seismic-py: Reading seismic data with Python

Kurt Schwehr

Center for Coastal and Ocean Mapping, University of New Hampshire

Abstract. The field of seismic exploration of the Earth has changed dramatically over the last half a century. The Society of Exploration Geophysicists (SEG) has worked to create standards to store the vast amounts of seismic data in a way that will be portable across computer architectures. However, it has been impossible to predict the needs of the immense range of seismic data acquisition systems. As a result, vendors have had to bend the rules to accommodate the needs of new instruments and experiment types. For low level access to seismic data, there is need for a standard open source library to allow access to a wide range of vendor data files that can handle all of the variations. A new seismic software package, seismic-py, provides an infrastructure for creating and managing drivers for each particular format. Drivers can be derived from one of the known formats and altered to handle any slight variations. Alternatively drivers can be developed from scratch for formats that are very different from any previously defined format. Python has been the key to making driver development easy and efficient to implement. The goal of seismic-py is to be the base system that will power a wide range of experimentation with seismic data and at the same time provide clear documentation for the historical record of seismic data formats.

INTRODUCTION

Seismic data systems use acoustic pulses to send sound waves through water and the solid earth to map layers within the subsurface. They vary from simple single source and single receiver systems to multiple sources and long arrays of geophones or hydrophones. The processing of the received sound waves requires a range of data storage and signal analysis techniques. Python can support both the data archival and precessing tasks.

In this paper, I will use the example of a single source seismic instrument towed behind a ship (e.g. Figure 1a,b). The device (known as a tow-fish, or just a *fish*) is a 2m long device that emits a pulse of sound energy over a range of frequencies straight down using piezoelectric transducers. The energy travels as waves through the water and bottom material. As the sound velocity of the medium changes, a small portion of the energy is reflected back up towards the fish where it is collected by the receivers and stored for later processing. Each pulse of outgoing energy is referenced to as a *shot* and the resulting returned data are collectively called a *trace*. A GPS on-board the ship records the position and time of the ship for each shot. When traces are combined and georeferenced, a ribbon view is created that is call a seismic line (Figure 1c). The inset in Figure 1c shows the seismic lines combined with the bathymetry to give an overall picture of the ocean bottom. The process of going from shots to a 3D model with interpretation through to a publication can be arduous, especially when there are terabytes data.

There are literally thousands of pieces of code around the world for reading and writing seismic data both in the commercial and academic world. Do we need yet another one? The Society of Exploration Geophysicists (SEG) has worked to provide a number of well thought out standards for seismic data (e.g. SEGY Rev 0) [*Barry et al.*, 1975]. The SEG has continued evaluating the needs of the community and has released an updated format that attempts to accommodate changes in the industry (e.g. SEGY Rev 1) [*Norris and Faichney*, 2002]. However, the reality is that no one software package can read all of the variations on these standard

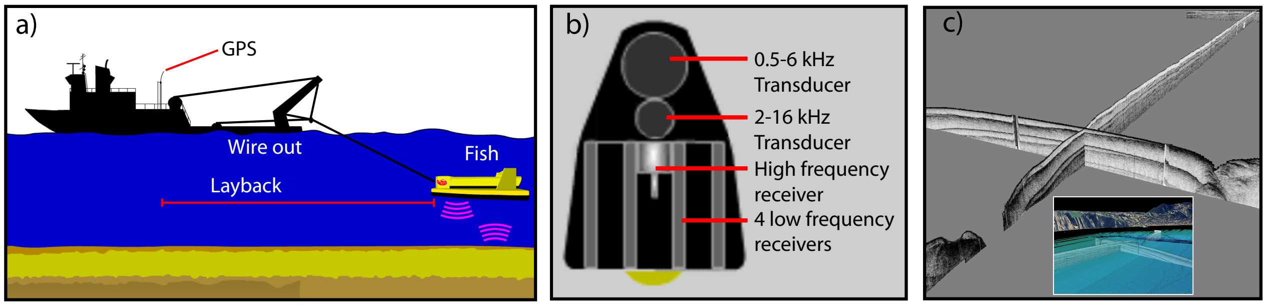

Figure 1. a) Schematic of a ship towing a seismic subbottom profiler (*fish*). Base image courtesy Genevieve Tauxe. b) Underside of an EdgeTech Chirp fish showing the transducers that produce the acoustic *shot* and the sensors that receive the reflected energy, which are stored as a *trace*. Image courtesy Laurent Beguery. c) Traces are shown together as a curtain to show a *seismic line* and are often combined with multibeam bathymetry as visualized by Fledermaus in the inset. Data courtesy Neal Driscoll and Pat Iampietro.

