MESS 2012
Python & ObsPy Introduction

Tobias Megies, Robert Barsch, J. Wassermann
Department for Earth and Environmental Sciences (Geophysics)
Ludwig-Maximilians-Universit at München

Fighting for Python

Why a scripting language?

- Higher productivity of scientists (computer scientists possibly - but not naturally - excluded)
- Simplified syntax
- Integration of simulation, visualization and data analysis
- Scientists often change data formats - higher flexibility

Why Python?

- Gluing your favorite simulation, visualization and data analysis in an easy way
- Many different data formats supported
- Python offers an exhaustive library to create GUIs in a very professional way
- Python can serve as modern interface to old codes

Python vs. Matlab?

Many things in common - **BUT**:

- Python is more powerful !!!!!!
- Python is for free and completely open
- Tons of additional function modules and packages available
- Nested, heterogenous data structures easily realizable
- Object orientated programming
- Much better interface to low-level codes (C,C++,Fortran)

Python Introduction

This course will **not** teach you basic programming

We assume you already know:

- Variables
- Loops
- Conditionals (if / else), control flow (for, while)
- Standard data types, int, foat, string, lists / arrays
- Reading/writing data from files

This lecture will show you how to do these well in **Python**

Why is Python so perfect for science?

1. Readability
2. Batteries included (oh boy - what does that mean)
3. Speed
4. Language interoperability
5. ...

Why is Python so perfect for Science?

Readable syntax:

✓Does an element exist in a list/dict?

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

✓Does a substring exist in a string?

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

✓Readable boolean values and logical operators

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

Why is Python so perfect for Science?

- 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
```

Why is Python so perfect for Science?

- Very minimalistic clean syntax & semantics

➡ Shorter code = Less errors!
➡ But also faster development, quicker understanding, faster typing, faster finding errors, easier to modify ...

Why is Python so perfect for Science?

- 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 or Qt
➡ ...

Why is Python so perfect for Science?

- 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

Why is Python so perfect for Science?

- Strong scientific 3rd party libraries:

  ➡ **Matplotlib** - 2D plotting library which produces publication quality figures via a set of functions familiar to MATLAB users

  ➡ Interfaces to Matlab, Mathematica, Maple, Octave...

  ➡ **mplot3d** toolkit (Matplotlib), **Mayavi2**, ...

⇒ No need to switch languages in order to write matrix manipulation code, communicate with a web server or automate an operating system task..

Why is Python so perfect for Science?

- 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!

Why is Python so perfect for Science?

- 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!)

Why is Python so perfect for Science?

- 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!

Why is Python so perfect for Science?

- 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, Cython

Why is Python so perfect for Science?

- Language Interoperability

Python excels at gluing other languages together:

➡ FORTRAN: F2py - Fortran to Python interface generator (part of NumPy)

➡ General C or C++ libraries: Cython, Ctypes, or SWIG are three ways to interface to it

➡ R: RPy - simple, robust Python interface to the Programming Language. It can manage all kinds of R objects and can execute arbitrary R functions (including the graphic functions).

Why is Python so perfect for Science?

- 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

Why is Python so perfect for Science?

- Further Reasons

➡ Easy to learn for beginners, yet very powerful for advanced users
➡ 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

IPython

- Enhanced interactive Python shell; Main features:

➡ Dynamic introspection and help
➡ Searching through modules and namespaces
➡ Tab completion
➡ Complete system shell access
➡ Session logging & restoring
➡ Verbose and colored exception traceback printouts
➡ Debugger included
➡ Highly configurable, programmable (Macros, Aliases)

IPython

- Getting Help
  - ➡Get help for a function:
    In[1]: command?
  - ➡Have a look at the implementation:
    In[2]: command??
  - ➡Search for variables/functions/modules starting with 'ab':
    In[3]: ab<Tab>
  - ➡Which objects are assigned anyway?
    In[4]: whos
  - ➡What attributes/methods are there?
    In[5]: object.<Tab>
  - ➡Get help for a object/class method/attribute:
    In[6]: object.command?

Python Data Types: Numbers

```
>>> a = 17
>>> type(a)
<type 'int'>
>>> a / 10
1
>>> a % 10
7
>>> a / 10.0
1.7
>>> _
1.7
>>> type(_)
<type 'float'>
```

Python Data Types: Numbers

