Python Workshop
File Input/Output

Mike Fienen

U.S. Geological Survey
Wisconsin Water Science Center, Middleton, Wisconsin USA

USGS National Groundwater Workshop, August 2012

Outline

1. Reading and Writing Strings
2. Text File Reading and Writing
3. Data from the Web

Overview

- Much of what is useful to do in Python is reading files, manipulating the data, and writing out results in another format
- Python and Numpy provide ways to read and write ASCII and binary files. We will focus on ASCII files

Reading and Writing with Strings

- The simplest way to write information to a string is using `str`

```
>>>a = 5.4
>>>str(a)
'5.4'
```

- We typically want more control. Two main ways to do it. Old school (`%`) and new school (`format`)
- Formatted input and output are a key difference between Python 2.X and 3.X

Writing Strings the Old School Way (%)

- The general syntax is to make a string with conversion types for variables. For example:

```
>>>outstr = 'I have %21.1f kg of %s and %d bins of %s' %(3.99983,'eggs',53,'spam')
>>> outstr
'I have                   4.0 kg of eggs and 53 bins of spam'
>>>outstr = 'I have %-21.1f kg of %s and %0.3d bins of %s' %(3.99983,'eggs',53,'spam')
>>> outstr
'I have 4.0                   kg of eggs and 053 bins of spam'
```

- The general idea is to make a string including `'%'`, a conversion flag (optional), a width and resolution (optional), and a conversion type (required).

  For example:

  `%<flag><width>.<resolution><type>`

  `%-12.3f` Is a left-justified, floating point value with width of 12 and 3 decimal places.

- Following the format string must be a list of values as a tuple identified by `%`

Writing Strings the Old School Way (%)

Details about formatted output available at:

`http://docs.python.org/library/stdtypes.html`

- Conversion flag characters
  - `'#'` Invokes alternate behavior (see website for details)
  - `'0'` Pads numeric values with zeros
  - `'-'` Left-adjusts the output
  - `' '` Leave a space before signed positive values so they line up with negative ones

Writing Strings the Old School Way (%)

Details about formatted output available at:

http://docs.python.org/library/stdtypes.html

- Conversion flag characters

'#'	Invokes alternate behavior (see website for details)
'0'	Pads numeric values with zeros
'–'	Left-adjusts the output
' '	Leave a space before signed positive values so they line up with negative ones

- Most common conversion types.

%d or %i	Signed integer
%f or %F	Floating point
%e or %E	Floating point exponential (lower or upper case)
%g or %G	Combination of %f and %e depending on resolution
%s or %r	String. Width is used, but not resolution

Writing Strings the New School Way (`format`)

Details about new school string formatting at:

http://docs.python.org/library/string.html#formatstrings

- The general syntax is similar, but conversion information is supplied differently. For example:

```
>>>outstr = 'I have %{0:21.1f} kg of {1:s} and {2:0=3d} bins of {3:s}'.
format(3.99983,'eggs',53,'spam')
>>> outstr
'I have                   4.0 kg of eggs and 053 bins of spam'
```

- In this case, make a string including {. . .}, statements with conversion information.
  The general pattern is {`[index]`:`[format]`}
  - The `[index]` argument refers to the item index being mapped in
  - The `[format]` argument is similar to those in the old school way, but with some additional flexibility

Now that we can write strings, how about we write them to files?
We should also know how to read stuff back in from files

Interacting with Text Files

- The first thing is to open a file and make a file object

```
ifp = open('somefile.txt','r')
ofp = open('someotherfile.txt','w')
```

  - This object can be used to read or write from.
    I use `ifp` for “input file pointer” and
    `ofp` for “output file pointer”
    The arguments `′r′` and `′w′` indicate “read” and “write” respectively.

