it previously in our program) is equal to `'W'`, in which case we print `'Moving forwards'`, or any of `'A'`, `'S'`, or `'D'`, and we print the appropriate message. Finally, if the key was not equal to any of `'W'`, `'A'`, `'S'`, or `'D'`, we print `'Unknown command'`.

If we wanted to not print `'Unknown command'`, we could just skip the `else` part of that code.

But again we can use these conditional terms at any point in our programs, to choose between two or more outcomes.

Remember we can also just use the `if` term:

```
...
>>> if price < 100:
        print("You've got a great deal!")
```

But there is no need to include any of the `else` or `elif` terms if they are not suitable.

However, we cannot use `else` or `elif` if there is not first an `if` term.

Assignment

Now that you have knowledge about Python, including the basics, and the `format()` method, creating your own methods, and conditionals, it's time for your first assignment!

It won't take up much of your time, but will be **essential** to firm up your knowledge and make sure you are taking maximum value from the course.

Here is what the assignment should do as a minimum, but feel free to expand on it!

1. Create a variable that holds a "magic number" between 0 and 10;
2. Tell the user to pick a number between 0 and 10;
3. If the user picks the magic number, tell them they've won; and
4. If the user does not pick the right number, tell them to run the program again.

You can do this with the tools given to you in this section.

Good luck!

Introduction

In this section we will be looking at understanding how the web works from a programming perspective, including getting the content of web pages, parsing them to extract information, and looking at common ways in which applications exchange data between them.

To complete the lectures and reading for this section will take you approximately 2 hours. The assignment should take a couple hours more of your time.

Remember, by programming you will learn a lot more than just by reading or watching! Work on the assignments or on something that interests you.

Development Environment

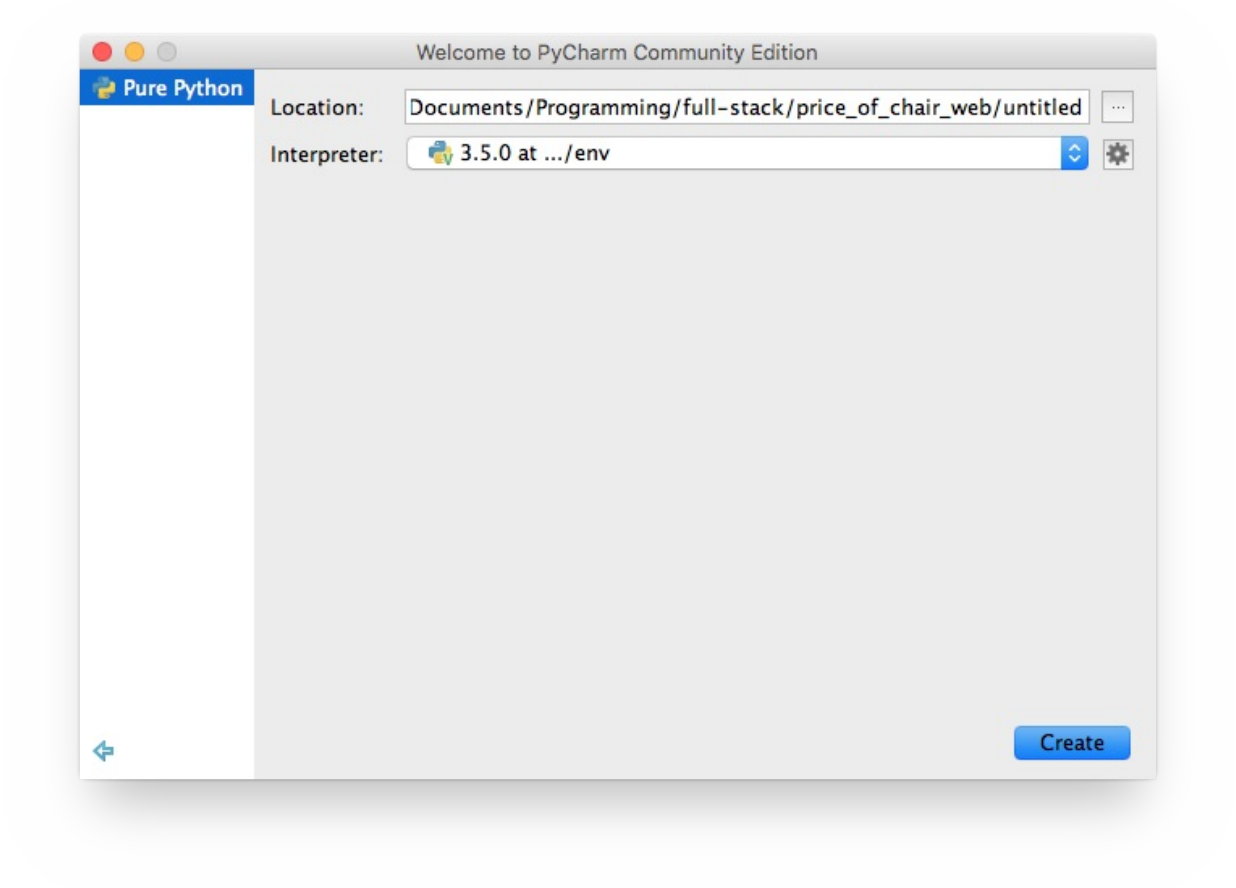
The IDLE is useful to create short programs or try things in Python. In fact, I nearly always have an IDLE open when I'm working, as that lets you easily try things out and make sure that the things you are writing are valid. However, it is not great to write long and complex programs!

It is easier to do using an IDE (Integrated Development Environment). Undoubtedly the most popular IDE for Python is one called PyCharm, which I would advise installing and using. A link is available here.

You can download and install it for Mac, Windows, or Linux. Downloading PyCharm CE (Community Edition) is free, whereas the Professional Edition is not free.

Once it is downloaded, you can open it and create a new project. A project will be one application, such as the "age application" from the last section, or one of the many web applications we will be creating in the next sections.

When you select "Create a new project" you will be prompted with a window like below:



And you will also have to choose an interpreter. This is what version of Python you will be using for this project. Remember that different versions of Python may be slightly different.