```
>>> a = 3.0 + 4.0j
>>> float(a)
Traceback (most recent call last):
...
TypeError: can't convert complex to float
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a) #sqrt(a.real ** 2 + a.imag ** 2)
5.0
```

Python Data Types: Numbers

```
>>> a = 17
>>> a = a + 1
>>> a
18
>>> a += 2
>>> a
20
>>> a++
a++
^
SyntaxError: invalid syntax
```

Python Data Types: Numbers

```
>>> a = 17
>>> a = a + 1
>>> a
18
>>> a += 2
>>> a
20
>>> a++
a++
^
SyntaxError: invalid syntax
```

NOTE: Numbers are immutable

a = 5; b = a ; a = 6; a is b:
False

Python Data Types: Numbers conversions

```
>>> s = ‘13.8’
>>> f = float(s)
>>> i = int(s)
ValueError: invalid literal for int() with base 10: '13.8'
>>> i = int(f)
>>> i
13
>>> complex(s)
(13.8+0j)
```

Division:

```
>>>p=3; q=6; p/q
0
>>>float(p)/q
0.5
```

Python Data Types: Strings

```
>>> 'spam eggs'
'spam eggs'
>>> "doesn't"
"doesn't"
>>> 'doesn\'t'
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Python Data Types: Strings

```
>>> ‘MESS’ + ‘ing around’
‘MESSing around’
>>> ‘MESS’ * 5
‘MESSMESSMESSMESSMESS’
>>> a = "Winter School"
>>> a[0]
'W'
>>> a[0:1]
'W' # different than in other languages!
>>> a[1:5]
'inter'
>>> a[-7:]
'School'
```

Python Data Types: Strings

```
>>> a = 'spam'
>>> a[3] = 'n' # strings are immutable !!
Traceback (most recent call last):
...
TypeError: 'str' object does not support
item assignment
>>> b = a[:-1] + 'n'
>>> b
'span'
>>> len(b)
4
```

Python Data Types: Strings

Strings are objects with many useful methods:

```
>>> a = "Winter School 2012"
>>> a.find('School')
7
>>> a.split()
['Winter', 'School', '2012']
>>> ' * '.join(_)
'Winter * School * 2012'
```

There are more useful string methods like startswith, endswith, lower, upper, ljust, rjust, center, ... See Python Library Reference.

Python Data Types: Lists

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 2*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Python Data Types: Lists

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23 # lists are mutable
>>> a
['spam', 'eggs', 123, 1234]
>>> a[0:2] = [1, 12] # Replace some items
>>> a
[1, 12, 123, 1234]
>>> a[0:2] = [] # Remove some
>>> a
[123, 1234]
>>> a[1:1] = ['bletch', 'xyzzy'] # Insert some
>>> a
[123, 'bletch', 'xyzzy', 1234]
```

Python Data Types: Lists

Important list operations:

Adding

```
>>>  arglist = []; arglist.append(myvar)
[myvar]
```

Extracting

```
>>>  [filename,plotfile,psfile] = arglist
```

Indexing

```
>>>  filename = arglist(0)
```

Searching

```
>>> i = arglist.index('tmp.ps')
>>> del arglist[i]
```

Python Data Types: Lists

Slicing:

```
>>> a = ‘demonstrate slicing in Python’.split()
>>> a[-1] #the last entry
[‘Python’]
>>>  a[:-1]  #everything NOT including the last  entry
['demonstrate', 'slicing', 'in']
>>> a[:] #everything
>>> a[2:] #everyting from index 2 upwards
['in', 'Python']
>>> a[-2:] #last two entries
>>> a[1:3]
>>> a[:0] = (‘here we’).split() #as append but starts at the
beginning of the list
['here', 'we', 'demonstrate', 'slicing', 'in', 'Python']
```

There are more useful list methods like append, insert, remove, sort, pop, index, reverse, ... See Python Library Reference.

Python Data Types: Tuples, Boolean, None

Tuple:

- Immutable lists created by round parentheses

```
>>> t = (12345, 54321, 'hello!')
>>> t[0]
12345
```

Boolean:

```
>>> type(True)
<type 'bool'>
```

None:

```
>>> a = None
>>> type(a)
<type 'NoneType'>
```

Python Data Types:Dictionary

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

Python Data Types: Dictionary

Important dictionary operations:

```
>>> d = {'dt': 10, 'xd': 20}
>>>  d[‘dt’] #key dt
10
>>>  d.keys() #returns a copy of list of keys
['dt', 'xd']
>>> d.has_key(‘dt’) #searching for a specific key
True
>>>  d.get(‘dt’,1.0) #looks for value of ‘dt’ if not existing takes default 1.0
>>> d.items() #list the content of dictionary as list of tuples
[('dt', 10), ('xd', 20)]
```

Python Data Types: NumPy Arrays

We need to import numpy for the following examples:

```
>>> import numpy as np
>>> a = np.array([2, 3, 4])
>>> a
array([2, 3, 4])
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array( [ (1.5, 2, 3), (4, 5, 6) ] )
>>> b
array([[ 1.5, 2. , 3. ],
[ 4. , 5. , 6. ]])
```

Python Data Types: NumPy Arrays

```
>>> b.ndim #number of dimensions
2
>>> b.shape #the dimensions
(2, 3)
>>> b.dtype #the type (8 byte floats)
dtype('float64')
>>> b.itemsize #the size of the type
8
>>> c = np.array([[1, 2], [3, 4]], dtype=complex)
>>> c
array([[ 1.+0.j, 2.+0.j],
[ 3.+0.j, 4.+0.j]])
```

Python Data Types: NumPy Arrays

```
>>> np.zeros((3, 4)) # parameter to specify the shape
array([[0., 0., 0., 0.],
[0., 0., 0., 0.],
[0., 0., 0., 0.]])
>>> np.ones((2, 3, 4), dtype=int16) # dtype also
specified
array([[[ 1, 1, 1, 1],
[ 1, 1, 1, 1],
[ 1, 1, 1, 1]],
[[ 1, 1, 1, 1],
[ 1, 1, 1, 1],
[ 1, 1, 1, 1]]], dtype=int16)
```

Supported data types: bool, uint8, uint16, uint32, uint64, int8, int16, int32, int64,float32, float64, float96, complex64, complex128, complex192

Python Data Types: NumPy Arrays

```
>>> np.arange(10, 30, 5)
array([10, 15, 20, 25])
>>> np.arange(0, 2, 0.3) # accepts float arguments
array([ 0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
>>> np.linspace(0, 2, 9) # 9 numbers from 0 to 2
array([ 0. , 0.25, 0.5 , 0.75, ..., 2. ])
>>> x = np.linspace(0, 2*np.pi, 100)
>>> f = np.sin(x)
```

Python Data Types: NumPy Arrays

NOTE: Difference of Lists and Numpy Arrays when slicing:

```
>>> a = [10,20,5]
>>> b = a # b is referencing a
>>> b is a
True
>>> a[2] = 30
>>> b is a
True
>>> b = a[:] #this is a copy of a
>>> a == b
True #b is a copy of a
>>> a is b
False
>>> b[1] = 30
>>> b == a
False
```

Python Data Types: NumPy Arrays

NOTE: Difference of Lists and Numpy Arrays when slicing:

```
>>> from numpy import *
>>> a = asarray(a) #makes an array out of a list
>>> b = a #b is referencing a
>>> b is a
True
>>> b = a[:]
>>> b[2] = 80
>>> b == a
True
>>> b = a.copy() #this is now a true copy
>>> b == a #b is a gives a False
True
>>> b[2] = 90
>>> b == a
False
```

Python Data Types: NumPy Arrays

```
>>> A = np.array([[1,1], [0,1]])
>>> B = np.array([[2,0], [3,4]])
>>> A * B # element wise product
array([[2, 0],
[0, 4]])
>>> np.dot(A, B) # matrix product
array([[5, 4],
[3, 4]])
>>> np.mat(A) * np.mat(B) # matrix product
matrix([[5, 4],
[3, 4]])
```

There are further functions for array creation, conversions, manipulation, querying, ordering, operations, statistics, basic linear algebra. See NumPy documentation.

Python Plotting: Matplotlib

Matplotlib is **the** plotting library for Python.

- syntax is close to Matlab's plotting commands
- advanced users can control all details of the plots

We need to import matplotlib for the following examples:

```
>>> import matplotlib.pyplot as plt
>>> x = [0, 2, 2.5]
>>> plt.plot(x)
>>> plt.show()
>>> x = [0, 2, 2.5]
>>> y = [3, -3.5, -1]
>>> plt.plot(x, y, 'ro')
>>> plt.show()
```

Python Plotting: Matplotlib

```
>>> t = np.arange(0.0, 2.0, 0.01)
>>> s = np.sin(2 * np.pi * t)
>>> plt.plot(t, s, linewidth=1.0)
>>> plt.xlabel('time (s)')
>>> plt.ylabel('voltage (mV)')
>>> plt.title('This is a very simple plot')
>>> plt.grid(True)
>>> plt.show()
```

See the Matplotlib homepage for basic plotting commands and especially the Matplotlib Gallery for many, many of plotting examples with source code!

Python Flow control: if statement

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...         print 'Negative'
...     elif x == 0:
...         print 'Zero'
...     elif x == 1:
...         print 'Single'
...     else:
...         print 'More'
...
More
```

Python Flow control: for statement

```
>>> a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...       print x, len(x)
...
cat 3
window 6
defenestrate 12
>>> for i in range(0, 6, 2):
... print i
...
0
2
4
```

Python Flow control: while statement

```
>>> import time
>>> i = 1
>>> while True:
...         i = i * 1000 # same as: i *= 1000
...         print repr(i)
...         time.sleep(1) # wait one second
...
1000
1000000
1000000000
1000000000000L # <- type conversion occurred!
1000000000000000L
# ... continues until memory is exhausted!
```

Python Flow control: continue & break

The break statement breaks out of the smallest enclosing for or while loop.

```
>>> for i in range(0, 100000):
...         if i > 50:
...             print i
...             break
...
51
```

The continue statement continues with the next iteration of the loop.

```
>>> for i in range(0, 100000):
...         if i != 50:
...             continue
...          print i
...
50
```

Python Functions:

Defining a function which returns a Fibonacci series up to n.

```
>>> def fib2(n):
... """Return the Fibonacci series up to n."""
... result = []
... a, b = 0, 1
... while a < n:
...     result.append(a)
...     a, b = b, a+b
... return result
```

Now call the function we just defined:

```
>>> fib(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Python Functions:

```
>>>def birthday2(name, age=1):
...:  if age < 40:
...:      msg = "Happy birthday, %s! You're %d today."
...:  else:
...:      age = 29
...:      msg = “Happy birthday, %s! You re %d today.”
...:   print msg % (name, age)
>>> birthday2("Katherine")
Happy birthday, Katherine! You're 1 today.
>>> birthday2(age=12, name="Katherine")
Happy birthday, Katherine! You're 12 today.
>>> birthday2("Katherine", 41)
Happy birthday, Katherine! You’re 29 today.
```

Python In & Out: File Handling

Use open(filename, mode) to open a file. Returns a File Object.

```
fh = open('/path/to/file', 'r')
```

- Some possible modes:
  - ➡ r: Open text file for read.
  - ➡ w: Open text file for write.
  - ➡ a: Open text file for append
  - ➡ rb: Open binary file for read
  - ➡ wb: Open binary file for write

Use close() to close a given File Object.

```
fh.close()
```

Python In & Out: Reading Files

- Read a quantity of data from a file:

```
>>> s = fh.read(size) # size: number of bytes to read
```

- Read entire file:

```
>>> s = fh.read()
```

- Read one line from file:

```
>>> s = fh.readline()
```

- Get all lines of data from the file into a list:

```
>>> list = fh.readlines()
```

- Iterate over each line in the file:

```
>>> for line in fh:
>>> print line,
```

Python In & Out: Writing Files

- Write a string to the file:

```
mystring = "spam"
fh.write(mystring)
```

- Write several strings to the file:

```
mysequence = ["eggs", "ham", "spam"]
fh.writelines(mysequence)
```

Python In & Out: Reading and Writing ASCII Based Data Files

```
>>> import numpy as np
```

Write an array as ASCII to a file:

```
>>> x = np.arange(0, 1.4, 0.2)
>>> x
array([ 0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2])
>>> np.savetxt('myarray.txt', x)
```

Load ASCII data from a file to an array:

```
>>> x2 = np.loadtxt('myarray.txt')
>>> x2
array([ 0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2])
```

ASCII format can me manipulated, see help:

```
>>> np.savetext?
```

Python In & Out: Reading and Writing ASCII Based Data Files

Write more than one timeseries column-wise to a file:

```
>>> y = np.cos(x)
>>> np.savetxt('myarray2.txt', [x, y])
>>> z = np.column_stack([x, y])
>>> z
array([[ 0. , 1. ],
[ 0.2 , 0.98006658],
[ 0.4 , 0.92106099],
[ 0.6 , 0.82533561],
[ 0.8 , 0.69670671],
[ 1. , 0.54030231],
[ 1.2 , 0.36235775]])
>>> np.savetxt('myarray3.txt', z)
```

Python Errors & Exceptions: Exceptions

```
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + muh*3
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: name 'muh' is not defined
>>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Python Errors & Exceptions: Handling Exceptions

```
>>> def divide(x, y):
...         try:
...             result = x / y
...         except ZeroDivisionError:
...             print "division by zero!"
...         except TypeError:
...             print "unsupported type!"
...         else:
...             print "result is", result
>>> divide(2, 1)
result is 2
>>> divide(2, 0)
division by zero!
>>> divide(2, 'bbb')
unsupported type!
```

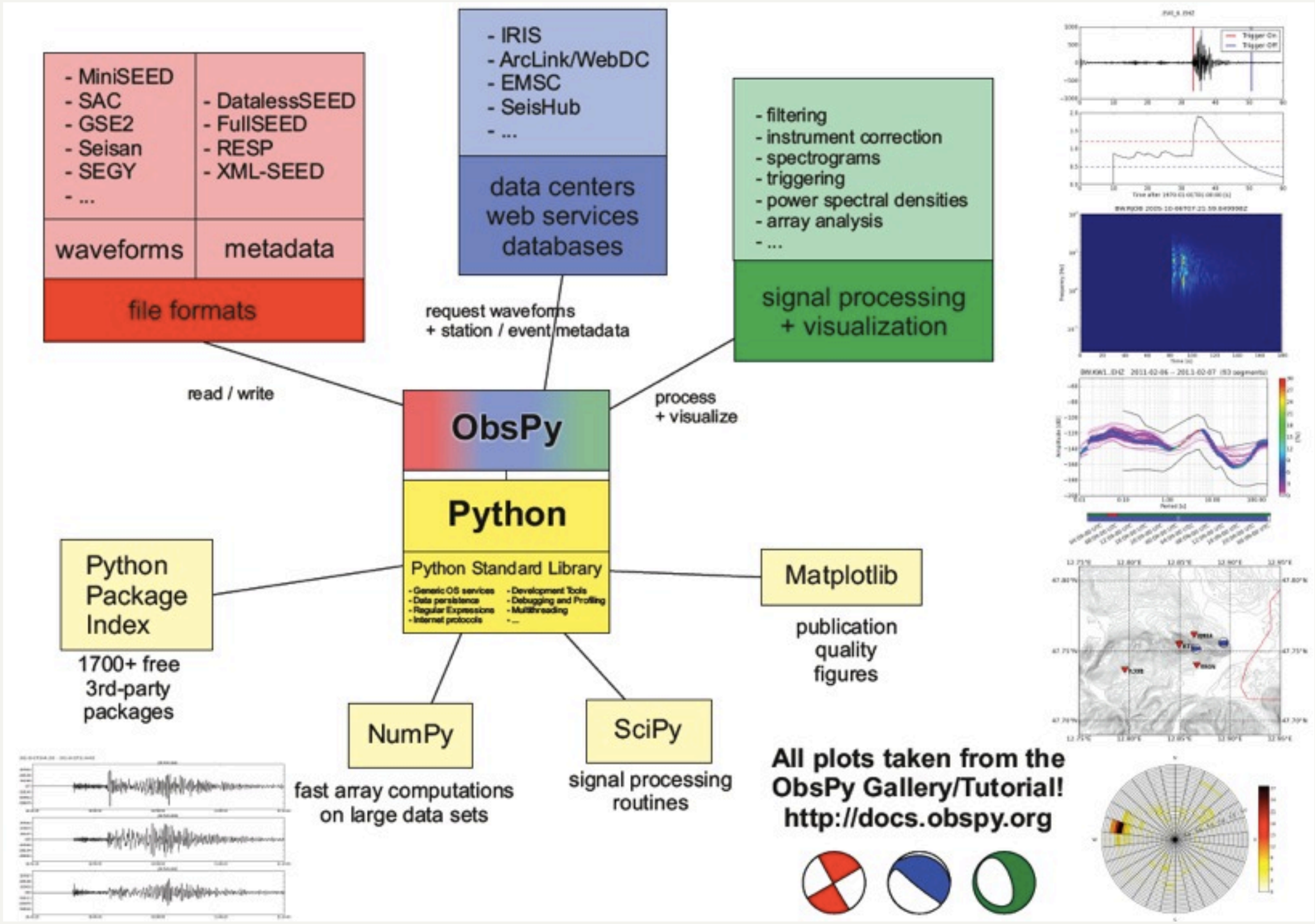
Literature:

- The Python Tutorial (http://docs.python.org/tutorial/)
- Sebastian Heimann - The Informal Python Boot Camp (http://emolch.org/pythonbootcamp.html)
- Hoyt Koepke - 10 Reasons Python Rocks for Research (http://www.stat.washington.edu/ hoytak/blog/whypython.html)
- Software Carpentry (http://software-carpentry.org/4 0/python/)
- Kent S Johnson - Python Rocks! and other rants (http://personalpages.tds.net/ kent37/stories/00020.html)
- Hans Petter Langtangen - Python Scripting for Computational Science, 3rd Edition, SpringerVerlag Berlin - Heidelberg, 2009.

ObsPy Development at the LMU - Munich

R. Barsch, T. Megies, M. Beyreuther, L. Krischer, M. Simon, J. Wassermann

ObsPy


- MiniSEED
- SAC
- GSE2
- Seisan
- SEGY
- ...
- DatalessSEED
- FullSEED
- RESP
- XML-SEED
waveforms
metadata
file formats
- IRIS
- ArcLink/WebDC
- EMSC
- SeisHub
- ...
data centers
web services
databases
request waveforms
+ station / event metadata
read / write
process
+ visualize
- filtering
- instrument correction
- spectrograms
- triggering
- power spectral densities
- array analysis
- ...
signal processing
+ visualization
ObsPy
Python
Python Standard Library
- Generic OS services
- Data persistence
- Regular Expressions
- Internet protocols
- Development Tools
- Debugging and Profiling
- Multithreading
- ...
Python
Package
Index
1700+ free
3rd-party
packages
Matplotlib
publication
quality
figures
NumPy
fast array computations
on large data sets
SciPy
signal processing
routines
All plots taken from the
ObsPy Gallery/Tutorial!
http://docs.obspy.org

ObsPy

ObsPy

ObsPy

- Python toolbox for seismologists

ObsPy

- Python toolbox for seismologists
- Goal: facilitate rapid application development for seismology (lectures, exercises, science)

ObsPy

- Python toolbox for seismologists
- Goal: facilitate rapid application development for seismology (lectures, exercises, science)
- Modular extensible architecture

ObsPy

- Python toolbox for seismologists
- Goal: facilitate rapid application development for seismology (lectures, exercises, science)
- Modular extensible architecture
- Waveform data: GSE2/GSE1, MiniSEED, SAC, SEG-Y, SEG-2, SH Q/ASCII, SEISAN, IRIS TSPAIR & SLIST, WAV

ObsPy

- Python toolbox for seismologists
- Goal: facilitate rapid application development for seismology (lectures, exercises, science)
- Modular extensible architecture
- Waveform data: GSE2/GSE1, MiniSEED, SAC, SEG-Y, SEG-2, SH Q/ASCII, SEISAN, IRIS TSPAIR & SLIST, WAV
- Inventory data: Dataless SEED, XML-SEED, RESP

ObsPy

- Python toolbox for seismologists
- Goal: facilitate rapid application development for seismology (lectures, exercises, science)
- Modular extensible architecture
- Waveform data: GSE2/GSE1, MiniSEED, SAC, SEG-Y, SEG-2, SH Q/ASCII, SEISAN, IRIS TSPAIR & SLIST, WAV
- Inventory data: Dataless SEED, XML-SEED, RESP
- Data request clients: ArcLink/WebDC, IRIS DHI/Fissures, SeisHub, IRIS Web Service, NERIES Web Service

