Slides from INF3331 lectures

Ola Skavhaug and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2009

About this course

Teachers

- Ola Skavhaug
- Joakim Sundnes
- We use Python to create efficient working (or problem solving) environments
- We also use Python to develop large-scale simulation software (which solves partial differential equations)
- We believe high-level languages such as Python constitute a promising way of making flexible and user-friendly software!
- Some of our research migrates into this course
- There are lots of opportunities for master projects related to this course

Contents

- Scripting in general
- Quick Python introduction (first two weeks)
- Python problem solving
- More advanced Python (class programming++)
- Regular expressions
- Combining Python with C, C++ and Fortran
- The Python C API and the NumPy C API
- Distributing Python modules (incl. extension modules)
- Verifying/testing (Python) software
- Documenting Python software
- Optimizing Python code
- Python coding standards and 'Pythonic' programming
- Basic Bash programming

What you will learn

- Scripting in general, but with most examples taken from scientific computing
- Jump into useful scripts and dissect the code
- Learning by doing
- Find examples, look up man pages, Web docs and textbooks on demand
- Get the overview
- Customize existing code
- Have fun and work with useful things

Teaching material

- Slides from lectures
  (by H. P. Langtangen and O. Skavhaug et al), download from

  `http://www.uio.no/studier/emner/matnat/ifi/INF3331/h09/`

- Associated book (for the Python material):
  H. P. Langtangen: *Python Scripting for Computational Science*, 2nd edition, Springer 2005

- You must find the rest: manuals, textbooks, google

- Good Python litterature:
  Harms and McDonald: The Quick Python Book (tutorial+advanced)
  Beazley: Python Essential Reference
  Grayson: Python and Tkinter Programming

What is a script?

- Very high-level, often short, program
  written in a high-level scripting language
- Scripting languages: Unix shells, Tcl, Perl, Python, Ruby, Scheme, Rexx, JavaScript, VisualBasic, ...
- This course: Python
  + a taste of Bash (Unix shell)

Characteristics of a script

- Glue other programs together
- Extensive text processing
- File and directory manipulation
- Often special-purpose code
- Many small interacting scripts may yield a big system
- Perhaps a special-purpose GUI on top
- Portable across Unix, Windows, Mac
- Interpreted program (no compilation+linking)

Why not stick to Java or C/C++?

Features of scripting languages compared with Java, C/C++ and Fortran:

- shorter, more high-level programs
- much faster software development
- more convenient programming
- you feel more productive

Two main reasons:

- no variable declarations,
  but lots of consistency checks at run time
- lots of standardized libraries and tools

Scripts yield short code (1)

- Consider reading real numbers from a file, where each line can contain an arbitrary number of real numbers:

```
1.1   9   5.2
1.762543E-02
0 0.01 0.001

9 3 7
```

- Python solution:

```
F = open(filename, 'r')
n = F.read().split()
```

Using regular expressions (1)

- Suppose we want to read complex numbers written as text

  `(-3, 1.4)   or   (-1.437625E-9, 7.11)   or   (  4, 2 )`

- Python solution:

```
m = re.search(r'\(\s*([^,]+)\s*,\s*([^,]+)\s*\)',
              '(-3,1.4)')
re, im = [float(x) for x in m.groups()]
```

Using regular expressions (2)

- Regular expressions like

  ```
  \(\s*([^,]+)\s*,\s*([^,]+)\s*\)
  ```

  constitute a powerful language for specifying text patterns
- Doing the same thing, without regular expressions, in Fortran and C requires quite some low-level code at the character array level
- Remark: we could read pairs (-3, 1.4) without using regular expressions,

  ```
  s = '(-3,  1.4 )'
  re, im = s[1:-1].split(',')
  ```

Script variables are not declared

- Example of a Python function:

```
def debug(leading_text, variable):
    if os.environ.get('MYDEBUG', '0') == '1':
        print leading_text, variable
```

- Dumps any printable variable
  (number, list, hash, heterogeneous structure)
- Printing can be turned on/off by setting the environment variable `MYDEBUG`

The same function in C++

- Templates can be used to mimic dynamically typed languages
- Not as quick and convenient programming:

```
template <class T>
void debug(std::ostream& o,
           const std::string& leading_text,
           const T& variable)
{
  char* c = getenv("MYDEBUG");
  bool defined = false;
  if (c != NULL) {  // if MYDEBUG is defined ...
    if (std::string(c) == "1") {  // if MYDEBUG is true ...
      defined = true;
    }
  }
  if (defined) {
    o <<  leading_text << " " << variable << std::endl;
  }
}
```

The relation to OOP

- Object-oriented programming can also be used to parameterize types
- Introduce base class `A` and a range of subclasses, all with a (virtual) print function
- Let `debug` work with `var` as an `A` reference
- Now `debug` works for all subclasses of `A`
- Advantage: complete control of the legal variable types that `debug` are allowed to print (may be important in big systems to ensure that a function can allow make transactions with certain objects)
- Disadvantage: much more work, much more code, less reuse of `debug` in new occasions

Flexible function interfaces

- User-friendly environments (Matlab, Maple, Mathematica, S-Plus, ...) allow flexible function interfaces
- Novice user:

```
# f is some data
plot(f)
```

- More control of the plot:

```
plot(f, label='f', xrange=[0,10])
```

- More fine-tuning:

```
plot(f, label='f', xrange=[0,10], title='f demo',
     linetype='dashed', linecolor='red')
```

Keyword arguments

- Keyword arguments = function arguments with keywords and default values, e.g.,

```
def plot(data, label='', xrange=None, title='',
         linetype='solid', linecolor='black', ...)
```

- The sequence and number of arguments in the call can be chosen by the user

Classification of languages (1)

- Many criteria can be used to classify computer languages
- Dynamically vs statically typed languages
  Python (dynamic):

```
c = 1               # c is an integer
c = [1,2,3]         # c is a list
```

C (static):

```
double c; c = 5.2;    # c can only hold doubles
c = "a string..."     # compiler error
```

Classification of languages (2)

- Weakly vs strongly typed languages
  Perl (weak):

```
$b = '1.2'
$c = 5*$b;    # implicit type conversion: '1.2' -> 1.2
```

  Python (strong):

```
b = '1.2'
c = 5*b       # illegal; no implicit type conversion
```

Classification of languages (3)

- Interpreted vs compiled languages
- Dynamically vs statically typed (or type-safe) languages
- High-level vs low-level languages (Python-C)
- Very high-level vs high-level languages (Python-C)
- Scripting vs system languages

Turning files into code (1)

- Code can be constructed and executed at run-time
- Consider an input file with the syntax

```
a = 1.2
no of iterations = 100
solution strategy = 'implicit'
c1 = 0
c2 = 0.1
A = 4
c3 = StringFunction('A*sin(x)')
```

- How can we read this file and define variables `a`, `no_of_iterations`, `solution_strategi`, `c1`, `c2`, `A` with the specified values?
- And can we make `c3` a function `c3(x)` as specified?

Yes!

Turning files into code (2)

- The answer lies in this short and generic code:

```
file = open('inputfile.dat', 'r')
for line in file:
    # first replace blanks on the left-hand side of = by _
    variable, value = line.split('=').strip()
    variable = re.sub(' ', '_', variable)
    exec(variable + '=' + value)   # magic...
```

- This cannot be done in Fortran, C, C++ or Java!

