Supplementary Modules

Python 3

home (../handouts.html)

Lightweight data storage with JSON

JSON (JavaScript Object Notation) is a simple "data interchange" format for sending structured data through text.

Structured simply means that the data is organized into standard programmatic containers (lists and dictionaries). In fact, JSON uses the same notation as Python (and vice versa) so it is immediately recognizable to us.

Here is some simple JSON with an arbitrary structure, saved into a file called **mystruct.json**:

```
{
   "key1":  ["a", "b", "c"],
   "key2":  {
              "innerkey1": 5,
              "innerkey2": "woah"
            },
   "key3":  55.09,
   "key4":  "hello"
}
```

Initializing a Python structure read from JSON

We can load this structure from a file read from a string:

```
fh = open('mystruct.json')
mys = json.load(fh)           # load from a file
fh.close()

fh = open('mystruct.json')
file_text = fh.read()

mys = json.loads(file_text)   # load from a string

fh.close()

print(mys['key2']['innerkey2'])   # woah
```

Note: although it resembles Python structures, JSON notation is slightly less forgiving than Python -- for example, double quotes are required around strings, and no trailing comma is allowed after the last element in a dict or list (Python allows this).

For example, I added a comma to the end of the outer dict in the example above:

```
  "key4":  "hello",
```

When I then tried to load it, the **json** module complained with a helpfully correct location:

```
ValueError: Expecting property name: line 9 column 1 (char 164)
```

Dumping a Python structure to JSON

```
json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])

json.dump(['streaming API'], io)
```

python object serialization with 'pickle'

Python has modules that can *serialize* (save on disk) Python structures directly. This works very similarly to **JSON** encoding except that a pickle file is not human-readable.

The process is simply to pass the structure to pickle for dumping or loading. Like

```
import pickle
br = [('joe', 22, 'clerk', 33000.0), ('pete', 34, 'salesman', 51000.0)]
F = open('datafile.pickle', 'w')
pickle.dump(br, F)
F.close()

## later
G = open('datafile.txt')
J = pickle.load(G)
print(J)
    #  [('joe', 22, 'clerk', 33000.0), ('pete', 34, 'salesman', 51000.0)]
```

This code simply pickles and then unpickles the br data structure. It shows that the structure is stored in a file and can be retrieved from it and brought back into the code using pickle.

The other main difference between **pickle** and **json** is that you can store objects of any type. You can even pickle custom objects, although the class to which they belong must be imported first.

To serialize functions, classes, modules, etc. without any prior importing, it is possible to serialize Python into bytecode strings using the **marshal** module. There are certain caveats regarding this module's implementation; it is possible that marshal will not work passing bytecode between different Python versions.

sqlite3: local file-based database

An **sqlite3** lightweight database instance is built into Python and accessible through SQL statements. It can act as a simple storage solution, or can be used to prototype database interactivity in your Python script and later be ported to a production database like MySQL, Postgres or Oracle.

sqlite3 commands are very similar to standard databases:

```
import sqlite3
conn = sqlite3.connect('example.db')  # a db connection object

c = conn.cursor()                     # a cursor object for issuing queries
```

Once a cursor object is established, SQL can be used to write to or read from the database:

```
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')
```

Note that **sqlite3** datatypes are nonstandard and don't reflect types found in databases such as **MySQL**:

INTEGER	all int types (TINYINT, BIGINT, INT, etc.)
REAL	FLOAT, DOUBLE, REAL, etc.
NUMERIC	DECIMAL, BOOLEAN, DATE, DATETIME, NUMERIC
TEXT	CHAR, VARCHAR, etc.
BLOB	BLOB (non-typed (binary) data, usually large)

Insert a row of data

```
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")
```

Larger example that inserts many records at a time

```
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
             ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
             ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
            ]
c.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)', purchases)
```

Commit the changes -- this actually executes the insert

```
conn.commit()
```

Retrieve single row of data

```
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)

tuple_row = c.fetchone()
print(tuple_row)           # (u'2006-01-05', u'BUY', u'RHAT', 100, 35.14)
```