formats and there is no central repository documenting how these formats differ from the SEGY standards. Many seismic processing packages come with tools that allow binary level inspection of data files to attempt to ascertain how a particular SEGY file is structured. Users currently need a range of tools in their arsenal to extract critical data from recorded data streams.

seismic-py is a package designed to alleviate the problems of changing data file formats in the seismic industry. It provides a Python Application Programming Interface (API) for seismic data that relies on a set of drivers that specify and document the actual layout of a particular file format. seismic-py provides this critical functionality in a library released under the GNU General Public License (GPL) [*Stallman*, 1984–], an Open Source Initiative (OSI) [*Raymond*, 1998–] approved license. This means that students and professionals alike have the right to modify and improve the seismic-py package. The source code is available for download here:

`http://schwehr.org/software/seismic-py/`

SEGY FILE LAYOUT

Before proceeding into the details of the software system, it is important to have an understanding of the layout of SEGY Rev 1 data. The format is a Fortran style series of binary data records preceded by a header. The overall layout of SEGY files breaks the content into two major sections. First is a header group starting with a 3200 byte text block (either ASCII or EBCDIC) that is either free form or grouped into 40 predefined 80 character records. Following the text block is a well defined 400 byte binary header region. After this are zero or more Extended Textual File Headers that do not have their format defined in the standard. The rest of the file consists of seismic trace records. These trace records are not required to all be the same size, but they are required to have a 240 byte binary header at the beginning of each trace. Vendors and people processing seismic data frequently create their own format by changing the meaning of these binary fields.

DESIGN

The choice of computer language is the most pivotal design element of a software project. This choice alters which tasks will be hard or easy. For most seismic packages, Fortran and/or C/C++ are the usual choices for implementation. Fortran is the most common language for the geophysical community, but is rather rigid. The C/C++ family of languages provides extreme flexibility, dynamically loadable modules, object oriented design and much more, but at a cost of complexity and frequency of bugs. Python appears to be an excellent compromise between the two groups of languages. Python comes with additional functionality not easily available with either of the other alternatives. Students are able to quickly pickup skills in Python faster than Fortran or C. Python's additional functionality simplified the initial design and implementation of seismic-py.

With the wide range of vendor implementations of SEGY writers, it is critical that SEGY readers be able to easily handle a large number of drivers and allow driver writers to quickly produce the needed changes. With C/C++, this task is possible with dynamically loaded, shared libraries or by parsing specification files,

```
def createDriverName(drvStr):
    '''Make Python filename to load:
    xstar            -> segy_drv_xstar
    drv_xstar        -> segy_drv_xstar
    segy_drv_xstar.py -> segy_drv_xstar
    '''
    if -1 != drvStr.find('.py'):
        drvStr = drvStr[:-3]
    if -1 == drvStr.find('drv_'):
        drvStr = 'segy_drv_'+drvStr
    if -1 == drvStr.find('segy_'):
        drvStr = 'segy_'+drvStr
    return drvStr

def getDriverModule(drvStr='segy_drv_rev1'):
    drivername = createDriverName(drvStr)
    file, pathname, description =
         imp.find_module(drivername)
    drv = imp.load_module(drivername,file,
                          pathname,description)
    return drv
```

Figure 2. The `getDriverModule` function wraps the Python module loaded. By wrapping the standard Python module loader with the `createDriverName` function, seismic-py is able to let the user use shorthand driver names such as "rev1" instead of "`segy_drv_rev1.py`".

but is prone to errors and difficulties with dynamic linkers. Python provides this functionality with the `imp` module [*Python Software Foundation*, 2008b] allowing device loading to be coded in python. The `imp` module provides the components required to create a custom import function in python. seismic-py provides a `getDriverModule` function that wraps the `imp` module allowing the user to specify the driver name in any one of four forms (Figure 2). Python loads a driver module from the users `PYTHONPATH` and returns the `drv` object. All Python code is then able to access the driver data just as it would any other Python module.