Scripts can be slow

- Perl and Python scripts are first compiled to byte-code
- The byte-code is then *interpreted*
- Text processing is usually as fast as in C
- Loops over large data structures might be very slow

```
for i in range(len(A)):
    A[i] = ...
```

- Fortran, C and C++ compilers are good at optimizing such loops at compile time and produce very efficient assembly code (e.g. 100 times faster)
- Fortunately, long loops in scripts can easily be migrated to Fortran or C

Scripts may be fast enough (1)

Read 100 000 (x,y) data from file and
write (x,f(y)) out again

- Pure Python: 4s
- Pure Perl: 3s
- Pure Tcl: 11s
- Pure C (fscanf/fprintf): 1s
- Pure C++ (iostream): 3.6s
- Pure C++ (buffered streams): 2.5s
- Numerical Python modules: 2.2s (!)
- Remark: in practice, 100 000 data points are written and read in binary format, resulting in much smaller differences

Scripts may be fast enough (2)

Read a text in a human language and generate random nonsense text in that language (from "The Practice of Programming" by B. W. Kernighan and R. Pike, 1999):

Language	CPU-time	lines of code
C	0.30	150
Java	9.2	105
C++ (STL-deque)	11.2	70
C++ (STL-list)	1.5	70
Awk	2.1	20
Perl	1.0	18

Machine: Pentium II running Windows NT

When scripting is convenient (1)

- The application's main task is to connect together existing components
- The application includes a graphical user interface
- The application performs extensive string/text manipulation
- The design of the application code is expected to change significantly
- CPU-time intensive parts can be migrated to C/C++ or Fortran

When scripting is convenient (2)

- The application can be made short if it operates heavily on list or hash structures
- The application is supposed to communicate with Web servers
- The application should run without modifications on Unix, Windows, and Macintosh computers, also when a GUI is included

When to use C, C++, Java, Fortran

- Does the application implement complicated algorithms and data structures?
- Does the application manipulate large datasets so that execution speed is critical?
- Are the application's functions well-defined and changing slowly?
- Will type-safe languages be an advantage, e.g., in large development teams?

Some personal applications of scripting

- Get the power of Unix also in non-Unix environments
- Automate manual interaction with the computer
- Customize your own working environment and become more efficient
- Increase the reliability of your work
  (what you did is documented in the script)
- Have more fun!

Some business applications of scripting

- Python and Perl are very popular in the open source movement and Linux environments
- Python, Perl and PHP are widely used for creating Web services (Django, SOAP, Plone)
- Python and Perl (and Tcl) replace 'home-made' (application-specific) scripting interfaces
- Many companies want candidates with Python experience

What about mission-critical operations?

- Scripting languages are free
- What about companies that do mission-critical operations?
- Can we use Python when sending a man to Mars?
- Who is responsible for the quality of products?

The reliability of scripting tools

- Scripting languages are developed as a world-wide collaboration of volunteers (open source model)
- The open source community as a whole is responsible for the quality
- There is a single repository for the source codes (plus mirror sites)
- This source is read, tested and controlled by a very large number of people (and experts)
- The reliability of *large* open source projects like Linux, Python, and Perl appears to be very good - at least as good as commercial software

Practical problem solving

- Problem: you are not an expert (yet)
- Where to find detailed info, and how to understand it?
- The efficient programmer navigates quickly in the jungle of textbooks, man pages, README files, source code examples, Web sites, news groups, ... and has a gut feeling for what to look for
- The aim of the course is to improve your practical problem-solving abilities
- *You think you know when you learn, are more sure when you can write, even more when you can teach, but certain when you can program* (Alan Perlis)

Basic Python Constructs

First encounter with Python

```
#!/usr/bin/env python

from math import sin
import sys

x = float(sys.argv[1])
print "Hello world, sin(%g) = %g." % (x, sin(x))
```

Running the Script

Code in file `hw.py`.
Run with command:

```
> python hw.py 0.5
Hello world, sin(0.5) = 0.479426.
```

Linux alternative if file is executable (`chmod a+x hw.py`):

```
> ./hw.py 0.5
Hello world, sin(0.5) = 0.479426.
```

Quick Run Through

On *nix; find out what kind of script language (interpreter) to use:

```
#!/usr/bin/env python
```

Access library functions:

```
from math import sin
import sys
```

Read command line argument and convert it to a floating point:

```
x = float(sys.argv[1])
```

Print out the result using a format string:

```
print "Hello world, sin(%g) = %g." % (x, sin(x))
```

Simple Assignments

```
a = 10 # a is a variable referencing an
       # integer object of value 10

b = True # b is a boolean variable

a = b # a is now a boolean as well
      # (referencing the same object as b)

b = increment(4) # b is the value returned by a function

is_equal = a == b # is_equal is True if a == b
```

Simple control structures

- Loops:

```
while condition:
    <block of statements>
```

Here, `condition` must be a boolean expression (or have a boolean interpretation), for example: `i < 10` or `!found`

```
for element in somelist:
    <block of statements>
```

Note that `element` is a copy of the list items, not a reference into the list!

- Conditionals:

```
if condition:
    <block of statements>
elif condition:
    <block of statements>
else:
    <block of statements>
```

Ranges and Loops

- `range(start, stop, increment)` constructs a list. Typically, it is used in for loops:

```
for i in range(10):
    print i
```

- `xrange(start, stop, increment)` is better for fat loops since it constructs an iterator:

```
for i in xrange(10000000):
    sum += sin(i*pi*x)
```

- Looping over lists can be done in several ways:

```
names = ["Ola", "Per", "Kari"]
surnames = ["Olsen", "Pettersen", "Bremnes"]
for name, surname in zip(names, surnames):
    print name, surname # join element by element

for i, name in enumerate(names):
    print i, name # join list index and item
```

Lists and Tuples

```
mylist  = ['a string', 2.5, 6, 'another string']
mytuple = ('a string', 2.5, 6, 'another string')
mylist[1]  = -10
mylist.append('a third string')
mytuple[1] = -10  # illegal: cannot change a tuple
```

A tuple is a constant list (immutable)

List functionality

`a = []`	initialize an empty list
`a = [1, 4.4, 'run.py']`	initialize a list
`a.append(elem)`	add `elem` object to the end
`a + [1,3]`	add two lists
`a[3]`	index a list element
`a[-1]`	get last list element
`a[1:3]`	slice: copy data to sublist (here: index 1, 2)
`del a[3]`	delete an element (index `3`)
`a.remove(4.4)`	remove an element (with value `4.4`)
`a.index('run.py')`	find index corresponding to an element's value
`'run.py' in a`	test if a value is contained in the list

More list functionality

`a.count(v)`	count how many elements that have the value `v`
`len(a)`	number of elements in list `a`
`min(a)`	the smallest element in `a`
`max(a)`	the largest element in `a`
`min(["001", 100])`	tricky!
`sum(a)`	add all elements in `a`
`a.sort()`	sort list `a` (changes `a`)
`as = sorted(a)`	sort list `a` (return new list)
`a.reverse()`	reverse list `a` (changes `a`)
`b[3][0][2]`	nested list indexing
`isinstance(a, list)`	is `True` if `a` is a list

Functions and arguments

- User-defined functions:

```
def split(string, char):
    position = string.find(char)
    if  position > 0:
        return string[:position+1], string[position+1:]
    else:
        return string, ""

# function call:
message = "Heisann"
print split(message, "i")
```

  prints out `('Hei', 'sann')`.

- Positional arguments must appear before keyword arguments:

```
def split(message, char="i"):
    [...]
```

How to find more Python information

- The book contains only fragments of the Python language (intended for real beginners!)
- These slides are even briefer
- Therefore you will need to look up more Python information
- Primary reference: The official Python documentation at `docs.python.org`
- Very useful: The Python Library Reference, especially the index
- Example: what can I find in the `math` module? Go to the Python Library Reference index, find "math", click on the link and you get to a description of the module
- Alternative: `pydoc math` in the terminal window (briefer)
- Note: for a newbie it is difficult to read manuals (intended for experts) – you will need a lot of training; just browse, don't read everything, try to dig out the key info

eval and exec

- Evaluating string expressions with `eval`:

```
>>> x = 20
>>> r = eval('x + 1.1')
>>> r
21.1
>>> type(r)
<type 'float'>
```

- Executing strings with Python code, using `exec`:

```
exec("""
def f(x):
    return %s
""" % sys.argv[1])
```

Exceptions

- Handling exceptions:

```
try:
    <statements>
except ExceptionType1:
    <provide a remedy for ExceptionType1 errors>
except ExceptionType2, ExceptionType3, ExceptionType4:
    <provide a remedy for three other types of errors>
except:
    <provide a remedy for any other errors>
...
```

- Raising exceptions:

```
if z < 0:
    raise ValueError\
          ('z=%s is negative - cannot do log(z)' % z)
a = math.log(z)
```

File reading and writing

- Reading a file:

```
infile = open(filename, 'r')
for line in infile:
    # process line

lines = infile.readlines()
for line in lines:
    # process line

for i in xrange(len(lines)):
    # process lines[i] and perhaps next line lines[i+1]

fstr = infile.read()
# process the while file as a string fstr

infile.close()
```

- Writing a file:

```
outfile = open(filename, 'w')    # new file or overwrite
outfile = open(filename, 'a')    # append to existing file
outfile.write("""Some string
....
""")
```

