Seismological data analysis using Python

GPHS445

May 12, 2016

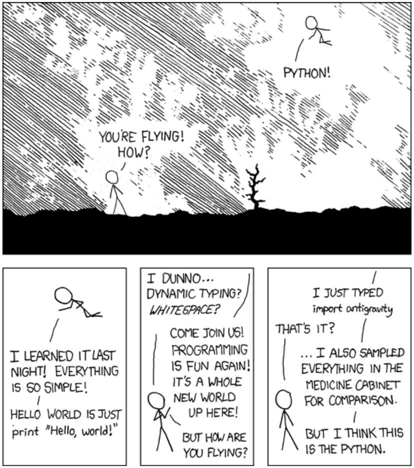

PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

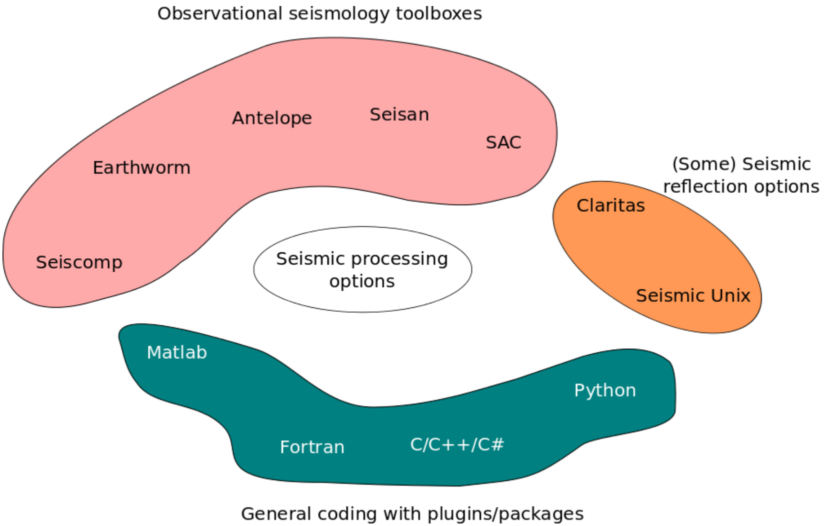
Figure: xkcd comic 353: http://www.xkcd.com/353/

Outline

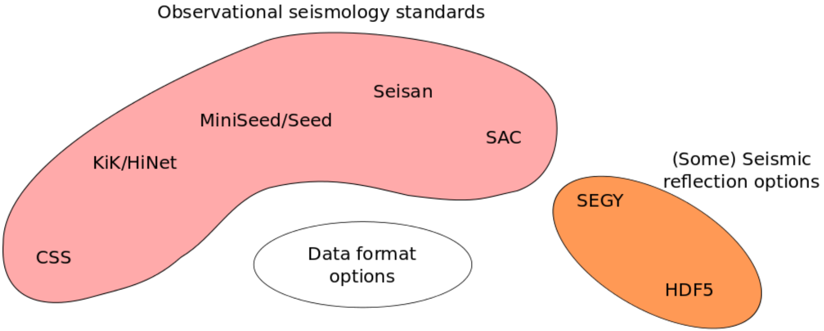
Why Python?

1. Focus in readability of code (low entry level, good for scientists);
2. Excellent code-base, easily accessed via PyPI (Python Package Index);
3. Can include other, faster code into Python with minimal effort;
4. Can do anything from making websites (Django), Data-mining, making creative things (Disney), and many more;
5. Promotes functional programming and re-usability;
6. Great tools for debugging, testing, distributing and contributing to projects.

Other options


Observational seismology toolboxes
Antelope
Seisan
SAC
Earthworm
Seiscomp
Seismic processing options
(Some) Seismic reflection options
Claritas
Seismic Unix
Matlab
Python
Fortran
C/C++/C#
General coding with plugins/packages

Data formats


Observational seismology standards
Seisan
MiniSeed/Seed
SAC
KiK/HiNet
CSS
Data format options
(Some) Seismic reflection options
SEGY
HDF5

Python IDEs

IDE: Interactive Development Environment (think Matlab GUI, but Python, so better).

Spyder is a good one to try.
Run scripts and test codes in iPython (a nice interactive Python shell).

Basic data types

- int (integer, e.g. 0)
- float (floating-point number, e.g. 0.123, 12.634, 1.0e10,...)
- bool (boolean, either True or False, note bool(0)==False, bool(1)==True)
- str (string, e.g. 'Ab_1')
- unicode (unicode - represents more characters than string, default for Python 3.x, e.g. u'Spam')

Basic data types cont.

