ICS 32 Winter 2014
Code Example: Downloading data from the web

Background

Thus far in this course and the preceding one, you've written Python programs that read data from text files and that exchange data over a network via sockets, which are two big steps that push outward the boundaries of what we can accomplish in Python. However, there is an elephant in the room, so to speak; if we think about where most of the interesting data on the Internet resides, it's on the web. *Web sites* display content and allow human users to interact with web-based data, while *web services* provide a similar ability to other programs. In both cases, the same fundamentals we've already seen apply: a connection is initiated by a *client* connecting to a *server* (usually on the server's port 80), and a protocol is followed that governs what the conversation looks like. So if we want to interact with web data — the simplest example of which is to download the content of a web page — we need to know enough about that protocol to be able to implement the conversation.

URLs

When we use a browser to visit a web page, all we need to do is tell the browser where we want to go and it handles the rest. The notion of "where you want to go" is encapsulated by a *URL (Uniform Resource Locator)*, which specifies a few things:

- What protocol should be used to download the web page?
- From what host (i.e., an IP address or the name of a machine, like **www.ics.uci.edu**) should the web page be downloaded?
- What page on that machine should be downloaded?

One of the earlier code examples included a link to a short Python module called **oops.py**. The complete URL for that link is: http://www.ics.uci.edu/~thornton/ics32/CodeExamples/ExceptionalControlFlow/oops.py. Here's what that URL means:

- The first few characters (preceding the colon) indicate what protocol should be used for the network conversation. For most web pages, that protocol will be listed as **http**, which means that we'd like to use the protocol called HTTP (HyperText Transfer Protocol). Another common alternative is **https**, which uses HTTP over a secure (encrypted) connection, to make eavesdropping substantially difficult.
- After the colon and the two slashes is the host. In this case, that host is listed as **www.ics.uci.edu**, which is the machine on which the ICS web site is hosted. It is possible also to specify a port, by following the host with a colon and a port number (e.g., **www.ics.uci.edu:8080**). The default port number for HTTP traffic is 80, and since most web sites use this port, port numbers are not usually specified in a browser except in the rare instances that they're something other than the default. Web services (consumed by programs, as opposed to human users) often use alternative ports, though.
- The rest of the URL specifies what web page we'd like to download from the given host using the given protocol. In this case, that page is **/~thornton/ics32/CodeExamples/ExceptionalControlFlow/oops.py**, which is a page in the web directory that's under my control.

Given that information, a browser will know just what it needs to do:

- Initiate a socket connection to port 80 on www.ics.uci.edu.
- Use HTTP to request the page **/~thornton/ics32/CodeExamples/ExceptionalControlFlow/oops.py**.
- Parse the HTTP response and draw the page in the browser window.

But browsers aren't the only programs that can have conversations using HTTP; our Python programs can do it, too. But we need to know a little bit about HTTP in order to do so effectively.

Some background on HTTP

HTTP (HyperText Transfer Protocol) is the protocol with which most web traffic on the Internet is transacted. Its latest version is HTTP/1.1, though a newer version is under development. We'll stick with HTTP/1.1, since it's the predominant version of HTTP used across the web today.

HTTP is a *request-response protocol*, which means that its conversations go something like this:

- Client initiates connection to server
- Server accepts connection
- Client makes a request
- Server sends a response

After that single request and response, both sides close the connection (though there are performance optimizations available that let a client specify that the connection should be kept open if, for example, the client knows that it needs not just a web page's text but also several images from the same server). For our purposes, we'll stick with a single request and response per connection.

Python programs can make these requests and parse these responses, but that requires us to know a little bit about the format of each. HTTP requests come in a few flavors, but the most common of them is called a *GET*, which means that the client would like to "get" a resource (a web page, an image, etc.) from the server. (We may see other alternatives later if we find a need for them.) A GET request in HTTP/1.1 looks like this.

```
GET /~thornton/ics32/CodeExamples/ExceptionalControlFlow/oops.py HTTP/1.1
Host: www.ics.uci.edu
```

The first line of a GET request begins with the word **GET**, is followed by the web resource you want to download (the part of the URL that follows the protocol and host), and finally is followed by **HTTP/1.1**, as a way to indicate what protocol we expect to be using for the conversation.

The second line (and subsequent lines) are what are called *headers*, which allow us to specify a variety of supplementary information that the server can use to figure out how to send us a response. In our case, we've included just one, a header called **Host:**, which specifies the name or IP address of the host we think we're connecting to; this is useful in the case that the same machine has multiple names (e.g., more than one web site being served up by the same machine), and is generally included in most HTTP requests. Additional headers include specifying what browser (and what version) is being used — so, for example, a server can send back different output for a small-sized screen like an iPhone than to a larger-sized screen like a laptop or desktop — or a variety

of performance optimizations that are available.

