Scraping class Documentation

Release 0.1

IRE/NICAR

August 22, 2016

Contents

A step-by-step guide to writing a web scraper with Python.

The course assumes the reader has little experience with Python and the command line, covering a number of fundamental skills that can be applied to other problems.

This guide was initially developed by Chase Davis, Jackie Kazil, Sisi Wei and Matt Wynn for bootcamps held by Investigative Reporters and Editors at the University of Missouri in Columbia, Missouri in 2013 and 2014.

It was modified by Ben Welsh in December 2014 for workshops at The Centre for Cultura Contemporania de Barcelona, Medialab-Prado and the Escuela de Periodismo y Comunicación at Universidad Rey Juan Carlos.

- Code repository: github.com/ireapps/first-web-scraper/
- Documentation: first-web-scraper.rtfd.org/
- Issues: github.com/ireapps/first-web-scraper/issues/

What you will make

This tutorial will guide you through the process of writing a Python script that can extract the roster of inmates at the Boone County Jail in Missouri from a local government website and save it as comma-delimited text ready for analysis.

Prelude: Prerequisites

Before you can begin, your computer needs the following tools installed and working to participate.

1. A command-line interface to interact with your computer

2. A text editor to work with plain text files

3. Version 2.7 of the Python programming language

4. The pip package manager for Python

Note: Depending on your experience and operating system, you might already be ready to go with everything above. If so, move on to the next chapter. If not, don't worry. And don't give up! It will be a bit of a slog but the instructions below will point you in the right direction.

2.1 Command-line interface

Unless something is wrong with your computer, there should be a way to open a window that lets you type in commands. Different operating systems give this tool slightly different names, but they all have some form of it, and there are alternative programs you can install as well.

On Windows you can find the command-line interface by opening the "command prompt." Here are instructions for Windows 8 and earlier versions. On Apple computers, you open the "Terminal" application. Ubuntu Linux comes with a program of the same name.

2.2 Text editor

A program like Microsoft Word, which can do all sorts of text formatting like change the size and color of words, is not what you need. Do not try to use it below.

You need a program that works with simple "plain text" files, and is therefore capable of editing documents containing Python code, HTML markup and other languages without dressing them up by adding anything extra. Such programs are easy to find and some of the best ones are free, including those below.

For Windows, I recommend installing Notepad++. For Apple computers, try TextWrangler. In Ubuntu Linux you can stick with the pre-installed gedit text editor.

2.3 Python

If you are using Mac OSX or a common flavor of Linux, Python is probably already installed and you can test to see what version, if any, is there waiting for you by typing the following into your terminal.

```
$ python -V
```

Note: You don't have to type the "$" It's just a generic symbol geeks use to show they're working on the command line.

If you don't have Python installed (a more likely fate for Windows users) try downloading and installing it from here.

In Windows, it's also crucial to make sure that the Python program is available on your system's PATH so it can be called from anywhere on the command line. This screencast can guide you through that process.

Python 2.7 is preferred but you can probably find a way to make most of this tutorial work with other versions if you futz a little.

2.4 pip

The pip package manager makes it easy to install open-source libraries that expand what you're able to do with Python. Later, we will use it to install everything needed to create a working web application.

If you don't have it already, you can get pip by following these instructions. In Windows, it's necessary to make sure that the Python Scripts directory is available on your system's PATH so it can be called from anywhere on the command line. This screencast can help.

Verify pip is installed with the following.

```
$ pip -V
```

Act 1: The command line

Working with Python (and pretty much any other programming language) means becoming comfortable with your computer's command line environment. If you haven't seen it before, it looks something like this:
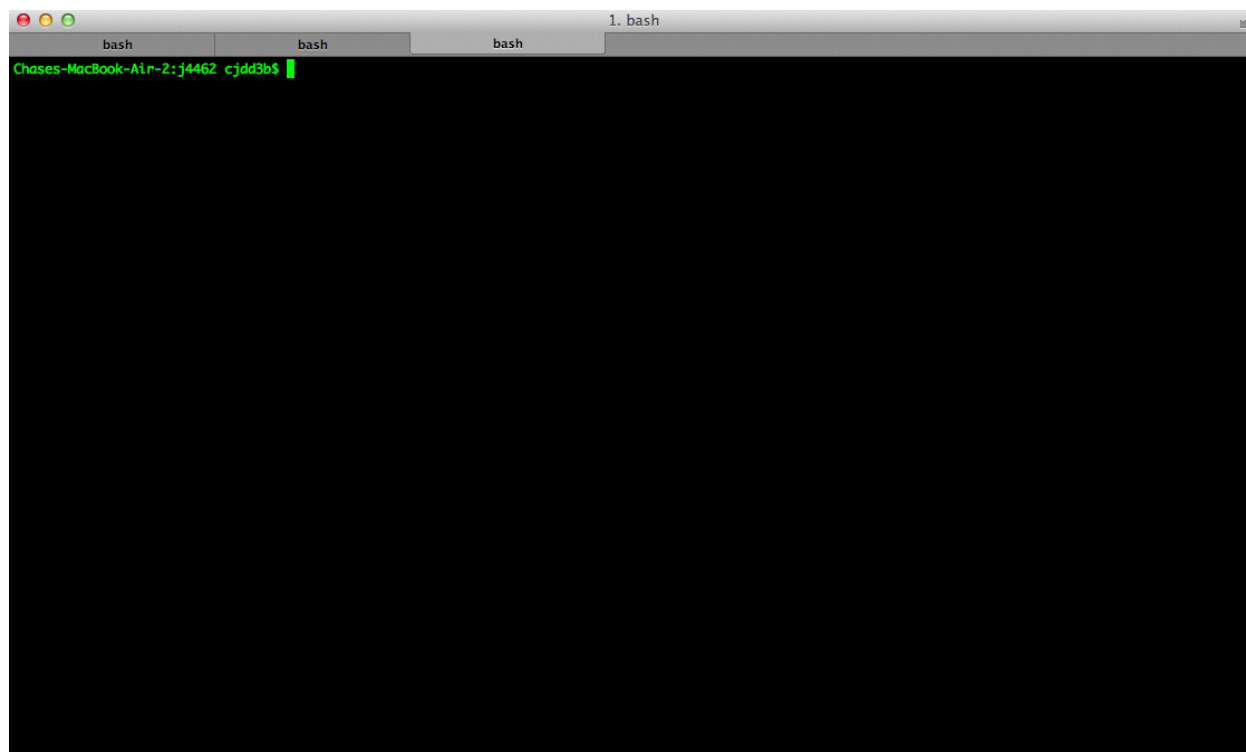
In this lesson we'll be using it to give the computer direct commands to manage files, navigate through directories and execute Python scripts. Don't worry, it'll only require only a few basic commands we'll cover now.