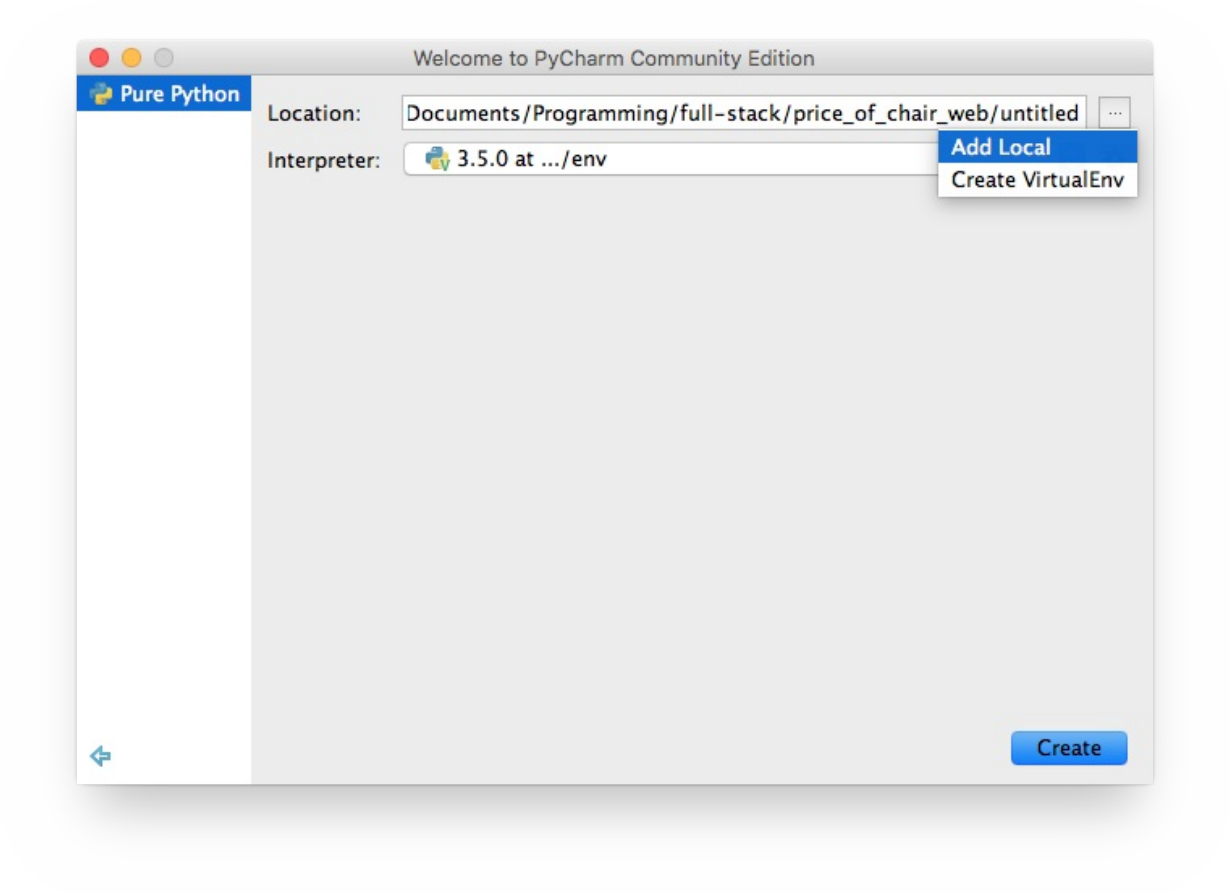
This is a good point to introduce Virtual Environments

Virtual Environments

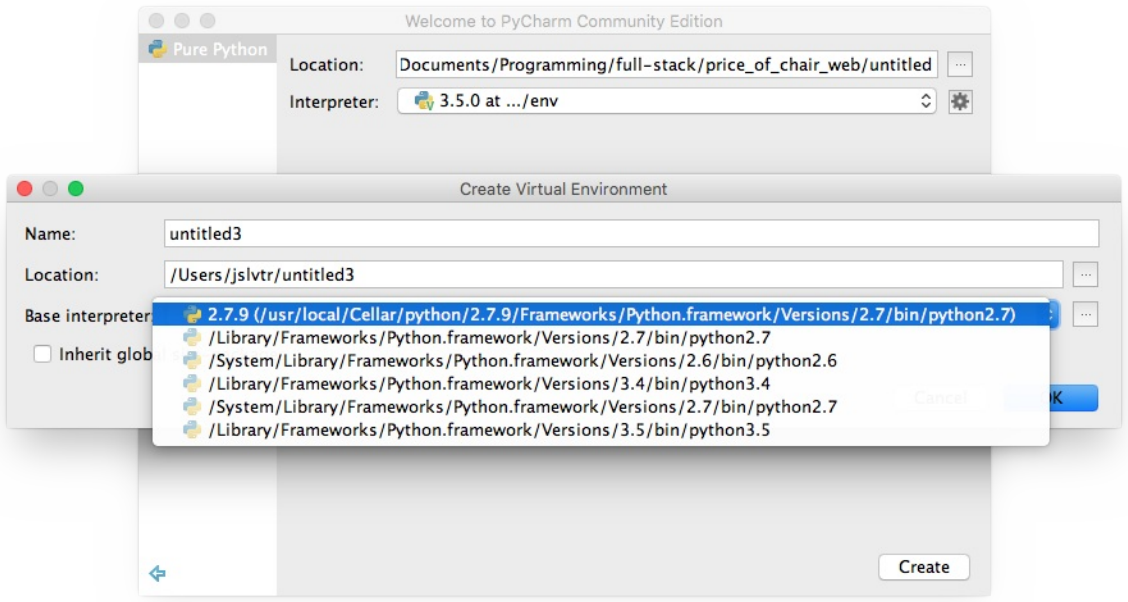
A Virtual Environment is a Python installation that runs separately to another Python installations. This means that you can use Python 2.7 for one project and Python 3 for a different project, for example.

It also means that you can create a Virtual Environment for each project, and only install libraries that are necessary for a project in that project's Virtual Environment.

Let's create a Virtual Environment for our new project, by pressing the cog icon beside the "Interpreter" drop-down:

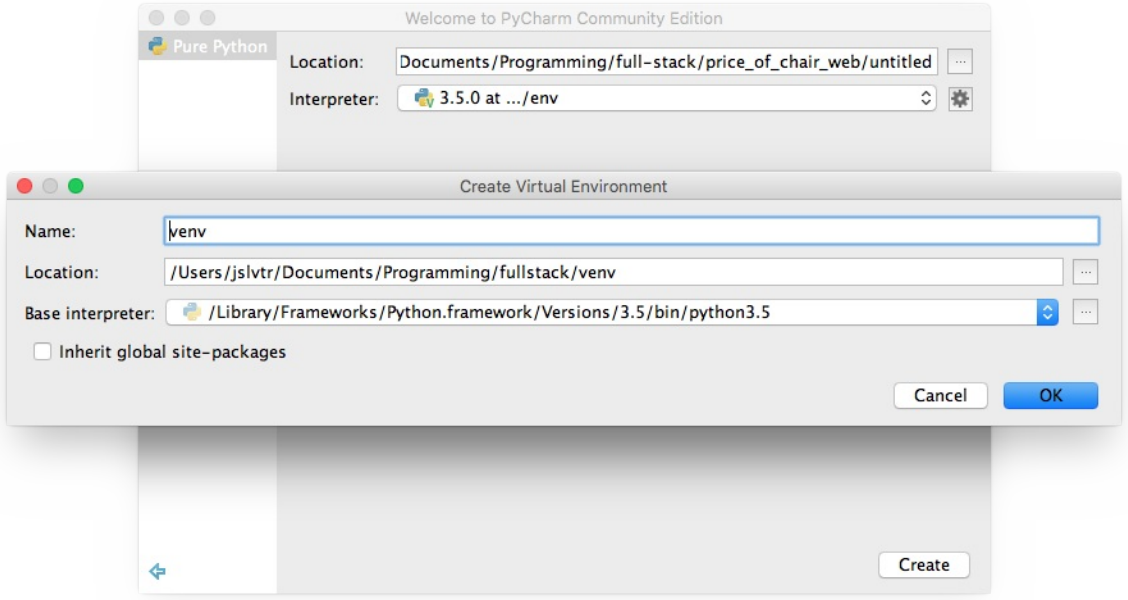


We now have to choose the "Base interpreter". This is so we can create a Virtual Environment using a specific Python version (i.e. one of 2.7, 3.3, 3.5, or any other Python version). The drop-down will show the available Python versions in your computer:



Remember to choose the location of the Virtual Environment and the name of the folder where the Virtual Environment will live. The name is usually `venv`.

Finally, we can create!



When your Virtual Environment has been created, you now have a separate Python installation that will be used by PyCharm to run your project. If your project requires any libraries (which it will, in this section), PyCharm will install those libraries in the Virtual Environment, as opposed to in your general system install.

Part of the good thing about this is that you then are not saturating your system install with required libraries, instead only install required libraries in the appropriate Virtual Environment.

Creating the Age App in PyCharm

Creating your first app in PyCharm is extremely easy.

In your newly-created project, just create a new Python file (right-click on the project), and call it something reasonable, like `app.py`.

The `.py` extension tells PyCharm this is indeed a Python file. PyCharm also works with a number of other file types, like HTML, CSS, JavaScript, and more.

Inside your `app.py` file, you can write your code for the age application.

This is the code we wrote in IDLE:

```
>>> def get_user_age():
        user_age_string = input("Enter your age: ")
        user_age_int = int(user_age_string)
        return user_age_int
>>> def calculate_age_in_seconds(age_years):
        return age_years * 365 * 24 * 60 * 60
>>> def run():
        print("Your age in seconds is {}.".format(calculate_age_in_seconds(get_user_ag
e())))
>>> run()
```

And we can write that in PyCharm like so, all in our `app.py` file:

```
def get_user_age():
    user_age_string = input("Enter your age: ")
    user_age_int = int(user_age_string)
    return user_age_int
def calculate_age_in_seconds(age_years):
    return age_years * 365 * 24 * 60 * 60
def run():
    print("Your age in seconds is {}.".format(calculate_age_in_seconds(get_user_age())
))

run()
```

Notice how the `run()` method **execution** (not the definition) is at the end. This is because at the end the methods have already been defined.

If we placed the `run()` execution at the top of the file, Python would try to execute that method **before** having created the method.

For example, the code below would not work, because we would execute `run()` before defining the `run` method:

```
def get_user_age():
    user_age_string = input("Enter your age: ")
    user_age_int = int(user_age_string)
    return user_age_int
def calculate_age_in_seconds(age_years):
    return age_years * 365 * 24 * 60 * 60

run()

def run():
    print("Your age in seconds is {}.".format(calculate_age_in_seconds(get_user_age())
))
```

Then, right-click your `app.py` file and press "Run". This will prompt you to enter your age, and then will give you a result!

JSON and XML

JSON and XML are two technologies used to transfer data between applications.

JSON

JSON stands for **JavaScript Object Notation**, and is an open standard, more readable than XML but can sometimes be limited, especially in terms of describing structure and allowing for searching.

JSON is thus simpler, and can be better to use in a number of occasions. For example, when sending data or retrieving data from a Web API where the size of the request and the response are impactful.

Here is a (slightly biased) comparison of JSON and XML: http://www.json.org/xml.html.

A less biased one is here.

Below, a possible JSON representation describing a person:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

XML

XML stands for **eXtensible Markup Language**. It is document-oriented and is extensible in the sense that tags can be created and easily re-used. JSON does not need this because it is optimized for data, but XML can in some scenarios be more versatile.

In this course we will not be using XML, but it is worth knowing that it exists, and, to some extent, what it looks like.

The person representation from above, here depicted as XML:

```
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumbers>
    <phoneNumber>
      <type>home</type>
      <number>212 555-1234</number>
    </phoneNumber>
    <phoneNumber>
      <type>fax</type>
      <number>646 555-4567</number>
    </phoneNumber>
  </phoneNumbers>
  <gender>
    <type>male</type>
  </gender>
</person>
```

Or it can also be described using tag attributes instead of only tags:

```
<person firstName="John" lastName="Smith" age="25">
  <address streetAddress="21 2nd Street" city="New York" state="NY" postalCode="10021"
 />
  <phoneNumbers>
     <phoneNumber type="home" number="212 555-1234"/>
     <phoneNumber type="fax"  number="646 555-4567"/>
  </phoneNumbers>
  <gender type="male"/>
</person>
```

XML looks very much like HTML, which we will be studying in this course.

In fact, HTML also uses tags, attributes, and tags can contain other tags and also literal content, like strings.

HTML stands for **HyperText Markup Language**, and is used to describe the structure of web-pages (i.e. what **is in the page**, but not **what it looks like**).

Making a GET request

In order to understand how the next few applications we are going to be developing work, we need to understand how the internet works.

A question you may get in an interview is the following:

> What happens when you type a URL in browser and press Enter?

An extremely in-depth answer is available here.

However, a lot of those steps are not needed in our case, as for now we are only worried with the network side of things.

A more suitable example is here: http://edusagar.com/articles/view/70/What-happens-when-you-type-a-URL-in-browser.

But really the steps we are focusing in are just a couple of them. Let's have a go at explaining it while programming in Python.

Install the required library

The required library is called `requests`, so you can include that library in your `requirements.txt` file. I would recommend, as always, to find the current version of the library and using that in your `requirements.txt` file. At the time of writing, the latest version was `2.7.2`, and my requirements.txt file looked like this:

```
requests==2.7.2
```

Import the library for use in your application

Before using a library in Python, we need to import it. First, create the Python file which will run your application. I tend to call this `app.py` or `run.py`.