ObsPy

- Python toolbox for seismologists
- Goal: facilitate rapid application development for seismology (lectures, exercises, science)
- Modular extensible architecture
- Waveform data: GSE2/GSE1, MiniSEED, SAC, SEG-Y, SEG-2, SH Q/ASCII, SEISAN, IRIS TSPAIR & SLIST, WAV
- Inventory data: Dataless SEED, XML-SEED, RESP
- Data request clients: ArcLink/WebDC, IRIS DHI/Fissures, SeisHub, IRIS Web Service, NERIES Web Service
- Signal processing: Filters, triggers, instrument correction, rotation, array analysis, beamforming

ObsPy

- Python toolbox for seismologists
- Goal: facilitate rapid application development for seismology (lectures, exercises, science)
- Modular extensible architecture
- Waveform data: GSE2/GSE1, MiniSEED, SAC, SEG-Y, SEG-2, SH Q/ASCII, SEISAN, IRIS TSPAIR & SLIST, WAV
- Inventory data: Dataless SEED, XML-SEED, RESP
- Data request clients: ArcLink/WebDC, IRIS DHI/Fissures, SeisHub, IRIS Web Service, NERIES Web Service
- Signal processing: Filters, triggers, instrument correction, rotation, array analysis, beamforming
- Plotting: spectrograms, beachballs and waveforms

ObsPy

- Python toolbox for seismologists
- Goal: facilitate rapid application development for seismology (lectures, exercises, science)
- Modular extensible architecture
- Waveform data: GSE2/GSE1, MiniSEED, SAC, SEG-Y, SEG-2, SH Q/ASCII, SEISAN, IRIS TSPAIR & SLIST, WAV
- Inventory data: Dataless SEED, XML-SEED, RESP
- Data request clients: ArcLink/WebDC, IRIS DHI/Fissures, SeisHub, IRIS Web Service, NERIES Web Service
- Signal processing: Filters, triggers, instrument correction, rotation, array analysis, beamforming
- Plotting: spectrograms, beachballs and waveforms
- Waveform indexer

ObsPy

- Open source (LGPL or GPL)
- 4.5 core developers
- Platform independent (Win, Mac, Linux) and tested
- Test-driven development (TDD), currently ~500 unit tests (http:/tests.obspy.org)
- Reliance on well-known third-party libraries (numpy, scipy, matplotlib)
- Reusing well established code, e.g. libmseed, GSE UTI (via ctypes or cython)
- Automated build of API documentation from source
- Binary distributions: PyPI (http://pypi.python.org), MAC OSX & Windows Installer
- Source code & community webpage containing **tutorials**, installation instruction, ticket system, mailing lists

http://www.obspy.org

ObsPy: Data Types

- ... we want to unify data from different sources in a common structure.

```
>>>st = read("file.mseed")
>>>st += read("file.sac")
>>>st += client_arclink.getWaveform(...)
>>>st += client_iris.getWaveform(...)
```

ObsPy: Data Types

- ... they know how to behave by themselves if we tell them once.

```
>>>utcdatetime + 10
>>>st += st2
>>>st.filter("lowpass", freq=1)
```

- ... there is less room for user errors.

```
>>>#st = client.getWaveform(..., channel="BHZ")
>>>st = client.getWaveform(..., channel="HHZ")
>>>data = st[0].data
>>>data = obspy.signal.lowpass(data, freq=1, df=20)
```

ObsPy: Data Types

- ... the code gets much shorter and better readable.

How about...

```
st = read("file")
from obspy.signal import lowpass
num_traces = len(st)
for i in range(num_traces):
df = st[i].stats.sampling_rate
st[i].data = lowpass(st[i].data, freq=1, df=df)
```

..compared to:

```
st = read("file")
st.filter("lowpass", freq=1)
```

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 'fresh' IPython 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 'ab': >>> 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
  - ➡ API http://docs.obspy.org/
- Python/Numpy/Scipy API
  - ➡ http://docs.python.org/
  - ➡ http://docs.scipy.org/doc/numpy/reference/
  - ➡ http://docs.scipy.org/doc/scipy/reference/

ObsPy Data Types: Overview (all in obspy.core)

- UTCDateTime
  - ➡ replacement/extension for 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
  - ➡ list-like container object
  - ➡ stores multiple traces (e.g. Z, N, E traces of one station)