Open the command-line program for your operating system and let's get started. If you need help finding it refer to the prequisite instructions for the *Command-line interface*.

3.1 Print the current directory

Once your terminal window is open the first thing we want to do if find out where you are. If you're using OSX or Linux, type this:

```
$ pwd
```

Note: You don't have to type the "$". It's a generic symbol geeks use to show they're working on the command line.

If you're on Windows try:

```
$ cd
```

The terminal should print out your current location relative to the root of your computer's filesystem. In this case, you're probably in the default directory for your user, also known as your **home** directory.

It's easy to lose track of which folder you're in when you're working from the command line, so this is a helpful tool for finding your way. You'll end up using it a lot more than you might think.

Note: In case you're curious, `pwd` standards "present working directory" and `cd` stands for "change directory," a tool we'll use again soon to move between folders on your file system.

3.2 List files in a directory

In order to see all the files and folders in a directory, there's another command you need to learn. On OSX and Linux, type:

```
$ ls
```

On Windows:

```
$ dir
```

You should now see a list of files and folders appear, such as Downloads, Documents, Desktop, etc. These should look a little familiar. The command line is just another way of navigating the directory structure you're probably used to seeing when clicking around your computer's folders in the user-interface provided by your operating system.

3.3 Change directories

Now let's move. In order to change directories from the command line, we'll return to the `cd` command we saw earlier, which works for OSX, Linux and Windows.

The only thing you need to do is tell it which directory to move into. In this case, the following will probably drop you on your desktop.

```
$ cd Desktop
```

Now run `ls` or `dir` to see what files we can find there. They should mirror what you see as you look at your desktop in your operating system's user interface.

To move back to our home folder, we'll use the `cd` command again, but with a little twist.

```
$ cd ..
```

You'll notice that will move you back to the home directory where we began. When you're working from the command line, it helps to think of your directory structure as a tree. Navigating through the directories is like going higher and lower on various branches. The convention for moving backwards is `..`

3.4 Creating directories and files

You might also find it useful sometimes to create files and directories from the command line. Let's create a folder called `Code` under our home directory that we can use to store code from this class.

Using OSX or Linux, here's how:

```
$ mkdir Code
```

In Windows, try this:

```
$ md Code
```

Next let's jump into the directory. If you remember, that goes like this:

```
$ cd Code
```

If you type `ls` or `dir` you'll notice that nothing is there. That's because all we've done so far is create a directory, but we haven't put any files in it yet.

You won't have to do this very often, but the command for creating a blank file in OSX and Linux is called `touch`. So here's how you make a new file named `test.py`.

```
$ touch test.py
```

There's no similar command in Windows, but you can accomplish the same thing by saving a file from a text editor or other program into our new directory.

3.5 Deleting directories and files

If you wanted to remove the file you just made, here's how on OSX and Linux:

```
$ rm test.py
```

And here's how in Windows:

```
$ del test.py
```

> **Warning:** **This must be done with caution**. Files you delete from the command line do not go into the recycle bin. They are gone. **Forever**.

And that's it! You've learned all the basic command-line tricks necessary to move on.

Act 2: Python

Python can be used for almost any application you can imagine, from building websites to running robots.

A thorough overview of the language would take months, so our class is going to concentrate on the absolute basics – basic principles that you need to understand as you complete this course.

4.1 How to run a Python program

A Python file is nothing more than a text file that has the extension ".py" at the end of its name. Any time you see a ".py" file, you can run it from the command line by typing into the command line:

```
$ python filename.py
```

That's it. And it works for both OSX and Windows.

Python also comes with a very neat feature called an **interactive interpreter**, which allows you to execute Python code one line at a time, sort of like working from the command line.

We'll be using this a lot in the beginning to demonstrate concepts, but in the real world it's often useful for testing and debugging.

To open the interpreter, simply type `python` from your command line, like this.

```
$ python
```

And here's what you should get.

