Intro to Research Computing with Python: File Input and Output

Erik Spence

SciNet HPC Consortium

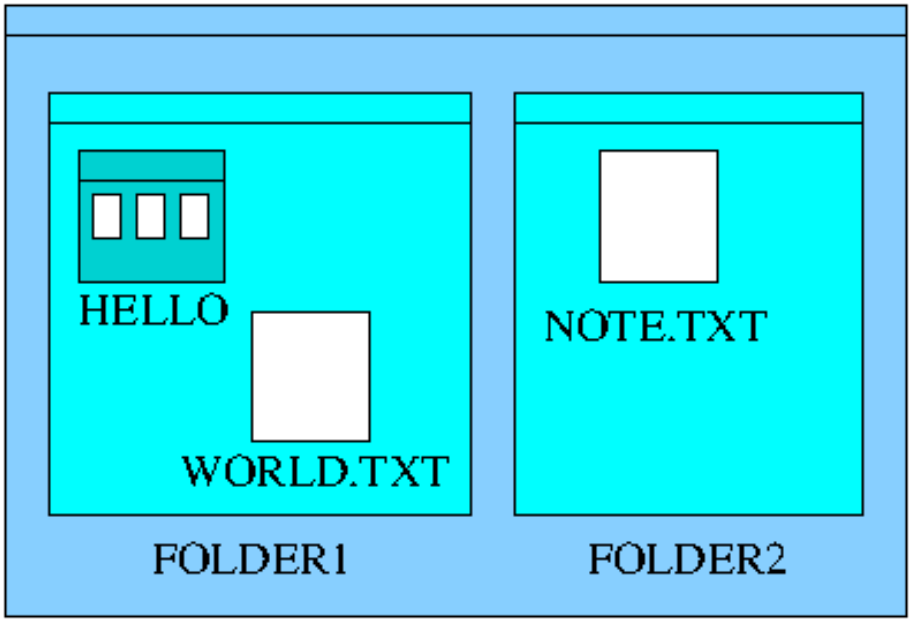
18 November 2014

Today's Lecture

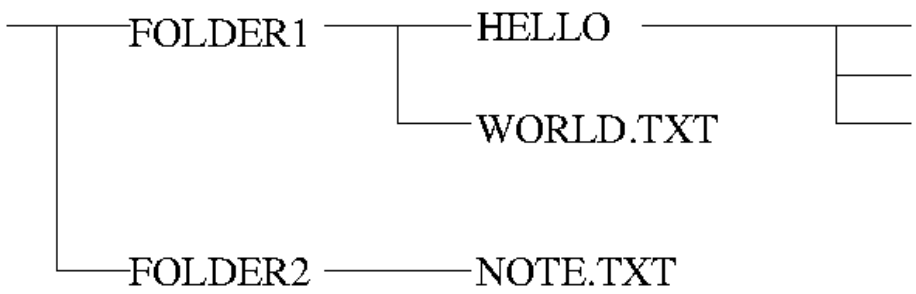
- Basic File Input and Output in Python.
- File system theory, IOPs.
- Different file formats (and how to use them).

SciNet compute • calcul CANADA

Basic File Input and Output in Python



Tree:



Files:

```
FOLDER1/WORLD.TXT
FOLDER2/NOTE.TXT
FOLDER1/HELLO/...
```

- Files contain your data.
- Files are organized in directories or folders.
- A directory is a file too.
- Path: sequence of directories to get to a file.

Directories

Create

```
In [1]: import os
In [2]: os.mkdir('FOLDER1')
```

Change current directory

```
In [3]: os.chdir('FOLDER1')
In [4]: os.chdir('..')
```

Note that, when using ipython standard Unix commands, such as mkdir, cd, and pwd, will work from the ipython prompt. These will not work within a Python script.

Write to a file

Writing

```
In [5]: f = open('FOLDER1/WORLD.TXT','w')
In [6]: line = "Hello\n"
In [7]: f.write(line)
In [8]: f.close()
```

Appending

```
In [9]: f = open('FOLDER1/WORLD.TXT','a')
In [10]: line = "World\n"
In [11]: f.write(line)
In [12]: f.close()
```

Note that on some Unix systems, append will append to the end of the file, regardless of the current seek position.

