The *formerly* Informal Python Boot Camp

http://emolch.org/pythonbootcamp.html

by Sebastian Heimann

Lesson 2: Modules, Classes, Regular Expressions

Modules

The slowness of Python

Object-oriented programming

Regular expressions

Things to remember from today's lesson

Further reading

Modules

Importing functionality of a module

The normal and safe aproach:

```
>>> import math
>>> math.pi
3.141592653589793l
>>> math.cos(math.pi)
-1.0
```

Not for the faint-hearted:

Importing directly into the local namespace:

```
>>> from math import *
>>> pi
3.141592653589793l
>>> cos(pi)
-1.0
```

Caution with `from module import *`

Consider different modules providing functions with the same name.

Example name collision:

```
>>> from os import *
>>> f = open( "lesson5.tex", "r" )  # fails!

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: an integer is required

# open() has been mapped to os.open() !!!
```

The builtin open() could still be accessed though:

```
>>> f = __builtins__.open("lesson5.tex","r")
```

Less typing while avoiding collisions:

Binding to local names:

```
>>> import math
>>> cos = math.cos
>>> pi = math.pi
>>> cos(pi)
-1.0
```

Import module under a different/shorter name:

```
>>> import math as m
>>> m.cos(m.pi)
-1.0
```

Import only what is needed:

```
>>> from math import pi, cos
>>> cos(pi)
-1.0
```

Writing your own module

seismo.py:

```
"""Some seismological utility functions."""

import math

def lame_parameters( alpha, beta, density ):
    """Convert seismic velocities to Lame’s parameters.

       Returns Lame’s parameters as (lambda, mu)."""

    return ( (alpha**2-2.0*beta**2)*density,
             beta**2*density )

def velocities( lambd, mu, density ):
    """Convert lame parameters to seismic velocities.

       Returns tuple with velocities (alpha, beta)."""

    return ( math.sqrt((lambd+2.0*mu)/density),
             math.sqrt(mu/density) )
```

Using your module

As with any other module:

```
>>> import seismo

>>> seismo.lame_parameters( 4000., 2100., 2600. )
(18668000000.0, 11466000000.0)

>>> seismo.velocities( *(_+(2600,)) )
(4000.0, 2100.0)
```

Help!

```
>>> import seismo
>>> help(seismo)  # makes man page from docstrings!
```

```
    Help on module seismo:

NAME
    seismo - Some seismological utility functions.

FILE
    /scratch/local1/sebastian/home-ext/bootcamp/seismo.py

FUNCTIONS
    lame_parameters(alpha, beta, density)
        Convert seismic velocities to Lame’s parameters.

        Returns Lame’s parameters as (lambda, mu).

    velocities(lambd, mu, density)
        Convert lame parameters to seismic velocities.

        Returns tuple with velocities (alpha, beta).
```

Introspection

You can look at the contents of any module:

```
>>> import seismo
>>> dir(seismo)
['__builtins__', '__doc__', '__file__', '__name__',
 'lame_parameters', 'math', 'velocities']

 # dir without argument looks at local namespace:
 >>> dir()
 ['__builtins__', '__doc__', '__name__', 'seismo']
```

Eight most essential modules

sys	`argv`, `stdin`, `stdout`, `stderr`, `exit()`
os	Directory, file and process management
shutil	Higher level copying/moving of files
math	Standard mathematical functions
random	Several random number generators
time	Time and Date manipulation, `sleep()`
re	Regular expressions
subprocess	Process management, IPC with pipes

Where to search for documentation

- `pydoc modulename`
- Python library reference
- Python library reference index
- Python global module index

The slowness of Python

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!

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!)

When speed matters

- Use specialized array math modules: numpy, scipy.
- Write or use C/Fortran extensions for time critical tasks.

We will see more on these topics in later sessions.