- To read the file can use `readline()` or `readlines()`
  - The difference is that `readlines()` reads the entire file into memory rather than `readline()` which reads one line at a time. Most of the time, `readlines()` is better
  - With `readlines()` once the data are read in, the result is a list with each element representing a line in the text file

Parsing input strings

- Using `strip` and `split`

```
>>>line = 'USGS        430406089232901 2010-12-03      15.04   P\t\r\n'
>>>line.strip()
'USGS\t430406089232901\t2010-12-03\t15.04\tP'
```

  - `strip()` removes newline and tab characters from the end

```
>>>line.strip().split()
['USGS', '430406089232901', '2010-12-03', '15.04', 'P']
>>>line.strip().split('0')
['USGS\t43', '4', '6', '892329', '1\t2', '1', '-12-', '3\t15.', '4\tP']
```

  - `split()` breaks up a string on whitespace
  - Can take any character as an argument (usually ',' or ' ')
  - Stacking `strip` and `split` is a common violation of the general rule not to stack up function calls

Parsing input strings (continued)

- Using `pop`

```
>>>>>>line = 'USGS        430406089232901 2010-12-03       15.04    P\t\r\n'
>>>a=line.strip().split()
>>>a.pop()
'P'
>>>a
['USGS', '430406089232901', '2010-12-03', '15.04']
>>>a.pop(1)
'430406089232901'
>>>a
['USGS', '2010-12-03', '15.04']
```

  - `pop()` *both* returns an element from a list *and* removes it from the remaining list

- Regular expressions are very flexible but another topic

```
>>>import re
>>>allints = re.findall("[0-9]",line)
>>>allints
['4', '3', '0', '4', '0', '6', '0', '8', '9', '2', '3', '2', '9', '0', '1', '2',
'0', '1', '0', '1', '2', '0', '3', '1', '5', '0', '4']
```

An Example Text File from NWIS

```
# ---------------------------------- WARNING ----------------------------------------
# Provisional data are subject to revision. Go to
# http://waterdata.usgs.gov/nwis/help/?provisional for more information.
#
# File-format description:  http://waterdata.usgs.gov/nwis/?tab_delimited_format_info
# Automated-retrieval info: http://waterdata.usgs.gov/nwis/?automated_retrieval_info
#
# Contact:   gs-w_support_nwisweb@usgs.gov
# retrieved: 2012-07-16 17:24:35 EDT (vaas01)
#
# Data for the following 2 site(s) are contained in this file
#    USGS 430406089232901 DN-07/09E/23-1297
#    USGS 430427089284901 DN-07/09E/19-0064
# -----------------------------------------------------------------------------------
#
# Data provided for site 430406089232901
#    DD parameter statistic   Description
#    01   72019     00001     Depth to water level, feet below land surface (Maximum)
#
# Data-value qualification codes included in this output:
#     P  Provisional data subject to revision.
#
agency_cd site_no datetime 01_72019_00001 01_72019_00001_cd
5s 15s 20d 14n 10s
USGS 430406089232901 2010-12-03 15.04 P
USGS 430406089232901 2010-12-04 14.92 P
```

Reading NWIS Output File

```
# loop over the input data, keep only proper data rows. Parse and assign to lists
for lnum, line in enumerate(tmpdat):
  # first read the lookup information from the header of the file
  if ("data for the following" in line.lower()):
    nWells = int(re.findall("[0-9]+",line)[0])
    statnums = []
    countynums = []
    for cwell in np.arange(nWells):
      nextline = lnum+1+cwell
      tmp = tmpdat[nextline].strip().split()
      statnums.append(tmp[2])
      countynums.append(tmp[3])
    station_lookup = dict(zip(statnums,countynums))

  if (('usgs' in line.lower()) and ('#' not in line)):
    tmp = line.strip().split() # strip newline off the end and split on whitespace
    Site_ID.append(tmp[1])
    dates.append(datetime.strptime(tmp[2],indatefmt)) #convert date to a time tuple
    DTW.append(tmp[3])
    prov_code.append(tmp[4].lower()) # --> note conversion to lower case!
```

`np.genfromtxt`: flexible way to read columns

Example file: `STATE_FIPS.csv`

```
State Abbreviation,FIPS Code,State Name
AK,02,ALASKA
AL,01,ALABAMA
AR,05,ARKANSAS
AS,60,AMERICAN SAMOA
AZ,04,ARIZONA
CA,06,CALIFORNIA
...
```

np.genfromtxt: flexible way to read columns

Example file: STATE_FIPS.csv

```
State Abbreviation,FIPS Code,State Name
AK,02,ALASKA
AL,01,ALABAMA
AR,05,ARKANSAS
AS,60,AMERICAN SAMOA
AZ,04,ARIZONA
CA,06,CALIFORNIA
...
```

```
import numpy as np
infilename = 'STATE_FIPS.csv'
indat = np.genfromtxt(infilename,delimiter=',',dtype=None,names=True)
```

