First Steps in ObsPy

ObsPy Workshop at the MESS 2013

Lion Krischer

Ludwig-Maximilians-University in Munich
Department of Earth and Environmental Sciences
Geophysics

Sudelfeld, March 12 2013

Rough Schedule

Morning: Introduction to ObsPy

Afternoon: Advanced ObsPy Exercise

What is ObsPy and What Can it Do?

A Python toolbox for seismology/seismological observatories.

The goal of the ObsPy project is to **facilitate rapid application development for seismology**. Nowadays it is also used as a tool for interactive data discovery and analysis.

http://www.obspy.org

What is ObsPy and What Can it Do?

- **Read and write waveform data in various formats** (MiniSEED, SAC, GSE, SEG Y, ...) with a unified interface.
- **Database and webservice access clients** for NERIES, IRIS DMC, ArcLink, SeisHub, SeedLink, and Earthworm (experimental).
- **Many seismological signal processing routines** like filters, trigger, instrument correction, array analysis, beamforming, ...
- **Support for inventory data** (SEED, XSEED, RESP and planned StationXML support)
- **Event data handling** (QuakeML)
- **Simple visualization routines** (Waveforms, Spectrograms, Beachballs, ...)
- **Experimental real-time support**
- **Utility functionality** (global arrival times, geodetic functions, ...)

$+$

The full power and flexibility of Python.

What is ObsPy and What Can it Do?

- In development since 2008
- 6 core developers
- Many more people contribute
- Thoroughly unit tested
- Written in Python (performance critical parts are written in C)
- Uses well established libraries (libmseed, GSE_UTI, ...)
- **Open source and cross platform**
- Starts to get widely used in the community

www.obspy.org

- Documentation and extensive tutorial.
- Gallery to showcase some features.
- **mailing list** - subscribe for updates and discussions about the project.
- Source code repository and bug tracker.
- Automatic, daily running tests bots (http://tests.obspy.org)
- Get in touch!

- Moritz Beyreuther et al. (2010) **ObsPy: A Python Toolbox for Seismology**, SRL, 81(3), 530-533
- Tobias Megies et al. (2011) **ObsPy – What can it do for data centers and observatories?** Annals Of Geophysics, 54(1), 47-58. doi:10.4401/ag-4838

Goal: Familiarize Yourself With ObsPy's Main Objects and Functions

obspy.core

This central module is the glue between all other ObsPy modules.

- Unified interface and functionality for handling waveform data in form of the **Stream** and **Trace** classes.
- All absolute time values within ObsPy are consistently handled with the **UTCDateTime** class.
- Event data is handled with the **Event** class.
- Generally useful utility classes and functions like the **AttribDict** class.
- Management via plugin discovery and binding, a global test script, . . .

Handling Time - The UTCDateTime Class

- All absolute time values are consistently handled with this class.
- No need to worry about timezones.
- Based on a high precision POSIX timestamp and not the Python datetime class because precision was an issue.

Features of UTCDateTime

- Initialization

```
>>> from obspy import UTCDateTime
>>> UTCDateTime("2012-09-07T12:15:00")
UTCDateTime(2012, 9, 7, 12, 15)
>>> UTCDateTime(2012, 9, 7, 12, 15, 0)
UTCDateTime(2012, 9, 7, 12, 15)
>>> UTCDateTime(1347020100.0)
UTCDateTime(2012, 9, 7, 12, 15)
```

- Time zone support

```
>>> UTCDateTime("2012-09-07T12:15:00+02:00")
UTCDateTime(2012, 9, 7, 10, 15)
```

Features of UTCDateTime

- Attribute access

```
>>> time = UTCDateTime("2012-09-07T12:15:00")
>>> time.year
2012
>>> time.julday
251
>>> time.timestamp
1347020100.0
>>> time.weekday
4
```

Features of UTCDateTime

- Handling time differences

```
>>> time = UTCDateTime("2012-09-07T12:15:00")
>>> print time + 3600
2012-09-07T13:15:00.000000Z
>>> time2 = UTCDateTime(2012, 1, 1)
>>> print time - time2
21644100.0
```

UTCDateTime - Exercises

1. Calculate the number of days passed since your birth.
    - The current date and time can be obtained with **“UTCDateTime()”**
    - Optional: Include the correct time zone
2. Get a list of 10 UTCDateTime objects, starting yesterday at 10:00 with a spacing of 90 minutes.