- tuple (tuple, e.g. (1, 2, 3, 'Spam') - note, items in tuple can be of mixed type)
- list (list of same types, e.g. [1, 2, 3], note a ‘word’ is a list of individual strings, ‘Spam’.split() == [‘S’, ‘p’, ‘a’, ‘m’])
- dict (like a tuple but with keys, e.g. test_dict={'bob': 0, 'malcolm': 'help'}, then test_dict['bob'] == 0)

Note, there are others, but these are the main ones you will encounter.

Looping

Looping in Python is simple and runs on iterators or lists.

```
for i in range(10):
  print(i * i)
# Is the same as...
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
  print(i * i)
# Which is generalised to...
for item in list:
  print(item)
```

Iterators are better (reduced memory consumption), and range in python 3.x produces an iterator rather than a list.

Conditions

Conditions allow you to do different things based on a variable, e.g.:

```
bob = 3
# Obviously this is trivial, we will get on
# to more responses next slide.
if bob == 3:
  print('bob is 3!')
elif bob == 'albatross':
  print('bob is an albatross?!')
else:
  print('bob is a dull being')
```

Conditions

More conditions, *if* checks whether something is **True**.

```
if bob != albatross
# is the same as
if not bob == albatross
# You can also check is something is in a list
if bob in list
# Or if something is not in a list
if bob not in list
```

Looping with conditions

Combining loops and conditions is useful...

```
for name in class_list:
  if name == 'bob':
    return 'bob is present'
# Note that this loop is rubbish, you should do:
if 'bob' in list:
  return 'bob is present'
```

List comprehension and generators

List comprehension can be done in-line, or to create generator functions:

```python
list_squares = [i*i for i in range(10)]
# This generate a list, which is the same as:
list_squares = []  # Empty list
for i in range:
  list_squares.append(i * i)
# This next one is a true generator expression:
squares_generator = (i*i for i in range(10))
# This returns an iterator.
```

In-place variable math

In Python you can use C-style variable overwriting with math functions:

```
# Rather than:
i = i + 1
# You can do:
i += 1
# Similar operations include:
i /= 10
i *= 10
i -= 2
```

Note that floating point math and int math are handled differently in Python 2.x and 3.x.

Importing modules

Possibly the best thing about Python is that lots of things are already written and easy to find.
Installing modules is usually done by *pip*, *easy_install* or *conda*.
To use the modules you have to import them. One of the most useful modules is *numpy*:

```
import numpy as np
# Now you can use all the amazing things in numpy!
random_vector = np.random.randn(1000)
# type(random_vector) == numpy.ndarray
random_sum = np.sum(random_vector)
# This is > 10x faster than Pythons in-built sum method
    !
```

Functions

Functional programming is useful for many reasons, one very good thing is to break programs down into useful, clear sections - read http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745.

```python
def add_integers(a, b):
  """
  Simple function to add two integers.

  :type a: int
  :param a: first integer to be summed
  :type b: int
  :param b: integer to add to a

  :returns: int, Sum of a and b
  """
  c = a + b
  return c
```

Dictionaries

A dictionary in Python is essentially a series of keys with associated values.
Dictionaries are encased in curly {} braces, and keys are separated from their variables by a :
Example:

```
test_dict = {'Name': 'Monty', 'Profession': 'Gymnast',
             'Numbers': [0,2,6,3]}
```

Objects

Python is an object-oriented language. You can make your own objects, as-well as use in-built objects, or objects from other modules.
Objects can contain both variables (e.g. st.data is the data within a stream object in ObsPy), and methods (e.g. st.resample(20) is an in-place method for resampling data in a stream object).
Attributes of objects can be accessed using the *class.attribute* syntax.

Classes

Classes are user-defined prototypes of objects. These prototypes define what the object can be made up of. For example:

```
class Things:
  """Class to define a few handy things"""

  def __init__(self, name, profession, numbers):
    self.name = name
    self.profession = profession
    self.numbers = numbers

  def sum(self):
    return sum(self.numbers)
```

NumPy

In NumPy's own words:
NumPy is the fundamental package for scientific computing with Python.
Almost everything in Python uses NumPy, you will too!
NumPy is mostly written in C, Cython and Python. C and Cython are compiled languages that allow large speed-ups: NumPy passes some of these speed-ups to Python with a simple and intuitive syntax.

Useful types

NumPy comes with all sorts of extra data types based more closely on c-types to allow for data transfer between Python and C. These types include:

- float16, float32 and np.float64
- intc, intp, int8, int16, tnt32 and np.int64
- complex64 and complex128
- unint8, uint16, uint32, uint64.