Numpy example

```
import os
from numpy import *

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

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

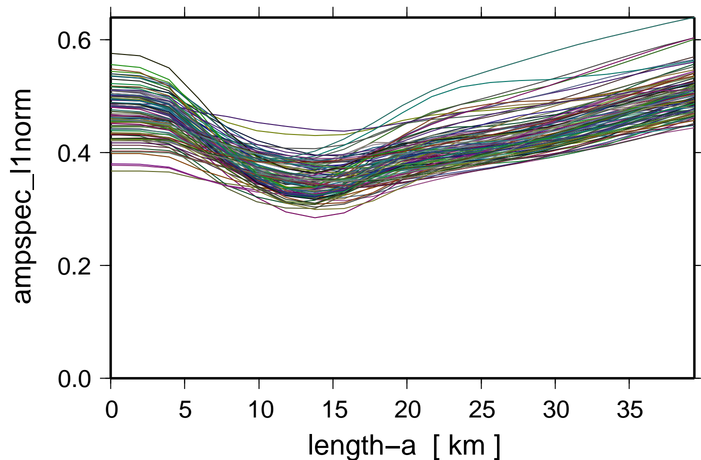
- 37 MB memory usage
- Runs about 0.3 s

The OOP paradigm

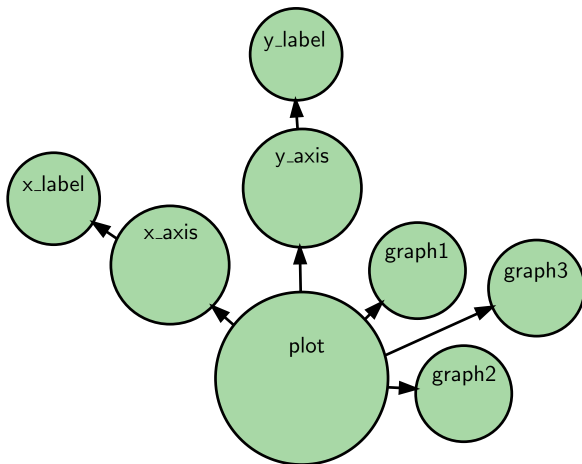
Object-oriented programming is a bunch of concepts helping to structure a (larger) program in a ”natural” way.

Typical OOP introductions in text books suck!

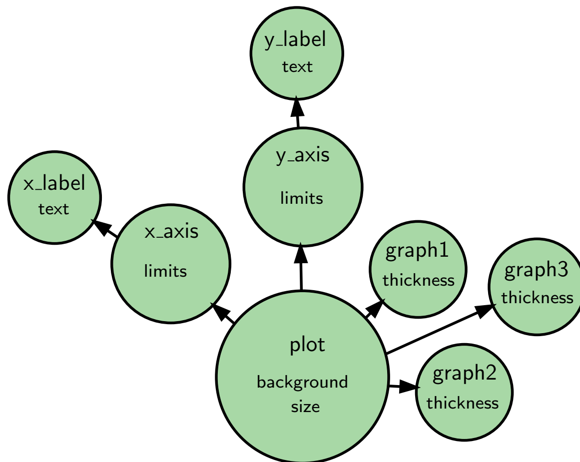
Think about a plotting program



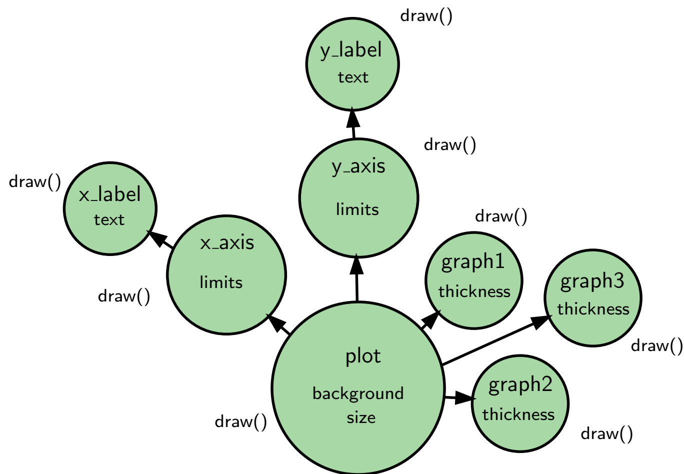
A possible subdivision into objects



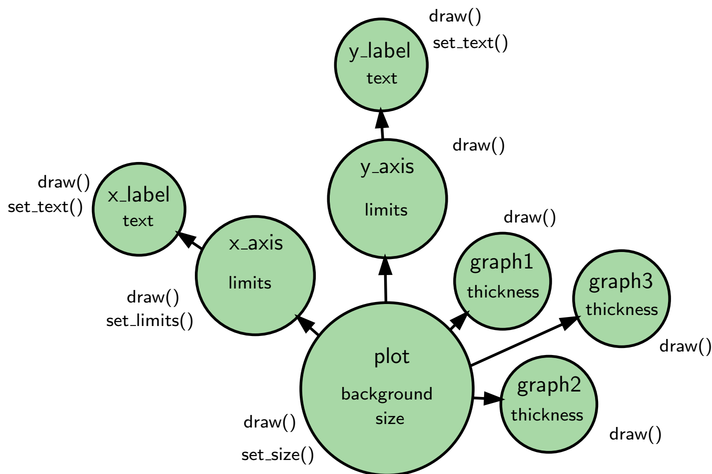
Data attributes



Common methods



Individual methods



Classes and objects

Fragmentary implementation of a label class:

```
class Label:

    def __init__(self, text):
        self.text = text
        self.position = (0,0)

    def set_text(self, new_text):
        self.text = new_text

    def draw(self, painter):
        painter.drawText(self.text, self.position)

x_label = Label('time [t]')
x_label.draw(...)
```

Classes and objects

- The `class` keyword introduces a class.
- To create an instance of the class, use function notation:

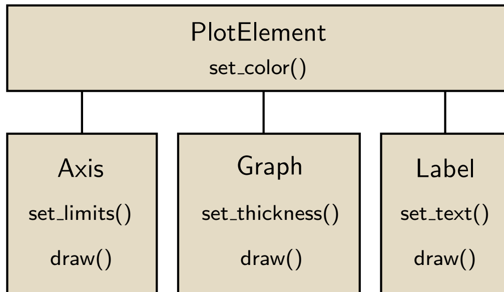
```
x_label = Label('time [t]')
```

- The `__init__()` method is invoked when an instance of the class is created.
- Class methods receive a reference to the instance as first argument.
  (By convention it is called `self`)

Classes and objects

- An *instance object* is an entity encapsulating state (*data attributes*) and behaviour (*methods*).
- A *class* is the blueprint from which individual objects (*instances*) are created.

Inheritance




```
x_axis.set_limits(...)
x_axis.set_color(...)
...
for element in [ x_axis, y_axis, graph1, ... ]:
    element.set_color(...)
    element.draw()
    element.set_limits()   # <= won't work!
```

Inheritance

```
class PlotElement:
    def __init__(self, color='black'):
        self.color=color

    def set_color(self,new_color):
        self.color = new_color

class Graph(PlotElement):
    def __init__(self):
        PlotElement.__init__( self, 'red' )

    def draw(self, painter):
        ...

graph1 = Graph()
graph1.set_color('blue')
graph1.draw(...)
```

Object orientation in Python for C++ programmers

In C++ terminology, in Python

- all class methods and data attributes are *public*.
- all methods are *virtual*.
- built-in types can be used as base classes (unlike in C++).
- most operators may be overloaded.
- multiple inheritance is supported (unlike in Java).
- dispatching is always done at runtime.

1. Download and unpack exercise material