Handling Waveform Data

```
>>> from obspy import read
>>> st = read("waveform.mseed")
>>> print st
1 Trace(s) in Stream:
BW.FURT..EHZ | 2010-01-04... | 200.0 Hz, 7204234 samples
```

- Automatic file format detection.
- Always results in a Stream object.
- Raw data available as a numpy.ndarray.

```
>>> st[0].data
array([-426, -400, ... , -489, -339], dtype=int32)
```

The Stream Object

- A **Stream** object is a collection of **Trace** objects

```
>>> from obspy import read
>>> st = read()
>>> type(st)
obspy.core.stream.Stream
>>> print st
3 Trace(s) in Stream:
BW.RJOB..EHZ | 2009-08-24T00: ... | 100.0 Hz, 3000 samples
BW.RJOB..EHN | 2009-08-24T00: ... | 100.0 Hz, 3000 samples
BW.RJOB..EHE | 2009-08-24T00: ... | 100.0 Hz, 3000 samples
>>> st.traces
[<obspy.core.trace.Trace at 0x1017c8390>, ...]
>>> print st[0]
BW.RJOB..EHZ | 2009-08-24T00: ... | 100.0 Hz, 3000 samples
>>> type(st[0])
obspy.core.trace.Trace
```

The Trace Object

- A **Trace** object is a single, continuous waveform data block
- It furthermore contains a limited amount of metadata

```
>>> tr = st[0]
>>> print tr
BW.RJOB..EHZ | 2009-08-24T00: ... | 100.0 Hz, 3000 samples
>>> print tr.stats
         network: BW
         station: RJOB
        location:
         channel: EHZ
       starttime: 2009-08-24T00:20:03.000000Z
         endtime: 2009-08-24T00:20:32.990000Z
   sampling_rate: 100.0
           delta: 0.01
            npts: 3000
           calib: 1.0
```

The Trace Object

- For custom applications it is often necessary to directly manipulate the metadata of a Trace.
- The state of the Trace will stay consistent, as all values are derived from the starttime, the data and the sampling rate and are updated automatically.

```
>>> print tr.stats.delta, "|", tr.stats.endtime
0.02 | 2009-08-24T00:20:27.980000Z
>>> tr.stats.sampling_rate = 5.0
>>> print tr.stats.delta, "|", tr.stats.endtime
0.2 | 2009-08-24T00:23:27.800000Z
>>> print tr.stats.npts
3000
>>> tr.data = tr.data[:100]
>>> print tr.stats.npts, "|", tr.stats.endtime
100 | 2009-08-24T00:20:27.800000Z
```

The Trace Object

- Working with them is easy, with a lot of attached methods.

```
>>> print tr
BW.RJOB..EHZ | 2009-08-24T00: ... | 100.0 Hz, 3000 samples
>>> tr.resample(sampling_rate=50.0)
>>> print tr
BW.RJOB..EHZ | 2009-08-24T00: ... | 50.0 Hz, 1500 samples
>>> tr.trim(tr.stats.starttime + 5, tr.stats.endtime - 5)
>>> print tr
BW.RJOB..EHZ | 2009-08-24T00: ... | 50.0 Hz, 500 samples
>>> tr.detrend("linear")
>>> tr.filter("highpass", freq=2.0)
```

Stream Methods

- Most methods that work on a **Trace** object also work on a **Stream** object. They are simply executed for every trace.
  - **st.filter()** - Filter all attached traces.
  - **st.trim()** - Cut all traces.
  - **st.resample() / st.decimate()** - Change the sampling rate.
  - **st.trigger()** - Run triggering algorithms.
  - **st.plot() / st.spectrogram()** - Visualize the data.
  - **st.simulate(), st.merge(), st.normalize(), st.detrend(), ...**
- A **Stream** object can also be exported to many formats making ObsPy a good tool for converting between different file formats.

```
>>> st.write("output_file.sac", format="SAC")
```

Waveform Data - Exercises

Later on a useful example application will be developed. For now the goal is to get to know the Stream and Trace classes.

Several possibilies to obtain a Stream object:

- The empty **read()** method will return some example data.
- Passing a filename to the **read()** method.
- Using one of the webservices. This will be dealt with in the next part.
- Passing a URL to **read()**. See e.g. *http://examples.obspy.org* for some files.

Trace Exercise 1

- Make a trace with all zeros (e.g. *numpy.zeros(200)*) and an ideal pulse at the center
- Fill in some station information (*network*, *station*)
- Print trace summary and plot the trace
- Change the sampling rate to 20 Hz
- Change the *starttime* to the start time of this session
- Print the trace summary and plot the trace again

Trace Exercise 2

- Use *tr.filter(...)* and apply a lowpass filter with a corner frequency of 1 second.
- Display the preview plot, there are a few seconds of zeros that we can cut off.
- Use *tr.trim(...)* to remove some of the zeros at start and at the end.