Read a file

```
In [13]: f = open('FOLDER1/WORLD.TXT','r')
In [14]: line = f.readline()
In [15]: print line
Hello
In [16]: f.close()
```

Read/Write

```
In [17]: f = open('FOLDER1/WORLD.TXT','r+')
In [18]: f.seek(1)
In [19]: f.write('a')
In [20]: line = f.readline()
In [21]: print line
Hallo
In [22]: f.close()
```

The '+' above means that the file is being opened for updating.

Glob

The glob package does only one thing: it finds all files or paths matching a specific Unix-style regular expression pattern, and returns them in a list.

```
In [23]: import glob
In [24]: f = glob.glob('*/*.TXT')
In [25]: print f
['NOTE.TXT', 'WORLD.TXT']
In [26]:
```

os.path

There are a number of useful file and directory-testing functions in os.path.

```
In [26]: import os
In [27]: print f
['NOTE.TXT', 'WORLD.TXT']
In [28]: os.path.isfile(f[0])
True
In [29]: os.path.isdir(f[1])
False
In [30]: os.path.abspath(f[1])
'/home/s/scinet/ejspence/FOLDER1/WORLD.TXT'
In [31]: os.path.expanduser('~')
'/home/s/scinet/ejspence'
```

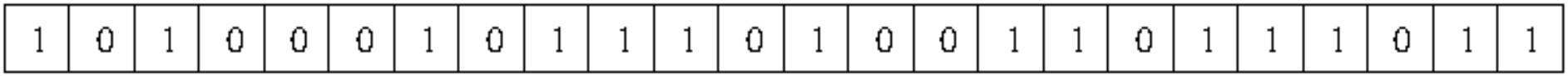
If you're looking for a directory-testing function, it's likely in os.path.

Computer Data Storage

Media:

- Memory
- Disks
- Flash (USB)
- CD/DVD
- Tape
- ...

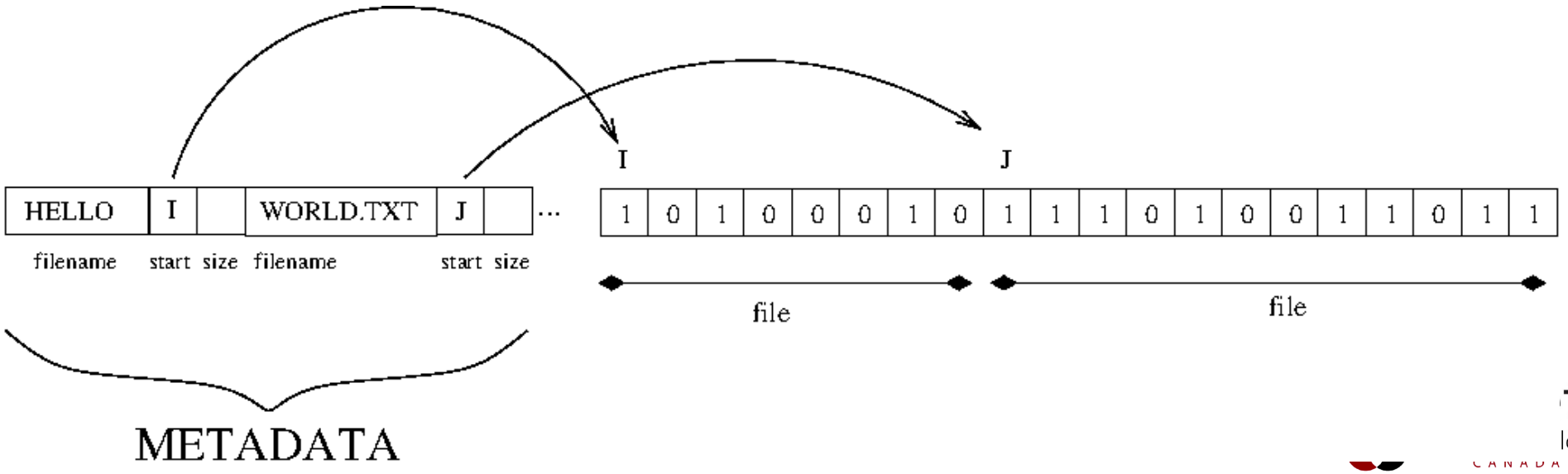
All media are essentially linear strings of bits:



In and of itself, this is useless. What do these bits mean?

