FILE IO

SEBASTIAN HAHN

FILE FORMATS

- ASCII
- Binary
- Numpy
- Matlab
- IDL
- NetCDF
- HDF5
- Excel
- Pickle - Python object serialization

ASCII

- A text file in which each byte represents one character according to the ASCII code.
- Contrast with a binary file, in which there is no one-to-one mapping between bytes and characters.
- Files that have been formatted with a word processor must be stored and transmitted as binary files to preserve the formatting.

PYTHON STANDARD LIBRARY

Input output routines

- `open()`, `close()`
- `read()`, `readline()`, `readlines()`
- `write()`, `writelines()`

BEFORE WE START

```
import os

f_dir = os.path.join('.', 'files')
if not os.path.exists(f_dir):
    try:
        os.mkdir(f_dir)
        print("Created files folder")
    except OSError:
        f_dir = ''
        print("Failed creating files folder")
```

```
>>> >>> ... ... ... ... ... ... ... Created files folder
```

ASCII EXAMPLE 1

```
filename = os.path.join(f_dir, 'example1.txt')

f = open(filename, 'w')
f.write("Hello World!")
f.close()

# Using the context manager
with open(filename) as f:
    print(f.read())
```

```
>>> >>> >>> >>> >>> ... ... ... Hello World!
```

NUMPY

- np.fromregex: Construct an array from a text file, using regular expression parsing.
- np.fromstring: A new 1-D array initialized from raw binary or text data in a string.
- np.ndarray.tofile: Write array to a file as text or binary (default).

CSV

- A comma-separated values (CSV) file stores **tabular data** (numbers and text) in plain-text form.
- Plain text means that the file is a sequence of characters, with no data that has to be interpreted as binary numbers.
- A CSV file consists of any number of records, separated by line breaks of some kind; each record consists of fields, separated by some other character or string, most commonly a literal comma or tab.
- Usually, all records have an identical sequence of fields.

PYTHON STANDARD LIBRARY

Input output routines

- `open()`, `close()`
- `read()`, `readline()`, `readlines()`
- `write()`, `writelines()`

CSV EXAMPLE 1

```
filename = os.path.join(f_dir, 'example1.csv')

with open(filename, 'w') as f:
    f.write("lon,lat,temperature\n")
    data = [[16, 48, 20], [17, 48, 21], [18, 48, 18]]
    for record in data:
        f.writelines(','.join(map(str, record)))
        f.write("\n")

with open(filename) as f:
    for record in f.readlines():
        print(record)
```

```
>>> ... ... ... ... ... ... >>> ... ... ... lon,lat,temperature

16,48,20

17,48,21

18,48,18
```

NUMPY

- np.savetxt: Save an array to a text file.
- np.loadtxt: Load data from a text file.
- np.genfromtxt: Load data from a text file, with missing values handled as specified.

CSV EXAMPLE 2

```
import numpy as np

filename = os.path.join(f_dir, 'example2.csv')
data =  np.tile(np.arange(0, 5), (3, 1)).T

# save data
np.savetxt(filename, data, header='x,y,z', delimiter=',')

# load data
x, y, z = np.loadtxt(filename, skiprows=1, \
    delimiter=',', unpack=True)
print(x, y, z)
```

```
>>> >>> >>> >>> ... >>> >>> ... ... >>> (array([ 0.,  1.,  2.,  3.,
 4.]), array([ 0.,  1.,  2.,  3.,  4.]), array([ 0.,  1.,  2.,  3.,
 4.]))
```

PANDAS

The pandas I/O API is a set of top level reader functions, e.g. for CSV

- The Series and DataFrame objects have an instance method to_csv() which allows storing the contents of the object as a comma-separated-values file.
- The two workhorse functions for reading text files are read_csv() and read_table(). They both use the same parsing code to intelligently convert tabular data into a DataFrame object. See the cookbook for some advanced strategies

BINARY

- A binary file is a computer file that is not a text file.
- Binary files typically contain bytes that are intended to be interpreted as something other than text characters.
- Some binary files contain headers, blocks of metadata used by a computer program to interpret the data in the file.
- If a binary file is opened in a text editor, each group of eight bits will typically be translated as a single character,
- Binary itself is meaningless, until such time as an executed algorithm defines what should be done with each bit, byte, word or block.