A non object-oriented design works just as well and should be more approachable to scientists who may not be familiar with object-oriented design. For most projects of this nature, the obvious choice for a design would be to create a parent class and derive drivers from the parent class or from sibling drivers. With the history of the SEGY formats, a straight inheritance tree would probably be rather difficult. It is expected that as the pool of drivers increases, new drivers will pull pieces

```
1. textFileHeaderEntries (Optional)
2. binaryHeaderEntries
3. extTextFileHeaderEntries (Optional)
4. traceHeader
5. fileHeaderTables
6. traceHeaderTables
7. fileHeaderShortList
8. traceHeaderShortList
```

Figure 3. Lookup tables (dictionaries) for a SEGY file driver. Each table specifies all of the valid field names and byte locations for each field. All of these tables are required except the text and extended text file header entries.

from a wide variety of existing drivers. A true object-oriented design would potentially create a complicated path of multiple inheritances. The driver approach here appears to simplify this problem and allows drivers to reuse pieces from where ever they exist without code duplications. If one were to try to draw the historical relationships of SEGY format variations, it might look something like the attempts to graph Unix system lineages: very complicated and never truly accurate.

Driver Specification File

Each specification driver is simply a Python file with a set of required dictionaries (lookup tables; Figure 3). These lookup tables have a variety of tasks ranging from acting as pointers into binary data to allowing decoding of data elements. Each section dictionary contains byte offset ranges for each data field. The "`segy_drv_rev1.py`" provides the reference driver.

Items 1-4 in Figure 3 provide the core lookup tables. These tables specify the location for each field in the headers. The `binaryHeaderEntries` and `traceHeader` tables together dictate how to decode the data in the traces. The majority of these fields are integers. For those integers that are enumerated values, it is important to be able to create human readable text representations of values. Take the `dataField` dictionary as an example. A '5' means that the data will be a "4-byte IEEE floating-point." The lookup tables provide byte offsets in items 5 and 6 for each field. Figure 4 shows a code example showing how to use the tables contained in a driver.

The short lists (items 7 and 8) are used for programs that wish to show a smaller list of items considered to be the most critical. The short list provides a less overwhelming view of the trace header and are the items

```
>>> import segy
>>> s = segy.Segy('file.sgy')
>>> print s.drv.binaryHeaderEntries
              ['SampleFormat']
    [3225, 3226, 'Data sample format code']
>>> s.header.getBinEntry('SampleFormat')
    1
>>> print s.drv.fileHeaderTables
              ['SampleFormat'][1]
    4-byte IBM floating point
```

Figure 4. The Python command line is a quick way to explore a SEGY file. Once a file is loaded, it is possible to query for the raw values as with getBinEntry or get the English translation by using one of the lookup tables.

```
from segy_drv_rev1 import dataFormats
from segy_drv_rev1 import dataFormatStruct
from segy_drv_rev1 import traceSortingCodes
from segy_drv_rev1 import sweepTypeCodes
```

Figure 5. Reuse of common driver functionality is encouraged. This can also be used to show the heritage of file format. For example, if a driver is essentially SEGY Rev 1 with a few modifications, this will immediately be clear to anyone who reads the driver file.

that the driver author decided are the most important for users to examine. For example, the short list for a trace header might consist of only the shot number (*Shotpoint*), the geographic location of the GPS (X, Y), and the delay from the shot firing to the time the receivers start recording (*Delay*). The standard trace header has an overwhelming 90 items, whereas the short list might have just 4 or 5 entries.

Deriving Variant Specifications

Once a basic driver has been created for a family of SEGY formats, it is easy to create derivative drivers that only modify small portions of an existing driver. The `segy_drv_xstar.py` file provides an example of a derivative driver. The SIO EdgeTech Chirp XStar format is similar to SEGY Rev 1. All of the components that remain the same are directly imported (Figure 5).

Python tries to keep only references to objects when they are used elsewhere within a Python program. For items that need to be changed, it is important to make a completely new and separate local copy of the data. This is done with what Python calls a `deepcopy` [*Python Software Foundation*, 2008a]. Figure 6 is an example with the binaryHeaderEntries. The XStar format does not fill in a number of fields. Missing entries are removed from the local copy after the SEGY Rev 1 entries are deep copied.

```
binaryHeaderEntries = copy.deepcopy (
    segy_drv_rev1.binaryHeaderEntries
    )
del binaryHeaderEntries['JobId']
del binaryHeaderEntries['ReelNo']
del binaryHeaderEntries['TracesPerEnsemble']
```

Figure 6. It is critical to use deepcopy when deriving tables from drivers. This prevents the original driver from being corrupted when altering of deleting entries in a new driver.