Next we'll use the interpreter to walk through a handful of basic concepts you need to understand if you're going to be writing code, Python or otherwise.
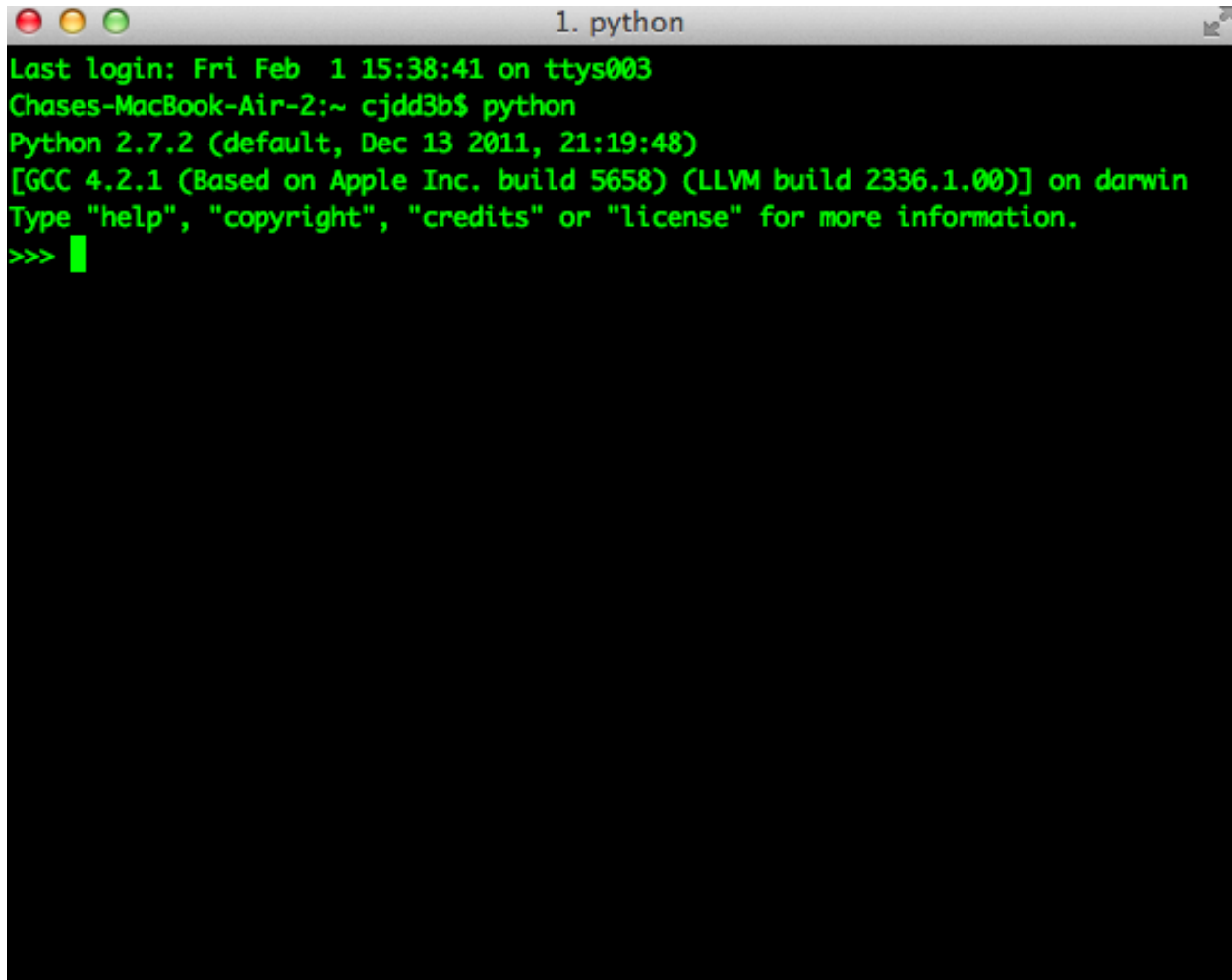
4.2 Variables

Variables are like containers that hold different types of data so you can go back and refer to them later. They're fundamental to programming in any language, and you'll use them all the time.

To try them out, open your Python interpreter.

```
$ python
```

Now let's start writing Python!

```
>>> greeting = "Hello, world!"
```

In this case, we've created a **variable** called `greeting` and assigned it the **string value** "Hello, world!".

In Python, variable assignment is done with the = sign. On the left is the name of the variable you want to create (it can be anything) and on the right is the value that you want to assign to that variable.

If we use the `print` command on the variable, Python will output "Hello, world!" to the terminal because that value is stored in the variable.

```
>>> print greeting
Hello world!
```

4.3 Data types

Variables can contain many different kinds of data types. There are integers, strings, floating point numbers (decimals), and other types of data that languages like SQL like to deal with in different ways.

Python is no different. In particular, there are six different data types you will be dealing with on a regular basis: strings, integers, floats, lists, tuples and dictionaries. Here's a little detail on each.

4.3.1 Strings

Strings contain text values like the "Hello, world!" example above. There's not much to say about them other than that they are declared within single or double quotes like so:

```
>>> greeting = "Hello, world!"
>>> goodbye = "Seeya later, dude."
>>> favorite_animal = 'Donkey'
```

4.3.2 Integers

Integers are whole numbers like 1, 2, 1000 and 1000000. They do not have decimal points. Unlike many other variable types, integers are not declared with any special type of syntax. You can simply assign them to a variable straight away, like this:

```
>>> a = 1
>>> b = 2
>>> c = 1000
```

4.3.3 Floats

Floats are a fancy name for numbers with decimal points in them. They are declared the same way as integers but have some idiosyncracies you don't need to worry about for now.

```
>>> a = 1.1
>>> b = 0.99332
>>> c = 100.123
```

4.3.4 Lists

Lists are collections of values or variables. They are declared with brackets like these `[]`, and items inside are separated by commas. They can hold collections of any type of data, including other lists. Here are several examples:

```
>>> list_of_numbers = [1, 2, 3, 4, 5]
>>> list_of_strings = ['a', 'b', 'c', 'd']
>>> list_of_both = [1, 'a', 2, 'b']
>>> list of lists = [[1, 2, 3], [4, 5, 6], ['a', 'b', 'c']]
```

Lists also have another neat feature: The ability to retrieve individual items. In order to get a specific item out of a list, you just pass in its position.