Dictionary functionality

`a = {}`	initialize an empty dictionary
`a = {'point':[2,7], 'value':3}`	initialize a dictionary
`a = dict(point=[2,7], value=3)`	initialize a dictionary
`a['hide'] = True`	add new key-value pair to a dictionary
`a['point']`	get value corresponding to key `point`
`'value' in a`	`True` if `value` is a key in the dictionary
`del a['point']`	delete a key-value pair from the dictionary
`a.keys()`	list of keys
`a.values()`	list of values
`len(a)`	number of key-value pairs in dictionary `a`
`for key in a:`	loop over keys in unknown order
`for key in sorted(a.keys()):`	loop over keys in alphabetic order
`isinstance(a, dict)`	is `True` if `a` is a dictionary

String operations

```
s = 'Berlin: 18.4 C at 4 pm'
s[8:17]          # extract substring
s.find(':')      # index where first ':' is found
s.split(':')     # split into substrings
s.split()        # split wrt whitespace
'Berlin' in s    # test if substring is in s
s.replace('18.4', '20')
s.lower()        # lower case letters only
s.upper()        # upper case letters only
s.split()[4].isdigit()
s.strip()        # remove leading/trailing blanks
', '.join(list_of_words)
```

Modules

Import module as namespace:

```
import sys
x = float(sys.argv[1])
```

Import module member `argv` into current namespace:

```
from sys import argv
x = float(argv[1])
```

Import everything from sys into current namespace (evil)

```
from sys import *
x = float(argv[1])
```

Import `argv` into current namespace under an alias

```
from sys import argv as a
x = float(a[1])
```

Frequently encountered tasks in Python

Overview

- file globbing, testing file types
- copying and renaming files, creating and moving to directories, creating directory paths, removing files and directories
- directory tree traversal
- parsing command-line arguments
- running an application
- file reading and writing
- list and dictionary operations
- splitting and joining text
- basics of Python classes
- writing functions

Python programming information

Man-page oriented information:

- `pydoc somemodule.somefunc`, `pydoc somemodule`
- `doc.html`! Links to lots of electronic information
- The Python Library Reference (go to the index)
- Python in a Nutshell
- Beazley's Python reference book
- Your favorite Python language book
- Google

These slides (and exercises) are closely linked to the "Python scripting for computational science" book, ch. 3 and 8

File globbing

- List all .ps and .gif files (Unix):

```
ls *.ps *.gif
```

- Cross-platform way to do it in Python:

```
import glob
filelist = glob.glob('*.ps') + glob.glob('*.gif')
```

This is referred to as file globbing

Testing file types

```
import os.path
print myfile,

if os.path.isfile(myfile):
    print 'is a plain file'
if os.path.isdir(myfile):
    print 'is a directory'
if os.path.islink(myfile):
    print 'is a link'

# the size and age:
size = os.path.getsize(myfile)
time_of_last_access       = os.path.getatime(myfile)
time_of_last_modification = os.path.getmtime(myfile)

# times are measured in seconds since 1970.01.01
days_since_last_access = \
(time.time() - os.path.getatime(myfile))/(3600*24)
```

More detailed file info

```
import stat

myfile_stat = os.stat(myfile)
filesize = myfile_stat[stat.ST_SIZE]
mode = myfile_stat[stat.ST_MODE]
if stat.S_ISREG(mode):
    print '%(myfile)s is a regular file '\
          'with %(filesize)d bytes' % vars()
```

Check out the `stat` module in Python Library Reference

Copy, rename and remove files

- Copy a file:

```
import shutil
shutil.copy(myfile, tmpfile)
```

- Rename a file:

```
os.rename(myfile, 'tmp.1')
```

- Remove a file:

```
os.remove('mydata')
# or os.unlink('mydata')
```

Path construction

- Cross-platform construction of file paths:

```
filename = os.path.join(os.pardir, 'src', 'lib')

# Unix:    ../src/lib
# Windows: ..\src\lib

shutil.copy(filename, os.curdir)

# Unix:  cp ../src/lib .

# os.pardir : ..
# os.curdir : .
```

Directory management

- Creating and moving to directories:

```
dirname = 'mynewdir'
if not os.path.isdir(dirname):
    os.mkdir(dirname) # or os.mkdir(dirname,'0755')
os.chdir(dirname)
```

- Make complete directory path with intermediate directories:

```
path = os.path.join(os.environ['HOME'],'py','src')
os.makedirs(path)

# Unix: mkdirhier $HOME/py/src
```

- Remove a non-empty directory tree:

```
shutil.rmtree('myroot')
```

Basename/directory of a path

- Given a path, e.g.,

```
fname = '/home/hpl/scripting/python/intro/hw.py'
```

- Extract directory and basename:

```
# basename: hw.py
basename = os.path.basename(fname)

# dirname: /home/hpl/scripting/python/intro
dirname  = os.path.dirname(fname)

# or
dirname, basename = os.path.split(fname)
```

- Extract suffix:

```
root, suffix = os.path.splitext(fname)
# suffix: .py
```

Platform-dependent operations

- The operating system interface in Python is the same on Unix, Windows and Mac
- Sometimes you need to perform platform-specific operations, but how can you make a portable script?

```
# os.name         : operating system name
# sys.platform    : platform identifier

# cmd:  string holding command to be run
if os.name == 'posix':                # Unix?
    failure, output = commands.getstatusoutput(cmd + '&')
elif sys.platform[:3] == 'win':       # Windows?
    failure, output = commands.getstatusoutput('start ' + cmd)
else:
    # foreground execution:
    failure, output = commands.getstatusoutput(cmd)
```

Traversing directory trees (1)

- Run through all files in your home directory and list files that are larger than 1 Mb
- A Unix find command solves the problem:

```
find $HOME -name '*' -type f -size +2000 \
     -exec ls -s {} \;
```

- This (and all features of Unix find) can be given a cross-platform implementation in Python

Traversing directory trees (2)

- Similar cross-platform Python tool:

  ```
  root = os.environ['HOME']  # my home directory
  os.path.walk(root, myfunc, arg)
  ```

  walks through a directory tree (`root`) and calls, for each directory `dirname`,

  ```
  myfunc(arg, dirname, files)  # files is list of (local) filenar
  ```

- `arg` is any user-defined argument, e.g. a nested list of variables

Example on finding large files

```
def checksize1(arg, dirname, files):
    for file in files:
        # construct the file’s complete path:
        filename = os.path.join(dirname, file)
        if os.path.isfile(filename):
            size = os.path.getsize(filename)
            if size > 1000000:
                print ’%.2fMb %s’ % (size/1000000.0,filename)

root = os.environ[’HOME’]
os.path.walk(root, checksize1, None)

# arg is a user-specified (optional) argument,
# here we specify None since arg has no use
# in the present example
```

Make a list of all large files

- Slight extension of the previous example
- Now we use the `arg` variable to build a list during the walk

```
def checksize1(arg, dirname, files):
    for file in files:
        filepath = os.path.join(dirname, file)
        if os.path.isfile(filepath):
            size = os.path.getsize(filepath)
            if size > 1000000:
                size_in_Mb = size/1000000.0
                arg.append((size_in_Mb, filename))

bigfiles = []
root = os.environ['HOME']
os.path.walk(root, checksize1, bigfiles)
for size, name in bigfiles:
    print name, 'is', size, 'Mb'
```

arg must be a list or dictionary

- Let's build a tuple of all files instead of a list:

```
def checksize1(arg, dirname, files):
    for file in files:
        filepath = os.path.join(dirname, file)
        if os.path.isfile(filepath):
            size = os.path.getsize(filepath)
            if size > 1000000:
                msg = '%.2fMb %s' % (size/1000000.0, filepath)
                arg = arg + (msg,)

bigfiles = []
os.path.walk(os.environ['HOME'], checksize1, bigfiles)
for size, name in bigfiles:
    print name, 'is', size, 'Mb'
```

- Now `bigfiles` is an empty list! Why? Explain in detail... (Hint: `arg` must be mutable)