Retrieve multiple rows of data

```
for tuple_row in c.execute('SELECT * FROM stocks ORDER BY price'):
    print(tuple_row)

### (u'2006-01-05', u'BUY', u'RHAT', 100, 35.14)
### (u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
### (u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
### (u'2006-04-05', u'BUY', u'MSFT', 1000, 72.0)
```

Close the database

```
conn.close()
```

time and datetime

time: work with time values

```
import time

print(time.time())     # 1450818009.925441 (epoch seconds)

time.ctime()          # 'Tue Dec 22 16:02:13 2015'
time.ctime(1)         # 'Wed Dec 31 19:00:01 1969'  (epoch begins 12/31/69 19:00)

time.sleep(10)        # pause execution 10 seconds
```

datetime.datetime and **datetime.timedelta**

Python uses the datetime.datetime library to allow us to work with dates. Using it, we can convert string representations of date and time (like "4/1/2001" or "9:30") to datetime objects, and then compare them (see how far apart they are) or change them (advance a date by a day or a year).

```
from datetime import datetime    # load the datetime library
dt = datetime.now()              # create a new date object set to now
dt = datetime.today()            # same

dt = datetime(2011, 7, 14, 14, 22, 29)
                                 # create a new date object by setting
                                 # the year, month, day, hour, minute,
                                 # second (milliseconds if desired)

dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
                                 # create a new date object
                                 # by giving formatted date and time,
                                 # then telling datetime what format we used
```

Once a datetime object has been created, we can view the date in a number of ways

```
print(dt)                         # print formatted (ISO) date
                                 # 2011-07-14 14:22:29.045814

print(dt.strftime("%m/%d/%Y %H:%M:%S"))
                                 # print formatted date/time
                                 # using string format tokens
                                 # '07/14/2011 14:22:29'

print(dt.year)                    # 2011
print(dt.month)                   # 7
print(dt.day)                     # 14
print(dt.hour)                    # 14
print(dt.minute)                  # 22
print(dt.second)                  # 29
print(dt.microsecond)
print(dt.weekday())               # 'Tue'
```

Comparing datetime objects

Often we may want to compare two dates. It's easy to see whether one date comes before another by comparing two date objects as if they were numbers:

```
d1 = datetime(2011, 7, 14, 9, 40, 15)
                                 # new date object:  July 14, 2011  9:40:15am

d2 = datetime(2011, 6, 14, 9, 30, 00)
                                 # new date object:  June 14, 2011  9:30:00am

print(d1 < d2)                    # True

print(d2 > d1)                    # False
```

"Delta" means change. If we want to measure the difference between two dates, we can subtract one from the other. The result is a timedelta object:

```
td = d1 - d2        # a new timedelta object
print(td.days)       # 30
print(td.seconds)    # 615
```

Between June 14, 2011 at 9:30am and July 14, 2011 at 9:45am, there is a difference of 30 days, 10 minutes and 15 seconds. timedelta doesn't show difference in minutes, however: instead, it shows the number of seconds -- seconds can easily be converted between seconds and minutes/hours with simple arithmetic.

Changing a date using the timedelta object

Timedelta can also be constructed and used to change a date. Here we create timedelta objects for 1 day, 1 hour, 1 minute, 1 second, and then change d1

```
from datetime import timedelta
add_day = timedelta(days=1)
add_hour = timedelta(hours=1)
add_minute = timedelta(minutes=1)
add_second = timedelta(seconds=1)

print(d1.strftime("%m/%d/%Y %H:%M:%S"))             # '07/14/2011 9:40:15'

d1 = d1 - add_day
d1 = d1 - add_hour
d1 = d1 - add_minute
d1 = d1 - add_second

print(d1.strftime("%m/%d/%Y %H:%M:%S"))             # '07/13/2011 8:39:14'
```

random

The **random** module is aptly named...

```
import random

r1 = random.random()        # random between 0.0 and 1.0 (full float precision)

print(len(set([ random.random() for x in range(10000000) ])))  # 10000000


r2 = random.randint(1, 10)   # random integer between 1 and 10, inclusive


r3 = random.choice(['this', 'that', 'other'])
```