Why care?
Seismic data are stored as int32 in miniseed STEIM2 files - if you want to write out seismic data to this format (which is the most common archive format) then you need to have a basic idea of types...
I care because of memory consumption, precision and truncation errors.

Plotting

If you like Matlab style plotting you are in for a treat!
Python library *Matplotlib* is designed to have a Matlab style syntax for plotting in Python.
Matplotlib works really well for plotting ‘standard’ data-sets, and makes (IMO) prettier plots than Matlab.

`http://matplotlib.org/gallery.html`

ObsPy

ObsPy is pretty much THE package for handling seismic data in Python. We will focus on this and run through some of the tutorials here: `https://docs.obspy.org/tutorial/` in the lab tomorrow.
Essentially, if there is something you want to do with seismic data, there is probably a function for it in ObsPy, and it's probably very simple to use and well documented.

IO

ObsPy has read/write support for a lot of waveform types, and, if it's not there but you know how to read it (this is unlikely) then you can easily create a way to read it in using the ObsPy *Stream* and *Trace* objects.
https://docs.obspy.org/packages/autogen/obspy.core.stream.read.html#obspy.core.stream.read

IO cont.

Reading seismic data is simple and ObsPy will work out what format your data are in. Data will be read in to a Stream object, which is essentially a list of Trace objects. Each Trace contains the waveform data as a NumPy array, and a series of stats (AttribDict class, wrapper for dictionary) containing the header information.

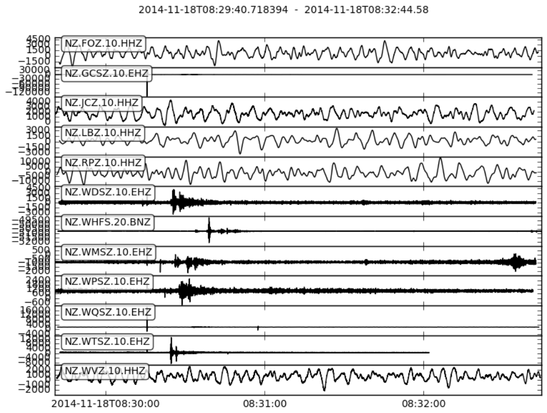
```python
from obspy import read

st = read('test_data.ms')
print(st.stats.station)
# Might return 'COSA' or some other station name.
```

Plotting

Simple waveform plotting is handled in place:

```python
st.plot()
```

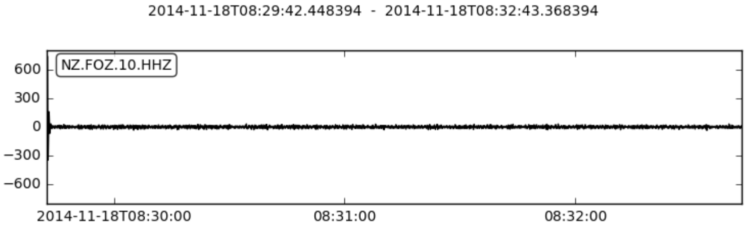


Filtering

In the previous image you can see the earthquake on some channels, but not on others. The obvious ones are short-period (4.5 Hz) sensors, all the others are broadband and include long-period noise.

Filtering is an in-built function for the Stream and Trace classes:

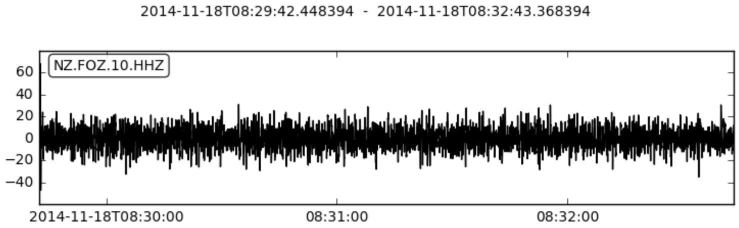
```
tr = st.select(station='FOZ', channel='*Z').copy()
tr.filter('bandpass', freqmin=2.0, freqmax=10.0)
tr,plot()
```



Filtering cont.

The data do not look good! You should always take care to de-trend your data before filtering, steps (spikes, delta functions) in the time-domain contain all frequencies, so are not stable in the frequency domain.

```
tr = st.select(station='FOZ', channel='*Z').copy()
tr.detrend('simple').filter('bandpass', freqmin=2.0,
    freqmax=10.0)
tr.plot()
```



Triggers

ObsPy has multiple standard event detection routines built in osbpy.signal.trigger, and these can be applied directly to Stream or Trace objects.

- recursive sta/lta;
- carl sta routine;
- classic sta/lta;
- delayed sta/lta;
- z-detector;
- Baer picker routine;
- Auto-regressive picker.

