LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

MESS 2011
Python & ObsPy Introduction

Robert Barsch, Tobias Megies

Department für Geo- and Umweltwissenschaften (Geophysik)
Ludwig-Maximilians-Universität München

2011-02-21

Python Introduction

Trinity:

> *I know why you're here, Neo. I know what you've been doing... why you hardly sleep, why you live alone, and why night after night, you sit by your computer. [...] It's the question that brought you here. You know the question, just as I did.*

Neo:

> *What is Python?**

Trinity:

> *The answer is out there, Neo, and it's looking for you, and it will find you if you want it to.*

*grossly inaccurate quote

Python Introduction

- This course will **not** teach you basic programming
- Assume you already know:
  - variables
  - loops
  - conditionals (if / else)
  - standard data types, int, float, string, lists / arrays
  - reading/writing data from files
- This lecture will show you how to do these well in Python

Python Introduction

- This course will **not** teach you basic programming
- Assume you already know:
  - variables
  - loops
  - conditionals (if / else)
  - standard data types, int, float, string, lists / arrays
  - reading/writing data from files
- This lecture will show you how to do these well in Python

Python Introduction

- This course will **not** teach you basic programming
- Assume you already know:
  - variables
  - loops
  - conditionals (if / else)
  - standard data types, int, float, string, lists / arrays
  - reading/writing data from files
- This lecture will show you how to do these well in Python

Reasons Why Python Rocks for Research

1. Readability
2. Batteries included
3. Speed
4. Language Interoperability
5. Others

Python Rocks: Readability

- Readable syntax
  1. Does an element exist in a list/dict:

```
>>> 3 in [1, 2, 3, 4, 5]
True
```

  2. Does a substring exist in a string:

```
>>> 'sub' in 'string'
False
```

  3. Readable boolean values and logical operators

```
>>> a = True
>>> not a
False
>>> 'sub' not in ['string', 'hello', 'world']
True
>>> a or True and not 1==2
True
```

Python Rocks: Readability

- Readable syntax
  1. Does an element exist in a list/dict:

```
>>> 3 in [1, 2, 3, 4, 5]
True
```

  2. Does a substring exist in a string:

```
>>> 'sub' in 'string'
False
```

  3. Readable boolean values and logical operators

```
>>> a = True
>>> not a
False
>>> 'sub' not in ['string', 'hello', 'world']
True
>>> a or True and not 1==2
True
```

Python Rocks: Readability

- Readable syntax
  1. Does an element exist in a list/dict:

```
>>> 3 in [1, 2, 3, 4, 5]
True
```

  2. Does a substring exist in a string:

```
>>> 'sub' in 'string'
False
```

  3. Readable boolean values and logical operators

```
>>> a = True
>>> not a
False
>>> 'sub' not in ['string', 'hello', 'world']
True
>>> a or True and not 1==2
True
```

Python Rocks: Readability

- Readable syntax
  1. Does an element exist in a list/dict:

```
>>> 3 in [1, 2, 3, 4, 5]
True
```

  2. Does a substring exist in a string:

```
>>> 'sub' in 'string'
False
```

  3. Readable boolean values and logical operators

```
>>> a = True
>>> not a
False
>>> 'sub' not in ['string', 'hello', 'world']
True
>>> a or True and not 1==2
True
```

Python Rocks: Readability

- Indentation
  - Code blocks are defined by their indentation.
  - No explicit begin or end, and no curly braces to mark where a block starts and stops. The only delimiter is a colon (:) and the indentation of the code itself.

```
>>> for i in [1, 2, 3, 4, 5]:
...     if i<3:
...         print i,
...     else:
...         print i*2,
...
1 2 6 8 10
```

- Very minimalistic clean syntax & semantics
  - Short code = Less errors!
  - But also faster development, quicker understanding, faster typing, faster finding errors, easier to modify ...

Python Rocks: Readability

- Indentation
  - Code blocks are defined by their indentation.
  - No explicit begin or end, and no curly braces to mark where a block starts and stops. The only delimiter is a colon (:) and the indentation of the code itself.

```
>>> for i in [1, 2, 3, 4, 5]:
...     if i<3:
...       print i,
...     else:
...       print i*2,
...
1 2 6 8 10
```

- Very minimalistic clean syntax & semantics
  - Short code = Less errors!
  - But also faster development, quicker understanding, faster typing, faster finding errors, easier to modify ...