Then, the amongst the first lines you should write some code to tell Python that this file is going to be using the `requests` library. *Note: the line does not have to be at the top necessarily, but that is the most common place for it.*

```
import requests

__author__ = "Your Name"
```

That first line now tells the Python interpreter that it needs to load the contents of the `requests` library for use when executing any code from this file.

Get the content of a page

All of the internet-related communications we are going to be doing in this course happen in a network layer that uses a specific protocol to transfer data: the HyperText Transfer Protocol. You'll know this as it often appears in front of URLs as `http://...`

This protocol just states **how** the transfer happens, and what data is transferred to some extent. The name itself tells us what type of data is transferred: hypertext, which is just another name for text that has links to other pieces of text. This protocol is as old as the internet, when pages were just bits of text with links to other pages.

Today, we use HTTP to transfer the pages themselves, images, videos, and everything in between.

Thus, the pages we will be writing are all going to be text. The browser (e.g. Google Chrome, Safari, or others) interpret that text (which is HTML and CSS code, mostly) to show us a *renderized* version of the page. The way the browser gets the content of the page is by **requesting** it from a server. A server is just a computer that has a program designed to answer these requests.

Thus, when a browser connects to the server, the server gives it the content of the page that it has asked for, and then the browser renders it and shows you a version that isn't all just plain text.

When we make a request using Python, we do not have a browser, so all we are going to be getting back is the text that makes up the page--the HTML and CSS code. Let's make our first request!

```
import requests

__author__ = "Your Name"

requests.get("http://google.com")
```

We've made a program that will ask one of the servers hosting http://google.com for the contents of the page!

However we are not storing that request anywhere, so there's not much we can do with it. Let's store the contents of the request in a variable, and then print the content of the page:

```
import requests

__author__ = "Your Name"

r = requests.get("http://google.com")
print(r.content)
```

If you run this program, you'll see an extremely long line be printed out to your console. That's the Google page!

A GET request

What we have done is `requests.get("<page>")`. This has replicated what a browser would do when asking Google for a page, and it's called a `GET` request.

> A `GET` request just retrieves something from a server. In this case, the page content.

There are many other types of requests that HTTP supports: `POST`, `PUT`, `DELETE`, and many more. Some a self-explanatory, whereas others are not:

HTTP "verb"	Meaning
GET	Retrieve something from the server
POST	Create a new element in the server, using the data provided in the request
PUT	Update an existing element in the server, using the data provided in the request
DELETE	Remove an element in the server

The URL

The `GET` request retrieves something from the server. But when we access `http://google.com/` we are not retrieving anything specific. We aren't telling the server what we want. Are we?

It turns out we are, and the key is that last character of the URL: `/`.

The forward slash character by itself means "the root". In the case of web applications, the root of the application tends to be the home page. So we can access pages, and we are always accessing the root:

```
http://google.com/
http://schoolofcode.me/
http://facebook.com/
```

If the forward slash character is not at the end, then it is assumed, so sometimes you may not see it!

Accessing other parts of the page

If we wanted to access School of Code's courses, we could go into the `courses` "folder":

```
http://schoolofcode.me/courses
```

I read this as a folder of courses because it makes sense to then access a specific course, living inside that folder:

```
http://schoolofcode.me/courses/complete-python-web
```

What does HTTP look like?

When the browser does the request to view the courses page, really it's doing something like this:

```
HTTP/1.1 GET http://schoolofcode.me/courses
```

That is sent to the server, which knows that the browser is expecting something back: the content of the page.

Is it really this simple?

Well... No. Not really. There's a lot more going on. HTTP isn't magical, traffic still has to travel from one server to another, and for that we need a lot of things. Feel free to look these up: Physical Network Layer, Ethernet, TCP, IP, DNS. That will help understand a bit more of what is happening behind the scenes, although it is not required for this course.

What price is that chair?

Now that we are able to get the contents of a page, there isn't much we can't do! We are one step closer--and really one step away--from being able to make our programs understand basically any piece of information on the internet.

What's left is telling the program what the information is and what it looks like, so it can be found.

In order to do this, we need to be able to understand the content of the pages. The page content is written in HTML, which is extremely similar in structure to XML. We can use a library called `BeautifulSoup` to allow our program to search through the HTML code.

Installing the required library

Like previously, we will first have to install the library, and then import it. Add the following code to your `requirements.txt`:

```
beautifulsoup4==4.4.0
```

Then, go into your main Python file (which I tend to call `app.py`), and import the library at the top, like so:

```
from bs4 import BeautifulSoup
```

What this import statement means is that installing the `beautifulsoup4` library actually is downloading a file called `bs4`. This file contains a section of code (called a **class**, but we will look at this later on) called `BeautifulSoup`. That is the code we are importing, and now we can start using it!

Parsing page contents

The class `BeautifulSoup` accepts at least one parameter: the page content to parse. Lets include it in our page:

```
import requests
from bs4 import BeautifulSoup

r = requests.get("http://google.com")
content = r.content
soup = BeautifulSoup(content)
```

Now the `soup` variable contains ways to search through the content (by tag name and by attributes). For example, if the page content looked like this:

```
<data>
  <students>
    <student id="3514m">
      <name>Jose</name>
      <subject>Computer Science</subject>
    </student>
    <student id="881749h">
      <name>Rolf</name>
      <subject>Computer Science</subject>
    </student>
  </students>
</data>
```

We could perform a search using BeautifulSoup like so:

```
soup.find("student", {"id": "3514m"})
```

Which would find a `<student>` tag (of which there are two), where the tag contains the attribute `"id"`, and it has the value `"3514m"`. Easy, isn't it!

What Chair do we want?

Now, lets go into an online store website and find an item you are interested in. We are going to make a program that will tell us whether we should buy the item or not, based on our budget.

> Note: some websites may not work as they block traffic coming from a robot, which our program will be. If you finish the program and it does not work, it may be because of this. Check the page content as it may give you some information.

The site I found was http://johnlewis.com/items/. I know this page works, so it may be a good idea to follow along using that, and then change it afterwards when you know it all works.

Open the page in your browser, and then right-click the large price text, and press "Inspect Element". This will bring up the contents of the page in a panel. It looks very complicated--but it's exactly the same structure as the XML we looked at previous, only at a larger scale.

In the page content, we want to find a line which contains the large price tag that we 'inspected'. It will look something like this:

```
<span itemprop="price" class="now-price"> £115.00 </span>
```

Then, lets go into our Python program and make the necessary changes:

```
import requests
from bs4 import BeautifulSoup

r = requests.get("http://johnlewis.com/items/")
content = r.content
soup = BeautifulSoup(content)
element = soup.find("span", {"itemprop": "price", "class": "now-price"})

print(element.text)
```

We still have to import both required libraries, but now our request is going to the online store website. We're looking for a `"span"` tag that has two attributes: `itemprop`, with value `"price"` and `class`, with value `"now-price"`.