File systems

- Most non-volatile media use a file system.
- This entails storing data describing the meaning of the data: **metadata**.
- Storage media is usually subdivided into files.
- Files have a name, size and usually other metadata.
- The metadata allows the operating system to understand and use that which has been stored, *e.g.*



Metadata

Metadata is the data which describes the file and its properties:

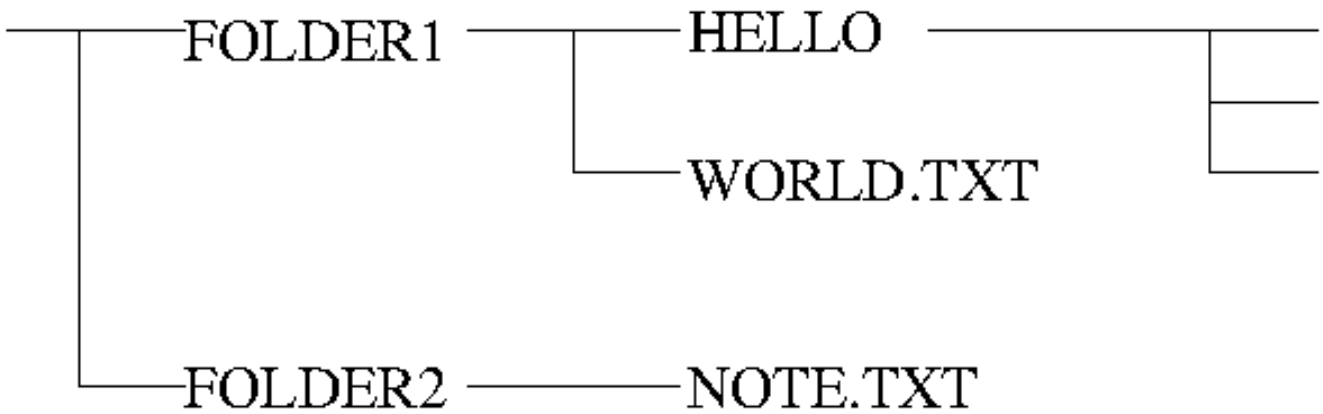
- File name
- File size
- Location on disk
- File type (though often through magic identifiers)
- Timestamp
- Read/write permissions
- Position in the directory tree
- ...

Directories or Folders

So we now have files, but this can become disorganized quickly. Imagine looking for the file 'NOTE.TXT' in a list of 10,000,000 files.

Directories

- Like special files that contain a list of (metadata for) other files.
- A directory can contain other directories, leading to a tree.



I/O Operations: opening

What really happens when we open a file? The OS:

Opening a file

1. finds the file in the directory, or creates a new entry in the directory.
2. checks permissions on the file.
3. finds the location of the file on disk.
4. initializes a file 'handle' and a file 'pointer'. The file handle is what `open` returns.

I/O Operations: writing

What really happens when write to a file?

Writing to a file

1. Python convert data to a stream of bytes.
2. The OS puts those bytes in a buffer.
3. The OS updates the file pointer.
4. If the buffer is full, the OS writes the buffer contents to file.

SciNet compute • calcul CANADA

I/O Operations: reading

What really happens when we read a file? Assuming the file has already been opened:

Reading from a file

1. if the data is not yet in a buffer: the OS reads the data into a buffer.
2. Python reads the bytes from the buffer into a variable, performing any needed conversions.
3. the OS updates file pointer.

I/O Operations: closing

What really happens when we close a file? The OS:

Closing a file

1. ensures all buffers are flushed to disk.
2. updates any metadata.
3. release the buffers associated with the file handle.

