



Community Experience Distilled

Automate it!

**Put your life on Autopilot with the magic
and power of Python**

Chetan Giridhar

[PACKT] open source*
PUBLISHING community experience distilled

Table of Contents

Chapter 1: Working with the Web	1
Introduction	1
Making HTTP Requests	2
Getting ready...	2
How to do it...	2
How it works...	4
There's more...	5
Parsing HTML Content	5
Getting ready...	5
How to do it...	5
How it works...	8
There's more...	8
Downloading content from the Web	9
Getting ready	9
How to do it...	9
How it works...	11
Web Scraping	12
Getting ready	12
How to do it...	12
How it works...	15
There's more...	15
Working with third Party Rest APIs	16
Getting ready	16
How to do it...	16
How it works...	22
See also	22
Writing you own Web App in Python	22
Getting ready	22
How to do it...	23
How it works...	26
There's more...	27
Asynchronous HTTP Server with Python	27
Getting ready	28
How to do it...	28

How it works...	29
There's more...	30
Web automation with selenium bindings	30
Getting ready...	30
How to do it...	30
How it works...	32
There's more...	33

1

Working with the Web

In this chapter, we will cover the following recipes:

- Making HTTP Requests
- Parsing HTML Content
- Downloading content from the web
- Web scraping
- Working with 3rd party Rest APIs
- Writing your own Web Application in Python
- Asynchronous HTTP Server in Python
- Web Automation with selenium bindings

Introduction

Internet has made life so easy that sometimes you just don't realize the powers of it. Checking out your friend's status, calling your parents, responding to an important business email or playing a game – for almost everything we rely on the World Wide Web today.

Thankfully, Python has rich set of modules that help us perform various tasks on the Web. Not only could you make simple HTTP requests to retrieve data from websites or download pages and images, you could also parse the page content to gather information and analyze it to generate meaningful insights with Python. And wait; did I mention that you could spawn a browser in an automated fashion to perform a daily mundane task?

The recipes in this chapter will primarily focus on Python modules that can be treated as the tool of choice while performing above operations on the web. Specifically, we will focus on the following Python modules in this chapter:

- `requests` (<http://docs.python-requests.org/en/master/>)
- `urllib2` (<https://docs.python.org/2/library/urllib2.html>)
- `lxml` (<https://pypi.python.org/pypi/lxml>)
- `beautifulsoup4` (<https://pypi.python.org/pypi/beautifulsoup4>)
- `selenium` (<http://selenium-python.readthedocs.org/>)



While the recipes in this chapter will give you an overview of how to interact with web using Python modules, I encourage you to try out and develop code for multiple use cases, which will benefit you as an individual and your project at an organization scale.

Making HTTP Requests

Throughout the following recipes in this section, we will use Python v2.7 and `requests` (v2.9.1) module of Python. This recipe will show you how to make HTTP Requests to web pages on the Internet.

But before going there, let's understand HTTP (Hypertext Transfer Protocol) in brief. HTTP is a stateless application protocol for data communication on the World Wide Web. A typical HTTP Session involves a sequence of request/response transaction. Client initiates TCP connection to the Server on dedicated IP and Port, when the Server receives the request, it responds with the response code and text. HTTP defines request methods (HTTP verbs like GET, POST) that indicate the desired action to be taken on the given Web URL.

In this section, we'll learn how make HTTP GET/POST request using Python `requests` module. We'll also learn how to POST json data and handle HTTP Exceptions.

Getting ready...

To step through this recipe, you will need to install Python v2.7. Once installed, you need to install Python pip. Pip stands for “pip installs packages” and is a program that can be used to download and install required Python packages on your computer. Lastly we'll need `requests` module to make HTTP Requests.

We will start by installing `requests` module. (I'll leave the Python and Pip installation for you to perform on your machine based on your Operating System) No other prerequisites are required. So hurry up and let's get going!

How to do it...

1. On your Linux/Mac computer go to Terminal and run command 'pip install -U requests'. You only need to use sudo if you don't have permissions to Python site packages else sudo is not required.
2. Below code helps you make a HTTP GET request with Python's requests module.

```
import requests r = requests.get('http://ip.jsontest.com/')
print("Response object:", r) print("Response Text:", r.text)
```

You will observe the following output.

```
('Response object:', <Response [200]>)
('Response Text:', u'{"ip": "117.213.178.109"}\n')
```

3. Creating a HTTP GET request with data payload is also simple. Below code helps you in achieving this.

```
payload = {'q': 'chetan'} r =
requests.get('https://github.com/search',
params=payload) print("Request URL:", r.url)
```

```
('Request URL:', u'https://github.com/search?q=chetan')
```

4. Let's now make HTTP POST request using requests module. This like POSTing to a login or signup form using requests module.

```
payload = {'key1': 'value1'} r =
requests.post("http://httpbin.org/post",
data=payload) print("Response text:", r.json())
```

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "key1": "value1"
  },
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
    "Accept-Encoding": "gzip, deflate",
    "Accept-Language": "en-US,en;q=0.5",
    "Cache-Control": "no-cache",
    "Content-Length": "11",
    "Content-Type": "application/x-www-form-urlencoded; charset=UTF-8",
    "Host": "httpbin.org",
    "Pragma": "no-cache",
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:38.0) Gecko/20100101"
  },
  "json": null,
  "origin": "117.213.178.109",
  "url": "http://httpbin.org/post"
}
```

5. Handling Errors and Exceptions is also trivial with requests. Below code snippet shows an example of error handling. If you run this code without Internet connection on your machine, it would result in an exception. Exception handler catches the exception and states that it failed to establish a new connection as expected.

```
try:
    r = requests.get("http://www.google.com/")
except requests.exceptions.RequestException
    as e: print("Error Response:", e.message)
```

How it works...

In the above recipes, we looked at how to make different type of HTTP Request with Python requests' module. Let's look at how this code works:

1. In the first example, we made GET request to `http://ip.jsontest.com` and get the response code and response text
2. Second example, we made a HTTP GET request with payload data. Look how the

request URL contains "`?q=chetan`" and it searches all the repositories by name "Chetan" on github.

3. Third, we made a POST with payload data being {'key1', 'value1'}. This is like submitting an online form as we observed in the 'How to do it' section.
4. requests' module has a Response object 'r' that includes various methods. These methods help in extracting response, status code and other information that is required while working with the web:
 - `r.status_code` – Returns the response code
 - `r.json()` – Converts the response to json format
 - `r.text` – Returns the response data for the query
 - `r.content` – includes the HTML and XML tags in the response content
 - `r.url` – Defines the Web URL of the request made
 - We also looked at the exception handling with requests module, where in, if there was no Internet, an exception occurred and requests module could easily catch this exception.

There's more...

Making HTTP requests on the web is just the beginning. There's still more in terms of what we can do with Python. So, what's next?

Parsing HTML Content

Well, now we're confident of making HTTP Requests to multiple URLs. We also know how to get the response code and content. But this isn't enough. If we want to query the web and make sense of the data, we should also know how to parse the different formats in which data is available on web. In this section we discuss how to parse HTML Content.

Getting ready...

To understand how to parse HTML content, we take an example of file. We'll learn how to select certain HTML elements and extract the desired data. For this recipe, you need to install `BeautifulSoup` module of Python. `BeautifulSoup` is one of the most comprehensive Python modules that will do a good job in parsing html content. So let's get started.

How to do it...

1. We start by installing BeautifulSoup on our Python instance. Below command will help us install the module. We install the latest version, which is `beautifulsoup4`.