PYTHON STANDARD LIBRARY

Input output routines

- `open()`, `close()`
- `read()`, `readline()`, `readlines()`
- `write()`, `writelines()`

BINARY EXAMPLE 1

```
import struct

filename = os.path.join(f_dir, 'example1.bin')
points = [(1, 2), (3, 4), (9, 10), (23, 14), (50, 90)]

msg = bytearray()
msg.extend(struct.pack('I', len(points)))

for x, y in points:
    msg.extend(struct.pack('II', x, y))

# write data
with open(filename, 'wb') as f:
    f.write(msg)

# read data
with open(filename, 'rb') as f:
     n_points = struct.unpack('I', f.read(4))[0]
     print(n_points)
```

```
>>> >>> >>> >>> >>> >>> >>> ... ... >>> ... ... ... >>> ... ... ...
... 5
```

NUMPY

- np.fromfile: Construct an array from data in a text or binary file
- ndarray.tofile: Write array to a file as text or binary (default)

BINARY EXAMPLE 2

```
filename = os.path.join(f_dir, 'example2.bin')

points = np.array([(1, 2), (3, 4), (9, 10), (23, 14), (50, 90)],
    dtype=np.int32)
points.tofile(filename)

data = np.fromfile(filename, dtype=[('x', np.int32),
    ('y', np.int32)])
print(data['x'], data['y'])
```

```
>>> ... >>> >>> >>> ... >>> (array([ 1,  3,  9, 23, 50], dtype=int32
), array([ 2,  4, 10, 14, 90], dtype=int32))
```

NUMPY

- There are a number of various Numpy IO routines (e.g. text, binary and NPZ)
- **NPZ**: a standard binary file format for persisting a single arbitrary Numpy array on disk. The format stores all of the shape and dtype information necessary to reconstruct the array correctly even on another machine with a different architecture.

NUMPY

- np.load: load arrays or pickled objects from .npy, .npz or pickled files.
- np.save: Save an array to a binary file in NumPy .npy format.
- np.savez: Save several arrays into a single file in uncompressed .npz format.
- np.savez_compressed: Save several arrays into a single file in compressed .npz format.

NUMPY EXAMPLE 1

```
filename = os.path.join(f_dir, 'example1.npy')
x = np.arange(10)

# save data
np.save(filename, x)

# load data
data = np.load(filename)
print(data)
```

```
>>> >>> ... >>> >>> ... >>> [0 1 2 3 4 5 6 7 8 9]
```

NUMPY EXAMPLE 2

```
filename = os.path.join(f_dir, 'example2.npz')
x = np.arange(4).reshape(2, 2)
y = np.sin(x)

# save data
np.savez(filename, x, y=y)

# load data
data = np.load(filename)

print(data)
print('x:', data['arr_0'])
print('y:', data['y'])
```

```
>>> >>> >>> ... >>> >>> ... >>> >>> <numpy.lib.npyio.NpzFile object
at 0x7fe392ec3190>
('x:', array([[0, 1],
       [2, 3]]))
('y:', array([[ 0.        ,  0.84147098],
       [ 0.90929743,  0.14112001]]))
```

MATLAB

- Scipy.io has many modules, classes, and functions available to read data from and write data to a variety of file formats.
- Saving and loading Matlab files (.mat)
  - supports: v4 (Level 1.0), v6 and v7 to 7.2 matfiles

SCIPY.IO

- loadmat: Load MATLAB file
- savemat: Save a dictionary of names and arrays into a MATLAB-style .mat file.
- whosmat: List variables inside a MATLAB file

MATLAB EXAMPLE 1

```
from scipy.io import loadmat, savemat, whosmat

filename = os.path.join(f_dir, 'example1.mat')
x = np.arange(10)

# save data
savemat(filename, {'var1': x})

# load data
data = loadmat(filename)

print(data['var1'])
```

```
>>> >>> >>> >>> ... >>> >>> ... >>> >>> [[0 1 2 3 4 5 6 7 8 9]]
```