Trace Exercise 3

- Scale up the amplitudes of the trace by a factor of 500
- Make a copy of the original trace
- Add standard normal gaussian noise to the copied trace (use *numpy.random.randn(..)*)
- Change the station name of the copied trace
- Display the preview plot of the new trace

Stream Exercise

- Read the example earthquake data into a stream object (*read()* without arguments)
- Print the stream summary and display the preview plot
- Assign the first trace to a new variable and then remove that trace from the original stream
- Print the summary for the single trace and for the stream
- Plot the spectrogram for the single trace

Waveform Data - Exercises

Some further ideas what you can do now to get a better grasp of the objects:

1. Read some files from different sources and see what happens
2. Have a look at the ObsPy Documentation on the homepage
3. Use IPython's tab completion and help feature to explore objects

obspy.xseed - Station Information

Inventory Data - obspy.xseed

- Can currently read/write/convert between SEED and XML-SEED.
- RESP file support.
- StationXML support is planned.

```
000001V 010009402.3121970,001,00:00:00.0000~2038,001,00:00:00.0000~
2009,037,04:32:41.0000~BayernNetz~~0110032002RJOB 000003RJOB 000008
...
```

⇕

```
<?xml version='1.0' encoding='utf-8'?>
<xseed version="1.0">
  <volume_index_control_header>
    <volume_identifier blockette="010">
      <version_of_format>2.4</version_of_format>
      <logical_record_length>12</logical_record_length>
      <beginning_time>1970-01-01T00:00:00</beginning_time>
      <end_time>2038-01-01T00:00:00</end_time>
...
```

obspy.xseed - Example usage

```
>>> from obspy.xseed import Parser
>>> p = Parser("dataless_SEED")
>>> print p
BW.FURT..EHZ | 2001-01-01T00:00:00.000000Z -
BW.FURT..EHN | 2001-01-01T00:00:00.000000Z -
BW.FURT..EHE | 2001-01-01T00:00:00.000000Z -
>>> p.getCoordinates("BW.FURT..EHZ")
{"elevation": 565.0, "latitude": 48.162899,
 "longitude": 11.2752}
>>> p.getPAZ("BW.FURT..EHZ")
{"digitizer_gain": 1677850.0,
 "gain": 1.0,
 "poles": [(-4.444+4.444j), (-4.444-4.444j), (-1.083+0j)],
 "seismometer_gain": 400.0,
 "sensitivity": 671140000.0,
 "zeros": [0j, 0j, 0j]}
```

obspy.xseed - Example usage

```
>>> p.writeXSEED("dataless.xml")
# Edit it ...
>>> p = Parser("dataless.xml")
>>> p.writeSEED("edit_dataless_SEED")
>>> p.writeRESP(".")
```

obspy.xseed - Exercise

- Read the **BW.FURT..EHZ.D.2010.005** waveform example file.
- Cut out some minutes of interest.
- Read the **dataless.seed.BW_FURT** SEED file.
- Correct the trimmed waveform file with the poles and zeros from the dataless SEED file using *st.simulate()*. This will, according to the SEED convention, correct to m/s.
- (Optional) Read the file again and convert to m by adding an extra zero. Choose a sensible waterlevel.
- (Optional) Convert the SEED file to XSEED, edit some values and convert it back to SEED again. This requires some knowledge of the general SEED file structure.

obspy.core.event - Event Handling

Events

- Aims to get a unified interface with read and write support independent of the data source, similar to how the Stream and Trace classes handle waveform data.
- Fully supports QuakeML 1.2 and is modelled after it

```
>>> from obspy import readEvents
>>> url = "http://www.seismicportal.eu/services/..."
>>> catalog = readEvents(url)
>>> print catalog
99 Event(s) in Catalog:
2012-04-11T10:43:09.400000Z |  ... | 8.2 Mw | ...
2012-04-11T08:38:33.000000Z |  ... | 8.4 M  | ...
...
```

Events - Basic Structure

- The **readEvents()** function always returns a **Catalog** object, which is a collection of **Event** objects.

```
>>> from obspy import readEvents
>>> cat = readEvents()
>>> type(cat)
obspy.core.event.Catalog
>>> type(cat[0])
obspy.core.event.Event
```

Events - Basic Structure

```
>>> event = cat[0]
>>> print event
Event:  2012-04-04T14:...| +41.818,  +79.689 | 4.4 mb

            resource_id: ResourceIdentifier(...)
             event_type: "not reported"
          creation_info: CreationInfo
             agency_uri: ResourceIdentifier(...)
             author_uri: ResourceIdentifier(...)
          creation_time: UTCDateTime(2012, 4, 4, 16, 40, 50)
                version: "1.0.1"
        ---------
                origins: 1 Elements
             magnitudes: 1 Elements
```