Creating Tar archives

- Tar is a widepsread tool for packing file collections efficiently
- Very useful for software distribution or sending (large) collections of files in email
- Demo:

```
>>> import tarfile
>>> files = 'NumPy_basics.py', 'hw.py', 'leastsquares.py'
>>> tar = tarfile.open('tmp.tar.gz', 'w:gz')   # gzip compression
>>> for file in files:
...     tar.add(file)
...
>>> # check what's in this archive:
>>> members = tar.getmembers()   # list of TarInfo objects
>>> for info in members:
...     print '%s: size=%d, mode=%s, mtime=%s' % \
...           (info.name, info.size, info.mode,
...            time.strftime('%Y.%m.%d', time.gmtime(info.mtime)))
...
NumPy_basics.py: size=11898, mode=33261, mtime=2004.11.23
hw.py: size=206, mode=33261, mtime=2005.08.12
leastsquares.py: size=1560, mode=33261, mtime=2004.09.14
>>> tar.close()
```

- Compressions: uncompressed (`w:`), gzip (`w:gz`), bzip2 (`w:bz2`)

Reading Tar archives

```
>>> tar = tarfile.open('tmp.tar.gz', 'r')
>>>
>>> for file in tar.getmembers():
...     tar.extract(file)         # extract file to current work.dir.
...
>>> # do we have all the files?
>>> allfiles = os.listdir(os.curdir)
>>> for file in allfiles:
...     if not file in files:  print 'missing', file
...
>>> hw = tar.extractfile('hw.py')  # extract as file object
>>> hw.readlines()
```

Measuring CPU time (1)

- The time module:

```
import time
e0 = time.time()      # elapsed time since the epoch
c0 = time.clock()     # total CPU time spent so far
# do tasks...
elapsed_time = time.time() - e0
cpu_time = time.clock() - c0
```

- The `os.times` function returns a list:

```
os.times()[0]  : user   time, current process
os.times()[1]  : system time, current process
os.times()[2]  : user   time, child processes
os.times()[3]  : system time, child processes
os.times()[4]  : elapsed time
```

- CPU time = user time + system time

Measuring CPU time (2)

- Application:

```
t0 = os.times()
# do tasks...
os.system(time_consuming_command) # child process
t1 = os.times()

elapsed_time = t1[4] - t0[4]
user_time    = t1[0] - t0[0]
system_time  = t1[1] - t0[1]
cpu_time     = user_time + system_time
cpu_time_system_call = t1[2]-t0[2] + t1[3]-t0[3]
```

- There is a special Python profiler for finding bottlenecks in scripts (ranks functions according to their CPU-time consumption)

A timer function

Let us make a function `timer` for measuring the efficiency of an arbitrary function. `timer` takes 4 arguments:

- a function to call
- a list of arguments to the function
- number of calls to make (repetitions)
- name of function (for printout)

```
def timer(func, args, repetitions, func_name):
    t0 = time.time();  c0 = time.clock()

    for i in range(repetitions):
        func(*args)  # old style: apply(func, args)

    print '%s: elapsed=%g, CPU=%g' % \
    (func_name, time.time()-t0, time.clock()-c0)
```

Parsing command-line arguments

- Running through `sys.argv[1:]` and extracting command-line info 'manually' is easy
- Using standardized modules and interface specifications is better!
- Python's `getopt` and `optparse` modules parse the command line
- `getopt` is the simplest to use
- `optparse` is the most sophisticated

Short and long options

- It is a ’standard’ to use either short or long options

```
-d dirname                    # short options -d and -h
--directory dirname  # long options --directory and --help
```

- Short options have single hyphen,
  long options have double hyphen
- Options can take a value or not:

```
--directory dirname --help --confirm
-d dirname -h -i
```

- Short options can be combined

```
-iddirname   is the same as  -i -d dirname
```

Using the getopt module (1)

- Specify short options by the option letters, followed by colon if the option requires a value
- Example: `'id:h'`
- Specify long options by a list of option names, where names must end with = if the require a value
- Example: `['help','directory=','confirm']`

Using the getopt module (2)

- `getopt` returns a list of (option,value) pairs and a list of the remaining arguments
- Example:

```
--directory mydir -i file1 file2
```

makes `getopt` return

```
[('--directory','mydir'), ('-i','')]
['file1','file2]'
```

Using the getopt module (3)

- Processing:

```
import getopt
try:
    options, args = getopt.getopt(sys.argv[1:], 'd:hi',
                    ['directory=', 'help', 'confirm'])
except:
    # wrong syntax on the command line, illegal options,
    # missing values etc.

directory = None; confirm = 0  # default values
for option, value in options:
    if option in ('-h', '--help'):
        # print usage message
    elif option in ('-d', '--directory'):
        directory = value
    elif option in ('-i', '--confirm'):
        confirm = 1
```

Using the interface

- Equivalent command-line arguments:

```
-d mydir --confirm src1.c src2.c
--directory mydir -i src1.c src2.c
--directory=mydir --confirm src1.c src2.c
```

- Abbreviations of long options are possible, e.g.,

```
--d mydir --co
```

- This one also works: `-idmydir`

Writing Python data structures

- Write nested lists:

```
somelist = ['text1', 'text2']
a = [[1.3,somelist], 'some text']
f = open('tmp.dat', 'w')

# convert data structure to its string repr.:
f.write(str(a))
f.close()
```

- Equivalent statements writing to standard output:

```
print a
sys.stdout.write(str(a) + '\n')

# sys.stdin         standard input as file object
# sys.stdout        standard input as file object
```

Reading Python data structures

- `eval(s)`: treat string `s` as Python code
- `a = eval(str(a))` is a valid 'equation' for basic Python data structures
- Example: read nested lists

```
f = open('tmp.dat', 'r')   # file written in last slide
# evaluate first line in file as Python code:
newa = eval(f.readline())
```

results in

```
[[1.3, ['text1', 'text2']], 'some text']

# i.e.
newa = eval(f.readline())
# is the same as
newa = [[1.3, ['text1', 'text2']], 'some text']
```

Remark about str and eval

- `str(a)` is implemented as an object function `__str__`
- `repr(a)` is implemented as an object function `__repr__`
- `str(a)`: pretty print of an object
- `repr(a)`: print of all info for use with `eval`
- `a = eval(repr(a))`
- `str` and `repr` are identical for standard Python objects (lists, dictionaries, numbers)

Persistence

- Many programs need to have persistent data structures, i.e., data live after the program is terminated and can be retrieved the next time the program is executed
- `str`, `repr` and `eval` are convenient for making data structures persistent
- pickle, cPickle and shelve are other (more sophisticated) Python modules for storing/loading objects

Pickling

- Write *any* set of data structures to file using the cPickle module:

```
f = open(filename, 'w')
import cPickle
cPickle.dump(a1, f)
cPickle.dump(a2, f)
cPickle.dump(a3, f)
f.close()
```

- Read data structures in again later:

```
f = open(filename, 'r')
a1 = cPickle.load(f)
a2 = cPickle.load(f)
a3 = cPickle.load(f)
```

Shelving

- Think of shelves as dictionaries with file storage

```
import shelve
database = shelve.open(filename)
database['a1'] = a1  # store a1 under the key 'a1'
database['a2'] = a2
database['a3'] = a3
# or
database['a123'] = (a1, a2, a3)

# retrieve data:
if 'a1' in database:
    a1 = database['a1']
# and so on

# delete an entry:
del database['a2']

database.close()
```

What assignment really means

```
>>> a = 3              # a refers to int object with value 3
>>> b = a              # b refers to a (int object with value 3)
>>> id(a), id(b )      # print integer identifications of a and b
(135531064, 135531064)
>>> id(a) == id(b)     # same identification?
True                   # a and b refer to the same object
>>> a is b             # alternative test
True
>>> a = 4              # a refers to a (new) int object
>>> id(a), id(b)       # let’s check the IDs
(135532056, 135531064)
>>> a is b
False
>>> b       # b still refers to the int object with value 3
3
```

Assignment vs in-place changes

```
>>> a = [2, 6]       # a refers to a list [2, 6]
>>> b = a            # b refers to the same list as a
>>> a is b
True
>>> a = [1, 6, 3]    # a refers to a new list
>>> a is b
False
>>> b                # b still refers to the old list
[2, 6]

>>> a = [2, 6]
>>> b = a
>>> a[0] = 1         # make in-place changes in a
>>> a.append(3)      # another in-place change
>>> a
[1, 6, 3]
>>> b
[1, 6, 3]
>>> a is b           # a and b refer to the same list object
True
```