```
pip install beautifulsoup4
```

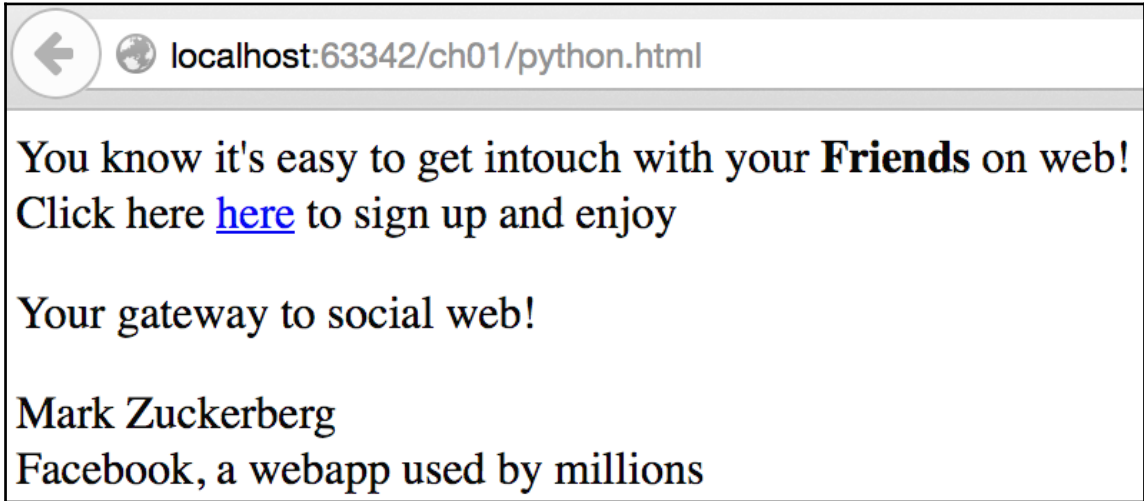
2. Now let us take a look at below html file that will help us learn how to parse html content.

```
<html xmlns="http://www.w3.org/1999/html">
<head>
  <title>Enjoy Facebook!</title>
</head>
<body>
  <p><span>You know it's easy to get in touch with your
  <strong>Friends</strong> on web!<br></span>
  Click here <a href="https://facebook.com">here</a>
  to sign up and enjoy<br>
</p>
  <p class="wow"> Your gateway to social web! </p>
  <div id="inventor">Mark Zuckerberg</div>
  Facebook, a webapp used by millions
</body>
</html>
```

3. Let us name this file as “python.html”. Our html file is hand crafted so that we can learn multiple ways of parsing. Python.html has typical html tags as given below:

```
<head> - Container for all head elements like <title>
<body> - Defines the body of the HTML document
<p> - p element in HTML defines a paragraph
<span> - Used to group inline elements in a document
<strong> - Bolds the text present under this tag
<a> - Represents a hyperlink or anchor. Contains
<href> that points to the hyperlink
<class> - It is an attribute that points to a class in style sheet
<div id> - Div is a container that encapsulates other page elements
and divides the
content into sections. Every section can be identified by attribute
'id'
```

If we open this html in a browser, this is how it'll look like:



4. Let us now write some Python code to parse this html file. We start by creating a BeautifulSoup object.



Note: we always need to define the parser. In this case we used “lxml” as the parser.

```
__author__ = 'Chetan' import bs4 myfile = open('python.html') soup =  
bs4.BeautifulSoup(myfile, "lxml")  
#Making the soup print "BeautifulSoup Object:", type(soup)
```

Output of the above code is:

```
BeautifulSoup Object: <class 'bs4.BeautifulSoup'>
```

5. We can select or find html elements using different ways. We could select elements with Id, CSS or Tags. Below code uses python.html to demonstrate this concept:

```
#Find Elements By tags print soup.find_all('a') print  
soup.find_all('strong') #Find Elements By id print
```

```
soup.find('div', {"id":"inventor"}) print
soup.select('#inventor') #Find Elements by css print
soup.select('.wow')
```

Output of the above code is:

```
[<a href="https://facebook.com">here</a>]
[<strong>Friends</strong>]
<div id="inventor">Mark Zuckerberg</div>
[<div id="inventor">Mark Zuckerberg</div>]
[<p class="wow"> Your gateway to social web! </p>]
```

- Now let's move on and get the actual content from the html file. Below are a few ways we can extract the data of interest.

```
print "Facebook URL:", soup.find_all('a')[0]['href']print
print "Inventor:", soup.find('div', {"id":"inventor"}).textprint
print "Span content:", soup.select('span')[0].getText()
```

Output of the above code snippet is:

```
Facebook URL: https://facebook.com
Inventor: Mark Zuckerberg
Span content: You know it's easy to get intouch with your Friends on web!
```

How it works...

In the above recipe, we looked at how to find or select different html elements based on Id, CSS or Tags. When we use `find_all()`, we get multiple instances of the match as an array. `select()` method helps you reach the element directly. We then use the object obtained from `select()` or `find_all()` and use it get the content. Methods like `getText()` and attributes like `text` (as seen iin the above examples) help in extracting the content.

There's more...

Hey, so we understood how to parse html file (or a web page) with Python. We learnt how we select or find html elements by id, css or tags. We also looked at examples on how to extract the require content from html.

Downloading content from the Web

So in the earlier recipes we saw how to make http requests and also learnt how to parse html content. It's time to move ahead and download content from the web. You know World Wide Web is not just about html pages. It contains other resources like txt files, documents and images amongst many formats. Here in this recipe we'll learn ways to download images with an example.

Getting ready

To download images, we will need two Python modules namely: BeautifulSoup and urllib2. We could use requests module instead of urllib2, but this will help you learn about urllib2 as an alternative that can be used for http requests.

How to do it...

1. We identify the kind of images we would like to download and from which location. In this recipe we download 'Avatar' movie images and from Google (https://google.com) images search. We download the top 5 images

For doing this lets import Python modules and define variables that we'll need:

```
from bs4 import BeautifulSoupimport reimport urllib2import os
image_type = "Project"movie = "Avatar"url =
"https://www.google.com/search?q="+movie+"&source=lnms&tbn=isch"
```

2. Ok then, let us now create a beautifulsoup object with URL parameters and appropriate headers

```
header = {'User-Agent': 'Mozilla/5.0'} soup =
BeautifulSoup(urllib2.urlopen(urllib2.Request(url, headers=header)))
```

3. Google images are hosted as static content under the domain name 'gstatic.com'.

So using the BeautifulSoup object, we now try to find_all() the images whose source contains 'gstatic.com'. Below code does exactly the same thing we need:

```
images = [a['src'] for a in soup.find_all("img",
{"src": re.compile("gstatic.com")})][:5]
for img in images:
    print "Image Source:", img
```

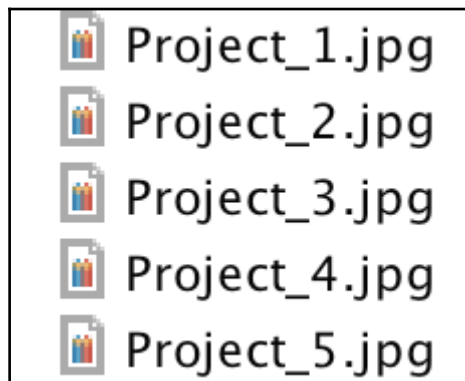
Output of the above code