Events - Basic Structure

- **Event** objects are again collections of other resources.

```
>>> type(event.origins[0])
obspy.core.event.Origin
>>> type(event.magnitudes[0])
obspy.core.event.Magnitude
>>> print event.origins[0]
Origin
         resource_id: ResourceIdentifier(...)
                time: UTCDateTime(...)
            latitude: 41.818
           longitude: 79.689
               depth: 1.0
          depth_type: "from location"
           method_id: ResourceIdentifier(...)
  used_station_count: 16
       azimuthal_gap: 231.0
       ...
```

The Catalog object

- The Catalog object contains some convenience methods to make working with events easier.
- Events can be filtered with various keys.

```
>>> small_magnitude_events = cat.filter("magnitude <= 4.0")
```

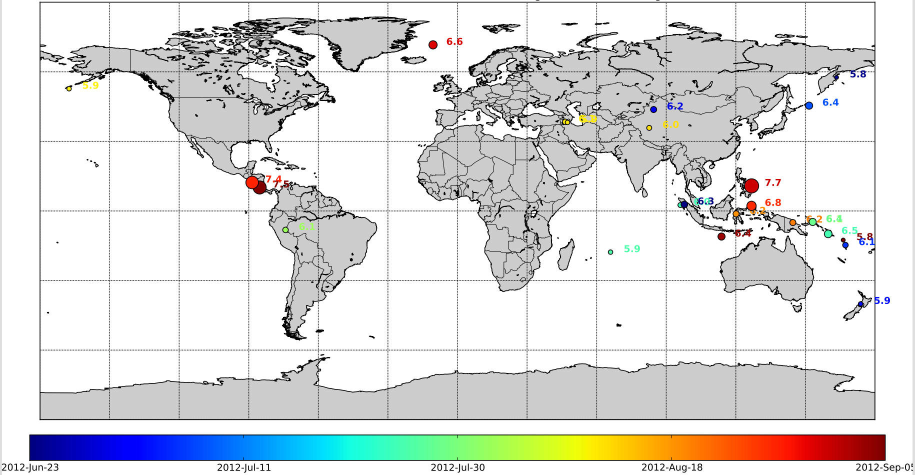
- They can be plotted using the basemap module.

```
>>> cat.plot()
```

- And they can be written.

```
>>> cat.write("modified_events.xml", format="quakeml")
```

25 events (2012-06-23 to 2012-09-05) - Color codes origin time, size the magnitude



obspy.core.event - Exercise

- Read the **example_catalog.xml** file.
- Plot the events.
- Print the resulting Catalog object and filter it, so it only contains events with a magnitude larger then 8.
- Now assume you did a new magnitude estimation and want to add it to one event. Create a new magnitude object, fill it with some values and append it to magnitude list of the largest event.
- Write the Catalog as a QuakeML object.

Waveform Plugins

Waveform Plugins

- Read and write support for all waveform formats is handled via plugins.
- The following formats are currently supported:
  - datamark
  - gse2
  - mseed
  - sac
  - seg2
  - segy
  - seisan
  - sh
  - wav

Waveform Plugins

- Format specific header values are stored in the stats object of the Trace, e.g. for files in the MiniSEED format:

```
>>> print tr.stats.mseed
AttribDict({"record_length": 512, "encoding": "STEIM1",
    "filesize": 28690432L, "dataquality": "D",
    "number_of_records": 56036L, "byteorder": ">"})
```

- Format specific header values are stored in the stats object of the Trace, e.g. for files in the MiniSEED format:

```
>>> st = read()
>>> st.write("output_file.mseed", format="mseed",
    "record_length"=1024, "encoding"="STEIM2")
```

Retriving Data - ObsPy Clients

Clients - Getting waveform data from the web

ObsPy has clients for **NERIES**, **IRIS**, **ArcLink**, **SeisHub**, **SeedLink** and **Earthworm**.

```
>>> from obspy import UTCDateTime
>>> from obspy.arclink.client import Client
>>> client = Client(user="test@obspy.org")
>>> t = UTCDateTime("2009-08-20 04:03:12")
>>> st = client.getWaveform("BW", "RJOB", "", "EH*",
        t - 3, t + 15)
>>> st.plot()
```

- Similar interfaces for the other clients.
- The returned Stream object is already known.
- In the end it does not matter if the data originally is from a file or from a webservice.