When you run this program, you should see something like this printed out!

```

                £115.00
```

It is a bit strange how it has all that whitespace around the price! Lets get rid of that quite easily by using the `strip()` method. We can call this method on any string in Python.

```
import requests
from bs4 import BeautifulSoup

r = requests.get("http://johnlewis.com/items/")
content = r.content
soup = BeautifulSoup(content)
element = soup.find("span", {"itemprop": "price", "class": "now-price"})

print(element.text.strip())
```

And now, running this should give us the expected result:

```
£115.00
```

Removing the pound sign

Slicing

In Python it is easy to get split a string into parts, or only get a part of the string that we want. It works like this:

```
my_string = "hello, world!"
my_string_too = my_string[:]  # This copies the string 'my_string' into 'my_string_too'


hello = my_string[0:5]  # This copies the characters 0 to 5 from 'my_string' into 'hel
lo'
print(hello)  # Would print "hello"

world = my_string[7:12]  # This copies characters 7 to 12 from 'my_string' into 'world'

print(world)  # Would print "world"

all_hello = my_string[:5]  # Just like hello, but the first 0 can be omitted
print(all_hello)  # Would print "hello"

world_all = my_string[7:]  # This copies from character 7 onwards
print(world_all)  # Would print "world!"
```

Modifying the price string

So now it is easy to go back and remove the pound sign from the price string:

```
import requests
from bs4 import BeautifulSoup

r = requests.get("http://johnlewis.com/items/")
content = r.content
soup = BeautifulSoup(content)
element = soup.find("span", {"itemprop": "price", "class": "now-price"})

price = element.text.strip()
price_no_currency = price[1:]  # This would copy all except the index 0, which is the
pound sign

print(price_no_currency)
```

The price is still a string though. This means we cannot add it to another price to calculate, for example, total prices. Lets convert it to a number.

Remember the price has a decimal point, so we cannot convert it to an `int`, as integers are whole numbers. We need to convert it to a `float` instead: a floating-point number.

```
import requests
from bs4 import BeautifulSoup

r = requests.get("http://johnlewis.com/items/")
content = r.content
soup = BeautifulSoup(content)
element = soup.find("span", {"itemprop": "price", "class": "now-price"})

price = element.text.strip()
price_no_currency = price[1:]
price_number = float(price_no_currency)

print(price_number)
```

That's it! Our program now knows the price of the chair, so we can write a small amount more of code to act as a budget. We already know how to do this!

```
import requests
from bs4 import BeautifulSoup

user_budget = int(input("What is your budget? Enter a whole number: "))

r = requests.get("http://johnlewis.com/items/")
content = r.content
soup = BeautifulSoup(content)
element = soup.find("span", {"itemprop": "price", "class": "now-price"})

price = element.text.strip()
price_no_currency = price[1:]
price_number = float(price_no_currency)

if price_number > user_budget:
  print("This item is over your budget... Sorry!")
else:
  print("Lets buy it!")
```

Good job getting here. This is the end of the second section, so next we'll be working on creating a blog!

The next section will introduce Object-Oriented Programming as well as our first database system that we are going to use: MongoDB.

Introduction

In this section we will be looking at creating a simple Blog application. We are going to be learning about databases and object-oriented programming while developing our last console application. In the next section we will start with web development!

To complete the lectures and reading for this section will take you approximately 2 hours. The assignment should take a few hours more of your time.

Remember, by programming you will learn a lot more than just by reading or watching! Work on the assignments or on something that interests you.

Introduction to MongoDB

Installing MongoDB

Lets get started with installing MongoDB by going to the downloads page (http://mongodb.org/downloads) and following the instructions on the page (also below).

> If you are using Windows, you may want to follow the instructions linked here, as opposed to installing MongoDB directly onto your system.

On Mac

I recommend installing Homebrew. Homebrew is a package manager: a software used to install other software. It makes installing MongoDB a breeze.

Open up your Terminal.app (you can find it by typing 'term' in Spotlight) and pasting the code below into your terminal:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Then, still in your terminal, type the following:

```
brew update
```

And then,

```
brew install mongodb
```

That's it!

On Linux

Each Linux distribution has specific instructions to install it. Please refer to the installation instructions for your distribution:

- Ubuntu 12.04 and 14.04
- Debian 7
- RHEL (CentOS) 7

Using MongoDB

Using MongoDB is, you'll be glad to hear, easier than installing it!

> If you are using Windows and have installed MongoDB using Docker, as linked above, follow these instructions inside the Kinematic application.

Open up your terminal (Terminal.app on Mac, Terminal on Linux, Console on Windows), unless you've got it already open, and type the following:

```
mongod
```

> You will not need to do this if you are running the application using Docker.

This starts a MongoDB server in your computer. Now you can start interacting with MongoDB. **Do not close the Terminal.app while the `mongod` process is running**.

Open another terminal window, and type the following:

```
mongo
```

This starts the interactive console that you can use to interact with the running `mongod` server.

You can now see the databases that are present in your system (potentially none), by typing:

```
show dbs
```

And you can tell `mongo` that you want to use a specific one (or create a new one) by issuing the command `use <db>`. In this case, we are going to create a new database called `fullstack`:

```
use fullstack
```

Then, we can see the collections, which hold data, in our database. Because we have just created this database, they will be empty:

```
show collections
```

However, we can insert data into a non-existing collection, and it will be created alongside the data. For example, we can create a new student and put it in our database:

```
db.students.insert({"name": "Jose", "mark": 99})
```

This creates the collection `students`, and puts a piece of data inside the collection. The data is `{"name": "Jose", "mark": 99}`. We can think of this set of key-value pairs (`name-Jose` and `mark-99`) as the data representing our first student.

Similarly, we can remove data:

```
db.students.remove({"name": "Jose"})
```

Here, MongoDB finds the data that matches the criteria (in this case, our only piece of data), and removes it from the database.

> MongoDB is a *schema-less* database, which means you can have pieces of data with different keys in the same collection.
>
> For example, you could have `{"name": "Jose", "mark": 99}` and `{"item": "chair", "price": 129.5, "store": "IKEA"}` in the `students` collection. This would not make much sense, but you can do it!

Next up, lets look at using MongoDB from within our Python projects!

Python project with MongoDB

The first thing to do when using MongoDB with Python is to install the required library.

Installing the required library

If you are using PyCharm, as I recommend, the only thing you have to do is create a blank project, and then inside the project create a file called `requirements.txt`.

Then, inside this file, write the name of the library that you want to install. In our case, we want to install the library `pymongo`.

Optionally, we can also specify (as we did in the last section) the version number of the library. Type the following into your requirements file:

```
pymongo==3.2.2
```

If you omit `==3.2.2`, then the latest version will be installed, although this is not recommended.