```
Image Source: https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9Gc0_BRDYS6jrzyTZVIXuIjysRu20IGopCbcSt0QINWzBeoK0cuoyI9bhzp0
Image Source: https://encrypted-tbn3.gstatic.com/images?q=tbn:ANd9GcTu5xVwLkYHKP8HE5w-651EHJduNexoolYW0-eaC4SGYTPeI-9z-sa_ohY9
Image Source: https://encrypted-tbn1.gstatic.com/images?q=tbn:ANd9GcT-utBDuyVz8B5REY79H8rrNBDDC6s-WvZ84IcHrZueNBw47uBfFArenWqI
Image Source: https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcSzKtM9_qK0NOPBy83f2_tdjdBxJTob2Tvw_hKepXHfcSLjBUIyhKdtPFk
Image Source: https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9Gc0k40NaH2Ephi34PKmBrC5F4v6yyzjKXCsggxchhnEW9FSLd_YlhmogwLk
```

4. Now that we have source url of all the images, lets download them. Below Python code uses urlopen() method to read() the image and downloads it on the local file system.

```
for img in images:      raw_img = urllib2.urlopen(img).read()      cntnr
=
len([i for i in os.listdir(".") if image_type in i]) +
1f = open(image_type + "_" + str(cntnr) + ".jpg", 'wb')
f.write(raw_img)      f.close()
```

When the images get downloaded, we can see them on our Editor. Below snapshot shows the top 5 images we downloaded and Project_3.jpg looks like this:





How it works...

So in the above recipe we looked at downloading content from the web. First we defined the parameters for download. Parameters are like configuration that defines the location where the downloadable resource is available and what kind of content is to be downloaded. In our example, we defined that we have to download Avatar movie images and that too from Google.

Then we created the BeautifulSoup object which will make the URL request using the urllib2 module. Actually, urllib2.Request() prepares the request with the configuration like headers and the URL itself and urllib2.urlopen() actually makes the request. We wrap html response of urlopen() method and create a BeautifulSoup object so that we can parse the html response.

Next, we use the soup object to search for top 5 images present in the html response. We search for images based on 'img' tag with find_all() method. As we know, find_all() returns list of image URLs where the picture is available on Google.

We then iterate through all URLs and again use urlopen() method on URLs to read() the images. Read() returns the image in raw format as binary data. We then use this raw image to write to a file on our local file system. We have also added logic in between so that image names increment so that they're uniquely identified in local file system.

Web Scraping

Before we know how to do scraping, let's understand what scraping means. In Web world, scraping is a way to sift through the web pages of a website with the intention of extracting required information in the said format with the help of computer program. For example, if I want to get the title and date of all the articles published on a blog, I could write a program to scrape through the blog and get the required data and store in database or a flat file based on the requirement.

Web scraping is often confused with web crawling. Web crawler is a bot that systematically browses the web with the purpose of web indexing and used by search engines to index web pages so that users can search the web more effectively.

But scraping is not easy. The data, which is interesting to us, is available on the blog or website in a particular format, say XML tags or embedded in HTML tags. So it is important for us to know the format before we begin to extract the data we need. Also the web scraper should know the format in which extracted data needs to be stored to act on it later.

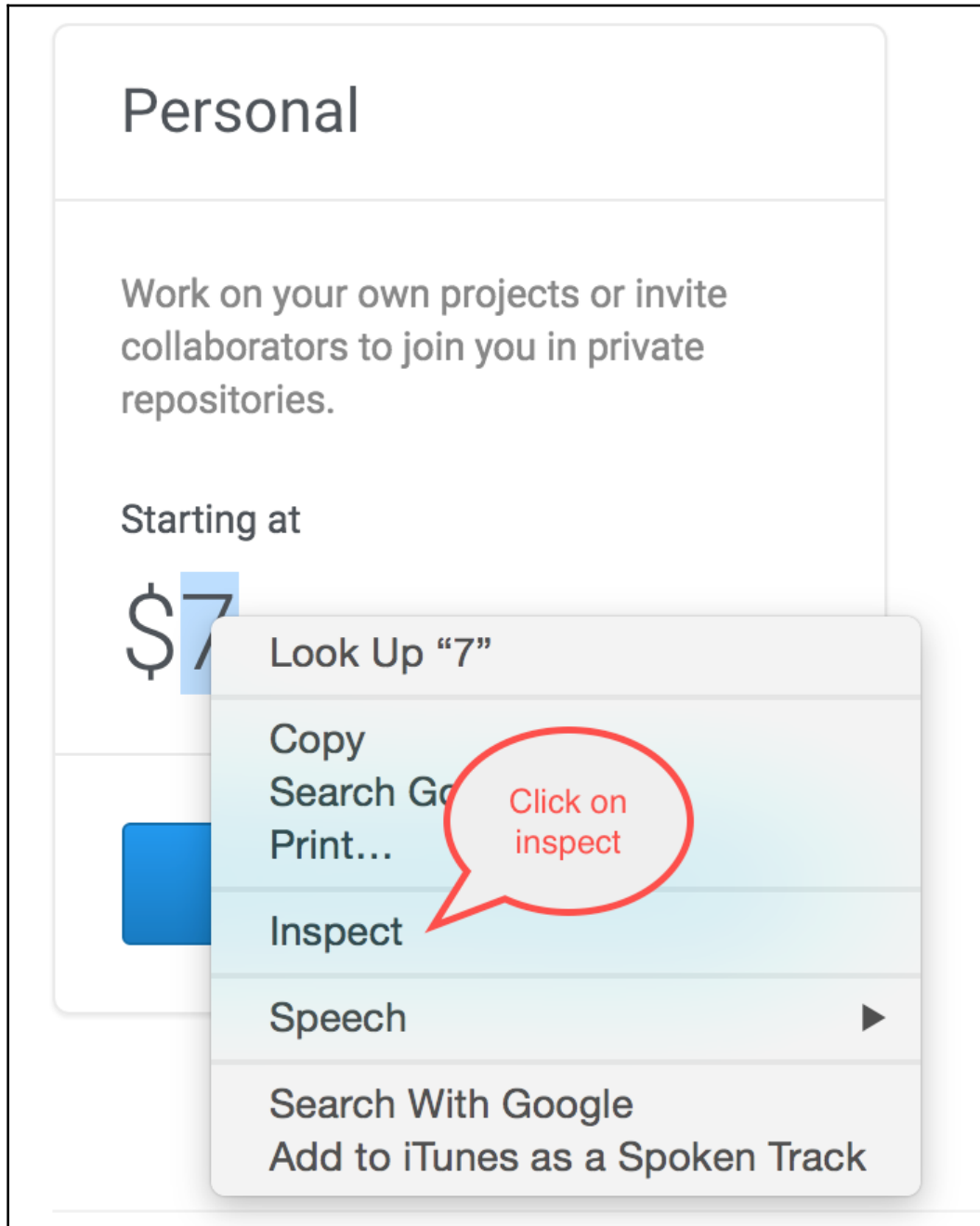
Getting ready

We take two examples to demonstrate web scraping with Python. In the first example, we scrape the pricing data from <https://github.com/> website. In the second example, we work with <http://www.stackoverflow.com> and get query and response data.

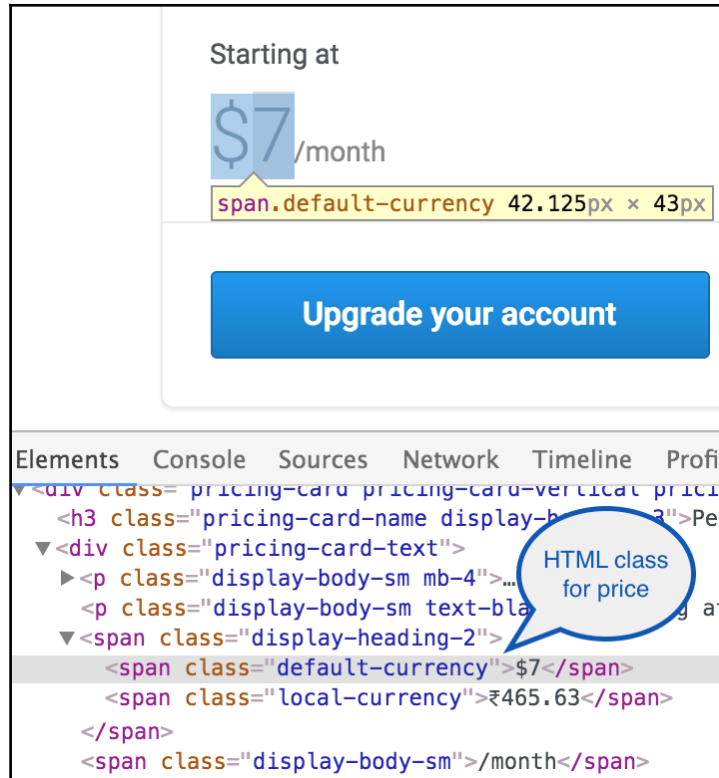
Lets see how to go about doing this.

How to do it...

1. Open Google chrome browser on your computer and open <https://github.com/pricing/> web page
2. On this page you would notice multiple pricing plans namely, Personal, Organization and Enterprise
3. Now on your browser, right click on the pricing of the Personal plan and click on "Inspect" element like in the below screenshot



4. Once you click inspect, Chrome browser's console log opens up which will help you understand the HTML structure of github's pricing page like below



5. If you look at the highlighted HTML span – `$7`, you'd know this webpage uses default-currency class to list down the pricing of plans. We'll now use this property to extract prices of multiple Github plans
6. But before doing that let's install python module lxml that would be needed to extract content from above HTML document. Install lxml and requests module

