California State University Fullerton

CPSC-223P

Python Programming

Stephen T. May

Python Standard Library

Internet Protocols and Support

https://docs.python.org/release/3.9.6/tutorial/index.html

Slide Notes

- Command typed at the Linux command prompt ($)

```
$ python3.9
```

- Command typed at the Python interpreter command prompt (>>>)

```
>>> Ctrl-D
```

- Python source code

```
print(“Hello world!”)
```

- Mixed example

```
>>> the world is flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

urllib — urllib.request

- `urllib` is a package that collects several modules for working with URLs:
- `urllib.request` for opening and reading URLs
- `urllib.parse` for parsing URLs
- `urllib.error` containing the exceptions raised by `urllib.request`

- `urllib.request` is a Python module for fetching URLs.
- It offers a very simple interface, in the form of the `urlopen` function.
- This is capable of fetching URLs using a variety of different protocols.
- `urllib.request` supports fetching URLs for many “URL schemes” (identified by the string before the ":" in URL - for example "ftp" is the URL scheme of "ftp://python.org/") using their associated network protocols (e.g. FTP, HTTP).
- For straightforward situations `urlopen` is very easy to use.
- But as soon as you encounter errors or non-trivial cases when opening HTTP URLs, you will need some understanding of the HyperText Transfer Protocol.

`urllib.request` - Fetching URLs

- HTTP is based on requests and responses - the client makes requests and servers send responses.
- `urllib.request` mirrors this with a Request object which represents the HTTP request you are making.
- In its simplest form you create a Request object that specifies the URL you want to fetch.
- Calling `urlopen` with this Request object returns a response object for the URL requested.
- This response is a file-like object, which means you can for example call `.read()` on the response.
- Note that `urllib.request` makes use of the same Request interface to handle all URL schemes.

```
import urllib.request
with urllib.request.urlopen('http://python.org/') as response:
   html = response.read()
```

```
import urllib.request

req = urllib.request.Request('http://www.voidspace.org.uk')
with urllib.request.urlopen(req) as response:
   the_page = response.read()
```

```
req = urllib.request.Request('ftp://example.com/')
```

urllib.parse - Data

- In the case of HTTP, there are two extra things that Request objects allow you to do.
- First, you can pass data to be sent to the server.
- Second, you can pass extra information ("metadata") about the data or about the request itself, to the server - this information is sent as HTTP "headers".
- Sometimes you want to send data to a URL (often the URL will refer to a CGI (Common Gateway Interface) script or other web application).
- With HTTP, this is often done using what's known as a POST request.
- This is often what your browser does when you submit a HTML form that you filled in on the web.
- Not all POSTs have to come from forms: you can use a POST to transmit arbitrary data to your own application.
- In the common case of HTML forms, the data needs to be encoded in a standard way, and then passed to the Request object as the data argument.
- The encoding is done using a function from the `urllib.parse` library.

```
import urllib.parse
import urllib.request

url = 'http://www.someserver.com/cgi-bin/register.cgi'
values = {'name' : 'Michael Foord',
          'location' : 'Northampton',
          'language' : 'Python' }

data = urllib.parse.urlencode(values)
data = data.encode('ascii') # data should be bytes
req = urllib.request.Request(url, data)
with urllib.request.urlopen(req) as response:
   the_page = response.read()
```

`urllib.parse` – Data (cont.)

- If you do not pass the data argument, `urllib` uses a GET request.
- One way in which GET and POST requests differ is that POST requests often have “side-effects”.
- They change the state of the system in some way (for example by placing an order with the website for a hundredweight of tinned spam to be delivered to your door).
- Though the HTTP standard makes it clear that POSTs are intended to always cause side-effects, and GET requests never to cause side-effects, nothing prevents a GET request from having side-effects, nor a POST requests from having no side-effects.
- Data can also be passed in an HTTP GET request by encoding it in the URL itself.