Python Rocks: Readability

- Indentation
  - Code blocks are defined by their indentation.
  - No explicit begin or end, and no curly braces to mark where a block starts and stops. The only delimiter is a colon (:) and the indentation of the code itself.

```
>>> for i in [1, 2, 3, 4, 5]:
...     if i<3:
...       print i,
...     else:
...       print i*2,
...
1 2 6 8 10
```

- Very minimalistic clean syntax & semantics
  - Short code = Less errors!
  - But also faster development, quicker understanding, faster typing, faster finding errors, easier to modify ...

Python Rocks: Readability

Guido van Rossum (Python's original author)

> *This emphasis on readability is no accident. As an object-oriented language, Python aims to encourage the creation of reusable code. Even if we all wrote perfect documentation all of the time, code can hardly be considered reusable if it's not readable. Many of Python's features, in addition to its use of indentation, conspire to make Python code highly readable.*

$\Longrightarrow$ Design focus on productivity and code readability

Python Rocks: Readability

Guido van Rossum (Python's original author)

> *This emphasis on readability is no accident. As an object-oriented language, Python aims to encourage the creation of reusable code. Even if we all wrote perfect documentation all of the time, code can hardly be considered reusable if it's not readable. Many of Python's features, in addition to its use of indentation, conspire to make Python code highly readable.*

$\Longrightarrow$ Design focus on productivity and code readability

Python Rocks: Readability

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
...
```

Python Rocks: "Batteries included"

- Extensive standard libraries:
  - Data Persistence
  - Data Compression and Archiving
  - Cryptographic Services
  - Internet Protocols
  - Internet Data Handling
  - Structured Markup Processing Tools
  - Multimedia Services
  - Internationalization
  - Development Tools
  - Multithreading & Multiprocessing
  - Regular expressions
  - Graphical User Interfaces with Tk
  - ...

Python Rocks: "Batteries included"

- Well-documented
- Platform independent API, but optimized for each platform
- One place to look first for a proven solution
- Reuse instead of reinvent
- Strong scientific 3rd party libraries:
  - **NumPy/SciPy** - array and matrix structures, linear algebra routines, numerical optimization, random number generation, statistics routines, differential equation modeling, Fourier transforms and signal processing, image processing, sparse and masked arrays, spatial computation, and numerous other mathematical routines
  - **Matplotlib** - 2D plotting library which produces publication quality figures via a set of functions familiar to MATLAB users
  - **mplot3d** toolkit (Matplotlib), **Mayavi2**, ...

$\Longrightarrow$ No need to switch languages in order to write matrix manipulation code or automating an operating system task.

Python Rocks: "Batteries included"

- Well-documented
- Platform independent API, but optimized for each platform
- One place to look first for a proven solution
- Reuse instead of reinvent
- Strong scientific 3rd party libraries:
  - **NumPy/SciPy** - array and matrix structures, linear algebra routines, numerical optimization, random number generation, statistics routines, differential equation modeling, Fourier transforms and signal processing, image processing, sparse and masked arrays, spatial computation, and numerous other mathematical routines
  - **Matplotlib** - 2D plotting library which produces publication quality figures via a set of functions familiar to MATLAB users
  - **mplot3d** toolkit (Matplotlib), **Mayavi2**, ...

$\Longrightarrow$ No need to switch languages in order to write matrix manipulation code or automating an operating system task.

Python Rocks: "Batteries included"

- Well-documented
- Platform independent API, but optimized for each platform
- One place to look first for a proven solution
- Reuse instead of reinvent
- Strong scientific 3rd party libraries:
  - **NumPy/SciPy** - array and matrix structures, linear algebra routines, numerical optimization, random number generation, statistics routines, differential equation modeling, Fourier transforms and signal processing, image processing, sparse and masked arrays, spatial computation, and numerous other mathematical routines
  - **Matplotlib** - 2D plotting library which produces publication quality figures via a set of functions familiar to MATLAB users
  - **mplot3d** toolkit (Matplotlib), **Mayavi2**, ...

$\Longrightarrow$ No need to switch languages in order to write matrix manipulation code or automating an operating system task.

Python Rocks: Speed

"Python is extremely slow and wastes memory!"