Performance

Software performance is critical to seismic processing applications. Seismic instruments are capable of rapidly generating enormous quantities of data. If the code is not able to cope with this volume, users will quickly become frustrated. seismic-py takes the approach of using the mmap system call through the `mmap` Python module [*Python Software Foundation*, 2008c]. This call allows the operating system (OS) to page data into memory on demand via the paging system. Since these pages are marked as read only, the OS can dump pages quickly as memory pressure increases during processing runs. Locations of each name are stored in a Python dictionary (basically hash tables). With the small size of these dictionaries, the lookups proceed quickly. `mmap` brings in raw binary data that cannot be direct read with Python. However, Python provides the `struct` module [*Python Software Foundation*, 2008d] that can convert binary data to Python objects given a conversion string. The `struct` module can convert a range of integer types along with IEEE 32- and 64-bit floating point numbers. Much older seismic data is in IBM floating point format that is not supported by struct, therefore seismic-py can not yet read those seismic data files.

If the speed of the pure Python is not fast enough, it is possible to replace data parsing code with optimized C or C++ code. Originally, this was only possible with the Python/C programming API [*van Rosum*, 2008], but there now exist a wide range of tools for wrapping C++ for using python such as SWIG [*Beazley and Lomdhal*, 1997] or Boost.Python [*Abrahams*, 2002–], or

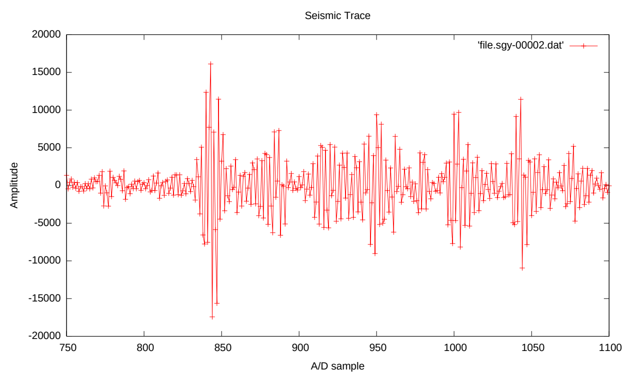

Figure 7. Gnuplot output from plotting the ASCII trace values written to disk by `segydump`. Plotted with "`plot 'file.sgy-00002.dat' with linespoints`".

alternatively using C inline within Python source code [*Mardal and Westlie*, 2007-; *Simpson*, 2001].

SAMPLE APPLICATIONS

To make it easier to get started with seismic-py, the package comes with many sample applications. I will discuss 3 applications to give a flavor of the possibilities. `Segydump` provides a quick look capability similar standard hex viewers, but with an understanding of the header field names. `Seqysql` loads the trace headers into a simple SQL database. `Segysqlgmt` combines the trace locations with a program to draw maps.

Segydump

`Segydump` provides internal listings and trace data dumps for SEGY data files. This is an excellent starting example as it exercises just about all of the functionality in the driver but hides most of it behind the `Segy` class. The most challenging portion of `Segydump` is handling all of the command line options such as being able to include the filename in front of each line of text. Figure 8 is a stripped down version of the dumping code.

By specifying the driver with the `Segy` class, all of the quirks of the XStar format are irrelevant at this level of the interface. The code starts by opening a SEGY data file with the specified driver on Line 3. Line 4 prints out the number of traces in the file. The `printBinaryHeaders` call in line 5 prints out all of the header entries. The user can request that traces be dumped out to disk, which is done in lines 6-8. Line 9 finishes by printing out the header information for each trace. Additional code in `segydump` (not shown here) handles looping over each of the provided files, selecting

```
1 traceNum = 123
2 filename = 'LaJolla-line101.xstar'
3 s = Segy(filename, drivername='xstar')
4 print 'traces = %s' % s.getNumberOfTraces()
5 s.header.printBinaryHeaders()
6 s.writeTraceToFile('%s-%05d.dat' \
7                    % (filename,traceNum),
8                    traceNum)
9 s.getTraceHdr(traceNum).printHeader()
```

Figure 8. This code snippet writes the data from a trace out to an ASCII text data file. This data file is suitable to loading into Octave or plotting with Gnuplot.

short or long output, and providing additional information. It is up to the user to use a tool like grep to pull out specific header fields. Think of `segydump` as the equivalent to the Unix `ls` or DOS `dir` commands.