All lists in Python are **zero-indexed**, which means the first item in them sits at position 0.

```
>>> my_list = ['a', 'b', 'c', 'd']
>>> my_list[0]
'a'
>>> my_list[2]
'c'
```

You can also extract a range of values by specifiying the first and last positions you want to retrieve with a colon in between them, like this:

```
>>> my_list[0:2]
['a', 'b']
```

4.3.5 Tuples

Tuples are a special type of list that cannot be changed once they are created. That's not especially important right now. All you need to know is that they are declared with parentheses `()`. For now, just think of them as lists.

```
>>> tuple_of_numbers = (1, 2, 3, 4, 5)
>>> tuple_of_strings = ('a', 'b', 'c', 'd')
```

4.3.6 Dictionaries

Dictionaries are probably the most difficult data type to explain, but also among the most useful. In technical terms, they are storehouses for pairs of keys and values. You can think of them like a phonebook.

An example will make this a little more clear.

```
>>> my_phonebook = {'Mom': '713-555-5555', 'Chinese Takeout': '573-555-5555'}
```

In this example, the keys are the names "Mom" and "Chinese takeout", which are declared as strings (Python dictionary keys usually are).

The values are the phone numbers, which are also strings, although dictionary values in practice can be any data type.

If you wanted to get Mom's phone number from the dictionary, here's how:

```
>>> my_phonebook['Mom']
713-555-5555
```

There's a lot more to dictionaries, but that's all you need to know for now.

4.4 Control structures

As a beginner your first Python scripts won't be much more complicated that a series of commands that execute one after another, working together to accomplish a task.

In those situations, it is helpful to be able to control the order and conditions under which those commands will run.

That's where control structures come in – simple logical operators that allow you to execute parts of your code when the right conditions call for it.

Here are two you will end up using a lot.

4.4.1 The if clause

If statements are pretty much exactly what they sound like. **If** a certain condition is met, your program should do something.

Let's start with a simple example.

```
>>> number = 10
>>> if number > 5:
>>>     print "Wow, that's a big number!"
>>>
Wow, that's a big number!
```

Our little program in this case starts with a variable, which we've called `number`, being set to 10. That's pretty simple, and a concept you should be familiar with by this point.

```
>>> number = 10
>>> if number > 5:
>>>     print "Wow, that's a big number!"
```

The next line, `if number > 5:` declares our `if` statement. In this case, we want something to happen if the `number` variable is greater than 5.

```
>>> number = 10
>>> if number > 5:
>>>     print "Wow, that's a big number!"
```

Most of the if statements we build are going to rely on equality operators like the kind we learned in elementary school: greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=) and plain old "equals". The equals operator is a little tricky, in that it is declared with two equals signs (==), not one (=). Why is that? Because you'll remember from above that a single equals sign is the notation we use to assign a value to a variable!

Next, take note of the indentation. In Python, whitespace matters. A lot. Notice that I said indents must be four spaces. Four spaces means four spaces – not a tab.

```
>>> number = 10
>>> if number > 5:
>>>     print "Wow, that's a big number!"
```

Tabs and spaces are different. To avoid problems, you should press the space bar four times whenever you indent Python code.

Note: There are some text editors that will automatically convert tabs to spaces, and once you feel more comfortable you might want to use one. But for now, get in the habit of making all indents four spaces.

If you look closely, there's another small detail you need to remember: The colon! When we declare an `if` statement, we always end that line with a colon.

```
>>> number = 10
>>> if number > 5:
>>>     print "Wow, that's a big number!"
>>>
>>> print "I execute no matter what your number is!"
```

It helps sometimes to think of your program as taking place on different levels.

In this case, the first level of our program (the one that isn't indented) has us declaring the variable `number = 10` and setting up our `if` condition, `if number > 5:`.

The second level of our program executes only on the condition that our `if` statement is true. Therefore, because it depends on that `if` statement, it is indented four spaces.

If we wanted to continue our program back on the first level, we could do something like this:

```
>>> number = 10
>>> if number > 5:
>>>     print "Wow, that's a big number!"
>>>
>>> print "I execute no matter what your number is!"
>>>
Wow, that's a big number!
I execute no matter what your number is!
```

The last statement doesn't depend on the `if` statement, so it will always run.

4.4.2 The else clause

Now let's talk about a common companion for `if` statement – the `else` clause. It can be combined with an `if` statement to have the script execute a block of code when it turns out not to be true.

You don't need to have an `else` condition for your `if` statements, but sometimes it helps. Consider this example:

```
number = 10
if number > 5:
    print "Wow, that's a big number!"
else:
    print "Gee, that number's kind of small, don't you think?"
```

In this case, we're telling our program to print one thing if `number` is greater than five, and something else if it's not. Notice that the `else` statement also ends with a colon, and as such its contents are also indented four spaces.

4.4.3 For loops

Remember earlier we discussed the concept of a list – the type of variable that can hold multiple items in it all at once?

Many times during your programming career, you'll find it helps to run through an entire list of items and do something with all of them, one at a time.

That's where for loops come in. Let's start by having Python say the ABC's:

```
>>> list_of_letters = ['a', 'b', 'c']
>>> for letter in list_of_letters:
>>>     print letter
>>>
```

```
a
b
c
```

The output of this statement is what you might guess. But there are still a few things to unpack here – some familiar and some not.

First, you'll notice from looking at the print statement that our indentation rules still apply. Everything that happens within the `for` loop must still be indented four spaces from the main level of the program. You'll also see that the line declaring the loop ends in a colon, just like the if and else statements.

Second, turn your attention to the syntax of declaring the loop itself.

```
>>> list_of_letters = ['a', 'b', 'c']
>>> for letter in list_of_letters:
>>>     print letter
```

All of our for loops start, unsurprisingly, with the word `for` and follow the pattern `for variable_name in list:`. The `variable_name` can be anything you want – it's essentially just a new variable you're creating to refer to each item within your list as the `for` loop iterates over it.