Minimizing IOPs

It is important, both for you and for other users, that you minimize file I/O.

- Disk I/O (Input/Output) is usually the slowest part of a data pipeline.
- If manipulating data from files is most of what you do, try and minimize IOPs.
- Load everything into memory once; reuse data if you can; use ramdisk.

Bad

```
s = 'Hi world\n'
for c in s:
  f = open('hiworld.txt','a')
  f.write(c)
  f.close()
```

Good

```
s = 'Hi world\n'

f = open('hiworld.txt','w')
f.write(s)
f.close()
```

What's in a file?

Files come in different formats. For our purposes there are two basic types:

Text

- On its face this seems attractive: you can just read it.
- But this is not as trivial as it may sound.
- A bit pattern must be assigned to each letter or symbol (encoding).
- Ideally there are unique assignments across languages.

Binary

- We covered the format of individual numbers in the Numerics class.
- Good binary formats include information on the data within the file, *e.g.*: HDF5, NetCDF.

Text format

An introduction to text:

- ASCII Encoding:
  7 bits = 1 character.
- 128 possible, but only 95 printable characters.
- Uses 8-bit bytes: storage efficiency 82% at best.
- ASCII representation of floating point numbers:
  - Needs about 18 bytes vs 8 bytes in binary: **inefficient**.
  - Representation must be computed: **slow**.
  - **Non-exact** representation.

ASCII

integers	characters
32	(space)
33-47	!"#$%&'()*+,-./
48-57	0-9
58-64	:;<=>?@
65-90	A-Z
91-96	[\]^_
97-122	a-z
123-126	{|}~

Text Encodings

ASCII: 7-bit encoding. For English.

Latin-1: 8-bit encoding. For western European Languages mostly.

UTF-8: *Variable-width* encoding that can represent every character in the Unicode character set.

Unicode: standard containing more than 110,000 characters.

Python can deal with these encodings:

```
# -*- coding: latin-1 -*-
print u"Comment ça va?"
```

This is especially true for Python 3.X, which was rewritten to make unicode encodings of strings the default.

SciNet compute • calcul CANADA

Binary format

We've discussed much of this in the Numerics class.

- The numbers are output to storage in the same format in which they are stored in memory.
- Why bother? Fast and space-efficient.

Writing 128M doubles:

SciNet file system:

ASCII	173 s
binary	6 s

ramdisk

ASCII	174 s
binary	1 s

- Not human readable.

  But is that really so bad? If you have 100 million numbers in a file, are you going to read them all?

Why you should not use raw binary data

Just dumping the memory is fast, but you lose the information on what it meant. For example:

- Suppose you dump a 2D array of 100x100 floating point numbers
- This gives you a file of 800,000 bytes.
- If you give this to someone else, how will he know what it is? It could be almost anything:
  - a 2D array of 100x100 numbers,
  - an array of 10,000 floating point numbers,
  - a string of 800,000 characters,
  - . . . ?

Obviously we need some metadata to go with the actual information we are trying to save.

Binary Formats

You could invent your own binary format, but it's better to take an existing standard: this saves you potential bugs, the burden of documentation and/or maintaining an I/O library.

Pickle: A Python-specific format. Portable for the same version.

NumPy: Has a binary format called `npy` or `npz`.

NetCDF: A self-describing format: contains not only data but names, descriptions of arrays (`scipy.io.netcdf`).

HDF5: Another standard, self-describing format (`pytables`)
Almost a filesystem in a file.

For both NetCDF and HDF5, there are tools to inspect/analyze the files.
We won't discuss HDF5 here.