> Always installing the latest version can seem like a good idea. However, if you create your application using v3.2.2, and then an update to the library comes along which you don't investigate, the update might break your application.
>
> **Always specify a version number in your requirements file.**

When you go into a Python file to start the project, PyCharm will prompt you to install the libraries that you have not already installed from your requirements file.

Next up, lets look at list comprehension in Python. It's a small aside from the project which will nevertheless be extremely useful!

Using the pymongo library

First, create a file called `app.py` in your project. We are going to write a small program that will help us understand how to interact with MongoDB in our Python programs.

Since we are using the library `pymongo` and it is not in the same file that we are writing, we have to import it. After importing, your file will look something like this:

```
import pymongo

__author__ = "Your Name"
```

Every `mongod` process has associated with it a **Universal Resource Identifier**. This identifier allows a program to find and connect with the process. The URI is often this:

```
mongodb://127.0.0.1:27017
```

> If you are using Docker to run MongoDB, your URI will look different. You should be able to see your URI in the Kinematic window.

Lets add our URI to our program:

```
import pymongo

__author__ = "Your Name"

mongodb_uri = "mongodb://127.0.0.1:27017"
```

The `pymongo` library allows our program to create a variable of type `MongoClient`, which is what gives us the functionality to interact with MongoDB. This variable needs the URI we have created, so it knows where the `mongod` process lives.

```
import pymongo

__author__ = "Your Name"

mongodb_uri = "mongodb://127.0.0.1:27017"
client = pymongo.MongoClient(mongodb_uri)
```

Next up, we need to get the database that our program is going to be using, and the specific collection where we want to insert data and retrieve data from.

```
import pymongo

__author__ = "Your Name"

mongodb_uri = "mongodb://127.0.0.1:27017"
client = pymongo.MongoClient(mongodb_uri)
database = client['fullstack']
collection = database['students']
```

Now we are ready to start inserting or retrieving data!

In order to retrieve data, you'll need to have some data in the collection. Refer to the top of this document for how to insert data into your `students` collection!

```
import pymongo

__author__ = "Your Name"

mongodb_uri = "mongodb://127.0.0.1:27017"
client = pymongo.MongoClient(mongodb_uri)
database = client['fullstack']
collection = database['students']

students = collection.find({})
print(students)
```

If we were to run this program (run it!), we would expect to see a list of students or something recognisable. However, we do not. We see a bunch of gibberish that says something about a `Cursor` object. What is this?

The `pymongo` library does not give us back a list of whatever it found in the database. Instead, it gives us a `Cursor`. We can use this `Cursor` to find what is in the database. We can also do other things with this, as we'll see later. Things like sorting, for example.

In order to get our list of students, we need to **iterate** over the `Cursor`. We do this by using a `for` loop. Let's look at the structure of the loop, and then explain what is happening:

```
import pymongo

__author__ = "Your Name"

mongodb_uri = "mongodb://127.0.0.1:27017"
client = pymongo.MongoClient(mongodb_uri)
database = client['fullstack']
collection = database['students']

students = collection.find({})

for student in students:
    print(student)
```

This code reads nicely: "for each of the students in the `students` element, print the student".

The code is creating a variable called `student`, and assigning to it the first element found in the `students` object (our `Cursor`). Then it runs the "body" of the loop, which says `print(student)`. After, it goes back to the top of the loop and gives the variable `student` the **second** value in our object. Then it repeats, with the third, fourth, etc., values.

Run the program, and you'll be able to see something recognisable!

Next, let's look at how we can simplify this and store the students in a variable, instead of printing them out.

List Comprehension

We have just looked at how to interact with MongoDB from within our Python programs, and how to retrieve data and print it out from within Python.

This was the program we wrote in the last few pages:

```
import pymongo

__author__ = "Your Name"

mongodb_uri = "mongodb://127.0.0.1:27017"
client = pymongo.MongoClient(mongodb_uri)
database = client['fullstack']
collection = database['students']

students = collection.find({})

for student in students:
    print(student)
```

However, we have not looked at how we can retrieve the data from the database and assign all of the data to a variable.

There are two ways to do it. One way does not use list comprehension, and requires more code, and the other does. Let's look at not using list comprehension first.

We could create a variable `students_list` which we initialise to be a Python list, and then append each of the students to it as we iterate over the `Cursor` with our for loop:

```
import pymongo

__author__ = "Your Name"

mongodb_uri = "mongodb://127.0.0.1:27017"
client = pymongo.MongoClient(mongodb_uri)
database = client['fullstack']
collection = database['students']

students = collection.find({})
students_list = []  # Create the empty list

for student in students:
    students_list.append(student)  # Append each student

print(students_list)
```

This is all good, and it works!

However, it requires adding more lines of code and it is a bit difficult to read. Instead of directly creating a list of students, we have to iterate over our `Cursor` and `.append` each other students to our list.

Lets look at list comprehension. I'll give you a couple of unrelated examples first using IDLE:

```
>>> values_range = [0, 1, 2, 3, 4, 5]
>>> [x for x in values_range]
[0, 1, 2, 3, 4, 5]

>>> [x*2 for x in range(10)]
[0, 2, 4, 6, 8, 10]
```

Read the first example as "get the value `x` for each `x` inside `values_range`, and make it part of a new list".

What that does is create a temporary variable `x`, and assign to it the first value of `values_range`. What we are putting in the new list is just `x` (this is the first `x` we encounter). We are telling Python we want a new list by enclosing this `for`-loop-like syntax inside square brackets.

Read the second example as "get the value `x*2` for each `x` inside `values_range`, and make it part of a new list".

Just as above, but this time multiplying `x` by two before putting it in the new list.

Using list comprehension with our data Cursor

We know we can iterate over our `Cursor` object, as we have done with a `for` loop.

We want to get each of the values in our `Cursor` and put them in a new list, unmodified.

I think you can do this! Try, and then continue reading once you've attempted it.

Since we want the unmodified values to go into a new list, we could just do as the first list comprehension example above:

```
import pymongo

__author__ = "Your Name"

mongodb_uri = "mongodb://127.0.0.1:27017"
client = pymongo.MongoClient(mongodb_uri)
database = client['fullstack']
collection = database['students']

students = collection.find({})
students_list = [student for student in students]

print(students_list)
```

In my case, this prints something like the following:

```
[{'name': 'Jose', 'mark': 75}, {'name': 'Rolf', 'mark': 85}]
```

There are two students in there: 'Jose' and 'Rolf'.