```
import os

xvec = range(2000000)
yvec = range(2000000)
zvec = [0.5*(x+y) for x,y in zip(xvec,yvec)]

os.system("ps v "+str(os.getpid()))
```

- 104 MB memory usage!
- Runs almost 8 seconds!

Python Rocks: Speed

Comparison with C

```
#include <stdlib.h>
#include <unistd.h>
main () {
    int *avec, *bvec;
    float *cvec;
    int i, n = 2000000;
    avec = (int*)calloc( n, sizeof(int) );
    bvec = (int*)calloc( n, sizeof(int) );
    for (i=0;i<n;i++) {
        avec[i] = bvec[i] = i;
    }
    cvec = (float*)calloc( n, sizeof(float) );
    for (i=0;i<n;i++) {
        cvec[i] = (avec[i]+bvec[i])*0.5;
    }
    int slen = 100;
    char *command = (char*)calloc( slen, sizeof(char) );
    snprintf(command, slen, "ps v %i", getpid() );
    system( command );
}
```

- 25 MB memory usage
- Runs about 0.1 s (almost a hundred times faster!)

Python Rocks: Speed

Using the "right tool" for the job: NumPy

```
import os
from numpy import *

xvec = arange(2000000)
yvec = arange(2000000)
zvec = (xvec+yvec)*0.5

os.system("ps v "+str(os.getpid()))
```

- 37 MB memory usage
- Runs about 0.3 s

Python Rocks: Speed

Implementation time vs. execution time

- Python is designed for productivity
- No (separate) compilation step
  - No compiler problems
  - No makefiles
  - No linker problems
  - Faster development cycles
- When execution speed matters:
  - Use specialized modules
  - Implement time critical parts in C/C++/Fortran
  - Prototyping & profiling
  - Use specialized JIT compiler

Python Rocks: Speed

Implementation time vs. execution time

- Python is designed for productivity
- No (separate) compilation step
  - No compiler problems
  - No makefiles
  - No linker problems
  - Faster development cycles
- When execution speed matters:
  - Use specialized modules
  - Implement time critical parts in C/C++/Fortran
  - Prototyping & profiling
  - Use specialized JIT compiler

Python Rocks: Speed

Implementation time vs. execution time

- Python is designed for productivity
- No (separate) compilation step
  - No compiler problems
  - No makefiles
  - No linker problems
  - Faster development cycles
- When execution speed matters:
  - Use specialized modules
  - Implement time critical parts in C/C++/Fortran
  - Prototyping & profiling
  - Use specialized JIT compiler

Python Rocks: Speed

Implementation time vs. execution time

- Python is designed for productivity
- No (separate) compilation step
  - No compiler problems
  - No makefiles
  - No linker problems
  - Faster development cycles
- When execution speed matters:
  - Use specialized modules
  - Implement time critical parts in C/C++/Fortran
  - Prototyping & profiling
  - Use specialized JIT compiler

Python Rocks: Language Interoperability

Python excels at gluing other languages together:

- **FORTRAN**: F2py - Fortran to Python interface generator (part of NumPy)
- General **C** or **C++** libraries: Ctypes, Cython, or SWIG are three ways to interface to it
- **R**: RPy - simple, robust Python interface to the R Programming Language. It can manage all kinds of R objects and can execute arbitrary R functions (including the graphic functions).

$\Longrightarrow$ Now, if only all these were two way streets ...

Python Rocks: Language Interoperability

Python excels at gluing other languages together:

- **FORTRAN**: F2py - Fortran to Python interface generator (part of NumPy)
- General **C** or **C++** libraries: Ctypes, Cython, or SWIG are three ways to interface to it
- **R**: RPy - simple, robust Python interface to the R Programming Language. It can manage all kinds of R objects and can execute arbitrary R functions (including the graphic functions).

$\Longrightarrow$ Now, if only all these were two way streets ...

Python Rocks: Language Interoperability

Python excels at gluing other languages together:

- **FORTRAN**: F2py - Fortran to Python interface generator (part of NumPy)
- General **C** or **C++** libraries: Ctypes, Cython, or SWIG are three ways to interface to it
- **R**: RPy - simple, robust Python interface to the R Programming Language. It can manage all kinds of R objects and can execute arbitrary R functions (including the graphic functions).

$\Longrightarrow$ Now, if only all these were two way streets ...

Python Rocks: Language Interoperability

Python excels at gluing other languages together:

- **FORTRAN**: F2py - Fortran to Python interface generator (part of NumPy)
- General **C** or **C++** libraries: Ctypes, Cython, or SWIG are three ways to interface to it
- **R**: RPy - simple, robust Python interface to the R Programming Language. It can manage all kinds of R objects and can execute arbitrary R functions (including the graphic functions).

$\Longrightarrow$ Now, if only all these were two way streets ...

Python Rocks: Further Reasons

- Free, open source (Python Software Foundation License)
- Platform independent
- Availability: standard component for many operating systems
- Very broadly applicable
  - can be used as a terminal calculator
  - can substitute shell scripts
  - can be used to create large GUI applications
  - can be used to do numerics in MATLAB style
- Easy to learn
- Very high level programming language
- Support for multiple programming paradigms (object oriented, imperative, and functional)
- Dynamic data types & automatic memory management & garbage collection
- Strong introspection capabilities

How to Work on the Practicals..

- Either..
  - work line by line in IPython shell
  - when it's working: save history and condense it