Assignment with copy

- What if we want `b` to be a copy of `a`?
- Lists: `a[:]` extracts a slice, which is a *copy* of all elements:

```
>>> b = a[:]    # b refers to a copy of elements in a
>>> b is a
False
```

  In-place changes in `a` will not affect `b`

- Dictionaries: use the `copy` method:

```
>>> a = {'refine': False}
>>> b = a.copy()
>>> b is a
False
```

  In-place changes in `a` will not affect `b`

Running an application

- Run a stand-alone program:

```
cmd = 'myprog -c file.1 -p -f -q > res'
failure = os.system(cmd)
if failure:
  print '%s: running myprog failed' % sys.argv[0]
  sys.exit(1)
```

- Redirect output from the application to a list of lines:

```
pipe = os.popen(cmd)
output = pipe.readlines()
pipe.close()

for line in output:
  # process line
```

- Better tool: the `commands` module (next slide)

Running applications and grabbing the output

- A nice way to execute another program:

```
import commands
failure, output = commands.getstatusoutput(cmd)

if failure:
    print 'Could not run', cmd; sys.exit(1)

for line in output.splitlines()  # or output.split('\n'):
    # process line
```

  (`output` holds the output as a string)

- `output` holds both standard error and standard output (`os.popen` grabs only standard output so you do not see error messages)

Running applications in the background

- `os.system`, pipes, or `commands.getstatusoutput` terminates after the command has terminated
- There are two methods for running the script in parallel with the command:
  - run the command in the background

    ```
    Unix:     add an ampersand (&) at the end of the command
    Windows:  run the command with the 'start' program
    ```

  - run the operating system command in a separate thread
- More info: see “Platform-dependent operations” slide and the `threading` module

The new standard: subprocess

- A module subprocess is the new standard for running stand-alone applications:

```
from subprocess import call
try:
    returncode = call(cmd, shell=True)
    if returncode:
        print 'Failure with returncode', returncode; sys,exit(1
except OSError, message:
    print 'Execution failed!\n', message; sys.exit(1)
```

- More advanced use of subprocess applies its `Popen` object

```
from subprocess import Popen, PIPE
p = Popen(cmd, shell=True, stdout=PIPE)
output, errors = p.communicate()
```

Output pipe

- Open (in a script) a dialog with an interactive program:

```
pipe = Popen('gnuplot -persist', shell=True, stdin=PIPE).stdin
pipe.write('set xrange [0:10]; set yrange [-2:2]\n')
pipe.write('plot sin(x)\n')
pipe.write('quit')  # quit Gnuplot
```

- Same as "here documents" in Unix shells:

```
gnuplot <<EOF
set xrange [0:10]; set yrange [-2:2]
plot sin(x)
quit
EOF
```

Writing to and reading from applications

- In theory, `Popen` allows us to have two-way comminucation with an application (read/write), but this technique is not suitable for reliable two-way dialog (easy to get hang-ups)
- The `pexpect` module is the right tool for a two-way dialog with a stand-alone application

```
# copy files to remote host via scp and password dialog
cmd = 'scp %s %s@%s:%s' % (filename, user, host, directory)
import pexpect
child = pexpect.spawn(cmd)
child.expect('password:')
child.sendline('&%$hQxz?+MbH')
child.expect(pexpect.EOF)   # wait for end of scp session
child.close()
```

File reading

- Load a file into list of lines:

```
infilename = '.myprog.cpp'
infile = open(infilename, 'r')   # open file for reading

# load file into a list of lines:
lines = infile.readlines()

# load file into a string:
filestr = infile.read()
```

- Line-by-line reading (for large files):

```
while 1:
    line = infile.readline()
    if not line: break
    # process line
```

File writing

- Open a new output file:

```
outfilename = '.myprog2.cpp'
outfile = open(outfilename, 'w')
outfile.write('some string\n')
```

- Append to existing file:

```
outfile = open(outfilename, 'a')
outfile.write('....')
```

Python types

- Numbers: `float`, `complex`, `int` (+ `bool`)
- Sequences: `list`, `tuple`, `str`, NumPy arrays
- Mappings: `dict` (dictionary/hash)
- Instances: user-defined class
- Callables: functions, callable instances

Numerical expressions

- Python distinguishes between strings and numbers:

```
b =  1.2         # b is a number
b = '1.2'        # b is a string
a = 0.5 * b      # illegal: b is NOT converted to float
a = 0.5 * float(b)    # this works
```

- All Python objects are compard with

```
==    !=   <   >   <=   >=
```

Potential confusion

- Consider:

```
b = '1.2'

if b < 100:    print b, '< 100'
else:          print b, '>= 100'
```

What do we test? string less than number!

- What we want is

```
if float(b) < 100:    # floating-point number comparison
# or
if b < str(100):      # string comparison
```

Boolean expressions

- A `bool` type is `True` or `False`
- Can mix `bool` with `int` 0 (false) or 1 (true)
- `if a:` evaluates `a` in a boolean context, same as `if bool(a):`
- Boolean tests:

```
>>> a = ''
>>> bool(a)
False
>>> bool('some string')
True
>>> bool([])
False
>>> bool([1,2])
True
```

- Empty strings, lists, tuples, etc. evaluates to `False` in a boolean context

Setting list elements

- Initializing a list:

```
arglist = [myarg1, 'displacement', "tmp.ps"]
```

- Or with indices (if there are already two list elements):

```
arglist[0] = myarg1
arglist[1] = 'displacement'
```

- Create list of specified length:

```
n = 100
mylist = [0.0]*n
```

- Adding list elements:

```
arglist = []  # start with empty list
arglist.append(myarg1)
arglist.append('displacement')
```

Getting list elements

- Extract elements form a list:

```
 filename, plottitle, psfile  = arglist
(filename, plottitle, psfile) = arglist
[filename, plottitle, psfile] = arglist
```

- Or with indices:

```
filename = arglist[0]
plottitle = arglist[1]
```

Traversing lists

- For each item in a list:

```
for entry in arglist:
    print 'entry is', entry
```

- For-loop-like traversal:

```
start = 0;  stop = len(arglist);  step = 1
for index in range(start, stop, step):
    print 'arglist[%d]=%s' % (index,arglist[index])
```

- Visiting items in reverse order:

```
mylist.reverse()  # reverse order
for item in mylist:
    # do something...
```

List comprehensions

- Compact syntax for manipulating all elements of a list:

```
y = [ float(yi) for yi in line.split() ]   # call function float
x = [ a+i*h for i in range(n+1) ]          # execute expression
```

(called list comprehension)

- Written out:

```
y = []
for yi in line.split():
    y.append(float(yi))
```

etc.

Map function

- `map` is an alternative to list comprehension:

```
y = map(float, line.split())
y = map(lambda i: a+i*h, range(n+1))
```

- `map` is faster than list comprehension but not as easy to read

Typical list operations

```
d = []                # declare empty list
d.append(1.2)         # add a number 1.2
d.append('a')         # add a text
d[0] = 1.3            # change an item
del d[1]              # delete an item
len(d)                # length of list
```

Nested lists

- Lists can be nested and heterogeneous
- List of string, number, list and dictionary:

```
>>> mylist = ['t2.ps', 1.45, ['t2.gif', 't2.png'],\
          { 'factor' : 1.0, 'c' : 0.9} ]
>>> mylist[3]
{'c': 0.90000000000000002, 'factor': 1.0}
>>> mylist[3]['factor']
1.0
>>> print mylist
['t2.ps', 1.45, ['t2.gif', 't2.png'],
 {'c': 0.90000000000000002, 'factor': 1.0}]
```

- Note: `print` prints all basic Python data structures in a nice format

Sorting a list

- In-place sort:

  ```
  mylist.sort()
  ```

  modifies `mylist`!

  ```
  >>> print mylist
  [1.4, 8.2, 77, 10]
  >>> mylist.sort()
  >>> print mylist
  [1.4, 8.2, 10, 77]
  ```

- Strings and numbers are sorted as expected

Defining the comparison criterion

```
# ignore case when sorting:

def ignorecase_sort(s1, s2):
    s1 = s1.lower()
    s2 = s2.lower()
    if   s1 <  s2: return -1
    elif s1 == s2: return  0
    else:          return  1

# or a quicker variant, using Python's built-in
# cmp function:
def ignorecase_sort(s1, s2):
    s1 = s1.lower();  s2 = s2.lower()
    return cmp(s1,s2)

# usage:
mywords.sort(ignorecase_sort)
```