```
pip install lxml
pip install requests
```

7. Now open your favorite editor and type this code

```
from lxml import html
import requests

page = requests.get('https://github.com/pricing/')
tree = html.fromstring(page.content)
print("Page Object:", tree)
plans = tree.xpath('//h3[@class="pricing-card-name display-
heading-3"]/text()')
pricing = tree.xpath('//span[@class="default-currency"]/text()')
print("Plans:", plans, "\nPricing:", pricing)
```

8. If you look at the above code, we used class 'default-currency' and 'pricing-card-name display-heading-3' to get the pricing and pricing plan

9. Output of the above code:

```
Page Object: <Element html at 0x104a6b188>
Plans: ['Personal', 'Organization', 'Enterprise']
Pricing: ['$7', '$25', '$2,500']
```

How it works...

Like we discussed earlier, we need to find out appropriate way to extract information. So in this example, we first get the HTML Tree for <https://github.com/pricing/> page. Then using lxml module and xpath() method look for class 'default-currency' and 'pricing-card-name display-heading-3' to get the pricing and pricing plans. Thus we scraped github page to the required data.

There's more...

Hey, you learnt what web scrappers are and how they go ahead and extract interesting information from web. You also understood how they are different from web crawlers. But then, there's more. Web scraping involves extraction, which cannot happen until we parse HTML content from the web page to get data interesting to us. In the next section we'll learn about parsing HTML and XML content in detail.

Working with third Party Rest APIs

Now that we've covered a bit about scraping, crawling and parsing, it's time for another interesting work that we can do with Python, that is working with 3rd Party Rest APIs. I'd assume many of us are aware and might have basic understanding of Rest APIs. So lets get started!

Getting ready

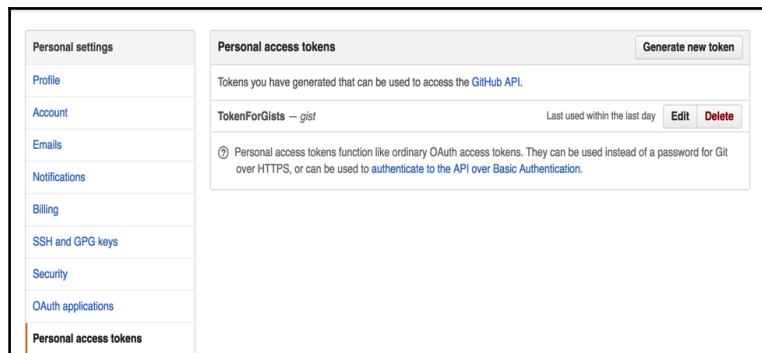
To demonstrate the understanding, we take case of Github gists. Gists in Github are the best way to share your work, a small code snippet that's help your colleague or a small app with multiple files that gives an understanding of a concept. Github allows creation, listing, deleting and updating gists and presents a classical case of working with Github Rest APIs.

So, in this section, we use our very own requests module to make HTTP requests to Github Rest API to create, update, list or delete gists.

The following steps will show you how to work with Github Rest APIs using Python.

How to do it...

1. To work with Github Rest APIs, we need to create a Personal access token. For doing that, login to github.com and browse to `https://github.com/settings/tokens` and click on 'Generate new token'.



- You'd now be taken to 'New personal access token' page. Enter a description at the top of the page and check 'gists' option among the scopes given out. (Note: scope represents the access for your token. For instance, if you just select gists, you can use Github APIs to work on gists resource but not on other resources like repo or users. For this section, gists scope is just what we need.)

Personal access tokens	<input type="checkbox"/> repo_deployment	Access deployment status
Repositories	<input type="checkbox"/> public_repo	Access public repositories
Organizations	<input type="checkbox"/> admin:org	Full control of orgs and teams
Saved replies	<input type="checkbox"/> write:org	Read and write org and team membership
	<input type="checkbox"/> read:org	Read org and team membership
	<input type="checkbox"/> admin:public_key	Full control of user public keys
	<input type="checkbox"/> write:public_key	Write user public keys
	<input type="checkbox"/> read:public_key	Read user public keys
	<input type="checkbox"/> admin:repo_hook	Full control of repository hooks
	<input type="checkbox"/> write:repo_hook	Write repository hooks
	<input type="checkbox"/> read:repo_hook	Read repository hooks
	<input type="checkbox"/> admin:org_hook	Full control of organization hooks
	<input checked="" type="checkbox"/> gist	Create gists

- Once you click on 'Generate token', you'd be presented with a screen containing your personal access token. Keep this token confidential with you.
- With the access token available, let's start working with Github Rest APIs and create a new gist. With create, we add a new resource and for doing this we make HTTP POST Request on Github Rest APIs like in the code below:

```
__author__ = 'Chetan' import requests import json BASE_URL =
'https://api.github.com' Link_URL = 'https://gist.github.com'
username = <username>      ## Fill in your github username
api_token = <api_token>    ## Fill in your token
header = { 'X-Github-Username': '%s' % username,
'Content-Type': 'application/json',
'Authorization': 'token %s' % api_token, }

url = "/gists" data = {  "description": "the description for this
gist",
"public": True,  "files": {      "file1.txt": {
"content": "String file contents"      }  } } r =
requests.post('%s%s' % (BASE_URL, url),          headers=header,
data=json.dumps(data)) print r.json()['url']
```

- If I now go to my gists page on Github, I should see the newly created gist. And

voila, it's available!



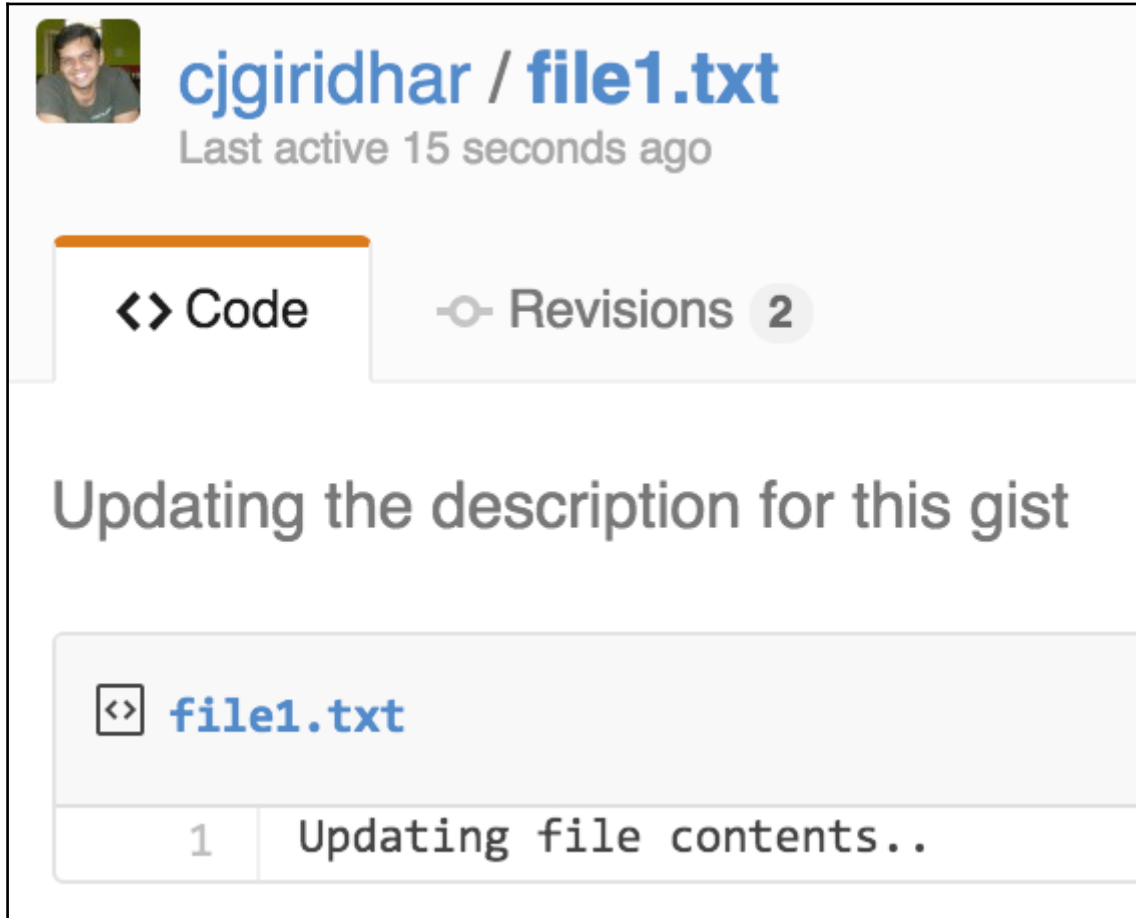
6. Hey we create the gist in step 5, but can we now list this gist? In the example of step 5, we print the URL of the newly created gist. It will be in the format `https://gist.github.com/<username>/<gist_id>`
We now use this `gist_id` to get the details of the gist, which means which make HTTP GET request on the `gist_id`:

```
__author__ = 'Chetan'
import requestsimport json
BASE_URL = 'https://api.github.com' Link_URL =
'https://gist.github.com'
username = 'cjgiridhar' api_token =
'77c7f875ecff38199f39d41838ee4a19a513e445' gist_id =
'de7cd21746ef39677c63ab3bc1b29b14'
header = { 'X-Github-Username': '%s' % username, 'Content-
Type': 'application/json', 'Authorization': 'token %s' % api_token, }
url = "/gists/%s" % gist_id r = requests.get('%s%s' %
(BASE_URL, url), headers=header)print r.json()
```

7. We created a new gist with HTTP POST Request and got the details of the gist with HTTP GET Request in the previous steps. Now let's update this gist with HTTP PATCH request. (Note: Many third party libraries choose to use PUT request to update a resource, but HTTP PATCH can also be used for this operation as is chosen by Github). Below code demonstrates updating the gist:

```
import requests import json BASE_URL =
'https://api.github.com' Link_URL = 'https://gist.github.com'
username = 'cjgiridhar' api_token =
'77c7f875ecff38199f39d41838ee4a19a513e445' gist_id =
'de7cd21746ef39677c63ab3bc1b29b14' header = { 'X-Github-
Username': '%s' % username, 'Content-Type': 'application/json',
'Authorization': 'token %s' % api_token, } data = { "description":
"Updating the description for this gist",
"files": { "file1.txt": { "content": "Updating file
contents.."
} } } url = "/gists/%s" % gist_id r = requests.patch('%s%s'
%(BASE_URL, url),
headers=header, data=json.dumps(data)) print r.json()
```

8. And now if I look at my Github login and browse to this gist, the contents of the gist have been updated. Not to forget to see the Revisions in the screenshot – it got update to revision 2.

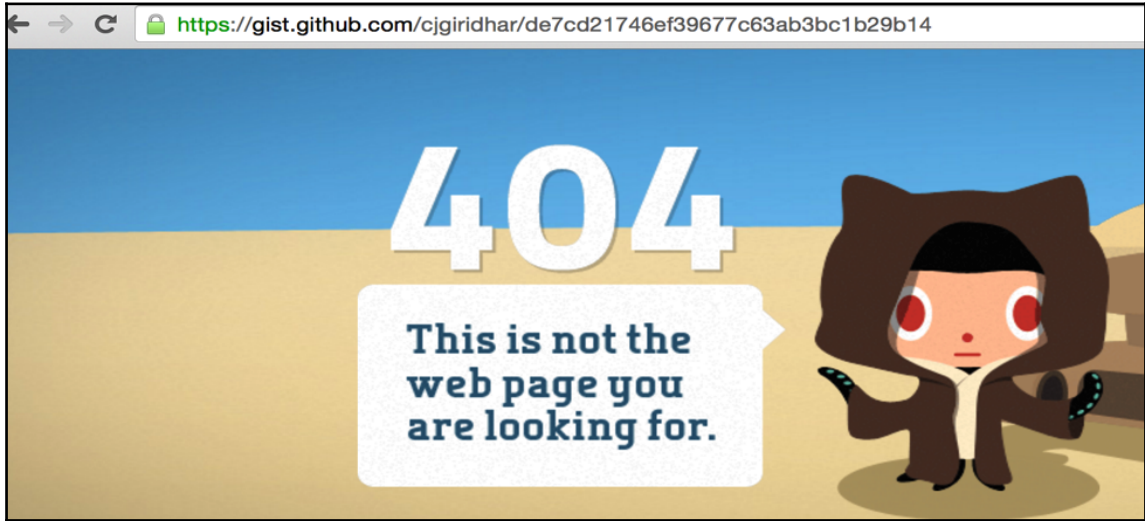


9. And now comes the most destructive API operation – yes deleting the gist. Github provides an API for removing the gist by making use of HTTP DELETE operation on its `/gists/<gist_id>` resource. Below code helps us delete the gist:

```
__author__ = 'Chetan'
import requests import json BASE_URL = 'https://api.github.com'
Link_URL = 'https://gist.github.com' username = 'cjgiridhar'
api_token =
'77c7f875ecff38199f39d41838ee4a19a513e445'
gist_id = 'de7cd21746ef39677c63ab3bc1b29b14'
header = { 'X-Github-Username': '%s' % username,
```

```
'Content-Type': 'application/json',
'Authorization': 'token %s' % api_token,}
url = "/gists/%s" % gist_id r = requests.delete('%s%s' % (BASE_URL,
url), headers=header, )
```

10. And now let us see if the gist is now available on github website? And it says 404, resource not found, so we have successfully deleted the gist!



11. One last thing let us list all the gists in your account. For this we make HTTP GET API call on /users/<username>/gists resource:

```
__author__ = 'Chetan' import requests BASE_URL =
'https://api.github.com' Link_URL = 'https://gist.github.com'
username = <username>
## Fill in your github username api_token = <api_token>
## Fill in your token
header = { 'X-Github-Username': '%s' % username,
'Content-Type': 'application/json',
'Authorization': 'token %s' % api_token, }
url = "/users/%s/gists" % username r = requests.get('%s%s' %
(BASE_URL, url),
headers=header) gists = r.json() for gist in gists:
data = gist['files'].values()[0] print data['filename'],
data['raw_url'], data['language']
```

12. Output of the above code for my account:


```
asyncio_parallel.py https://gist.githubusercontent.com/cjgiridhar/9813678fcaaac181ba18/raw/cf2253b0f3d80e83b56661a958fed0898900ed36b/asyncio\_parallel.py Python
tornadogenex.py https://gist.githubusercontent.com/cjgiridhar/b8fde85a77249d0e538f/raw/6d93fc45b9984641bf46096815e8629a281b5101/tornadogenex.py Python
tornadoasyncex.py https://gist.githubusercontent.com/cjgiridhar/3ac7550dba96fd526c2ff/raw/29f0e0337ea6df2bbbc44d3a2c4211e7da827728/tornadoasyncex.py Python
hello.js https://gist.githubusercontent.com/cjgiridhar/5045567/raw/eadcb76993d78f9ae4ce6c4fa41fb1836b2fe1c4/hello.js JavaScript
async.js https://gist.githubusercontent.com/cjgiridhar/4553831/raw/619897f0ed579b383a014e5b559f2b1396d45cf8/async.js JavaScript
helloworld.js https://gist.githubusercontent.com/cjgiridhar/4528692/raw/3fc7fd5f78bcd6590656021e0a5f1afa17ccff7b/helloworld.js JavaScript
test.py https://gist.githubusercontent.com/cjgiridhar/3870274/raw/c6117b0131ea03a2cb97ab098a3bd89379dd63a/test.py Python
fib.feature https://gist.githubusercontent.com/cjgiridhar/3870272/raw/63260ee02a030c7146178ac7bdbc66b4a5517dd3/fib.feature Cucumber
morelikethis.html https://gist.githubusercontent.com/cjgiridhar/3786714/raw/b01936b374a2892a1d58f4138c9adcla7752fc12/morelikethis.html HTML
tornadowhoosh_morelike.py https://gist.githubusercontent.com/cjgiridhar/3786711/raw/d9e41498024f2c60e7c8a05565c988c04e1e45ac/tornadowhoosh\_morelike.py Python
```

How it works...

Python's requests module helps in making HTTP GET/POST/PUT/PATCH and DELETE API calls on Github's resources. These operations, also known as HTTP Verbs in Rest terminology, are responsible for taking certain actions on the URL resources. Like we saw in the examples, HTTP GET request helps in listing gists, POST creates a new gist, PATCH updates a gist and DELETE completely removed the gist. Thus, in this section, we learnt how to work with 3rd party Rest APIs – an essential part of the World Wide Web today, using Python.

See also

- There are many 3rd party applications that are written as Rest APIs. You may want to try them out the same way we did for Github.

Writing you own Web App in Python

Now if you're a user interface developer or are working on a product that depends on APIs or you're interested in harnessing data from the web, then whatever we learnt till now in this chapter will definitely benefit you. But the web is not complete without writing your own web application that solves a given need. Thankfully, Python has multiple mini frameworks or large MVC frameworks that help us quickly write a web application. In this section, we work on writing our own web application that serves UI requests and returns data in the particular format.

Getting ready

In this section, we will develop a web application that manages task list. We use a simplistic Python we framework, bottle (<http://bottlepy.org/docs/dev/index.html>) to

demonstrate development of a web application. But before that, what is bottle? Bottle is a single file micro web framework. Yes single file! With bottle you could develop single page application and it supports routes, templates and serves static content that are essential parts of web development. Let's get started and work on our task app. During this section we support CRUD operations for our app. CRUD means – Create, Read, Update and Delete operations on our task app.

How to do it...

1. We start with something really basic. Say we have a list of tasks stored as list of Python dictionaries. We write a route in bottle framework that will return this list as JSON response.

```
__author__ = 'Chetan' from bottle import route, run from bottle
import response, request import json tasks = {
    "tasks": [ {"id": 1, "task": "Write a book", "status":
"OPEN"},
    {"id": 2, "task": "Wash my clothes", "status": "COMPLETE"},
    {"id": 3, "task": "Go for a movie", "status": "INPROGRESS"}
] } @route('/tasks') def task(): response.content_type =
'application/json' return json.dumps(tasks)

run(host='localhost', port=8080, debug=True)
```

2. Output of the above code segment is:

```
{
  "tasks": [
    {
      "status": "OPEN",
      "task": "Write a book",
      "id": 1
    },
    {
      "status": "COMPLETE",
      "task": "Wash my clothes",
      "id": 2
    },
    {
      "status": "INPROGRESS",
      "task": "Go for a movie",
      "id": 3
    }
  ]
}
```

3. Now we go ahead and support CRUD for our web app. Let's start by creating new task:

```
@route('/tasks', method='POST') def task_create():
    task = request.forms.get("task") new_task = {
        "id": 999, "task": task, "status": "OPEN", }
    tasks["tasks"].append(new_task)
    response.content_type = 'application/json' return json.dumps(tasks)
```

4. When our app receives a POST request to create a task, it will be assigned an id of "999" (just for the sake of this example) and default status of the task would be

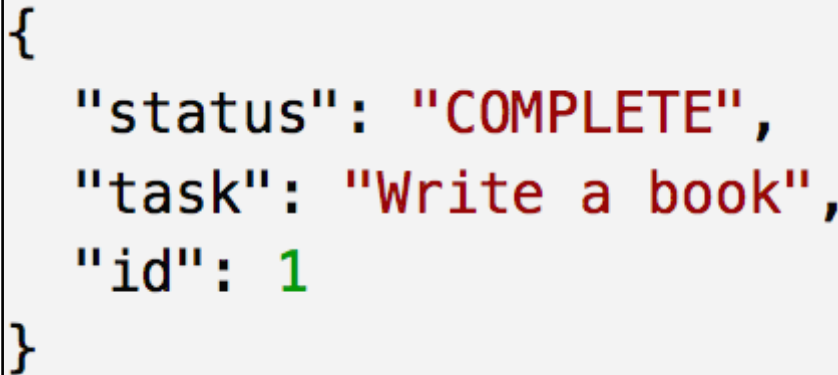
"OPEN". Task itself needs to be sent as request parameter to our app. So say, we send a POST request to `http://localhost:8080/tasks` with parameter `{"task":"This is a new task"}`, a new task is added to our list like below:

```
{
  "tasks": [
    {
      "status": "OPEN",
      "task": "Write a book",
      "id": 1
    },
    {
      "status": "COMPLETE",
      "task": "Wash my clothes",
      "id": 2
    },
    {
      "status": "INPROGRESS",
      "task": "Go for a movie",
      "id": 3
    },
    {
      "status": "OPEN",
      "task": "This is a new task",
      "id": 999
    }
  ]
}
```

5. Ok, now that we have created the task, we may find the need to get the details of our task. We have defined another route for GET operation that will help our app users to list an existing task based on task id. And, your next question would be, can I not update my task? Yes, our app supports updating a task with a HTTP PUT operation. Code for Read and Update operations are below:

```
@route('/tasks/<id>', method='GET') def task_list(id):
    for data in tasks["tasks"]: if data["id"] == int(id):
        response.content_type = 'application/json'
        return json.dumps(data)
@route('/tasks/<id>', method='PUT') def task_update(id,
status="COMPLETE"):
    for data in tasks["tasks"]: if data["id"] == int(id):
        data["status"] = status response.content_type = 'application/json'
return json.dumps(data)
```

6. If we update task id 1 by sending HTTP PUT request on `http://localhost:8080/1`, based on our code, the status of the task should be moved to “COMPLETE”. What better of way of finding out then to make a HTTP GET request on `http://localhost:8080/1`



```
{
  "status": "COMPLETE",
  "task": "Write a book",
  "id": 1
}
```

7. Indeed the status of task is now changed to “COMPLETE”!
8. And the final operation that we intend to support on our app is deleting the task. Below server code will enable deleting the task:

```
@route('/tasks/<id>', method='DELETE') def task_delete(id):
    for data in tasks["tasks"]: if data["id"] == int(id):
        tasks["tasks"].remove(data) response.content_type = 'application/json'
return json.dumps(tasks)
```

9. Perform a HTTP DELETE request on `http://localhost:8080/1` to delete the task with “id”:“1”

How it works...

In this recipe, we develop our own web application with help of bottle framework of Python. It was a very simplistic app and supported CRUD operations. This was possible because of server side routes and methods like we defined in the recipes.

We run the Python code by using the command:

```
python app.py
```

This command will run the server on port 8080 and our web application is ready to accept requests on this port.

When we made a POST request to our web application on `http://localhost:8080/tasks`, the bottle framework routes it to `task_create()` method. This happens because we have defined our own route `/tasks` to handle POST requests on our server. Once the request reaches `task_create()` method, a new task gets created with `id:999`, `status:COMPLETE` and with the task name "This is a new task" as we saw in the above section.

On similar lines, we defined a new route `/tasks/<id>` that supports HTTP DELETE under the method `task_delete(id)`. Method `task_delete()` accepts an input parameter "id", Id of the task that needs to be deleted. When our web app receives HTTP DELETE request on our configured route with task id, the id is located in the JSON object and deleted from the dictionary.

There's more...

We developed a minimalistic web app in this section, but there is much more like serving template, static files, and databases and more, so you should make an attempt to try these out on your own.

Asynchronous HTTP Server with Python

In the previous recipe we discussed how to develop our own web application with the help for bottle framework. We defined our own routes and methods to handle various HTTP requests and use cases. If you realize, all the operations in our web application were synchronous. A client connection gets established for every request made by the client and a callable method gets invoked at the server side. Server performs the business operation and writes the response body to the client socket. Once the response is exhausted, the client connection gets closed. All these operations happen in sequence one after the other, hence

synchronous.

But the web today, as we see it, cannot rely on synchronous mechanism only. Consider case of any website that queries data from the web and retrieves the information for you. Now if we develop this web application in a synchronous manner, for every request made by the client, the server would make an I/O call to either the database or over the network to retrieve information and then present it back to the client. If these I/O requests take longer time to respond, the server gets blocked waiting for the response. Typically web servers maintain a thread pool that handles multiple requests from the client. If a server waits long enough to serve requests, thread pool may get exhausted soon and the server shall get stalled.

Solution? In comes the asynchronous ways of doing things!

Getting ready

For this recipe, we will use Tornado, an asynchronous framework developed in Python. It has support for both Python 2 and Python 3 and was originally developed at FriendFeed. Tornado uses a non-blocking network I/O and solves the problem of scaling to tens of thousands of live connections (C10K problem). I like this framework and enjoy developing code with it. I hope you'd too! Before we get into “How to do it” section, let us first install tornado by executing below command:

```
pip install -U tornado
```

How to do it...

1. We're now ready to develop our own HTTP Server that works on asynchronous philosophy. Below code represents an asynchronous server developed in tornado web framework

```
__author__ = 'Chetan' import tornado.ioloop
import tornado.web import httplib2 class
AsyncHandler(tornado.web.RequestHandler):
@tornado.web.asynchronous def get(self):
http = httplib2.Http() self.response, self.content =
http.request("http://ip.jsontest.com/", "GET")
self._async_callback(self.response, self.content)
def _async_callback(self, response, content):
print "Content:", content print "Response:\nStatusCode:
%s Location: %s" %(response['status'],
```

```
response['content-location']) self.finish()
tornado.ioloop.IOLoop.instance().stop()
application = tornado.web.Application([
    (r"/", AsyncHandler)], debug=True)
if __name__ == "__main__":
    application.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

2. Run the server as:

```
python tornado_async.py
```

3. Server is now running on port 8888 and ready to receive requests
4. Now launch any browser of your choice and browse to `http://localhost:8888/`. On the server you'd see the following output

Content: {"ip": "117.213.178.109"}

Response:

StatusCode: 200 Location: <http://ip.jsontest.com/>

How it works...

Our asynchronous web server is now up and running and accepting requests on port 8888. But what is asynchronous about this? Actually tornado works on the philosophy of a single threaded event loop. This event loop keeps polling for events and passes it on to the corresponding event handlers.

In the above example, when the app is run, it starts by running the ioloop. IOLoop is single threaded event loop and is responsible for receiving requests from the clients. We have defined `get()` method that is decorated with `@tornado.web.asynchronous` which makes it asynchronous. When a user makes a HTTP GET request on `http://localhost:8888/get()` method is triggered that internally makes an I/O call to `http://ip.jsontest.com`.

Now a typical synchronous web server would wait for the response of this I/O call and block the request thread. But tornado being asynchronous framework, it triggers a task, adds it to a queue, makes the I/O call and returns the thread of execution back to the event loop. Event loop now keeps monitoring the task queue and polls for response of the I/O call. When the event is available, it executes the event handler, `async_callback()`, to print the

content and response and stops the event loop.

There's more...