In this case we chose `letter`, but you could just as easily call it `donkey`, like so:

```
>>> list_of_letters = ['a', 'b', 'c']
>>> for donkey in list_of_letters:
>>>     print donkey
```

The next thing you have to specify is the list you want to loop over, in this case `list_of_letters`. The line ends with a colon, and the next line starts with an indent. And that's the basics of building a loop!

4.4.4 Functions

Often it's helpful to encapsulate a sequence of programming instructions into little tools that can be used over and over again. That's where functions come in.

Think of functions like little boxes. They take input (known as **arguments**), perform some operations on those arguments, and then return an **output**.

In Python, a simple function might take an integer and divide it by two, like this:

```
>>> def divide_by_two(x):
>>>     return x / 2.0
```

In order to call that function later in the program, I would simply have to invoke its name and feed it an integer – any integer at all – like so:

```
>>> def divide_by_two(x):
>>>     return x / 2.0
>>> divide_by_two(10)
5
```

Once you write a function (assuming it works) you don't need to know what's inside. You can just feed it an input and expect an output in return.

Every function must be declared by the word `def`, which stands for "define". That is followed by the name of the function. Like the variable in a ''loop' you can call it anything you want.

```
>>> def get_half(x):
>>>     return x / 2.0
```

The name is then followed by a set of parentheses in which you can define the arguments the function should expect. In our example above, we've called the only argument``x``.

When we feed a value in, like the number 10, a variable by the name of our argument is created within the function. You can name that what you want too.

```
>>> def get_half(num):
>>>     return num / 2.0
```

After you finish declaring arguments, you'll see something familiar –the colon. Just like the `if` statements and `for` loops, the next line must be indented four spaces because any code within the function is nested one level deeper than the base level of the program.

Most functions return some kind of output. Arguments go in, some processing happens, and something comes out. That's what the `return` statement is for.

```
>>> def get_half(num):
>>>     return num / 2.0
```

Functions don't necessarily need arguments, nor do they always need to return a value using the `return` command. You could also do something like this:

```
def say_hello():
    print "Hello!"
```

But the idea of arguments and `return` values are still fundamental in understanding functions, and they will come up more often than not.

4.5 Python as a toolbox

Lucky for us, Python already has tools filled with functions to do pretty much anything you'd ever want to do with a programming language: everything from navigating the web to scraping and analyzing data to performing mathematical operations to building websites.

Some of these are built into a toolbox that comes with the language, known as the **standard library**. Others have been built by members of the developer community and can be downloaded and installed from the web.

There are two ways to import these tools into your scripts.

To pull in an entire toolkit, use the `import` command. In this case, we'll get the `urllib2` package, which allows us to visit websites with Python:

```
>>> import urllib2
>>> urllib2.urlopen("http://www.python.org/")
```

You can also import specific tools from inside a toolkit by working in the `from` command with something like this:

```
>>> from urllib2 import urlopen
>>> urlopen("http://www.python.org/")
```

In practice, you'll use both of these methods.

Note: There's no rule but most Python programmers try to keep things manageable by lining up all `import` statements at the top of each script.

Act 3: Web scraping

Now that we've covered all the fundamentals, it's time to get to work and write a web scraper.

The target is a regularly updated roster of inmates at the Boone County Jail in Missouri. Boone County is home to Columbia, where you can find the University of Missouri's main campus and the headquarters of Investigative Reporters and Editors.

5.1 Installing dependencies

The scraper will use Python's BeautifulSoup toolkit to parse the site's HTML and extract the data.

We'll also use the Requests library to open the URL, download the HTML and pass it to BeautifulSoup.

Since they are not included in Python's standard library, we'll first need to install them using `pip`, a command-line tool that can grab open-source libraries off the web. If you don't have it installed, you'll need to follow the prequisite instructions for *pip*.

In OSX or Linux try this:

```
$ sudo pip install BeautifulSoup
$ sudo pip install Requests
```

On Windows give it a shot with the `sudo`.

```
$ pip install BeautifulSoup
$ pip install Requests
```

5.2 Analyzing the HTML

HTML is the framework that, in most cases, contains the content of a page. Other bits and pieces like CSS and JavaScript can style, reshape and add layers of interaction to a page.

But unless you've got something fancy on your hands, the data you're seeking to scrape is usually somewhere within the HTML of the page and your job is to write a script in just the write way to walk through it and pull out the data. In this case, we'll be looking to extract data from the big table that makes up the heart of the page.

By the time we're finished, we want to have extracted that data, now encrusted in layers of HTML, into a clean spreadsheet.

In order to scrape a website, we need to understand how a typical webpage is put together.

Current Inmates of Boone County Jail

Date: 12/9/2014

Last Name	First Name	Middle	Gender	Race	Age	City	State
Adam	Ahmed	Abubaker	M	B	29	Columbia	MO
Anderson	Isaiah	Dwight	M	B	18	Columbia	MO
Anderson	Michael	Leon	M	B	23	Columbia	MO
Armendarez	Leonel		M	H	32	Columbia	MO
Baker	Cody	Marshall	M	W	32	Ashland	MO
Ball	Reginald	Eric	M	B	45	Columbia	MO
Banks	Antonio	Laron	M	B	22	Jefferson City	MO
Barney	Shaneika	Renne	F	B	26	Columbia	MO
Berends	Lorrie	Ann	F	W	56	Columbia	MO
Berryhill	Theodore	Bud	M	W	56	Columbia	MO
Birmingham	Andrew	Ryan	M	W	19	Auvasse	MO
Black	Aaron	Liam James	M	B	20	Columbia	MO
Bowen	Steven	Wayne	M	W	43	Columbia	MO
Breaux	Jason	Allen	M	W	32	Eldon	MO
Broderick	James	Edward	M	W	43	Columbia	MO
Brookins	Gary	Laren	M	B	56	Columbia	MO
Brown	Mattie	Lynn	F	W	20	Columbia	MO
Brown	Terrell		M	B	41	Columbia	MO
Bryant	Randi-Jo	Nikole	F	W	21	Columbia	MO
Bryant	Ronald	Dean	M	W	70	Columbia	MO
Buck	Robert	Lindsey	M	W	49	Columbia	MO