MATLAB EXAMPLE 2

```
filename = os.path.join(f_dir, 'example2.mat')
x = np.arange(4).reshape(2, 2)
y = np.sin(x)

# save data
savemat(filename, {'var1': x, 'y': y})

# show variables
print(whosmat(filename))

# load data
data = loadmat(filename)

print(data['y'])
```

```
>>> >>> >>> ... >>> >>> ... [('y', (2, 2), 'double'), ('var1', (2, 2
), 'int64')]
>>> ... >>> >>> [[ 0.          0.84147098]
 [ 0.90929743  0.14112001]]
```

IDL (EXELIS VIS)

Scipy.io also supports reading IDL (.sav) files

- readsav: Read an IDL .sav file

Overview of SAVE files (from Exelisvis): You can create binary files containing data variables, system variables, functions, procedures, or objects using the SAVE procedure. These SAVE files can be shared with other users who will be able to execute the program, but who will not have access to the IDL code that created it. Variables that are used from session to session can be saved as and recovered from a SAVE file.

NETCDF

- NetCDF (Network Common Data Form) is a set of software libraries and self- describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data.
- NetCDF data is:
  - **Self-Describing**: Metadata information about the data are included.
  - **Portable**: Accessable by computers with different ways of storing integers, characters, and floating-point numbers.

NETCDF

- NetCDF data is:
  - **Scalable**: A small subset of a large dataset may be accessed efficiently.
  - **Appendable**: Data may be appended to a properly structured netCDF file.
  - **Sharable**: One writer and multiple readers may simultaneously access the same netCDF file.
  - **Archivable**: Access to all earlier forms of netCDF data will be supported by current and future versions of the software.
- Python package: netCDF4

NETCDF4 EXAMPLE 1 - PART 1

```
import netCDF4

filename = os.path.join(f_dir, 'example1.nc')

f = netCDF4.Dataset(filename, 'w', format='NETCDF4')
grp_temp = f.createGroup("temperature")
subgrp_air = grp_temp.createGroup("air")
subgrp_soil = grp_temp.createGroup("soil")

print(grp_temp.groups)
```

```
>>> >>> >>> >>> >>> >>> >>> >>> OrderedDict([('air', <type 'netCDF4.
Group'>
group /temperature/air:
    dimensions(sizes):
    variables(dimensions):
    groups:
), ('soil', <type 'netCDF4.Group'>
group /temperature/soil:
    dimensions(sizes):
    variables(dimensions):
    groups:
)])
```

NETCDF4 EXAMPLE 1 - PART 2

```
# Create Dimension
lat = f.createDimension('lat', 50)
lon = f.createDimension('lon', 50)
time = f.createDimension('time', None)
print(f.dimensions)

# Datasets and Attributes
soil_temp = np.ones((50, 50))
soil_dset = subgrp_soil.createVariable("Soil Temperature",
    soil_temp.dtype.name, ('lat', 'lon'))
soil_dset[:] = soil_temp

# http://docs.python.org/2/library/functions.html#setattr
# setattr(x, 'foobar', 123) is equivalent to x.foobar = 123
setattr(soil_dset, 'unit', 'degree celsius')
soil_dset.scaling = 1
f.close()
```

```
>>> >>> >>> OrderedDict([('lat', <type 'netCDF4.Dimension'>: name =
'lat', size = 50
), ('lon', <type 'netCDF4.Dimension'>: name = 'lon', size = 50
), ('time', <type 'netCDF4.Dimension'> (unlimited): name = 'time', s
ize = 0
)])
```

NETCDF4 EXAMPLE 1 - PART 3

```
with netCDF4.Dataset(filename) as f:
    print(f.dimensions)
    print(f.groups['temperature'].groups['soil'].variables['Soil Tem
perature'])
```

```
... ... OrderedDict([(u'lat', <type 'netCDF4.Dimension'>: name = 'la
t', size = 50
), (u'lon', <type 'netCDF4.Dimension'>: name = 'lon', size = 50
), (u'time', <type 'netCDF4.Dimension'> (unlimited): name = 'time',
size = 0
)])
<type 'netCDF4.Variable'>
float64 Soil Temperature(lat, lon)
    unit: degree celsius
    scaling: 1
path = /temperature/soil
unlimited dimensions:
current shape = (50, 50)
filling on, default _FillValue of 9.96920996839e+36 used
```