Event driven web servers like tornado make use of kernel level libraries to monitor for events. These libraries are kqueue, epoll etc. If you're really interested you should do more reading on this and I'll be more than happy to help you with the resources.

Web automation with selenium bindings

In all the above recipes we had a dedicated URL to make HTTP requests; be it calling a Rest API or downloading content from the web. But then there are services that don't have a defined API resource or need login to the web to perform operations. How about controlling the browser itself to do some tasks? Interesting isn't it?

Getting ready...

For this recipe, we use Python's selenium module. Selenium (<http://www.seleniumhq.org/>) is a portable software framework for web applications. Selenium automates browsers. You could automate mundane tasks with selenium. Selenium spawns a browser and helps you perform tasks as though a human is doing them. Selenium has support of some of the largest used browsers like Firefox, Chrome, Safari, and Internet Explorer amongst others. Let us take an example of login to facebook with Python selenium binding.

How to do it...

1. We start by installing selenium bindings for Python. Installing selenium can be done with the below command:

```
pip install selenium
```

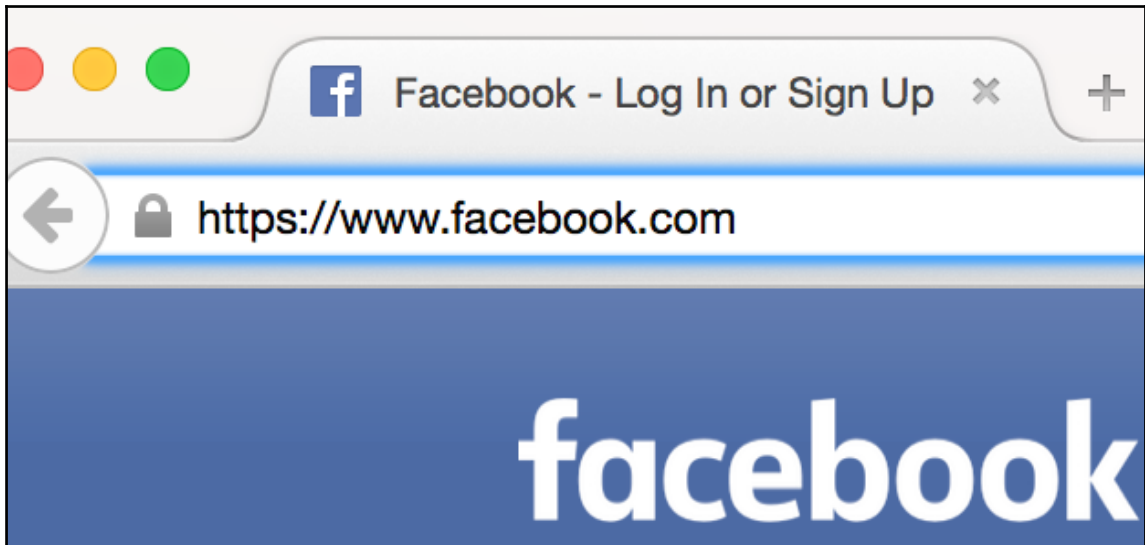
2. Let us start by first creating a browser object. We use Firefox browser for spawning the browser instance, `__author__ = 'Chetan'` from selenium **import** `webdriver` `browser = webdriver.Firefox()` **print "WebDriver Object"**, browser

```
WebDriver Object <selenium.webdriver.firefox.webdriver.WebDriver (session="aef75d40-879a-ae4a-a825-bd8f906e0e4e")>
```

3. Next, we ask the browser to browse to the facebook home page. Below code helps us achieve this:

```
browser.maximize_window() browser.get('https://facebook.com')
```

You will see something like the following:



4. As a next step, we locate the Email and Password elements and enter appropriate data

```
email = browser.find_element_by_name('email')
password = browser.find_element_by_name('pass')
print "Html elements:"
print "Email:", email, "\nPassword:", password
```

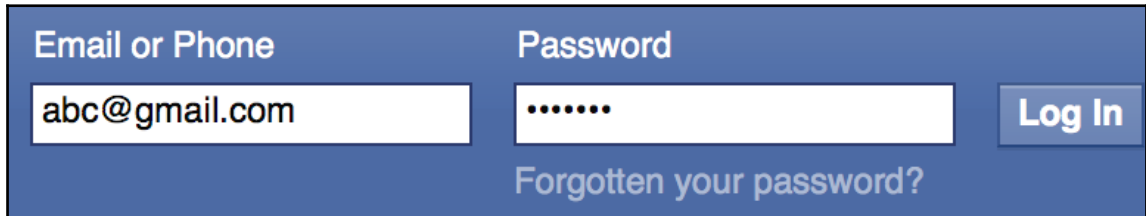
Output of the above code is:

```
Html elements:
Email: <selenium.webdriver.remote.webelement.WebElement (session="7b4f564a-4388-c94e-9d94-23df1a59899a", element="{8b95073f-6f28-db4d-be1a-368e24781d83}")>
Password: <selenium.webdriver.remote.webelement.WebElement (session="7b4f564a-4388-c94e-9d94-23df1a59899a", element="{b19df0ce-74a7-5e4d-bf95-227b22394232}")>
```

5. Once we have selected the Email and Password text inputs, let us now fill them

with the correct Email and Password. Below code will enable entering Email and Password:

```
email.send_keys('abc@gmail.com') #Enter correct email
addresspassword.send_keys('pass123') #Enter correct password
```

A screenshot of a login interface. It features a blue header bar. On the left, the text "Email or Phone" is above a white input field containing "abc@gmail.com". On the right, the text "Password" is above a white input field containing seven dots. To the right of the password field is a blue button with the text "Log In". Below the password field, the text "Forgotten your password?" is displayed in a lighter blue color.

6. Now that we have entered the Email and Password, the last thing to do is submit the form and click on Log In button. We do this by finding the element by Id and clicking on the element.

```
browser.find_element_by_id('loginbutton').click()
```

If you have entered the correct email Id and password, you'd have logged in to Facebook!

How it works...

For the above recipe, we used selenium webdriver Python APIs. **Webdriver** is the latest inclusion in selenium APIs and drives browser natively like a user. It can drive locally or on a remote machine using selenium server. In this example, we ran it on the local machine. Basically, selenium server runs on local machine on a default port 4444 and selenium webdriver APIs interacts with the selenium server to take actions on the browser.

In this recipe, we first created a webdriver instance using the Firefox browser. We then use the webdriver API to browse to the Facebook homepage. We then parse the Html page and locate the Email and Password input elements. How did we find the elements? Yes, like we did in the web-scraping example. Like in Chrome we have the developer console, in Firefox we can install the firebug plugin. Using this plugin we can get the html elements for Email and Password. See the screenshot below:

```
▼ <td>  
  <input id="email" class="inputtext" type="email" tabindex="1" value="" name="email"></input>  
</td>  
▼ <td>  
  <input id="pass" class="inputtext" type="password" tabindex="2" name="pass"></input>  
</td>
```

Once we figured the html element names, we programmatically create an html element object using webdriver's method `find_element_by_name()`. Webdriver API has a method `send_keys()` that can work on element objects and enter the required text (in this case email and password). Last operation is to submit the form and that we do by finding the 'Log In' object and clicking on it.

There's more...

We looked at a very basic example with selenium webdriver Python bindings. Now its up to your imagination on what all you can achieve with selenium and automate the mundane tasks.