```
> wget http://emolch.org/material.tar.gz
> tar -xzvf material.tar.gz
> cd material
> ls
circler.py  example1.py ... thingy.py
```

The package contains a python module `thingy.py`. This module contains two classes: `Thingy` and `Playground`.

3. Try out interactively with the python interpreter

Watch beamer screen for something to happen ;-)

```
# import classes Playground and
# Thingy from module thingy:
>>> from thingy import Playground, Thingy

# connect to the playground:
>>> p = Playground('http://wegener:8778',
...                loginname='yourname')

# create an instance of class Thingy:
>>> t = Thingy(p,'name-of-your-thingy')

# modify the thingy:
>>> t.set_color(1,0,0)
>>> t.set_speed(1.0)
>>> t.set_pensize(5)
```

4. Find out, what else you can do with the thingy

```
> pydoc thingy
# or look at the implementation:
> less thingy.py
```

... or from within the interpreter...

```
>>> import thingy
>>> help(thingy)
```

5. Let your thingy do something impressive

- Look at and try out the example scripts provided.
- You can create more than one thingy.
- Every thingy must have a unique name.
- `time.sleep(0.05)` pauses execution 0.05 s.
- `random.random()` returns float in range $[0, 1[$.
- `random.randint(1,5)` ret. int in range $[1, 5]$.
- `math.pi` is π.
- `math.atan2(y,x)` is $\arctan \frac{y}{x}$ with correct sign.

6. Inheritance

- The file `circler.py` contains an inheritance example.
- Try to understand what it does.
- Now write your own thingy enhancement by subclassing `Thingy`.

Exercise: Creating and using objects

Download and unpack:

`http://emolch.org/material.tar.gz`

```
# import classes Playground and Thingy from
# module thingy:

>>> from thingy import Playground, Thingy

# connect to the playground:
>>> p = Playground('http://wegener:8778',
...                      loginname='yourname')

# create an instance of class Thingy:
>>> t = Thingy(p,'name-of-thingy')

# modify the thingy:
>>> t.set_color(1,0,0)
>>> help(Thingy)
```

Exercise: Inheritance

```
from time import sleep
from thingy import Playground, Thingy

class Circler(Thingy):
    def __init__(self, p, name, turn_angle):
        Thingy.__init__(self, p, name)
        self.turn_angle = turn_angle
        self.set_color(1,1,1)
        self.set_pensize(2)
        self.set_speed(0.3)

    def update(self):
        self.add_direction(self.turn_angle)
```

...continued...

```
p = Playground('http://wegener:8778',
                   loginname='yourname')

# create a list with two circlers
thingys = [ Circler(p, 'heinz', 1),
            Circler(p, 'einstein', 0.5) ]

for i in range(1000):
    for t in thingys:
        t.update()

    sleep(0.05)
```

Regular expressions

```
[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
```

Introduction

Regular expressions are a powerful tool to do tasks like:

- Extract values from strings like `'lat=22.5, lon=5'`.
- Remove all HTML formatting from a web page.
- Strip off any filename extensions from a list of filenames.
- Extract all section names from a LaTeX file.
- Check user input to be in a specific format.
- Replace all whitespace sequences in a text with single spaces.

Use them, where string methods like `string.find()` *or* `string.replace()` *don't offer enough flexibility.*

Introductory example

Extract numbers from a string:

```
>>> import re
>>> p = re.compile(r'\d+')
>>> p.findall("""12 drummers drumming,
                 11 pipers piping,
                 10 lords a-leaping""")
['12', '11', '10']
```

String pattern matching with regular expressions

Regular expressions are in very wide use:

- Unix tools: awk, sed, grep, ...
- Editors: emacs, vi, nedit, kate, ...
- Programming languages: perl, ...
- Regex libraries exist for almost any programming language.

String pattern matching with regular expressions

Regex Trivia

- Regular expressions are written in their own language.
- Dialects differ slightly.
- Most newer tools use `perl` style regular expressions.
- Documentation:
  `http://perldoc.perl.org/perlre.html`
- Book: *Mastering Regular Expressions - by Jeffrey E. F. Friedl*

String pattern matching with regular expressions

Regular expressions in Python

- Python uses `perl` style regular expressions.
- Regular expression are provided through the `re` module.
- Python specific HOWTO:
  `http://www.amk.ca/python/howto/regex/`

Simple patterns

Matching characters:

```
>>> text = "Currywurst, Bratwurst, Wurst"
>>> re.findall( r'Brat', text )
['Brat']
>>> re.findall( r'Tofu', text )
[]
```

Simple patterns

These characters have a special meaning for the regex:

```
. ^ $ * + ? { [ ] \ | ( )
```

Yes, also the dot!

They must be backslash-escaped when searching for them:

```
>>> text = "lesson1.tex lesson1.pdf"
>>> re.findall( r'\.tex', text )
['.tex']
```

Character classes

- Brackets match any of the enclosed characters.

Example 1: `[12]` matches `1` as well as `2`.

```
>>> text = "lesson1.tex lesson2.tex"
>>> re.findall( r'lesson[12]', text )
['lesson1', 'lesson2']
```

Example 2: `[t-z]` matches any character between 't' and 'z'.

```
>>> text = "lesson1.tex lesson2.tex"
>>> re.findall( r'[t-z]', text )
['t', 'x', 't', 'x']
```

Character classes

- Use `[^...]` to match any characters not in the class.

Example: `[^a-z]` matches any non-lowercase letters:

```
>>> text = "lesson1.tex lesson2.tex"
>>> re.findall( r'[^a-z]', text )
['1', '.', ' ', '2', '.']
```

Character class shortcuts

- `\d` Matches any decimal digit; equivalent with `[0-9]`.
- `\D` Matches any non-digit character; `[^0-9]`.
- `\s` Matches any whitespace character; `[ \t\n\r\f\v]`.
- `\S` Matches any non-whitespace character; `[^ \t\n\r\f\v]`.
- `\w` Matches any alphanumeric character; `[a-zA-Z0-9_]`.
- `\W` Matches any non-alphanumeric character; `[^a-zA-Z0-9_]`.

Character class shortcuts

- The dot `.` matches any character.

Repeating things

- Match it one or more times by appending a `+` to it.
- Match it zero or more times by appending a `*` to it.

```
>>> text = "width=800, height=600"
>>> re.findall( r'\d+', text )
['600', '800']
```

- Match it between `n` and `m` times by appending `{n,m}` to it.

Optional parts of a pattern

- Use parantheses to group a part of a pattern.
- Append a questionmark to an optional part of a pattern.

Example: `r'((curry)?brat)?wurst'` matches any of `'wurst'`, `'bratwurst'` and `'currybratwurst'`.

By default patterns are *greedy*, so the longest possibe match will win.

Alternation

- Use the "or" operator | to specify alternatives.

Example: `r'(curry|brat)wurst'`
matches `'bratwurst'` as well as `'currywurst'`.

Anchors

- `^` matches the beginning of the string.
- `$` matches the end of the string.

Example:

```
>>> text = 'From Here to Eternity'
>>> re.findall( r'^From', text )
['From']

>>> text = 'Reciting From Memory'
>>> re.findall( r'^From', text )
[]
```

Compiling regular expressions

```
>>> import re
>>> re.findall(r'\d+', """12 drummers drumming,
                            11 pipers piping,
                            10 lords a-leaping""")
['12', '11', '10']
```

has the same effect as

```
>>> import re
>>> p = re.compile(r'\d+')
>>> p.findall("""12 drummers drumming,
                 11 pipers piping,
                 10 lords a-leaping""")
['12', '11', '10']
```

but the latter may be faster, if evaluated several times.

Main functions provided by the re module

Overview

`re.findall(...)`	returns all matches as a list of strings.
`re.search(...)`	returns a *match object* for first match.
`re.finditer(...)`	returns *match objects* for each match.
`re.split(...)`	splits string where the pattern matches.
`re.sub(...)`	replaces matching substrings.

Also available as:

- methods of objects returned by `re.compile(...)`