```
>>> %history [number_of_lines] [-n] [-f output_file]
```

- or..
  - work on your program in a text editor
  - in a second window, run program in an IPython shell and continue work at the end

```
$ ipython -i
>>> run -i PROGRAM.PY
```

    - (caution: best do this in a "freshÏPython shell)
  - extend program with appropriate lines of code and run it again in a new IPython shell

Getting Help..

IPython

- get help for a function: `>>> command?`
- have a look at the implementation: `>>> command??`
- search for variables/functions/modules starting with äb": `>>> ab<Tab>`
- what's the value? `>>> variable`
- what's the type? `>>> type(variable)`
- which variables are assigned anyway?? `>>> whos`
- what attributes/methods are there? `>>> variable.<Tab>`
- get help for a variable's method: `>>> variable.command?`
- what functions are available in a module? `>>> module.<Tab>`

Getting Help..

- ObsPy web pages
  - Tutorial
    - http://obspy.org/wiki/ObspyTutorial
    - file:///home/messuser/obspy/tutorial/ObspyTutorial.html
  - API
    - http://docs.obspy.org/
    - file:///home/messuser/obspy/docs/index.html
- Python/Numpy/Scipy API
  - http://docs.python.org/
  - file:///home/messuser/obspy/python/python-docs/index.html
  - http://docs.scipy.org/doc/numpy/reference/
  - file:///home/messuser/obspy/python/numpy-docs/index.html
  - http://docs.scipy.org/doc/scipy/reference/
  - file:///home/messuser/obspy/python/scipy-docs/index.html

ObsPy Data Types – Overview

- `UTCDateTime`
  - extension of the Python `datetime` object
  - stores a time stamp
- `Stats`
  - extension of the Python `dict` object
  - stores header information of waveforms
- `Trace`
  - stores a single-channel, continuous piece of waveform data
  - consisting of waveform data and header information
- `Stream`
  - stores multiple traces (e.g. Z, N, E traces of one station)
- all of them defined in `obspy.core`
  - `from obspy.core import UTCDateTime`

ObsPy Data Types – `UTCDateTime`

- `UTCDateTime`
  - used to handle all time information in ObsPy
  - initialize via
    - `t = UTCDateTime("2011-02-21T08:00:00.00Z")`
    - `t = UTCDateTime(2011, 2, 21, 8)`
    - ...
  - several attributes/methods
    (e.g. `t.microsecond`, `t.julday`, `t.weekday()`, ...)
  - important operations
    - subtracting two `UTCDateTime` objects gives time difference in seconds
    - adding/subtracting `int`/`float` returns new `UTCDateTime` object
  - see ObsPy documentation

Exercises – `UTCDateTime`

- A
  - the morning sessions start at 8 and are 3 hours..
    assume we want to have the coffee break 1234 seconds and 5 microseconds before the session ends. What time is the break?
  - assume you had your last cup of coffee yesterday at breakfast. How many minutes do you have to survive with that cup of coffee?
- B
  - how many days from today is your birthday this year?
  - what day of week is it?
  - you want to have your birthday party at the first saturday after your birthday, what date is the party?
- C
  - some of your friends always seem to find an excuse not to come... for the next fifty years, print the date of your party sticking to that scheme
  - we are superstitious and do not leave the house on friday 13th.. print a list of dates and count the days we have to take off from now till the end of next year.

Exercises – `UTCDateTime`

- A
  - the morning sessions start at 8 and are 3 hours..
    assume we want to have the coffee break 1234 seconds and 5 microseconds before the session ends. What time is the break?
  - assume you had your last cup of coffee yesterday at breakfast. How many minutes do you have to survive with that cup of coffee?
- B
  - how many days from today is your birthday this year?
  - what day of week is it?
  - you want to have your birthday party at the first saturday after your birthday, what date is the party?
- C
  - some of your friends always seem to find an excuse not to come..
    for the next fifty years print the date of your party sticking to that scheme
  - we are superstitious and do not leave the house on friday 13th..
    print a list of dates and count the days we have to take off from now till the end of next year..

ObsPy Data Types – `Stats`

- `Stats` – header information for waveform data