Tuples ('constant lists')

- Tuple = constant list; items cannot be modified

```
>>> s1=[1.2, 1.3, 1.4]    # list
>>> s2=(1.2, 1.3, 1.4)    # tuple
>>> s2=1.2, 1.3, 1.4      # may skip parenthesis
>>> s1[1]=0               # ok
>>> s2[1]=0               # illegal
Traceback (innermost last):
  File "<pyshell#17>", line 1, in ?
    s2[1]=0
TypeError: object doesn't support item assignment

>>> s2.sort()
AttributeError: 'tuple' object has no attribute 'sort'
```

- You cannot append to tuples, but you can add two tuples to form a new tuple

Dictionary operations

- Dictionary = array with text indices (keys)
  (even user-defined objects can be indices!)
- Also called hash or associative array
- Common operations:

```
d['mass']              # extract item corresp. to key 'mass'
d.keys()               # return copy of list of keys
d.get('mass',1.0)      # return 1.0 if 'mass' is not a key
d.has_key('mass')      # does d have a key 'mass'?
d.items()              # return list of (key,value) tuples
del d['mass']          # delete an item
len(d)                 # the number of items
```

Initializing dictionaries

- Multiple items:

```
d = { 'key1' : value1, 'key2' : value2 }
# or
d = dict(key1=value1, key2=value2)
```

- Item by item (indexing):

```
d['key1'] = anothervalue1
d['key2'] = anothervalue2
d['key3'] = value2
```

Dictionary examples

- Problem: store MPEG filenames corresponding to a parameter with values 1, 0.1, 0.001, 0.00001

```
movies[1]       = 'heatsim1.mpeg'
movies[0.1]     = 'heatsim2.mpeg'
movies[0.001]   = 'heatsim5.mpeg'
movies[0.00001] = 'heatsim8.mpeg'
```

- Store compiler data:

```
g77 = {
  'name'          : 'g77',
  'description'   : 'GNU f77 compiler, v2.95.4',
  'compile_flags' : ' -pg',
  'link_flags'    : ' -pg',
  'libs'          : '-lf2c',
  'opt'           : '-O3 -ffast-math -funroll-loops'
}
```

Another dictionary example (1)

- Idea: hold command-line arguments in a dictionary `cmlargs[option]`, e.g., `cmlargs['infile']`, instead of separate variables
- Initialization: loop through `sys.argv`, assume options in pairs: –option value

```
arg_counter = 1
while arg_counter < len(sys.argv):
    option = sys.argv[arg_counter]
    option = option[2:]   # remove double hyphen
    if option in cmlargs:
        # next command-line argument is the value:
        arg_counter += 1
        value = sys.argv[arg_counter]
        cmlargs[cmlarg] = value
    else:
        # illegal option
    arg_counter += 1
```

Another dictionary example (2)

- Working with `cmlargs` in simviz1.py:

```
f = open(cmlargs['case'] + '.', 'w')
f.write(cmlargs['m']      + '\n')
f.write(cmlargs['b']      + '\n')
f.write(cmlargs['c']      + '\n')
f.write(cmlargs['func']   + '\n')
...
# make gnuplot script:
f = open(cmlargs['case'] + '.gnuplot', 'w')
f.write("""
set title '%s: m=%s b=%s c=%s f(y)=%s A=%s w=%s y0=%s dt=%s';
""" % (cmlargs['case'],cmlargs['m'],cmlargs['b'],
       cmlargs['c'],cmlargs['func'],cmlargs['A'],
       cmlargs['w'],cmlargs['y0'],cmlargs['dt']))
if not cmlargs['noscreenplot']:
    f.write("plot 'sim.dat' title 'y(t)' with lines;\n")
```

- Note: all `cmlargs[opt]` are (here) strings!

Environment variables

- The dictionary-like `os.environ` holds the environment variables:

```
os.environ['PATH']
os.environ['HOME']
os.environ['scripting']
```

- Write all the environment variables in alphabethic order:

```
sorted_env = os.environ.keys()
sorted_env.sort()

for key in sorted_env:
    print '%s = %s' % (key, os.environ[key])
```

Find a program

- Check if a given program is on the system:

```
program = 'vtk'
path = os.environ['PATH']
# PATH can be /usr/bin:/usr/local/bin:/usr/X11/bin
# os.pathsep is the separator in PATH
# (: on Unix, ; on Windows)
paths = path.split(os.pathsep)
for d in paths:
    if os.path.isdir(d):
        if os.path.isfile(os.path.join(d, program)):
            program_path = d; break

try:  # program was found if program_path is defined
    print '%s found in %s' % (program, program_path)
except:
    print '%s not found' % program
```

Cross-platform fix of previous script

- On Windows, programs usually end with `.exe` (binaries) or `.bat` (DOS scripts), while on Unix most programs have no extension
- We test if we are on Windows:

```
if sys.platform[:3] == 'win':
    # Windows-specific actions
```

- Cross-platform snippet for finding a program:

```
for d in paths:
    if os.path.isdir(d):
        fullpath = os.path.join(dir, program)
        if sys.platform[:3] == 'win':    # windows machine?
            for ext in '.exe', '.bat':   # add extensions
                if os.path.isfile(fullpath + ext):
                    program_path = d; break
        else:
            if os.path.isfile(fullpath):
                program_path = d; break
```

Splitting text

- Split string into words:

```
>>> files = 'case1.ps case2.ps    case3.ps'
>>> files.split()
['case1.ps', 'case2.ps', 'case3.ps']
```

- Can split wrt other characters:

```
>>> files = 'case1.ps, case2.ps, case3.ps'
>>> files.split(', ')
['case1.ps', 'case2.ps', 'case3.ps']
>>> files.split(',  ')  # extra erroneous space after comma...
['case1.ps, case2.ps, case3.ps']  # unsuccessful split
```

- Very useful when interpreting files

Example on using split (1)

- Suppose you have file containing numbers only
- The file can be formatted 'arbitrarily', e.g,

```
1.432 5E-09
1.0

3.2 5 69 -111
4 7 8
```

- Get a list of all these numbers:

```
f = open(filename, 'r')
numbers = f.read().split()
```

- String objects's `split` function splits wrt sequences of whitespace (whitespace = blank char, tab or newline)

Example on using split (2)

- Convert the list of strings to a list of floating-point numbers, using `map`:

```
numbers = [ float(x) for x in f.read().split() ]
```

- Think about reading this file in Fortran or C!
(quite some low-level code...)
- This is a good example of how scripting languages, like Python, yields flexible and compact code

Joining a list of strings

- Join is the opposite of split:

```
>>> line1 = 'iteration 12:    eps= 1.245E-05'
>>> line1.split()
['iteration', '12:', 'eps=', '1.245E-05']
>>> w = line1.split()
>>> ' '.join(w)  # join w elements with delimiter ' '
'iteration 12: eps= 1.245E-05'
```

- Any delimiter text can be used:

```
>>> '@@@'.join(w)
'iteration@@@12:@@@eps=@@@1.245E-05'
```

Common use of join/split

```
f = open('myfile', 'r')
lines = f.readlines()                 # list of lines
filestr = ''.join(lines)              # a single string
# can instead just do
# filestr = file.read()

# do something with filestr, e.g., substitutions...

# convert back to list of lines:
lines = filestr.splitlines()
for line in lines:
    # process line
```

Text processing (1)

- Exact word match:

```
if line == 'double':
    # line equals 'double'

if line.find('double') != -1:
    # line contains 'double'
```

- Matching with Unix shell-style wildcard notation:

```
import fnmatch
if fnmatch.fnmatch(line, 'double'):
    # line contains 'double'
```

Here, double can be any valid wildcard expression, e.g.,

```
double*    [Dd]ouble
```

Text processing (2)

- Matching with full regular expressions:

```
import re
if re.search(r'double', line):
    # line contains 'double'
```

  Here, double can be any valid regular expression, e.g.,

```
double[A-Za-z0-9_]*   [Dd]ouble   (DOUBLE|double)
```

Substitution

- Simple substitution:

```
newstring = oldstring.replace(substring, newsubstring)
```

- Substitute regular expression `pattern` by `replacement` in `str`:

```
import re
str = re.sub(pattern, replacement, str)
```