NETCDF4 EXAMPLE 1 - PART 4

```
with netCDF4.Dataset(filename) as f:
    print(f.groups['temperature'].groups['soil'].\
        variables['Soil Temperature'][:])
    print(f.groups['temperature'].groups['soil'].\
        variables['Soil Temperature'].unit)
```

```
... ... ... ... [[ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]
 ...,
 [ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]]
degree celsius
```

HDF5

- HDF5 is a data model, library, and file format for storing and managing data.
- It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data.
- HDF5 is portable and is extensible, allowing applications to evolve in their use of HDF5.
- Python package h5py

HDF5 EXAMPLE 1 - PART 1

```
import h5py

filename = os.path.join(f_dir, 'example1.hdf5')

with h5py.File(filename, 'w') as f:
    grp_temp = f.create_group("temp")
    subgrp_soil = grp_temp.create_group("soil")
    soil_temp = np.arange(400)
    soil_dset = subgrp_soil.create_dataset("Soil Temperature", \
        data=soil_temp)
    soil_dset.attrs['unit'] = 'degree celsius'
    print(soil_dset)
```

```
>>> >>> >>> ... ... ... ... ... ... ... ... <HDF5 dataset "Soil Temp
erature": shape (400,), type "<i8">
```

HDF5 EXAMPLE 1 - PART 2

```
filename = os.path.join(f_dir, 'example1.hdf5')

with h5py.File(filename) as f:
    print(f['temp/soil'].keys())
    print(f['temp/soil/Soil Temperature'])
    print(f['temp/soil/Soil Temperature'][20:40])
```

```
>>> ... ... ... ... [u'Soil Temperature']
<HDF5 dataset "Soil Temperature": shape (400,), type "<i8">
[20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39]
```

EXCEL

- Openpyxl is a Python library for reading and writing Excel 2010 xlsx/xlsm/xltx/xltm files.
- It was born from lack of existing library to read/write natively from Python the Office Open XML format.
- The package openpyxl is used by Pandas to store e.g. DataFrames in Excel sheets

EXCEL EXAMPLE 1 - PART 1

There is currently an issue between pandas (0.16.0) and the latest openpyxl (2.2.2) version, but it works with an older version of openpyxl (1.8.6)

- pip install openpyxl==1.8.6
- pip install xlrd

EXCEL EXAMPLE 1 - PART 2

```
import pandas as pd

filename = os.path.join(f_dir, 'example1.xlsx')

df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df.to_excel(filename)

data = pd.io.excel.read_excel(filename)
print(type(data), data)
```

```
>>> >>> >>> >>> >>> >>> >>> (<class 'pandas.core.frame.DataFrame'>,
   A  B
0  1  4
1  2  5
2  3  6)
```

PICKLE

- The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure.
- Pickling is the process whereby a Python object hierarchy is converted into a byte stream, and unpickling is the inverse operation, whereby a byte stream is converted back into an object hierarchy.
- This documentation describes both the pickle module and the cPickle module.

PICKLE EXAMPLE 1

```
import pickle

filename = os.path.join(f_dir, 'example1.pickle')

data = {'a': [1, 2.0, 3, 4+6j],
        'b': ('string', u'Unicode string'),
        'c': None}

with open(filename, 'wb') as f:
    pickle.dump(data, f)

with open(filename, 'rb') as f:
    print(pickle.load(f))
```

```
>>> >>> >>> ... ... >>> >>> ... ... >>> ... ... {'a': [1, 2.0, 3, (4
+6j)], 'c': None, 'b': ('string', u'Unicode string')}
```

EXERCISE

CSV

Write a text file using `np.savetxt()` which contains 3 columns x, y, z:

```
x = 2, 4, 6, ..., 18
y = 4, 8, 12, ..., 36
z = 98, 95, 92, ..., 82
```

And try to read the file you just wrote using `np.loadtxt()`

HINT: `data = np.vstack((x, y, z)).T`

NPZ

Use x, y, z again and save them using `np.savez()` and read them using `np.load()`

BINARY

Write a binary file using `ndarray.tofile()` with 10 records, where a record looks like:

```
dtype = np.dtype([('id', np.float64),
                  ('height', np.float32),
                  ('weight', np.float32)])
```

And read it using `np.fromfile()`