Reading a file with 'with'

```
with open('myfile.txt') as fh:
    for line in fh:
        print(line)

## at this point (outside the with block), filehandle fh has been closed.
```

Implementing a 'with' context

```
class LinkResolve():
    """ this class implements __exit__() so is used with python 'with'.
        Exit of the 'with' block results in one of two behaviors:
        - with a string argument, creates a 'link' db connection
          and closes the connection at 'with' block exit
        - with a 'link' db connection argument, returns the
          link db connection and does not close it.  """

    def __init__(self, obj):
        if isinstance(obj, link.wrappers.dbwrappers.DBConnectionWrapper):
             self.link = obj
        elif isinstance(obj, str):
            self.link = lnk('dbs.' + obj)
        else:
            raise ValueError('dutils.LinkResolve():  LinkResolve() requires a '
                             'single string or link db connection argument')

    def __enter__(self):
        return self

    def __exit__(self, type, value, traceback):
        if not keepalive_all and not self.keepalive and not self.link.wrap_name.startswith('salesforce'):
            self.link.close()
```

Python test web servers: SimpleHTTPServer and CGIHTTPServer

A web server program (like **Apache** or **IIS**) runs continuously on a server on the internet and responds to web requests from browsers and other programs. It retrieves *static pages* or fields *dynamic requests*. In a dynamic request, we ask the server to execute a python program for us and feed us the output.

In the past server programs needed to be installed and extensively configured before use. But **Python provides a basic test server** through its **SimpleHTTPServer** and **CGIHTTPServer** modules.

starting a static page (HTTP) server through Python. This option allows you to serve out static pages or files like text files, HTML documents, images, etc.

1. Create/find a directory for your website

```
$ mkdir webtest
$ cd webtest
```

2. Create/move a static page (html or txt) into the directory. (I called mine **test.html**.)

```
<HTML>
  <HEAD><TITLE>My Page</TITLE></HEAD>
  <BODY><H1>Hello, static page!</H1></BODY>
</HTML>
```

3. Start the HTTP server *from the same directory*.

```
$ python -m SimpleHTTPServer
```

By default, the home directory for the new web server is the directory from which it was invoked.

4. Visit the web page: **http://localhost:8000/test.html**

You should see the page rendered in HTML or as plaintext.

starting a dynamic script (CGI) server through Python. This option allows you to serve static as well as dynamic (i.e., running Python scripts) requests.

1. Create/find/reuse the directory for your website and **cd** into it

2. Make a new directory, **cgi-bin**, inside the current directory

3. Create a python script that prints a **Content-type:** header and some plaintext or HTML (see next slide for a more elaborate test).

```
#!/usr/bin/env python

import random
myrand = random.randint(0,11)

print("Content-type:  text/html\n")   # tells the web server what type of text

print("<HTML><HEAD></HEAD><BODY>")
print("  <H1>Hello, dynamic page!</H1>")
print("  Your lucky number is {}".format(myrand))
print("</BODY></HTML>")
```

5. Start the web server

```
$ python -m CGIHTTPServer
```

You can now place static pages in the main directory from which you started the web server, and scripts to be executed inside the **cgi-bin** directory found there.