```
>>> import urllib.request
>>> import urllib.parse
>>> data = {}
>>> data['name'] = 'Somebody Here'
>>> data['location'] = 'Northampton'
>>> data['language'] = 'Python'
>>> url_values = urllib.parse.urlencode(data)
>>> print(url_values)  # The order may differ from below.
name=Somebody+Here&language=Python&location=Northampton
>>> url = 'http://www.example.com/example.cgi'
>>> full_url = url + '?' + url_values
>>> data = urllib.request.urlopen(full_url)
```

`urllib.parse` – Headers

- Some websites dislike being browsed by programs, or send different versions to different browsers.
- By default `urllib` identifies itself as `Python-urllib/x.y` (where x and y are the major and minor version numbers of the Python release, e.g. Python-urllib/2.5), which may confuse the site, or just plain not work.
- The way a browser identifies itself is through the User-Agent header.
- When you create a Request object you can pass a dictionary of headers in.
- The following example makes the same request as above, but identifies itself as a version of Internet Explorer.

```
import urllib.parse
import urllib.request

url = 'http://www.someserver.com/cgi-bin/register.cgi'
user_agent = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64)'
values = {'name': 'Michael Foord',
          'location': 'Northampton',
          'language': 'Python' }
headers = {'User-Agent': user_agent}

data = urllib.parse.urlencode(values)
data = data.encode('ascii')
req = urllib.request.Request(url, data, headers)
with urllib.request.urlopen(req) as response:
   the_page = response.read()
```

`urllib.error` – Handling Exceptions

- `urlopen` raises `URLError` when it cannot handle a response (though as usual with Python APIs, built-in exceptions such as `ValueError`, `TypeError` etc. may also be raised).
- `HTTPError` is the subclass of `URLError` raised in the specific case of HTTP URLs.
- The exception classes are exported from the `urllib.error` module.
- Often, `URLError` is raised because there is no network connection (no route to the specified server), or the specified server doesn't exist.
- In this case, the exception raised will have a '`reason`' attribute, which is a tuple containing an error code and a text error message.
- Every HTTP response from the server contains a numeric "status code".
- Sometimes the status code indicates that the server is unable to fulfil the request.

```
>>> req = urllib.request.Request('http://www.pretend_server.org')
>>> try: urllib.request.urlopen(req)
... except urllib.error.URLError as e:
...     print(e.reason)
...
(4, 'getaddrinfo failed')
```

```
>>> req = urllib.request.Request('http://www.python.org/fish.html')
>>> try:
...     urllib.request.urlopen(req)
... except urllib.error.HTTPError as e:
...     print(e.code)
...     print(e.read())
...
404
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">\n\n\n<html
  ...
  <title>Page Not Found</title>\n
  ...
```

`urllib.error` – Handling Exceptions (cont.)

- The default handlers will handle some of these responses for you (for example, if the response is a "redirection" that requests the client fetch the document from a different URL, `urllib` will handle that for you).
- For those it can't handle, `urlopen` will raise an `HTTPError`.
- Typical errors include '404' (page not found), '403' (request forbidden), and '401' (authentication required).
- So if you want to be prepared for `HTTPError` or `URLError` there are two basic approaches.
- Note The except `HTTPError` must come first, otherwise except `URLError` will also catch an `HTTPError`.

```
from urllib.request import Request, urlopen
from urllib.error import URLError, HTTPError
req = Request(someurl)
try:
    response = urlopen(req)
except HTTPError as e:
    print('The server couldn\'t fulfill the request.')
    print('Error code: ', e.code)
except URLError as e:
    print('We failed to reach a server.')
    print('Reason: ', e.reason)
else:
    # everything is fine
```

```
from urllib.request import Request, urlopen
from urllib.error import URLError
req = Request(someurl)
try:
    response = urlopen(req)
except URLError as e:
    if hasattr(e, 'reason'):
        print('We failed to reach a server.')
        print('Reason: ', e.reason)
    elif hasattr(e, 'code'):
        print('The server couldn\'t fulfill the request.')
        print('Error code: ', e.code)
else:
    # everything is fine
```

HTTP Client Server Example

- server
  - cgi-bin
    - contacts.py
    - contact.dat
    - SERVER_GET_get_contact.py
    - SERVER_GET_set_contact.py

- client
  - CLIENT_GET_get_contact.py
  - CLIENT_GET_set_contact.py

1. Open a terminal for the server and a separate terminal for the client
2. Start the server from the server directory: `python3 -m http.server --cgi 8080`
3. Set a contact from the client directory: `python3 -m CLIENT_GET_set_contact`
4. Get a contact from the client directory: `python3 -m CLIENT_GET_get_contact`