Now that we've got list comprehension set up, it is easy for us to **not** get the entire student. For example, we might only want the marks that they got (say, to calculate an average):

```
import pymongo

__author__ = "Your Name"

mongodb_uri = "mongodb://127.0.0.1:27017"
client = pymongo.MongoClient(mongodb_uri)
database = client['fullstack']
collection = database['students']

students = collection.find({})
marks_list = [student['mark'] for student in students]

print(marks_list)
```

This would print out the following:

```
[75, 85]
```

Hopefully I've convinced you of the power and flexibility of list comprehension. Let's move on to **Object-Oriented Programming**!

Object-Oriented Programming

Imagine we are making a program for a blog. We will be storing data related to posts that have been written for the blog.

An option for our application would be to just use a database, with no program written. Every time a post gets written, the author has to load up a database and put the title and content of the post in it.

This means that the author needs to have access to the database, and needs to know how to update the database with the new data.

We can't really expect that, so we agree that we need a program written with some sort of user interface. In this user interface we could load the posts, which would bring up some information about them, including, for example, the title or content.

While we develop our program, remember that each one of these posts is just a row with some data in a database somewhere.

In order to make our program understand that the data it is dealing with has these properties, we create a Class which has two properties: `title` and `content`. It could look something like this:

```
class Post:
  def __init__(self):
    self.title = "This is a post title"
    self.content = "This is some content"
```

Notice this isn't one specific post, but this is us just telling our program that this is the thing it is going to be working with. Now, lets create one post in our program:

```
import post

post_one = Post()
post_two = Post()
```

We are telling our program "hey, these things you understand called post_one and post_two are objects of type Post! You know they have these two properties: title, and content".

And then our program says "cool I got it."

In our class definition we stated that a post's title and content are "This is a post title" and "This is come content", respectively. Because it is defined at a class level, both `post_one` and `post_two` have the same title and content.

That's what a Class does: define that the properties exist (and, optionally, give them values).

This is not particularly useful, because whenever we create a `Post` it's going to have the same title and content. We need a way to change the title and content depending on the post we are creating. We can do this by passing arguments to the `__init__` method, like so:

```
class Post:
  def __init__(self, title, content):
    self.title = title
    self.content = content
```

Now we've got this constructor, we can tell our program "we're going to tell you that this post exists, and also give you its properties so you can use them".

```
import post

post_one = Post(title="Object-Oriented Programming", content="The content goes here!")
post_two = Post(title="Second blog post", content="You can have different content in t
his Post.")
```

Now the program knows post_one.title is "Object-Oriented Programming". But we can create many objects, not just one! They all have the same properties, but can have **different values** for them.

```
import post

post_one = Post(title="Object-Oriented Programming", content="The content goes here!")
post_two = Post(title="Second blog post", content="You can have different content in t
his Post.")
post_three = Post(title="Third post", content="This is empty, for now!")

print(post_one.title)
print(post_two.content)
print(post_three.title)
```

There we've got three objects, they all have different values, but we only had to define what their properties are (note: not their values) once, when we defined the Class. We did this by initializing properties in the `__init__` method, but setting their values via the arguments passed to it.

Object actions

Objects can not only store data. They can also store actions, or things the object can do. Usually, objects can do things on themselves.

For example, a `Post` object could clear its contents. To do this we would create a method inside the `Post` class like so:

```
class Post:
  def __init__(self, title, content):
    self.title = title
    self.content = content

  def clear_contents(self):
    self.title = ""
    self.content = ""
```

That method simply make the post's title and content equal to an empty string, which clears them, removing the values. This could be used in a program like so:

```
import post

post_one = Post(title="Object-Oriented Programming", content="The content goes here!")

print(post_one.title)  # prints "Object-Oriented Programming"

post_one.clear_contents()

print(post_one.title)  # prints ""
```

Our program now has the objects, and we've understood that objects can be used to model real-life objects, such as blog posts, but also could be used to model animals, computers, chairs, or anything else you can think of!

Creating the Database Class

In order to interact with the database, we can create a class that will model a Database. For the purposes of this class we will have to imagine that a database is a real-life object that we could interact with.

When thinking about creating a class we must think of two things:

- Does any data need to live in each of the objects?
- Do the objects need to be able to do things?

If there is data that needs to be in the objects, then the data names should be specified in the `__init__` method.

If there are things the objects should be able to do, then these actions should be defined as methods inside the class, for each object to have access to.

Data in the Database class

When the `mongod` process starts in your computer, it opens up a web port for applications to connect to it. By default, the port number is `27017`.

We cannot connect to a web port by itself, however. We also need a web address. Because `mongod` is running in your computer, the web address will be `127.0.0.1`.

The complete location, called URI (Universal Resource Identifier) will be `127.0.0.1:27017`.

We also need to specify the type of resource that we are connecting to when we access that location. We do this by appending the resource type to the URI: `mongodb://127.0.0.1:27017`.

Thus the Database class would need this URI to be defined in the `__init__` method, like so:

```
class Database:
  def __init__(self):
    self.uri = "mongodb://127.0.0.1:27017"
```

Instance and Class Properties

If we were to create two objects of type Database, they both would have a property `uri`.

```
db_one = Database()
db_two = Database()

print(db_one.uri)  # "mongodb://127.0.0.1:27017"
print(db_two.uri)  # "mongodb://127.0.0.1:27017"
```

This is something we already know! We can also change one of the objects' properties without affecting the property in the other object:

```
db_one = Database()
db_two = Database()

db_two.uri = "hello"

print(db_one.uri)  # "mongodb://127.0.0.1:27017"
print(db_two.uri)  # "hello"
```

This is very useful, if we want the URI for different objects to be different in some cases. Here, we want all our Database objects to connect to the same database, so it does not make sense to allow for the URI to differ between objects.

We can then create a **class property**, which is a variable that is not owned by each object, but rather by the class as a whole. Any object, including those not of type Database, can then access the property (or indeed change it).

We would change the `uri` to a class property like so:

```
class Database:
  uri = "mongodb://127.0.0.1:27017"

  def __init__(self):
    pass  # This method now doesn't do anything. We could just delete it.
```

Constants

Python does not have a way to define constants. Other languages, like Java or C++, allow the programmer to define constant values that cannot change.

In Python, we can define normal variables only, but if we give them fully uppercase names, this tells other programmers that the name should not change.

Let's change the `uri` variable to one named like a constant:

```
class Database:
  URI = "mongodb://127.0.0.1:27017"
```

Connecting to the database

Now that we have the URI defined, we need to be able to initiate a connection to the `mongod` process using that URI. We can do this using a pre-existing library called `pymongo`. Let's install this library by placing it in the `requirements.txt` file:

```
pymongo==2.7.2
```

Then, we can go back to our Database class and create our first method in the class:

```
import pymongo

class Database:
  URI = "mongodb://127.0.0.1:27017"

  def connect(self):
    # This initialises the connection to the URI
    client = pymongo.MongoClient(Database.URI)

    # This creates a variable which is the 'fullstack' database in that connection
    database = client['fullstack']
```

The `connect` method is shown with comments explaining what each line does.