4. Visit the web page:
http://localhost:8000/cgi-bin/test.cgi

5. Wow! Asi de facil.

Taking Parameter Input

URL parameters - the "query string"

In our usual Python programs, we take input through the command line (**sys.argv**) or **raw_input()**. In the CGI world, we receive input to our programs through the magic of *parameter input*. In this scheme, input is often passed us as key/value pairs as a *query string*. It looks like this:

```
http://www.mydomain.cgi/cgi-bin/input.cgi?key1=value1&key2=value2
```

This input can be built into the query string in a few ways:

1. it can be simply *typed directly* into the URL;
2. it can be part of an *html link* (see many of the links in the homework submission system); or

3. it can be created by a *form submission*

typed directly
When we are testing our input, we may simply change the parameters in the URL to test the behavior of our script.

html link

An html link with parameters looks like this:

```
<A HREF="http://i5.nyu.edu/~dbb212/cgi-bin/assignments.cgi?student_id=tosin&assignment_id=3.1&action=submit">3.1
```

This HTML creates a clickable link that visits the cgi shown and includes the parameters indicated.

form submission

We can use a form to allow the user to enter values and have them become part of the submitted query string:

```
<FORM ACTION="http://i5.nyu.edu/~dbb212/cgi-bin/take_input.cgi">
  Name:  <INPUT TYPE="text" NAME="name"><BR>
  Age:  <INPUT TYPE="text" NAME="age"><BR>
  <INPUT TYPE="submit">
</FORM>
```

This form, when the **submit** button is clicked, will make a new request to the **ACTION** url with the values entered into the form. The resulting query string will look like this:

```
http://i5.nyu.edu/~dbb212/cgi-bin/take_input.cgi?name=Joe&age=22
```

CGI: Decoding Parameter Input

To read parameter input from a form or URL, we use a module called **cgi**. The **FieldStorage()** method gives us access to the parameter values.

The below script, **test.py**, contains a form from which you can submit some text through a CGI parameter. The form submits its data to the same script, which then displays the input:

```
#!/usr/bin/env python

import cgi
import cgitb; cgitb.enable()  # for troubleshooting

print("Content-type: text/html")
print()

print("""
<html>

<head><title>Sample CGI Script</title></head>

<body>

  <h3> Sample CGI Script </h3>
""")

form = cgi.FieldStorage()
message = form.getvalue("message", "(no message)")

print("""

  <p>Previous message: %s</p>

  <p>form

  <form method="post" action="test.py">
    <p>message: <input type="text" name="message"/></p>
  </form>

</body>

</html>
""" % cgi.escape(message))
```

Replace your test script with this script's contents, make sure the form's **action** parameter points to the name of the script itself.

CGI: Debugging

An *Internal Server Error* occurs when something goes wrong with your script. The server masks the actual error (and in fact doesn't know what to do with it) by logging it in the web server's error log and returning **Internal Server Error** to the browser. Occasionally, we may even see a blank page result - this also indicates that there was an error but that the web server doesn't know what to do with it.

Running from the Command Line: error?
First thing to check is your script. Does it run without error? Open a new **Terminal** or **Command Prompt** window and run the script directly from the command line. If you see a Python error, correct it.

*Running from the Command Line: **Content-type:** header and blank line?*
Next thing to verify is that the script prints a valid **Content-type:** header and a blank line before printing anything else. If it does not do so, make sure it does.

Checking the log file
Every apache web server logs errors to a file called **error_log**. The log's location varies depending on the installation; on **i5** it is at **/var/apache2/2.2/logs/error_log**; on my Mac the location is **/var/apache2/2.2/logs/error_log**. Using **ssh** to reach the server and issuing the command **tail -f error_log** (where **error_log** is the path to the error log) will bring up a

dynamic session that displays every additional line added to the error log.

*Using **cgi.tb***

The **cgi traceback** module will deliver a nicely detailed traceback for certain types of errors (unfortunately it won't catch SyntaxError exceptions - for example a missing close quote - but these can be uncovered by running from the command line).

```
#!/usr/bin/env python

import cgitb
cgitb.enable()


print "Content-type:  text/plain\n"
print "Hello, world!"
5/0
```

The above script will result in a traceback showing the ZeroDivisionError exception in the last line.