Pickle

- Base64 encoding using readable ASCII
- Portable for the same version of python.
- In the `pickle` module.
- Flexible, can `serialize` almost any structure.

```
In [23]: import pickle, os
In [24]: a = zeros((10000,10000))
In [25]: f = open('a.pickle','w')
In [26]: pickle.dump(a,f)
In [27]: close(f)
In [28]: print os.path.getsize('a.pickle')
3200000198
In [29]: g = open('a.pickle','r')
In [30]: b = pickle.load(g)
In [31]: g.close()
```

pickle.dump wall time: 121.44 s

NumPy I/O Routines

- Remembers its shape.
- Straight binary dump of data.
- Surprisingly simple format but not ported too much.
- Just for NumPy arrays, doesn't work for other datatypes.

```
In [32]: import os
In [33]: a = zeros((10000,10000))
In [34]: save('a.npy',a)
In [35]: print os.path.getsize('a.npy')
799997952
In [36]: b = load('a.npy')
```

numpy.save wall time: 1.21 s

Numpy I/O Routines

Numpy has a number of handy functions for performing IO:

`save(FILE, ARRAY)` save a NumPy array to a .npy file

`savez(FILE, NAME1 = ARRAY1, NAME2 = ARRAY2)` save several NumPy arrays to an uncompressed zipped file with extension .npz

`savez_compressed(FILE, NAME1 = ARRAY1, NAME2 = ARRAY2)` save several NumPy arrays to a compressed zipped file with extension .npz

`load(FILE)` load NumPy array(s) from .npy (.npz) file. If FILE is an .npz, a dictionary with keys equal to the names supplied to savez is returned.

NetCDF files

There are three sections to a NetCDF file:

Dimensions How many points in each direction of our multidimensional array?

Variables The data in our multidimensional array

Attributes Variable and other annotations (e.g. units)

Python modules

- scipy.io.netcdf: for netCDF3 files
- netCDF4 (available in Canopy): for netCDF4 files

SciNet compute • calcul CANADA

NetCDF example

Can check the 'header' of a NetCDF file using the Linux utility `ncdump`:

```
ejspence@mycomp>
ejspence@mycomp> ncdump -h test.nc
netcdf test {
dimensions:
    x = 1000 ;
variables:
    double a(x, x) ;
        a:units = "Kelvin" ;
// global attributes:
        :history = "This is a test" ;
}
ejspence@mycomp>
```

Let's see how to create and use this file with scipy.io.netcdf.

scipy.io.netcdf: write file

```
In [37]: from scipy.io.netcdf import *

In [38]: f = netcdf_file('test.nc','w')      #create file

In [39]: f.history = 'This is a test' #set file attribute

In [40]: f.createDimension('x', 1000) #create dimension

In [41]: a = f.createVariable('a', 'd', ('x','x')) #array
In [42]: a[:] = zeros((1000,1000))                  #fill

In [43]: a.units='Kelvin'                 #array attribute

In [44]: f.close()                  #close file. Important!
```

scipy.io.netcdf: read file

```
In [45]: from scipy.io.netcdf import *
In [46]: f = netcdf_file('test.nc','r')
In [47]: print f.history
Created for a test

In [48]: a = f.variables['a']
In [49]: print a[100,300], a.units
0.0 Kelvin

In [50]: f.close()
```

scipy.io.netcdf overview

`HANDLE=netcdf_file(FILENAME,MODE)` Opens a netcdf file. `MODE='w'` for writing, `'r'` for reading, `MODE='rw'` for both.

`HANDLE.ATTRIBUTE=VALUE` Sets a file `ATTRIBUTE` to the value `VALUE`

`HANDLE.createDimension(NAME,VALUE)` Sets the dimension `NAME` (a string) to `VALUE`

`HANDLE.createVariable(NAME,SHAPE)` Creates the variabe `NAME` with `SHAPE` (a tuple of strings that were assigned a value with `createDimension`)

`HANDLE.variables[NAME]` The array variable `NAME`

`HANDLE.variables[NAME].ATTRIBUTE=VALUE` Set an attribute `ATTIBUTE` of the array variable `NAME` to the value `VALUE`

`HANDLE.close()` Flush everything to disk and close the file.

Final Tips

Some tips for optimizing your IOPS:

- If your data is not text, do not save it as text.
- Choose a binary format that is portable, such as NetCDF, HDF5, pickle.
- Minimize IOPS: write/read big chunks at a time; don't `seek` more than needed; try to reuse data or load more into memory.
- Don't create millions of files: it's unworkable and slows down directories.
- Stick to letters, numbers, underscores and periods in file names (no spaces!).