	A	B	C	D	E	F	G	H
1	Last	First	Middle	Gender	Race	Age	City	State
2	Adam	Ahmed	Abubaker	M	B	29	Columbia	MO
3	Anderson	Isaiah	Dwight	M	B	18	Columbia	MO
4	Anderson	Michael	Leon	M	B	23	Columbia	MO
5	Armendarez	Leonel		M	H	32	Columbia	MO
6	Baker	Cody	Marshall	M	W	32	Ashland	MO
7	Ball	Reginald	Eric	M	B	45	Columbia	MO
8	Banks	Antonio	Laron	M	B	22	Jefferson City	MO
9	Barney	Shaneika	Renne	F	B	26	Columbia	MO
10	Berends	Lorrie	Ann	F	W	56	Columbia	MO
11	Berryhill	Theodore	Bud	M	W	56	Columbia	MO
12	Birmingham	Andrew	Ryan	M	W	19	Auvasse	MO
13	Black	Aaron	Liam James	M	B	20	Columbia	MO
14	Bowen	Steven	Wayne	M	W	43	Columbia	MO
15	Breaux	Jason	Allen	M	W	32	Eldon	MO
16	Broderick	James	Edward	M	W	43	Columbia	MO
17	Brookins	Gary	Laren	M	B	56	Columbia	MO
18	Brown	Mattie	Lynn	F	W	20	Columbia	MO
19	Brown	Terrell		M	B	41	Columbia	MO
20	Bryant	Randi-Jo	Nikole	F	W	21	Columbia	MO
21	Bryant	Ronald	Dean	M	W	70	Columbia	MO
22	Buck	Robert	Lindsey	M	W	49	Columbia	MO

To view the HTML code that makesup this page () open up a browser and visit out target. Then right click with your mouse and select "View Source." You can do this for any page on the web.

We could fish through all the code to find our data, but to dig this more easily, we can use your web browser's inspector tool. Right click on the table of data that you are interested in and select 'inspect element.'

Note: The inspector tool might have a slightly different name depending on which browser you're using. To make this easy on yourself, consider using Google Chrome.

Your browser will open a special panel and highlight the portion of the page's HTML code that you've just clicked on.

There are many ways to grab content from HTML, and every page you scrape data from will require a slightly different trick.

At this stage, your job is to find a pattern or identifier in the code for the elements you'd like to extract, which we will then give as instructions to our Python code.
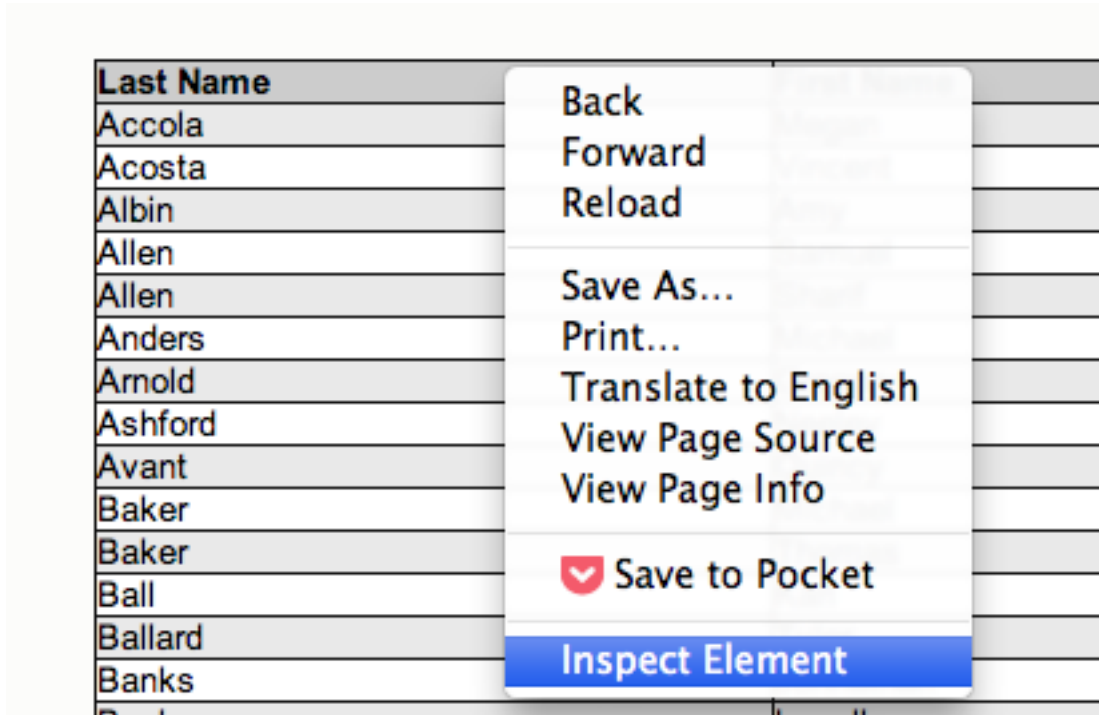
In the best cases, you can extract content by using the `id` or `class` already assigned to the element you'd like to extract. An 'id' is intended to act as the unique identifer a specific item on a page. A 'class' is used to label a specific type of item on a page. So, there maybe may instances of a class on a page.

On Boone County's page, there is only table in the HTML's `body` tag. The table is identified by a class.

```
<table class="resultsTable" style="margin: 0 auto; width: 90%; font-size: small;">
```

5.3 Extracting an HTML table

Now that we know where to find the data we're after, it's time to write script to pull it down and save it to a comma-delimited file.

