# A note on `urllib` and Python 3
## and a short introduction to `requests`

Towards the end of our penultimate lecture, a question was raised by someone in the class about an error they had been encountered when trying to run one of the XML examples (see snippet 1 below) with Python 3.

```
from urllib2 import Request, urlopen, URLError
import xml.etree.ElementTree as ET

request = Request("http://users.aber.ac.uk/mfc1/csm0120/api.php?"
                  "uuid0=0BA846D3-CBA2-4D81-AD4C-00004868EA28")
try:
    tree = ET.parse(urlopen(request))
    # Do something...
except URLError, e:
    print(e)
```

Code Snippet 1: Example API request as introduced at the end of Lecture 9

The example is fairly simple; import the necessary functions from the `urllib2` and `xml` packages and create a `Request` to the desired API endpoint. The code then attempts to open the `Request`'s URL with `urlopen` and catches any `URLError`s that may occur in the process. Snippet 2 demonstrates what happens when running this code in an interpreter using Python 3.

```
>>> from urllib2 import Request, urlopen, URLError
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ImportError: No module named "urllib2"
```

Code Snippet 2: Error encountered running the example with Python 3

The `urllib2` documentation[1] informs us that in Python 3, the module has been split into a pair of smaller modules; `urllib.request` and `urllib.error`, and so the `urllib2` module doesn't exist in the standard library anymore. Snippet 3 demonstrates how to import these packages and "fix" the original example.

```
1  from urllib.request import Request, urlopen
2  from urllib.error import URLError
3  import xml.etree.ElementTree as ET
4
5  request = Request("http://users.aber.ac.uk/mfc1/csm0120/api.php?"
6                    "uuid0=0BA846D3-CBA2-4D81-AD4C-00004868EA28")
7  try:
8      tree = ET.parse(urlopen(request))
9      # Do something...
10 except URLError as e:
11     print(e)
```

Code Snippet 3: Updating the API example for Python 3

[1] https://docs.python.org/2/library/urllib2.html

A subtle change you may not have noticed appears in the error handling on line 10 of snippet 3. A 'Python Enhancement Proposal' submitted as part of the Python 3 specification[2] cleaned up a potential ambiguity when using the `except` keyword and now requires use of the `as` keyword instead of the comma we have been using. Thankfully this syntax is valid in Python 2, too.

But this is a bit frustrating, what if we want a script to work for users without having to worry what version of Python they are using? Whilst it is possible to obtain the Python version number from the `sys` package as shown in snippet 4, it may not always be correct (depending on how the different versions of Python are actually installed).

```
sam@fi ~> python
>>> import sys
>>> sys.version
"2.7.5 (default, Jun 26 2014, 11:55:39) \n[GCC 4.8.2 20131212 (Red Hat 4.8.2-7)]"
>>> sys.version_info
sys.version_info(major=2, minor=7, micro=5, releaselevel="final", serial=0)
>>> sys.version_info[:2]
(2, 7)

sam@fi ~> python3
>>> import sys
>>> sys.version
"3.3.2 (default, Jun 30 2014, 12:45:18) \n[GCC 4.8.2 20131212 (Red Hat 4.8.2-7)]"
>>> sys.version_info
sys.version_info(major=3, minor=3, micro=2, releaselevel="final", serial=0)
>>> sys.version_info[:2]
(3, 3)
```

Code Snippet 4: Fetching Python's version number using the `sys` package

However, all is not lost. There is an even easier way that doesn't even involve inspecting the contents of the `version_info[:2]` tuple. You should now be familiar with `Exceptions` in Python and how we can use the `try` and `except` keywords to 'catch' errors our code may encounter. Snippet 5 shows how we can use the fact that Python raises an `ImportError` when a problem is encountered with importing modules to our advantage.

```
try:
    # Try and import using Python 2 compatible package
    from urllib2 import Request, urlopen, URLError
except ImportError:
    # Python 3 will raise an ImportError as the urllib2 package does not exist
    # and so the imports we place in this block will be run - only by Python 3!
    from urllib.request import Request, urlopen
    from urllib.error import URLError
```

Code Snippet 5: Taking advantage of `ImportError` to support Python 2 and 3

Simple! When Python 3 raises the `ImportError` as we saw in snippet 2, we can catch it and run the necessary imports for the program to work. Python 2 doesn't raise the `ImportError` and so skips over the additional imports inside the `except` block[3].

[2] PEP 3110 - Catching Exceptions in Python 3000 — https://www.python.org/dev/peps/pep-3110/

[3] There is some loss of generality in this example. Python 2 and 3 both encompass several different 'minor' versions and some packages actually differ between these minor versions. In fact the `xml` library we have been using has been known as four different packages in its lifetime! We don't need to worry about this as it's been the same since Python 2.5, but if you needed to support very old versions of Python (a **very rare** occurrence), you'd need to catch all three possible `ImportErrors`! See: Stack Overflow: Best way to import version-specific python modules — http://stackoverflow.com/questions/342437/best-way-to-import-version-specific-python-modules

* * *

As an aside, at the end of our final lecture, I mentioned my personal preference for the `Requests`[4] package. Self styled as "HTTP for Humans" its authors believe it is superior and simpler to use than the older `urllib2` package. The package is compatible with all version of Python that you have met on this course and according to the Anaconda package documentation[5], should be installed on your system already[6].