`np.genfromtxt`: flexible way to read columns

Example file: `STATE_FIPS.csv`

```
State Abbreviation,FIPS Code,State Name
AK,02,ALASKA
AL,01,ALABAMA
AR,05,ARKANSAS
AS,60,AMERICAN SAMOA
AZ,04,ARIZONA
CA,06,CALIFORNIA
...
```

```
import numpy as np
infilename = 'STATE_FIPS.csv'
indat = np.genfromtxt(infilename,delimiter=',',dtype=None,names=True)
```

`delimiter=’,’`	delimiter can be *anything*
`dtype=None`	Numpy interprets column data types. If unknown, makes it a string
`names=True`	Each column gets a data type and a name

`np.genfromtxt`: flexible way to read columns

Example file: `STATE_FIPS.csv`

```
State Abbreviation,FIPS Code,State Name
AK,02,ALASKA
AL,01,ALABAMA
AR,05,ARKANSAS
AS,60,AMERICAN SAMOA
AZ,04,ARIZONA
CA,06,CALIFORNIA
...
```

```
import numpy as np
infilename = 'STATE_FIPS.csv'
indat = np.genfromtxt(infilename,delimiter=',',dtype=None,names=True)
```

`delimiter=','` delimiter can be *anything*

`dtype=None` Numpy interprets column data types. If unknown, makes it a string

`names=True` Each column gets a data type and a name

```
In [10]: indat
Out[7]:
array([('AK', 2, 'ALASKA'), ('AL', 1, 'ALABAMA'), ('AR', 5, 'ARKANSAS'), ...
dtype=[('State_Abbreviation', '|S2'), ('FIPS_Code', '<i4'), ('State_Name', '|S20')])
```

np.genfromtxt: flexible way to read columns

Example file: STATE_FIPS.csv

```
State Abbreviation,FIPS Code,State Name
AK,02,ALASKA
AL,01,ALABAMA
AR,05,ARKANSAS
AS,60,AMERICAN SAMOA
AZ,04,ARIZONA
CA,06,CALIFORNIA
...
```

```
import numpy as np
infilename = 'STATE_FIPS.csv'
indat = np.genfromtxt(infilename,delimiter=',',dtype=None,names=True)
```

`delimiter=','`	delimiter can be *anything*
`dtype=None`	Numpy interprets column data types. If unknown, makes it a string
`names=True`	Each column gets a data type and a name

```
In [10]: indat
Out[7]:
array([('AK', 2, 'ALASKA'), ('AL', 1, 'ALABAMA'), ('AR', 5, 'ARKANSAS'), ...
dtype=[('State_Abbreviation', '|S2'), ('FIPS_Code', '<i4'), ('State_Name', '|S20')])
```

N.B.→underscore replaces space in names

Writing Back out to a File

- First need a file object in the same way as reading

```
ofp = open('some_outfile.txt','w')
```

- Next, create a string of output
- Write the string using

```
ofp.write(<string>)
```

- Remember to put a newline character `'\n'` at the end of each line

Pulling a data file from the Web

- An example using REST (Representational State Transfer a.k.a. a RESTful query) of USGS water data.

```
import urllib
fullURL = 'www.place.gov/some_path_to_a_data_file.txt'
datastream = urllib.urlopen(fullURL).read()
outfilename = 'local_filename.txt'
open(outfilename,'wb').write(datastream)
```

- `urllib` enables simple interaction with a URL
- BeautifulSoup allows for much more sophisticated complete web-scraping applications (not built in)
- Writing the local version of the file as binary is most robust:
  - Retains a copy of exactly what was downloaded
  - Writes the copy without respect to formatting issues
- One could work only in memory and not make a local copy

Some Useful Resources

- Building queries for RESTful queries of USGS water data
  `http://waterservices.usgs.gov/`
- USGS Water data type pm code lookup
  `http://nwis.waterdata.usgs.gov/nwis/pmcodes/`
- General I/O information in Python documentation
  `http://docs.python.org/tutorial/inputoutput.html`