The ability to write out individual traces should make a wide range of studies more convenient. Most processing environments and languages can read in ASCII data that is in (sample number,value) pairs. The simplest case is visual inspection of individual traces as shown in Figure 7, which shows the `gnuplot` results from running "`segydump -w -t 2 file.sgy`" followed by gnuplot. This idea can be extended to programs such as `MATLAB` and `IDL/ENVI` where additional signal processing is traditionally can be performed.

Segysql

One of the most common tasks of working with seismic data is trying to manage all of the metadata for SEGY files. Python provides a rich set of modules that simplify many of these tasks. Users often want to know which lines cross through a region or the shot closest to a feature, core, or station. The strategy most frequently used in the academic world is to create a wide range of text columns managed with `awk`, `sed`, and Perl scripts. SQL provides an easier way to query large data sets. The problem is that there has been no easy way to import the header data for files and traces into an SQL database. `Segysql` provides a complete example of database importing using the SQLite [*Wyrick and Hipp*, 2000–] database. SQLite was solely for its ease of use. There is no need to setup a database server. As of version 2.5, Python has a SQLite database interface called `sqlite3`, that simply uses a single file as the database repository. Older versions of Python can use pysqlite [*Owens and Haering*, 2001–]. Switching to any

```
01 import segy
02 import sqlite3
03 cx = sqlite3.connect('segy.db3')
04 cu = cx.cursor()
05 cu.execute(segy.sqlCreateFileTable('xstar'))
06 cu.execute(segy.sqlCreateTraceTable('xstar'))
07 cx.commit()
08 xstar = segy.Segy(filename,'xstar')
09 cu.execute(xstar.header.sqlInsert(filename))
10 cx.commit()
11 cu.execute('SELECT fileKey FROM segyFile WHERE filename=:1;',(filename,))
12 fileKey = cu.fetchone()['fileKey']
13 for i in range(1,xstar.getNumberOfTraces()+1):
14     cu.execute(xstar.getTraceHdr(i).sqlInsert(traceNumber=i,fileKey=fileKey))
15 cx.commit()
```

Figure 9. seismic-py provides helper mechanisms to simplify SQL database creation that can easily be combined with SQLite.

other database interface requires changing only a few lines. The seismic-py Python API provides methods that return the necessary SQL string for table creation and row insertion. Just pass these string into a new database interface.

Figure 9 demonstrates a stripped down version of code to create and fill an SQL database from a set of SEGY files. Lines 3-7 connect to a new database file and create the database tables. Once, the database has been created, the first task is to add a file to the database (Line 09). To insert all of the trace headers into the headers, first we have to get (with an SQL `SELECT`) the reference key created by the database for the file (Line 9-10). Finally, each trace is added in a loop over all of the traces (Line 13-14). The `commit` calls are part of the SQL database interface transaction handling. Nothing is actually added to the database until the `commit` call.

Segysqlgmt

Once header information is in a database, it becomes much easier to create mini-applications that add to the seismic processor's tool chest. Marine scientists typically use GMT [*Wessel and Smith*, 2006] and MBSystem [*Caress and Chayes*, 2001–] to make maps of areas that can incorporate other critical data. `segysqlgmt`, a program to display the tracks of seismic lines on a map, illustrates this concept. `mbm_grdplot`, a script with in MB System, reads a GMT `grd` and then outputs a default plotting script using GMT commands, providing a simple basemap. For example, with a Santa Barbara Basin, CA mutlibeam data set [*Hatcher and Maher*, 1999], the command is "`mbm_grdplot -Igmt.grd`." This is much easier that starting off writing your own GMT script. `Segysqlgmt` can then provide text format data files for the ship tracks and shot counts at intervals in a format suitable for GMT's `psxy` and `pstext` along with the shell script lines to add to the `mbm_grdplot` original script. `segysqlgmt` creates quick look shot plots for surveys on top of that base map that can be seen in Figure 10.

FUTURE DIRECTIONS

There is still much work to be done on seismic-py. This paper describes only the initial work done by one developer. seismic-py takes a different approach to seismic data processing compated to other academic packages such as sioseis [*Henkart*, 1975]), Seismic Unix [*Stockwell*, 1997], or pltsegy [*Harding*, 2005] by providing stand alone base level drivers. An open source contribution to the seismic community will hopefully spur more research into seismic data processing, visualization, and interpretation that will give the geoscience community new views into our rocky planets.