Diving straight to an example, snippet 6 presents our earlier example with `requests` in place of `urllib2`:

```
1  import requests
2  import xml.etree.ElementTree as ET
3
4  try:
5      # Make the request
6      response = requests.get("http://users.aber.ac.uk/mfc1/csm0120/api.php?"
7                              "uuid0=0BA846D3-CBA2-4D81-AD4C-00004868EA28", stream=True)
8  except requests.exceptions.RequestException as e:
9      # Catch any possible exceptions
10     print(e)
11 else:
12     tree = ET.parse(response.raw)
13     # Do something...
```

Code Snippet 6: Transplanting `requests` in to our earlier example

Here the `Request` and `urlopen` steps of snippet 1 are effectively rolled in to one function call to `requests.get` whose result is saved in the `response` variable. Note the `stream=True` keyword argument (line 7) which forces the get function to return a file-like response from the API which is what the `ElementTree` parser expects on line 12. We try to catch any possible `requests.exceptions.RequestException`s that may occur, else we move on to create the tree as before. Note also we pass `response.raw` to the parser, for the same reason as before[7].

It might seem like we've overcomplicated the example somewhat with this stream and raw stuff that I'm trying to circumvent a lengthy explanation of, but the benefits of the `requests` package become more obvious when we try to improve on the initial example like we have done in snippet 7:

* * *

[4] http://docs.python-requests.org/

[5] http://docs.continuum.io/anaconda/pkg-docs.html

[6] I've confirmed this for Python 2.7 but not for Python 3.3. Instructions on installing further packages with `conda` can be found at http://docs.continuum.io/anaconda/faq.html#install-packages

[7] We don't need to go in to much detail here but if we did not set `stream=True` and tried to just pass the plain XML response string to the parser, `parse` would assume this string was a file name and fail to build a tree. We could use `fromString` instead, but as we saw in the final lecture, that can lead to some troubles of its own and we're trying to reduce the number of changes needed to use this different package!

```
1  import requests
2  import xml.etree.ElementTree as ET
3
4  # Populate a "payload" dictionary to send to the API
5  payload = {"uuid0": "0BA846D3-CBA2-4D81-AD4C-00004868EA28",
6          "county0": "GREATER MANCHESTER"}
7
8  try:
9      # Make the request
10     response = requests.get("http://users.aber.ac.uk/mfc1/csm0120/api.php",
11                             params=payload, stream=True)
12 except requests.exceptions.RequestException as e:
13     # Catch any possible exceptions
14     print(e)
15 else:
16     # Check whether the request was actually successful
17     if response.status_code == 200:
18         tree = ET.parse(response.raw)
19         # Do something...
20     else:
21         # If the status is not HTTP 200 (OK), raise the status as an HTTPError
22         print(response.raise_for_status())
```

Code Snippet 7: A somewhat improved example using some other features of `requests`

Firstly we populate a dictionary with the key-value pairs that we want to use to query the API (see Assignment 2 for further information). Here we ask for some information on a particular house sale and the geographical co-ordinates of a county that happens to contain a sneaky space character.

We've added a second keyword argument on line 11 to the `requests.get` function; `params=payload`. This `params` argument accepts a dictionary which is used to build the URL of the API request automatically, including escaping characters such as spaces where necessary[8] (see snippet 8). Note we no longer need the ? at the end of the URL on line 10 either, it's taken care of.

Whilst not necessary due to the nature of this assignment and the API itself, in a real world application we might also want to check the status code of the response from the API. You've most likely come across '404' errors on the internet for pictures of cats that can no longer be found, such errors are not raised by `RequestException` as a '404' is still a valid response. Many APIs will return a status code of '400' for "Bad Request"[9] for incorrect or missing parameters in your query.

As shown on line 17, we can test whether the `status_code` attribute of our `response` is equal to the number 200, which represents the status "OK". If it is, we can get on with building the tree as before, otherwise we use the response's special `raise_for_status` function to elevate the HTTP status code to a proper exception; `HTTPError` that will stop the program and print an error. HTTP for Humans indeed!

```
>>> import requests
>>> response = requests.get("http://users.aber.ac.uk/mfc1/csm0120/api.php", params={
>>>     "uuid0": "0BA846D3-CBA2-4D81-AD4C-00004868EA28",
>>>     "county0": "GREATER MANCHESTER", "county1": ">^_^< i am a cat"})
>>> response.url
u"http://users.aber.ac.uk/mfc1/csm0120/api.php?county1=%3E%5E_%5E%3C+i+am+a+cat&"\
"county0=GREATER+MANCHESTER&uuid0=0BA846D3-CBA2-4D81-AD4C-00004868EA28"
```

Code Snippet 8: Given a dictionary `params`, `requests` can build your API endpoint URL for you

[8] You may wonder why the space has been encoded to `+` instead of the `%20` we were shown in class. The plus symbol is actually a valid replacement for the space character when being used in a query string of an URL, you'll notice other non-alphanumeric characters are correctly encoded to their %XX counterparts

[9] You can find a full list on Wikipedia: List of HTTP status codes — https://en.wikipedia.org/wiki/List_of_HTTP_status_codes