Let's start by creating a Python file to hold our scraper. First jump into the `Code` directory we made at the beginning of this lesson.

```
$ cd Code
```

Note: You'll need to `mkdir Code` (or `md Code` in Windows) if you haven't made this directory yet.

Then open your text editor and save an empty file into the directory name `scrape.py` and we're ready to begin. The first step is to import the requests library and download the Boone County webpage.

```python
import requests

url = 'http://www.showmeboone.com/sheriff/JailResidents/JailResidents.asp'
response = requests.get(url)
html = response.content
print html
```

Save the file and run this script from your command line and you should see the entire HTML of the page spilled out.

```
$ python scrape.py
```

Next import the `BeautifulSoup` HTML parsing library and feed it the page.

```python
import requests
from BeautifulSoup import BeautifulSoup

url = 'http://www.showmeboone.com/sheriff/JailResidents/JailResidents.asp'
response = requests.get(url)
html = response.content

soup = BeautifulSoup(html)
print soup.prettify()
```

Save the file and run the script again and you should see the page's HTML again, but in a prettier format this time. That's a hint at the magic happening inside BeautifulSoup once it gets its hands on the page.

```
$ python scrape.py
```

Next we take all the detective work we did with the page's HTML above and convert it into a simple, direct command that will instruct BeautifulSoup on how to extract only the table we're after.

```python
import requests
from BeautifulSoup import BeautifulSoup

url = 'http://www.showmeboone.com/sheriff/JailResidents/JailResidents.asp'
response = requests.get(url)
html = response.content

soup = BeautifulSoup(html)
table = soup.find('tbody', attrs={'class': 'stripe'})
print table.prettify()
```

Save the file and run `scrape.py` again. This time it only prints out the table we're after, which was selected by instructing BeautifulSoup to return only those `<table>` tags with `resultsTable` as their class attribute.

```
$ python scrape.py
```

Now that we have our hands on the table, we need to convert the rows in the table into a list, which we can then loop through and grab all the data out of.

5.3. Extracting an HTML table 23

BeautifulSoup gets us going by allowing us to dig down into our table and return a list of rows, which are created in HTML using `<tr>` tags inside the table.

```python
import requests
from BeautifulSoup import BeautifulSoup

url = 'http://www.showmeboone.com/sheriff/JailResidents/JailResidents.asp'
response = requests.get(url)
html = response.content

soup = BeautifulSoup(html)
table = soup.find('tbody', attrs={'class': 'stripe'})

for row in table.findAll('tr'):
    print row.prettify()
```

Save and run the script. You'll not see each row printed out separately as the script loops through the table.

```
$ python scrape.py
```

Next we can loop through each of the cells in each row by select them inside the loop. Cells are created in HTML by the `<td>` tag.

```python
import requests
from BeautifulSoup import BeautifulSoup

url = 'http://www.showmeboone.com/sheriff/JailResidents/JailResidents.asp'
response = requests.get(url)
html = response.content

soup = BeautifulSoup(html)
table = soup.find('tbody', attrs={'class': 'stripe'})

for row in table.findAll('tr'):
    for cell in row.findAll('td'):
        print cell.text
```

Again, save and run the script. (This might seem repetitive, but it is the constant rhythm of many Python programmers.)

```
$ python scrape.py
```

When that prints you will notice some annoying ` ` on the end of many lines. That is the HTML code for a **non-breaking space**, which forces the browser to render an empty space on the page. It is junk and we can delete it easily with this handy Python trick.

```python
import requests
from BeautifulSoup import BeautifulSoup

url = 'http://www.showmeboone.com/sheriff/JailResidents/JailResidents.asp'
response = requests.get(url)
html = response.content

soup = BeautifulSoup(html)
table = soup.find('tbody', attrs={'class': 'stripe'})

for row in table.findAll('tr'):
    for cell in row.findAll('td'):
        print cell.text.replace(' ', '')
```

Save and run the script. Everything should be much better.

```
$ python scrape.py
```

Now that we have found the data we want to extract, we need to structure it in a way that can be written out to a comma-delimited text file. That won't be hard since CSVs aren't any more than a grid of columns and rows, much like a table.

Let's start by adding each cell in a row to a new Python list.

```python
import requests
from BeautifulSoup import BeautifulSoup

url = 'http://www.showmeboone.com/sheriff/JailResidents/JailResidents.asp'
response = requests.get(url)
html = response.content

soup = BeautifulSoup(html)
table = soup.find('tbody', attrs={'class': 'stripe'})

for row in table.findAll('tr'):
    list_of_cells = []
    for cell in row.findAll('td'):
        text = cell.text.replace(' ', '')
        list_of_cells.append(text)
    print list_of_cells
```

Save and rerun the script. Now you should see Python lists streaming by one row at a time.

```
$ python scrape.py
```

Those lists can now be lumped together into one big list of lists, which, when you think about it, isn't all that different from how a spreadsheet is structured.

```python
import requests
from BeautifulSoup import BeautifulSoup

url = 'http://www.showmeboone.com/sheriff/JailResidents/JailResidents.asp'
response = requests.get(url)
html = response.content

soup = BeautifulSoup(html)
table = soup.find('tbody', attrs={'class': 'stripe'})

list_of_rows = []
for row in table.findAll('tr'):
    list_of_cells = []
    for cell in row.findAll('td'):
        text = cell.text.replace(' ', '')
        list_of_cells.append(text)
    list_of_rows.append(list_of_cells)

print list_of_rows
```

Save and rerun the script. You should see a big bunch of data dumped out into the terminal. Look closely and you'll see the list of lists.

```
$ python scrape.py
```

To write that list out to a comma-delimited file, we need to import Python's built-in `csv` module at the top of the file. Then, at the bottom, we will create a new file, hand it off to the `csv` module, and then lead on a handy tool it has called `writerows` to dump out our list of lists.