To date, seismic-py only implements the SEGY Rev 1 and the EdgeTech XStar format, but it holds promise for providing a vast range of data formats. Critical missing features include full handling of ASCII/EBCDIC headers, extended text headers, IBM floating point data, definition of non-integer header values, and many more vendors' formats. All of these are not hard to provide and are just a matter of additional developer

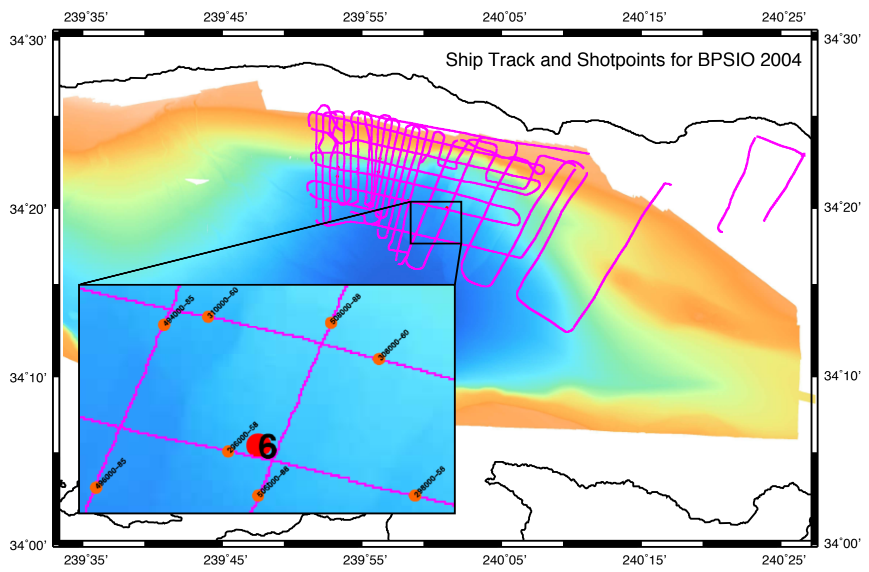

Figure 10. Ship tracks and shot points for a Santa Barbara Basin chirp cruise in 2004. The map was created with a combination of MB-System, GMT, and seismic-py. By drawing cores, ship tracks, and shot numbers, analysts can quickly find the relevant data.

time. Python has proved to be an ideal language for handling formats like this that are Fortran style binary data records. The built-in dictionary and list data types make the driver files appear very close to the original text specification documents.

The `segy` class interface needs a few additions. The most critical to making seismic-py more "Pythonic" is to add an iterator interface, such that a for loop on a `segy` object will loop over the traces. Additionally, the initial database interface only supports traditional SQL calls. Future versions need to add support for spatial databases such as PostGIS [*Refractions Research*, 2008] and [*Furieri*, 2008]

The initial work on seismic-py used Python dictionaries to define the SEGY file format and for the files variances produced by each instrument. This is effective for initial prototypes, but for larger impact on the community, future projects should use eXtensible Markup Language (XML) configuration files. XML allows software implementers to choose their language of preference (assuming that it has an XML reading library) or a compiler could be generated that emits source code for any particular programming language. A compiled version (resulting in Python code) could be made faster without the current run-time table lookups.

CONCLUSION

seismic-py provides a reference implementation of an interface to the wide variety of seismic data that end users encounter in processing seismic data. seismic-py will provide the basis for the rapid development of new tools for inspecting and processing seismic data. seismic-py removes the restriction of using one type of seismic data at a time. Users can open many different seismic streams at the same time with each stream utilizing a different driver matching each data file type.

Today, many students in geophysics and geology learn programming on non-geoscience type problems. Tools like seismic-py will allow beginning students learn the computer languages such as Python while working with data sets that are exciting and cutting edge. Why not have students start by accessing and viewing seismic data when they are learning to program if they are studying geophysics? The hope is to turn seismic-py into one component of an introduction to scientific computing class. The class can use packages such as SciPy [*Jones et al.*, 2001–], ScientificPython[*Hinsen*, 1999–], the pygsl interface to Gnu Scientific Library [*Gaedke and Schnizer*, 2001–]), and the Python Imaging Library [*PythonWare*, 2006] to dive right into processing real data while learning data structures and algorithms. The various tasks can then be combined to create figures suitable for scientific publications (e.g. Figure 1c).