Various string types

- There are many ways of constructing strings in Python:

```
s1 = 'with forward quotes'
s2 = "with double quotes"
s3 = 'with single quotes and a variable: %(r1)g' \
     % vars()
s4 = """as a triple double (or single) quoted string"""
s5 = """triple double (or single) quoted strings
allow multi-line text (i.e., newline is preserved)
with other quotes like ' and  "
"""
```

- Raw strings are widely used for regular expressions

```
s6 = r'raw strings start with r and \ remains backslash'
s7 = r"""another raw string with a double backslash: \\ """
```

String operations

- String concatenation:

```
myfile = filename + '_tmp' + '.dat'
```

- Substring extraction:

```
>>> teststr = '0123456789'
>>> teststr[0:5]; teststr[:5]
'01234'
'01234'
>>> teststr[3:8]
'34567'
>>> teststr[3:]
'3456789'
```

Mutable and immutable objects

- The items/contents of mutable objects can be changed in-place
- Lists and dictionaries are mutable
- The items/contents of immutable objects cannot be changed in-place
- Strings and tuples are immutable

```
>>> s2 = (1.2, 1.3, 1.4)    # tuple
>>> s2[1] = 0               # illegal
```

Implementing a subclass

- Class `MySub` is a subclass of `MyBase`:

```
class MySub(MyBase):

    def __init__(self,i,j,k):  # constructor
        MyBase.__init__(self,i,j)
        self.k = k;

   def write(self):
        print 'MySub: i=',self.i,'j=',self.j,'k=',self.k
```

- Example:

```
# this function works with any object that has a write func:
def write(v): v.write()

# make a MySub instance
i = MySub(7,8,9)

write(i)   # will call MySub's write
```

Functions

- Python functions have the form

```
def function_name(arg1, arg2, arg3):
    # statements
    return something
```

- Example:

```
def debug(comment, variable):
    if os.environ.get('PYDEBUG', '0') == '1':
        print comment, variable
...
v1 = file.readlines()[3:]
debug('file %s (exclusive header):' % file.name, v1)

v2 = somefunc()
debug('result of calling somefunc:', v2)
```

This function prints any printable object!

Keyword arguments

- Can name arguments, i.e., keyword=default-value

  ```
  def mkdir(dirname, mode=0777, remove=1, chdir=1):
      if os.path.isdir(dirname):
          if remove:  shutil.rmtree(dirname)
          elif :      return 0  # did not make a new directory
      os.mkdir(dir, mode)
      if chdir: os.chdir(dirname)
      return 1      # made a new directory
  ```

  Calls look like

  ```
  mkdir('tmp1')
  mkdir('tmp1', remove=0, mode=0755)
  mkdir('tmp1', 0755, 0, 1)             # less readable
  ```

- Keyword arguments make the usage simpler and improve documentation

Variable-size argument list

- Variable number of ordinary arguments:

```
def somefunc(a, b, *rest):
    for arg in rest:
        # treat the rest...

# call:
somefunc(1.2, 9, 'one text', 'another text')
#                ...........rest...........
```

- Variable number of keyword arguments:

```
def somefunc(a, b, *rest, **kw):
    #...
    for arg in rest:
        # work with arg...
    for key in kw.keys():
        # work kw[key]
```

Example

- A function computing the average and the max and min value of a series of numbers:

```
def statistics(*args):
    avg = 0;  n = 0;   # local variables
    for number in args:  # sum up all the numbers
        n = n + 1; avg = avg + number
    avg = avg / float(n) # float() to ensure non-integer divis

    min = args[0]; max = args[0]
    for term in args:
        if term < min: min = term
        if term > max: max = term
    return avg, min, max  # return tuple
```

- Usage:

```
average, vmin, vmax = statistics(v1, v2, v3, b)
```

The Python expert's version...

- The `statistics` function can be written more compactly using (advanced) Python functionality:

```
def statistics(*args):
    return (reduce(operator.add, args)/float(len(args)),
            min(args), max(args))
```

- `reduce(op,a)`: apply operation `op` successively on all elements in list `a` (here all elements are added)
- `min(a)`, `max(a)`: find min/max of a list `a`

Call by reference

- Python scripts normally avoid call by reference and return all output variables instead
- Try to swap two numbers:

```
>>> def swap(a, b):
        tmp = b; b = a; a = tmp;

>>> a=1.2; b=1.3; swap(a, b)
>>> print a, b      # has a and b been swapped?
(1.2, 1.3)   # no...
```

- The way to do this particular task

```
>>> def swap(a, b):
    return (b,a)    # return tuple

# or smarter, just say  (b,a) = (a,b)  or simply  b,a = a,b
```

Arguments are like variables

- Consider a function

```
def swap(a, b):
    b = 2*b
    return b, a
```

- Calling `swap(A, B)` is inside `swap` equivalent to

```
a = A
b = B
b = 2*b
return b, a
```

- Arguments are transferred in the same way as we assign objects to variables (using the assignment operator =)
- This may help to explain how arguments in functions get their values

In-place list assignment

- Lists can be changed in-place in functions:

```
>>> def somefunc(mutable, item, item_value):
        mutable[item] = item_value

>>> a = ['a','b','c']  # a list
>>> somefunc(a, 1, 'surprise')
>>> print a
['a', 'surprise', 'c']
```

- Note: `mutable` is a name for the same object as `a`, and we use this name to change the object in-place
- This works for dictionaries as well
  (but not tuples) and instances of user-defined classes

Input and output data in functions

- The Python programming style is to have input data as arguments and output data as return values

```
def myfunc(i1, i2, i3, i4=False, io1=0):
    # io1: input and output variable
    ...
    # pack all output variables in a tuple:
    return io1, o1, o2, o3

# usage:
a, b, c, d = myfunc(e, f, g, h, a)
```

- Only (a kind of) references to objects are transferred so returning a large data structure implies just returning a reference

Scope of variables

- Variables defined inside the function are local
- To change global variables, these must be declared as global inside the function

```
s = 1

def myfunc(x, y):
    z = 0  # local variable, dies when we leave the func.
    global s
    s = 2  # assignment requires decl. as global
    return y-1,z+1
```

- Variables can be global, local (in func.), and class attributes
- The scope of variables in nested functions may confuse newcomers (see ch. 8.7 in the course book)

Regular expressions

Contents

- Motivation for regular expression
- Regular expression syntax
- Lots of examples on problem solving with regular expressions
- Many examples related to scientific computations

More info

- Ch. 8.2 in the course book
- Regular Expression HOWTO for Python (see `doc.html`)
- perldoc perlrequick (intro), perldoc perlretut (tutorial), perldoc perlre (full reference)
- “Text Processing in Python” by Mertz (Python syntax)
- “Mastering Regular Expressions” by Friedl (Perl syntax)
- Note: the core syntax is the same in Perl, Python, Ruby, Tcl, Egrep, Vi/Vim, Emacs, ..., so books about these tools also provide info on regular expressions

Motivation

- Consider a simulation code with this type of output:

```
t=2.5  a: 1.0 6.2 -2.2   12 iterations and eps=1.38756E-05
t=4.25  a: 1.0 1.4   6 iterations and eps=2.22433E-05
>> switching from method AQ4 to AQP1
t=5  a: 0.9   2 iterations and eps=3.78796E-05
t=6.386  a: 1.0 1.1525   6 iterations and eps=2.22433E-06
>> switching from method AQP1 to AQ2
t=8.05  a: 1.0   3 iterations and eps=9.11111E-04
...
```

- You want to make two graphs:
  - iterations vs t
  - eps vs t
- How can you extract the relevant numbers from the text?