Python as a web client: the urllib2 module

A Python program can take the place of a browser, requesting and downloading HTML pages and other files. Your Python program can work like a web spider (for example visiting every page on a website looking for particular data or compiling data from the site), can visit a page repeatedly to see if it has changed, can visit a page once a day to compile information for that day, etc.

urllib2 is a full-featured module for making web requests. Although the **requests** module is strongly favored by some for its simplicity, it has not yet been added to the Python builtin distribution.

The **urlopen** method takes a url and returns a file-like object that can be **read()** as a file:

```
import urllib2
my_url = 'http://www.nytimes.com'
readobj = urllib2.urlopen(my_url)
text = readobj.read()
print text
readobj.close()
```

Alternatively, you can call **readlines()** on the object (keep in mind that many objects that can deliver file-like string output can be read with this same-named method:

```
for line in readobj.readlines():
  print line
readobj.close()
```

The text that is downloaded is HTML, Javascript, and possibly other kinds of data. It is not designed for human reading, but it does contain the info that we would need to retrieve from a page. To do so programmatically (instead of visually), we need to engage in *web scraping*. Most simply, this is done with regexes (coming up).

Encoding Parameters: urllib2.urlencode()

When including parameters in our requests, we must *encode* them into our request URL. The **urlencode()** method does this nicely:

```
import urllib
params = urllib.urlencode({'choice1': 'spam and eggs', 'choice2': 'spam, spam, bacon and spam'})
print "encoded query string: ", params
f = urllib.urlopen("http://i5.nyu.edu/~dbb212/cgi-bin/pparam.cgi?%s" % params)
print f.read()
```

this prints:

```
encoded query string: choice1=spam+and+eggs&choice2=spam%2C+spam%2C+bacon+and+spam

choice1:  spam and eggs<BR>
choice2:  spam, spam, bacon and spam<BR>
```

Beautiful Soup

Nested data can take the form of HTML or XML. Either type of document can be parsed using the Python module **Beautiful Soup**, which must be installed (it does not come included with the Python distribution).

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')

print(soup.title)             # <title>The Dormouse's story</title>

print(soup.title.name)        # u'title'

print(soup.title.string)      # u'The Dormouse's story'

print(soup.title.parent.name)  # u'head'

print(soup.p)                 # <p class="title"><b>The Dormouse's story</b></p>

print(soup.a)                 # <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

print(soup.find_all('a'))
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

print(soup.find(id="link3"))
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>

# all links in a page
for link in soup.find_all('a'):
    print(link.get('href'))

# pure text from a page
print(soup.get_text())

## Tag object
print(soup.a)                 # fetches the next <A> tag
```

Float precision and the Decimal object (1/3)

Because they store numbers in binary form, all computers are incapable of absolute float precision -- in the same way that decimal numbers cannot precisely represent 1/3 (0.33333333... only an infinite number could reach full precision!)

Thus, when looking at some floating-point operations we may see strange results. (Below operations are through the Python interpreter.)

```
>>> 0.1 + 0.2
0.30000000000000004          # should be 0.3?

>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17        # should be 0.0?

>>> round(2.675, 2)
2.67                         # round to 2 places - should be 2.68?
```

Float precision and the Decimal object (2/3)

What's sometimes confusing about this 'feature' is that Python doesn't always show us the true (i.e., imprecise) machine representation of a fraction:

```
>>> 0.1 + 0.2
0.30000000000000004

>>> print 0.1 + 0.2
0.3

>>> 0.3
0.3
```

The reason for this is that in many cases Python shows us a *print representation* of the value rather than the actual value. This is because for the most part, the imprecision will not get in our way - rounding is usually what we want.

Float precision and the Decimal object (3/3)

However, if we want to be sure that we are working with precise fractions, we can use the **decimal** library and its **Decimal** objects:

```
from decimal import Decimal          # more on module imports later

float1 = Decimal('0.1')     # new Decimal object value 0.1
float2 = Decimal('0.2')     # new Decimal object value 0.2
float3 = float1 + float2    # now float is Decimal object value 0.3
```

Although these are not **float** objects, **Decimal** objects can be used with other numbers and can be converted back to **floats** or **ints**.