Perhaps the most important side effect of seismic-py is the beginning of a library that documents seismic formats. There is a vast wealth of commercial and academic seismic data already collected to date. By keeping these older data sets readable, new experiments studying change in earth structures become more manageable. Huge amounts of money have already been spent collecting seismic data and it is important to simplify access to the valuable resource.

Acknowledgments

I would like to thank the reviewers and many other people who read drafts of this paper for their valuable input. Lisa Tauxe and Neal Driscoll provided much encouragement. NSF and BP provided funding the data collection that led to this software project.

References

Abrahams, D., Boost.Python, *http://www.boost.org/doc/libs/release/libs/python/doc/*, 2002–.

Barry, K. M., D. A. Cavers, and C. W. Kneale, Report on recommended standards for digital tape formats, *Geophysics*, *40*, 344–352, 1975.

Beazley, D. M., and P. S. Lomdhal, Feeding a Large–scale Physics Application to Python, *International Python Conference*, *6*, http://www.swig.org/papers/Py97/beazley.html, 1997.

Caress, D., and D. Chayes, MB-System Version 5, Open source software distributed from the MBARI and L-DEO web sites, 12 beta releases, *http://www.ldeo.columbia.edu/res/pi/MB-System/MB-System.intro.html*, 2001–.

Furieri, A., SpatiaLite - VirtualShape, *http://www.gaia-gis.it/spatialite-2.0/index.html*, 2008.

Gaedke, A., and P. Schnizer, PyGSL: Python interface for GNU Scientific Library, *http://pygsl.sourceforge.net/*, 2001–.

Harding, A., pltsegy, 2005.

Hatcher, G., and N. Maher, MBARI Santa Barbara Basin Multibeam Survey, *http://www.mbari.org/data/mapping/SBBasin/default.htm*, 1999.

Henkart, P., Sioseis, *http://sioseis.ucsd.edu/*, 1975.

Hinsen, K., ScientificPython, *http://starship.python.net/~hinsen/ScientificPython/*, 1999–.

Jones, E., T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, *http://www.scipy.org/*, 2001–.

Mardal, K., and M. Westlie, Instant, *http://www.fenics.org/wiki/Instant*, 2007-.

Norris, M. W., and A. K. Faichney, SEG Y rev 1 Data Exchange format, *http://seg.org/publications/tech-stand/*, 2002.

Owens, M., and G. Haering, pysqlite - A DB API v2.0 compatible interface to SQLite, *http://initd.org/tracker/pysqlite*, 2001–.

Python Software Foundation, copy – Shallow and deep copy operations, *http://docs.python.org/lib/module-copy.html*, 2008a.

Python Software Foundation, imp – Access the import internals, *http://docs.python.org/lib/module-imp.html*, 2008b.

Python Software Foundation, mmap – Memory-mapped file support, *http://docs.python.org/lib/module-mmap.html*, 2008c.

Python Software Foundation, struct – Interpret strings as packed binary data, *http://docs.python.org/lib/module-struct.html*, 2008d.

PythonWare, Python Imaging Library (PIL), *http://www.pythonware.com/products/pil/*, 2006.

Raymond, E. S., The Open Source Initiative, *http://www.opensource.org/*, 1998–.

Refractions Research, PostGIS, *http://postgis.refractions.net/*, 2008.

Simpson, K., PyInline, *http://www.fenics.org/wiki/Instant*, 2001.

Stallman, R., The GNU General Public License, *http://www.gnu.org/licenses/licenses.html*, 1984–.

Stockwell, J. W., Free Software in Education: A case study of CWP/SU: Seismic Un*x, *The Leading Edge*, 1997.

van Rosum, G., Python/C API Reference, *http://docs.python.org/api/api.html*, 2008.

Wessel, P., and W. H. F. Smith, Generic Mapping Tools, *http://gmt.soest.hawaii.edu/*, 2006.

Wyrick, G., and R. Hipp, SQLite, *http://www.sqlite.org/*, 2000–.

K. Schwehr, Center for Coastal and Ocean Mapping, University of New Hampshire, Chase Ocean Engineering 24 Colovos Rd, Durham, NH 03824, schwehr@ccom.unh.edu, http://schwehr.org