However, something to note in the method is that the parameter `self` is not used at all.

In this instance, it might be better to make the method a `@staticmethod`, which makes it a method that does not affect a specific object, but instead is a method that should be executed at the class level.

```
import pymongo

class Database:
  URI = "mongodb://127.0.0.1:27017"

  @staticmethod
  def connect(self):
    client = pymongo.MongoClient(Database.URI)
    database = client['fullstack']
```

However, the method still does nothing because it is not returning anything. Since it is not returning, the variable `database` it creates is lost at the end of the method.

Let's make the `database` variable into a class property.

```
import pymongo

class Database:
  URI = "mongodb://127.0.0.1:27017"
  database = None

  @staticmethod
  def connect(self):
    client = pymongo.MongoClient(Database.URI)
    database = client['fullstack']
```

Interacting with the database

Now that we have a class property which is the database that we are connected to, we can proceed and create the rest of the methods to interact with the database. For example, to insert or find data.

Finding data

The first method that we will implement will be used to find all elements matching certain criteria.

Remember the methods all go inside the Database class.

```
def find(collection, query):
  return Database.database[collection].find(query)
```

This method will be called from other parts of our app like so:

```
Database.find('users', {'username': 'jose'})
```

And that would find, in the collection 'users', all users that have the 'username' key with value 'jose'.

Often when we are looking for one specific user we want to get **just one element** returned from the database, as opposed to a list of users. For example, if we were trying to log a user in, we are not interested in all the users with that username, but rather just the only user with that username.

> It is a separate issue if you have more than one user with the same username. You probably shouldn't!

There are other scenarios where returning only one element is useful, and MongoDB allows us to query the database and get only one element back. We could do this like so:

```
def find_one(collection, query):
  return Database.database[collection].find_one(query)
```

Inserting data

Inserting data is also very simple using MongoDB. All we have to do is specify the collection into which data is going, and which data we want to put in the collection.

```
def insert(collection, data):
  return Database.database[collection].insert(data)
```

This would be called like so:

```
Database.insert('users', {
  'username': 'jose',
  'password': '1234',
  'email': 'email@example.com'
})
```

And it would insert the document into the database. For sake of completion, remember the document inserted would be as follows:

```
{
  '_id': ObjectID('...'),
  'username': 'jose',
  'password': '1234',
  'email': 'email@example.com'
}
```

Because MongoDB puts an `_id` field in every document it inserts.

Creating our Post Class

The application that we are building in this section is a blog system. Users will be able to register, create their blog, and then create posts within their blog.

Each post will have the following properties:

- A title;
- Some content;
- An author;
- The blog in which is was written; and
- A created date.

Finally, it is also going to have an extra property called `_id`, which MongoDB uses for storage. We will look at exactly why it is necessary further on.

As we know from the previous chapter, we can create a class which allows us to model these Post obects:

```
class Post:
  def __init__(self, title, content, author, blog_id, created_date, _id):
    self.title = title
    self.content = content
    self.author = author
    self.blog_id = blog_id
    self.created_date = created_date
    self._id = _id
```

JSON representation for storage in MongoDB

We have already seen that MongoDB only stores JSON data. It is necessary that we are able to represent our Post objects as JSON so that we can store them in MongoDB.

The `pymongo` library converts Python dictionaries to JSON for us, so all we have to do is represent the Post object as a Python dictionary, as opposed to a Python object. Here's one way we might go about doing that:

```
def json(self):
  return {
    '_id': self._id,
    'blog_id': self.blog_id,
    'author': self.author,
    'content': self.content,
    'title': self.title,
    'created_date': self.created_date
  }
```

Easy, right? All we have to do is give each of the properties a key, and put the key-property pair in a dictionary.

Writing to MongoDB

Then, writing to MongoDB just requires us to use the `Database` class we wrote earlier to insert that JSON into the database. This is what the Post class would look like now:

```
from src.common.database import Database


class Post(object):

  def __init__(self, blog_id, title, content, author, created_date, _id):
    self.blog_id = blog_id
    self.title = title
    self.content = content
    self.author = author
    self.created_date = created_date
    self._id = _id

  def save_to_mongo(self):
    Database.insert(collection='posts',
                    data=self.json())

  def json(self):
    return {
      '_id': self._id,
      'blog_id': self.blog_id,
      'author': self.author,
      'content': self.content,
      'title': self.title,
      'created_date': self.created_date
    }
```

Creating an object from data stored in MongoDB

When we retrieve data from MongoDB, we get a Python dictionary back. We can then use the data in the dictionary to create an object with the same properties as the object we saved.

We can retrieve the data related to one object by executing the following:

```
post_data = Database.find_one(collection='posts', query={'_id': id})
```

Then, we can use the data to create a Post object:

```
post_object = Post(
                blog_id=post_data['blog_id'],
                title=post_data['title'],
                content=post_data['content'],
                author=post_data['author'],
                created_date=post_data['created_date'],
                _id=post_data['_id'],
              )
```

Thus we can place the method inside the Post class, like so:

```
def from_mongo(self, post_id):
  post_data = Database.find_one(collection='posts', query={'_id': post_id})
  post_object = Post(
                  blog_id=post_data['blog_id'],
                  title=post_data['title'],
                  content=post_data['content'],
                  author=post_data['author'],
                  created_date=post_data['created_date'],
                  _id=post_data['_id'],
                )
  return post_object
```

However, we are not using `self` inside that method at all. That means the method isn't using one specific instance of an object. However, it does use `Post`, which is the class in which the method lives.

We can make this method into a `classmethod`, a type of method used when the method uses the class in which it lives, but not a specific object. To do this, we just add `@classmethod` and change one parameter of the method:

```
@classmethod
def from_mongo(cls, post_id):
  post_data = Database.find_one(collection='posts', query={'_id': post_id})
  post_object = cls(
                  blog_id=post_data['blog_id'],
                  title=post_data['title'],
                  content=post_data['content'],
                  author=post_data['author'],
                  created_date=post_data['created_date'],
                  _id=post_data['_id'],
                )
  return post_object
```

Argument unpacking

Finally, there is one more improvement we can make on this method, and that is to use **argument unpacking**.

> **Argument unpacking** allows us to pass a dictionary as parameters to a method, by using the keys as parameter names, and the values as parameter values.

Instead of doing this:

```
cls(
  blog_id=post_data['blog_id'],
  title=post_data['title'],
  content=post_data['content'],
  author=post_data['author'],
  created_date=post_data['created_date'],
  _id=post_data['_id'],
)
```

We can do this:

```
cls(**post_data)
```