5.3. Extracting an HTML table

```python
import csv
import requests
from BeautifulSoup import BeautifulSoup

url = 'http://www.showmeboone.com/sheriff/JailResidents/JailResidents.asp'
response = requests.get(url)
html = response.content

soup = BeautifulSoup(html)
table = soup.find('tbody', attrs={'class': 'stripe'})

list_of_rows = []
for row in table.findAll('tr'):
    list_of_cells = []
    for cell in row.findAll('td'):
        text = cell.text.replace(' ', '')
        list_of_cells.append(text)
    list_of_rows.append(list_of_cells)

outfile = open("./inmates.csv", "wb")
writer = csv.writer(outfile)
writer.writerows(list_of_rows)
```

Save and run the script. Nothing should happen – at least to appear to happen.

```
$ python scrape.py
```

Since there are no longer any print statements in the file, the script is no longer dumping data out to your terminal. However, if you open up your code directory you should now see a new file named inmates.csv waiting for you. Open it in a text editor or Excel and you should see structured data all scraped out.

There is still one obvious problem though. There are no headers!

Here's why. If you go back and look closely, our script is only looping through lists of <td> tags found within each row. Fun fact: Header tags in HTML tables are often wrapped in the slightly different <th> tag. Look back at the source of the Boone County page and you'll see that's what exactly they do.

But rather than bend over backwards to dig them out of the page, let's try something a little different. Let's just skip the first row when we loop though, and then write the headers out ourselves at the end.

```python
import csv
import requests
from BeautifulSoup import BeautifulSoup

url = 'http://www.showmeboone.com/sheriff/JailResidents/JailResidents.asp'
response = requests.get(url)
html = response.content

soup = BeautifulSoup(html)
table = soup.find('tbody', attrs={'class': 'stripe'})

list_of_rows = []
for row in table.findAll('tr')[1:]:
    list_of_cells = []
    for cell in row.findAll('td'):
        text = cell.text.replace(' ', '')
        list_of_cells.append(text)
    list_of_rows.append(list_of_cells)
```

	A	B	C	D	E	F	G	H
1								
2	Adam	Ahmed	Abubaker	M	B	29	Columbia	MO
3	Anderson	Isaiah	Dwight	M	B	18	Columbia	MO
4	Anderson	Michael	Leon	M	B	23	Columbia	MO
5	Armendarez	Leonel		M	H	32	Columbia	MO
6	Baker	Cody	Marshall	M	W	32	Ashland	MO
7	Ball	Reginald	Eric	M	B	45	Columbia	MO
8	Banks	Antonio	Laron	M	B	22	Jefferson City	MO
9	Barney	Shaneika	Renne	F	B	26	Columbia	MO
10	Berends	Lorrie	Ann	F	W	56	Columbia	MO
11	Berryhill	Theodore	Bud	M	W	56	Columbia	MO
12	Birmingham	Andrew	Ryan	M	W	19	Auvasse	MO
13	Black	Aaron	Liam James	M	B	20	Columbia	MO
14	Bowen	Steven	Wayne	M	W	43	Columbia	MO
15	Breaux	Jason	Allen	M	W	32	Eldon	MO
16	Broderick	James	Edward	M	W	43	Columbia	MO
17	Brookins	Gary	Laren	M	B	56	Columbia	MO
18	Brown	Mattie	Lynn	F	W	20	Columbia	MO
19	Brown	Terrell		M	B	41	Columbia	MO
20	Bryant	Randi-Jo	Nikole	F	W	21	Columbia	MO
21	Bryant	Ronald	Dean	M	W	70	Columbia	MO
22	Buck	Robert	Lindsey	M	W	49	Columbia	MO

5.3. Extracting an HTML table

```
outfile = open("./inmates.csv", "wb")
writer = csv.writer(outfile)
writer.writerow(["Last", "First", "Middle", "Gender", "Race", "Age", "City", "State"])
writer.writerows(list_of_rows)
```

Save and run the script one last time.

```
$ python scrape.py
```

Our headers are now there, and you've finished the class. Congratulations! You're now a web scraper.

	A	B	C	D	E	F	G	H
1	Last	First	Middle	Gender	Race	Age	City	State
2	Adam	Ahmed	Abubaker	M	B	29	Columbia	MO
3	Anderson	Isaiah	Dwight	M	B	18	Columbia	MO
4	Anderson	Michael	Leon	M	B	23	Columbia	MO
5	Armendarez	Leonel		M	H	32	Columbia	MO
6	Baker	Cody	Marshall	M	W	32	Ashland	MO
7	Ball	Reginald	Eric	M	B	45	Columbia	MO
8	Banks	Antonio	Laron	M	B	22	Jefferson City	MO
9	Barney	Shaneika	Renne	F	B	26	Columbia	MO
10	Berends	Lorrie	Ann	F	W	56	Columbia	MO
11	Berryhill	Theodore	Bud	M	W	56	Columbia	MO
12	Birmingham	Andrew	Ryan	M	W	19	Auvasse	MO
13	Black	Aaron	Liam James	M	B	20	Columbia	MO
14	Bowen	Steven	Wayne	M	W	43	Columbia	MO
15	Breaux	Jason	Allen	M	W	32	Eldon	MO
16	Broderick	James	Edward	M	W	43	Columbia	MO
17	Brookins	Gary	Laren	M	B	56	Columbia	MO
18	Brown	Mattie	Lynn	F	W	20	Columbia	MO
19	Brown	Terrell		M	B	41	Columbia	MO
20	Bryant	Randi-Jo	Nikole	F	W	21	Columbia	MO
21	Bryant	Ronald	Dean	M	W	70	Columbia	MO
22	Buck	Robert	Lindsey	M	W	49	Columbia	MO