Clients - Retrieving other data

The webservices are not limited to retrieving waveform data. Depending on the client module used, the available data includes:

- Event data
- Inventory and response data.
- Availability information.
- ...

obspy.arclink - Retrieving the Instrument Response

```
>>> from obspy import UTCDateTime
>>> from obspy.arclink.client import Client
>>> client = Client(user="test@obspy.org")
>>> dt = UTCDateTime(2009, 1, 1)
>>> paz = client.getPAZ("BW", "MANZ", "", "EHZ", dt)
>>> paz
AttribDict({"poles": [(-0.037004+0.037016j),
                (-0.037004-0.037016j), (-251.33+0j),
                (-131.04-467.29j), (-131.04+467.29j)],
            "sensitivity": 2516778600.0,
            "zeros": [0j, 0j],
            "name": "LMU:STS-2/N/g=1500",
            "gain": 60077000.0})
```

obspy.arclink - Requesting Inventory Data

```
>>> from obspy import UTCDateTime
>>> from obspy.arclink.client import Client
>>> client = Client(user="test@obspy.org")
>>> inv = client.getInventory("BW", "M*", "*", "EHZ",
        restricted=False, permanent=True,
        min_longitude=12, max_longitude=12.2)
>>> inv.keys()
["BW.MROB", "BW.MANZ..EHZ", "BW", "BW.MANZ", "BW.MROB..EHZ"]
>>> inv["BW"]
AttribDict({"description": "BayernNetz",
            "region": "Germany", ...
>>> inv["BW.MROB"]
AttribDict({"code": "MROB",
            "description": "Rosenbuehl, Bavaria", ...
```

obspy.arclink - Exercises

1. Use the obspy.arclink client and request some inventory information of your choice.
2. Use the gained information to download waveform and response information.
3. Correct for the instrument and save the file to disc.
4. (Optional) Use any of the other ObsPy clients. Some have additional functionality - refer to the ObsPy documentation for more information.

obspy.signal - Signal Processing Routines

sonic	cfrequency	fem
array_transff_wavenumber	bwith	fpm
array_transff_freqslowness	domperiod	em
relcalstack	logbankm	pm
envelope	logcep	tpg
normEnvelope	sonogram	rdct
centroid	cosTaper	fpg
instFreq	c_sac_taper	eg
instBwith	evalresp	pg
xcorr	cornFreq2Paz	plotTfMisfits
xcorr_3C	pazToFreqResp	plotTfGofs
xcorr_max	waterlevel	plotTfr
xcorrPickCorrection	specInv	recSTALTA
simple	seisSim	carlSTATrig
bandpass	paz2AmpValueOfFreqResp	classicSTALTA
bandstop	estimateMagnitude	delayedSTALTA
lowpass	estimateWoodAndersonA...	zDetect
highpass	konnoOhmachiSmoothing	triggerOnset
envelope	eigval	pkBaer
remezFIR	class PPSD	arPick
lowpassFIR	rotate_NE_RT	plotTrigger
integerDecimation	rotate_ZNE_LQT	coincidenceTrigger
lowpassCheby2	rotate_LQT_ZNE	utlGeoKm
polarizationFilter	cwt	utlLonLat

Filtering

```python
from obspy import read
st = read()
st.filter("highpass", freq=1.0, corners=2, zerophase=True)
```

Available filters:

- bandpass
- bandstop
- lowpass
- highpass
- lowpassCheby2
- lowpassFIR (experimental)
- remezFIR (experimental)

Instrument correction

```python
from obspy import read
from obspy.signal import cornFreq2Paz
paz_sts2 = {\
    "poles": ...,
    "zeros": [0j, 0j],
    "gain": 60077000.0,
    "sensitivity": 2516778400.0}
paz_1hz = cornFreq2Paz(1.0, damp=0.707)
st = read()
st.simulate(paz_remove=paz_sts2, paz_simulate=paz_1hz)
```

- The PAZ can also be retrieved from one the webservices, or from a SEED or RESP file.

Thanks for your Attention!

Appendix

Events - Resource References

- In QuakeML resources can refer to each other using a unique identifier string.
- These connections are preserved in obspy.core.event.
- This works across file boundaries assuming all necessary resources have been read before.

```
>>> magnitude = event.magnitudes[0]
# Retrieve the associated Origin object.
>>> print magnitude.origin_id
quakeml:eu.emsc/origin/rts/261020/782484
>>> origin = magnitude.origin_id.getReferredObject()
>>> print origin
Origin
  resource_id: ResourceIdentifier(...)
         time: UTCDateTime(2012, 4, 4, 14, 21, 42, 300000)
     latitude: 41.818
    longitude: 79.689
    ...
```