Using PuTTY (Windows) or Telnet (Mac), connect yourself to **www.ics.uci.edu** on port 80 and try sending the request above (plus a blank line following it, so the server will know there are no more headers) and you should get back a response very much like this one (some details left out here for brevity).

```
HTTP/1.1 200 OK
Date: Mon, 03 Feb 2014 07:56:07 GMT
Server: Apache/2.2.15 (CentOS)
...
...
Content-Length: 437
Content-Type: text/plain; charset=UTF-8

# oops.py
#
# ICS 32 Winter 2014
# Code Example
...
...
if __name__ == '__main__':
    f()
```

The first line of the response indicates that the server agrees to have an HTTP/1.1 conversation (that's the **HTTP/1.1** part), followed by what's called a *status code* (in this case, **200**) and a *reason phrase* (in this case, **OK**). There are forty or so status codes that are defined as part of the HTTP/1.1 standard; the two most common ones are:

- 200 (OK), which means that everything went as planned, the server's way of saying "Okay, cool, here's the web page you asked for!"
- 404 (Not Found), which means that the server doesn't have the page that you asked to download. (If you've ever seen "404" show up in a browser during your travels around the web, this is why; it's an HTTP status code, "geekspeak" for a web page that doesn't exist.)

The first line of the response is followed by *headers*, just as the first line of the request is. The server determines what headers to send, and the details there are too numerous to list, but I've included a few of the more interesting ones in the example above:

- **Date** is the date/time at which the response was generated.
- **Server** specifies what type of server is being run and what version. As of this writing, the ICS web server is running version 2.2.15 of a server called Apache (which is quite common on the web).
- **Content-Length** specifies the length, in bytes, of the content that will be sent back. This allows the client to know when the content has ended.
- **Content-Type** specifies what kind of content is being sent back (e.g., a web page, a text file, audio, video, etc.). Browsers respond to the content type by deciding what to do with the content: web pages are shown in the browser, video is often displayed in a video plugin or an external media player, etc. If a browser isn't sure what to do with content, it generally just asks you if you want to save the file somewhere on your hard drive.

After the last header is a blank line, followed by the desired content — in this case, the contents of the file **oops.py** that is linked from one of my code examples.

For those of you who are interested in the full details of HTTP, the specification for it can be found here. Don't feel obligated to read through it unless you're interested; it's not a part of the course. But if you want to get an idea of the complexity level of HTTP, and why we should be so quick to want to find a library that implements all of that complexity for us, take a quick look through it (and note that one of the main authors of the specification, Roy Fielding, was completing his Ph.D. here at UCI at the time it was written).

The urllib.request module in the Python standard library

Unlike the protocols we've implemented in this course, which had a fairly straightforward sequence of what needed to be sent from client to server and vice versa, HTTP is anything but simple. It is used for everything from fetching a simple web page, implementing the "guts" of the conversations happening behind the scenes while you use full-featured web sites like Gmail, and even for allowing non-browsers to interact with web data (e.g., programs that can send tweets via Twitter). While we could certainly implement an HTTP conversation using the techniques we've seen so far — opening a socket connection to a server's port 80, constructing and sending a GET request, parsing the response — this is a very complex task. In order to do the job right, we would need to implement the entire specification, which weighs in (when printed) at 114 pages.

Happily, HTTP support is something so fundamental to the needs of so many programmers, many programming language libraries include HTTP support; Python is no exception. Python's library includes a number of modules that implement different parts of the HTTP specification, with the main trick being to understand which module you need in a given circumstance.

Our job in lecture was simple: we just wanted to download the contents of a single web page in a Python program, given its URL. Your task in Project #3 is similar: given the URL that allows you to ask Yahoo for a set of stock quotes, you just want to download the stock quotes. More complex interactions require more complex tools, but the interactions we've needed thus far are the simplest ones, so the simplest part of the library will suffice. That module is called **urllib.request**.

The **urllib.request** module has one function that we're interested in: **urllib.request.urlopen()**. Looking through its documentation reveals many more details than we need to know if we only want to download a web page using a GET request; downloading one page can be done in the interpreter by doing just this:

```
>>> import urllib.request
>>> response = urllib.request.urlopen('http://www.ics.uci.edu/~thornton/ics32/CodeExamples/ExceptionalControlFlow/oops.py')
```

The **urlopen()** function returns an object called an **HTTPResponse**, which provides a few useful attributes and methods, the most important of which is the **read()** method, which retrieves all of the content from the response (i.e., the contents of the web page you asked for) and returns a **bytes** object containing those contents. Continuing the previous example in the interpreter:

```
Reading the content of the response (i.e., the data following the headers)
>>> data = response.read()

The data comes back as a bytes object, as opposed to a string
```

```
>>> data
b'"# oops.py\r\n#\r\n# ICS 32 Winter 2014\r\n# Code Example\r\n#\r\n.......'

Once we're done reading from the response, we should close it
>>> response.close()

We can decode the bytes into a string the same way we've done before
assuming that we know that the data is text and can be decoded this way.
The only way we can be absolutely sure is to use the information in the
HTTP response headers, particularly the Content-Type header, but
that's beyond the scope of this example.
>>> string_data = data.decode(encoding = 'utf-8')

This will give us a single string containing all of the data
>>> string_data
"# oops.py\r\n#\r\n# ICS 32 Winter 2014\r\n# Code Example\r\n#\r\n.......'

We can then split the string into lines using the str.splitlines() method.
str.split() will work, too, but str.splitlines() automatically handles the
differences in line endings between operating systems (e.g., Windows uses
'\r\n', while Mac, Unix, and Linux use '\n').
>>> lines = string_data.splitlines()

And that will give us a list of strings, in which each string is one line of text
>>> lines
['# oops.py', '#', '# ICS 32 Winter 2014', '# Code Example', '#', .......]
```

The code

Below is a link to a short program that asks the user to type a URL, then downloads the contents of that URL and prints them to the console, using the techniques demonstrated above.

- download_url.py