```
>>> re.findall(r'\d+', "lat=12, lon=11")
['12', '11']

>>> compiled_pattern = re.compile(r'\d+')
>>> compiled_pattern.findall("lat=12, lon=11")
['12', '11']
```

Case insensitive search

- Most re functions take an additional flags argument.
- The most useful flag is re.I (= re.IGNORECASE).

```
>>> re.findall(r'so', 'Mal so mal So')
['so']
>>> re.findall(r'so', 'Mal so mal So', re.I)
['so', 'So']
```

Capturing parts of a match

Use parantheses to capture parts of a match:

```
>>> line = 'event: 2007/10/18  03:19:55  M=5.5'

>>> m = re.search(r'(\d+):(\d+):(\d+)', line)

>>> int(m.group(1))
3
>>> int(m.group(2))
19
>>> int(m.group(3))
55
```

Capturing parts of a match

Nested groups

```
>>> line = 'event: 2007/10/18  03:19:55  M=5.5'

>>> m = re.search(r'((\d+:\d+):(\d+))', line)
# group number:      (------- 1 -----)
#                     (---2---)
#                               (-3-)

>>> m.group(1)
'03:19:55'
>>> m.group(2)
'03:19'
>>> m.group(3)
'55'
```

Count opening parantheses for group number.

Replacing: `sub(pattern, replacement, str)`

Convert DOS and Mac line endings to UNIX line feeds:

```
>>> text = re.sub(r'\r\n?', '\n', alien_text)
```

Remove any filename suffix:

```
>>> re.sub(r'\.[^.]+$', '', 'lesson5.tex')
'lesson5'
```

Replace multiple whitespace with single spaces:

```
>>> re.sub(r'\s+', ' ', """Hooray
                                     for \t  Dvorak!""")
'Hooray for Dvorak!'
```

Matching valid floating point numbers

```
22
5.202
+1.302
-5.2
.123
2e-3
2.E-3
```

Use this as a recipe:

```
[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
```

```
METACHARS                      FLAGS
^      string begin            re.I case insens.
$      str. end (before \n) re.M line based ^$
+      one or more             re.S . includes \n
*      zero or more            re.X ign. wh.space
?      zero or one             re.L locale aware \w,\b,\s
{3,7} repeat in range
()     capture               CHARCLASSES
(?:)  no capture             .  == [^\n]
[]     character class       \s == [ \t\n\r\f\v]
|      alternation           \w == [A-Za-z0-9_]
\b     word boundary         \d == [0-9]
\Z     string end            \S, \W and \D negate

FLOATS: [-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?

FUNCTIONS  p=pattern, s=string, f=flags, c=count
re.findall(p,s[,f])   returns list of strings
re.search(p,s[,f])    returns first match as match obj.
re.finditer(p,s,[,f]) iterates over match objects
re.split(p,s,[,c])    returns list of strings
re.sub(p,r,s[,c])     replaces matching substrings
re.escape(s)          escape regex metacharacters
```

Things to remember from today's lesson

- `help()`, `dir()` and `pydoc module`
- Python itself is slow.
- Python regex HOWTO:
  `http://www.amk.ca/python/howto/regex/`

Keep these pages open, when programming in Python:

- Tutorial:
  `http://docs.python.org/tut/`
- Library Reference:
  `http://docs.python.org/lib/lib.html`

Further reading

- Good introduction to ”Pythonic” programming
  `http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html`
- Coding conventions for Python
  `http://www.python.org/dev/peps/pep-0008/`
- Alex Martelli giving a Google tech talk ”Python for programmers”
  `http://www.youtube.com/watch?v=ujkzfC2lebA`