Also a network coincidence trigger, we will use these in the assignment.

Meta-data

Meta-data is everything other than seismic data.
Handled by obspy.core.event (event data) and obspy.core.inventory (station data).
Inventories includes response information, location info etc.
Events include (almost) all the information you could want about an earthquakes or explosion.
Based on QuakeML, a widely used meta-data format.
Lots of useful functions on the objects such as plotting!

UTCDateTime

UTCDateTime is ObsPy's wrapper on Python's native datetime.datetime object.

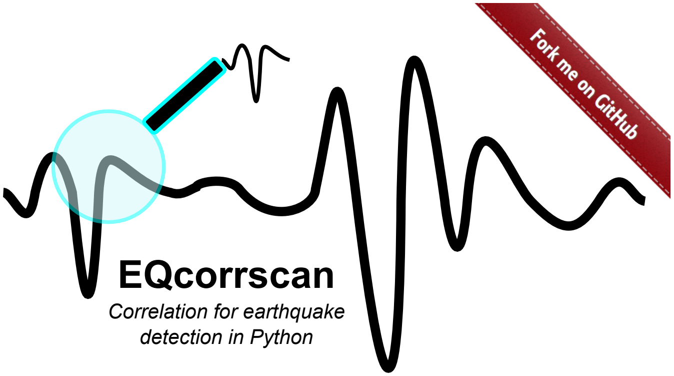
Necessary to cope with standard seismic date and time conventions, and with high-precision timing needed for seismic data.

Can easily convert between the two, and convert strings and numbers into UTCDateTime and datetime objects.

Can easily extract parts of the UTCDateTime object.

EQcorrscan

Matched-filter earthquake detection and processing.



A reasonable example of how easy it is to write useful things in Python, document them, and distribute them.
What it does:

- Generate templates from known events;
- Generate synthetic templates from un-known events;
- Use a source-scanning algorithm to detect and locate events;
- Multi-paralleled match-filter detection using template events;
- Automated local magnitude calculations;
- Singular-value decomposition magnitude calculations;
- Continues. . .

- Lots of neat plotting routines;
- Integrates with lots of other software via ObsPy and in-built hypoDD and Seisan IO (soon to be put into ObsPy);
- Linear and phase-weighted stacking;
- Pick corrected of near-repeating events (which can then be used for high-precision locations);
- Clustering of known events by location and by waveform similarity.

Building a set of tests, currently only testing 32% of codebase.
Multi-platform (Windows, OSX and Linux), tested via CI on travis and appveyor.
Correlations computed by fast C++ code (OpenCV).
Easily deployed on cluster computers, achieves up to 250x speed-ups over other implementations (Matlab).

Summary

1. Python is:
   - Simple to learn;
   - Extensible;
   - Not the fastest around in pure form, but see above;
   - Simple to document;
   - Simple to read (if you follow pep8 rules);
   - Simple to distribute, shared code is happy code.
2. Most things that you want to do have already been done (pip, github, bitbucket, etc.);
3. **Obspy is better than anything anyone has ever written in any other language for handling seismic data.**

Multprocessing

Python multiprocessing is simple at a low-level, but difficult to achieve large speed-ups in specific cases.
The GIL (Global Interpreter Lock) prevents memory over-writes, which is useful, but not good for accessing shared memory.
Python 3.x is better than Python 2.7, but not everything is written for 3.x (case-in-point, EQcorrscan isn't there yet).

Virtual environments

Use them!
Virtual environments mean you can install whatever packages you like, using whatever version of Python you like, and you won't (not likely to) break your computer! Yay!
Conda and virualenvwrapper are both useful for managing virtual environments, I like using conda (Anaconda) because lots of packages are pre-built. Install is streamlined, but might not be the fastest versions...

Useful idioms

```
a = [0,2,4,6]
b = ['cabbage', 'albatross', 'sausage', 'bob']
for i in range(len(a)):
    print(a[i])
    print(b[i])
# Better:
for i, number in enumerate(a):
    print(number)
    print(b[i])
# Even better:
for number, word in zip(a, b):
    print(number)
    print(word)
```

```python
# One-line variable defs
a, b, c = 'a', 1, [0, 1, 2]
# String 'addition'
color = 'y'
color += 'e'
color += 'llow'
# Joining lists of strings
sentence = ' & '.join(['bob', 'monty'])
# And the reverse
names = sentence.split(' & ')
```

```
# Appending items to lists
test_list = ['a', 'b', 'c']
test_list.append('d')
# Sort a list:
test_list.sort()
# Read data from text-file
f = open('file.txt')
for line in f:
    print(line)
```