  - contains at least the following keys
    - `stats.network` – network code (`str`)
    - `stats.station` – station code (`str`)
    - `stats.location` – location code (`str`)
    - `stats.channel` – channel code (`str`)
    - `stats.starttime` – time of first sample (`UTCDateTime`)
    - `stats.sampling_rate` – sampling rate in `Hz` (`float`)
    - `stats.npts` – number of samples (`int`)
  - derived keys
    - `stats.endtime` – time of last sample (`UTCDateTime`)
    - `stats.delta` – sampling interval (`float`)
  - optional keys
    - `stats._format` – format of original data file (`str`, e.g. `"MSEED"`)
    - `stats.paz` – poles, zeros, sensitivity and gain of instrument (`dict`)
    - `stats.coordinates` – longitude, latitude and elevation of station (`dict`)
    - ...
  - see ObsPy documentation

ObsPy Data Types – `Trace`

- `Trace` – continuous waveform data
  - usually constructed internally during `read(...)` or `getWaveform(...)`
  - consists of
    - `tr.data` – waveform data as a `numpy.ndarray` instance
    - `tr.stats` – header information as a `Stats` instance
  - built-in methods
    - `tr.id` – complete channel id in SEED standard (e.g. `"BW.RJOB..BHZ"`)
    - `tr.plot()` – shows preview plot of trace
    - `tr.copy()` – returns copy of trace (most operations work in-place)
    - `tr.trim(starttime, endtime)` – cut trace to specified time span
    - `tr.filter("type", **kwargs)` – filter waveform data
    - `tr.simulate(paz_remove, paz_simulate, **kwargs)` – apply instrument correction/simulation
    - `tr.write("filename", "format")` – write waveform to local file
    - ...
  - many built-in methods on `tr.data` (`numpy.ndarray`)!
  - see ObsPy documentation
  - see Numpy documentation – ndarray

Exercises – `Trace`

- A
  - 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 display the preview plot of the trace
  - change the sampling rate to 20Hz
  - change the `starttime` to the start time of this sessions
  - print trace summary and display the preview plot of the trace again

Exercises – `Trace`

- B
  - use `tr.filter(...)` and apply a lowpass with 1s corner frequency
  - 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 end
- C
  - 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

ObsPy Data Types – `Stream`

- `Stream` – collection of `Trace` objects in a `list`-like container
  - usually returned by a `read(...)` or `getWaveform(...)` call
  - `print st` – prints summary of all traces
  - `print len(st)` – prints number of traces in stream
  - `list`-like operations
    - `st[i]` – return trace at index `i`
    - `st.append(tr)` – add a single trace
    - `st.extend(st)` – add a list of traces
    - `st.remove(tr)` – remove specified trace from stream
    - `st.pop(i)` – remove trace at specified index and return it
    - `st.sort(...)` – sort traces in stream according to specified criteria
  - other built-in methods
    - `st.select(**kwargs)` – return new stream with matching traces (e.g. `component="Z"`)
    - `st.merge(method)` – merge traces with identical id
    - `st.printGaps()` – prints summary of gaps in the stream
  - many built-in methods of `Trace` (`trim`, `filter`, `simulate`,...)
  - see ObsPy documentation

Exercises – `Stream`

- A
  - 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

Exercises – `Stream`

- B
  - read the example earthquake data again
  - make a dictionary with `paz` information, assign `poles` at `[-0.037+0.037j, -0.037-0.037j]`, `zeros` at `[0j, 0j]`, the `sensitivity` of `2.517e9` and unity `gain`
  - remove the instrument response using this `paz` dictionary
  - print the data maximum and minimum of the first trace (now in m/s)
  - save the data to a local file in `MSEED` format
- C
  - read the example earthquake data again
  - change the station name for all traces in the stream
  - read the example earthquake data yet again and add the traces to the first stream
  - print the summary for the resulting stream and display the preview plot
  - `select` the Z traces and assign this stream to another variable
  - filter the Z components with a highpass at 5Hz
  - display the preview plot of the Z component stream
  - display the preview plot of the original stream