Regular expressions

- Some structure in the text, but `line.split()` is too simple (different no of columns/words in each line)
- Regular expressions constitute a powerful language for formulating structure and extract parts of a text
- Regular expressions look cryptic for the novice
- regex/regexp: abbreviations for regular expression

Specifying structure in a text

```
t=6.386  a: 1.0 1.1525   6 iterations and eps=2.22433E-06
```

- Structure: t=, number, 2 blanks, a:, some numbers, 3 blanks, integer, ' iterations and eps=', number
- Regular expressions constitute a language for specifying such structures
- Formulation in terms of a regular expression:

```
t=(.*)\s{2}a:.*\s+(\d+) iterations and eps=(.*)
```

Dissection of the regex

- A regex usually contains special characters introducing freedom in the text:

```
t=(.*)\s{2}a:.*\s+(\d+) iterations and eps=(.*)

t=6.386  a: 1.0 1.1525   6 iterations and eps=2.22433E-06
```

```
.           any character
.*          zero or more . (i.e. any sequence of characters)
(.*)        can extract the match for .* afterwards
\s          whitespace (spacebar, newline, tab)
\s{2}       two whitespace characters
a:          exact text
.*          arbitrary text
\s+         one or more whitespace characters
\d+         one or more digits (i.e. an integer)
(\d+)       can extract the integer later
iterations and eps=     exact text
```

Using the regex in Python code

```
pattern = \
r"t=(.*)\s{2}a:.*\s+(\d+) iterations and eps=(.*)"

t = []; iterations = []; eps = []

# the output to be processed is stored in the list of lines

for line in lines:

    match = re.search(pattern, line)

    if match:
        t.append          (float(match.group(1)))
        iterations.append(int   (match.group(2)))
        eps.append        (float(match.group(3)))
```

Result

- Output text to be interpreted:

```
t=2.5  a: 1 6 -2    12 iterations and eps=1.38756E-05
t=4.25  a: 1.0 1.4    6 iterations and eps=2.22433E-05
>> switching from method AQ4 to AQP1
t=5  a: 0.9   2 iterations and eps=3.78796E-05
t=6.386  a: 1 1.15    6 iterations and eps=2.22433E-06
>> switching from method AQP1 to AQ2
t=8.05  a: 1.0   3 iterations and eps=9.11111E-04
```

- Extracted Python lists:

```
t = [2.5, 4.25, 5.0, 6.386, 8.05]
iterations = [12, 6, 2, 6, 3]
eps = [1.38756e-05, 2.22433e-05, 3.78796e-05,
       2.22433e-06, 9.11111E-04]
```

Another regex that works

- Consider the regex

  ```
  t=(.*)\s+a:.*\s+(\d+)\s+.*=(.*)
  ```

  compared with the previous regex

  ```
  t=(.*)\s{2}a:.*\s+(\d+) iterations and eps=(.*)
  ```

- Less structure
- How ’exact’ does a regex need to be?
- The degree of preciseness depends on the probability of making a wrong match

Failure of a regex

- Suppose we change the regular expression to

  ```
  t=(.*)\s+a:.*(\d+).*=(.*)
  ```

- It works on most lines in our test text but not on

  ```
  t=2.5  a: 1 6 -2    12 iterations and eps=1.38756E-05
  ```

- 2 instead of 12 (iterations) is extracted
  (why? see later)
- Regular expressions constitute a powerful tool, but you need to develop understanding and experience

List of special regex characters

```
.         # any single character except a newline
^         # the beginning of the line or string
$         # the end of the line or string
*         # zero or more of the last character
+         # one or more of the last character
?         # zero or one of the last character

[A-Z]     # matches all upper case letters
[abc]     # matches either a or b or c
[^b]      # does not match b
[^a-z]    # does not match lower case letters
```

Context is important

```
.*     # any sequence of characters (except newline)
[.*]   # the characters . and *

^no    # the string 'no' at the beginning of a line
[^no]  # neither n nor o

A-Z    # the 3-character string 'A-Z' (A, minus, Z)
[A-Z]  # one of the chars A, B, C, ..., X, Y, or Z
```

More weird syntax...

- The OR operator:

```
(eg|le)gs   # matches eggs or legs
```

- Short forms of common expressions:

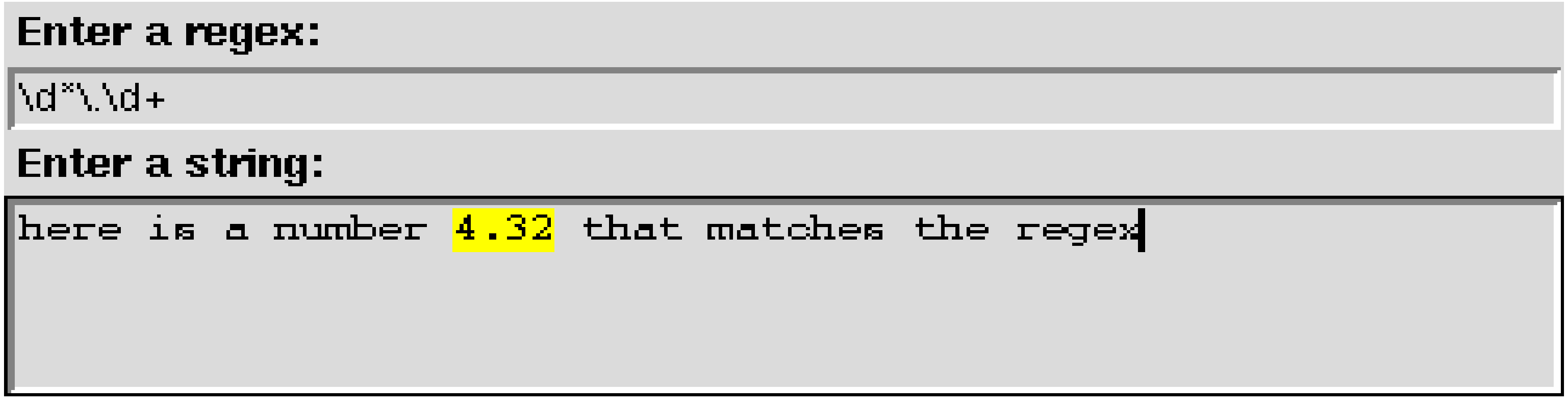
```
\n      # a newline
\t      # a tab
\w      # any alphanumeric (word) character
        # the same as [a-zA-Z0-9_]
\W      # any non-word character
        # the same as [^a-zA-Z0-9_]
\d      # any digit, same as [0-9]
\D      # any non-digit, same as [^0-9]
\s      # any whitespace character: space,
        # tab, newline, etc
\S      # any non-whitespace character
\b      # a word boundary, outside [] only
\B      # no word boundary
```

Quoting special characters

```
\.       # a dot
\|       # vertical bar
\[       # an open square bracket
\)       # a closing parenthesis
\*       # an asterisk
\^       # a hat
\/       # a slash
\\       # a backslash
\{       # a curly brace
\?       # a question mark
```

GUI for regex testing

src/tools/regexdemo.py:



The part of the string that matches the regex is high-lighted

Regex for a real number

- Different ways of writing real numbers:
  -3, 42.9873, 1.23E+1, 1.2300E+01, 1.23e+01
- Three basic forms:
  - integer: -3
  - decimal notation: 42.9873, .376, 3.
  - scientific notation: 1.23E+1, 1.2300E+01, 1.23e+01, 1e1

A simple regex

- Could just collect the legal characters in the three notations:

```
[0-9.Ee\-+]+
```

- Downside: this matches text like

```
12-24
24.-
--E1--
+++++
```

- How can we define precise regular expressions for the three notations?

Decimal notation regex

- Regex for decimal notation:

```
-?\d*\.\d+

# or equivalently (\d is [0-9])
-?[0-9]*\.[0-9]+
```

- Problem: this regex does not match '3.'
- The fix

```
-?\d*\.\d*
```

  is ok but matches text like '-.' and (much worse!) '.'
- Trying it on

```
'some text. 4. is a number.'
```

  gives a match for the first period!

Fix of decimal notation regex

- We need a digit before OR after the dot
- The fix:

```
-?(\d*\.\d+|\d+\.\d*)
```

- A more compact version (just "OR-ing" numbers without digits after the dot):

```
-?(\d*\.\d+|\d+\.)
```

Combining regular expressions

- Make a regex for integer or decimal notation:

  `(integer OR decimal notation)`

  using the OR operator and parenthesis:

  `-?(\d+|(\d+\.\d*|\d*\.\d+))`

- Problem: `22.432` gives a match for `22`
  (i.e., just digits? yes - `22` - match!)

Check the order in combinations!

- Remedy: test for the most complicated pattern first

  `(decimal notation OR integer)`

  ```
  -?((\d+\.\d*|\d*\.\d+)|\d+)
  ```

- Modularize the regex:

  ```
  real_in = r'\d+'
  real_dn = r'(\d+\.\d*|\d*\.\d+)'
  real = '-?(' + real_dn + '|' + real_in + ')'
  ```

Scientific notation regex (1)

- Write a regex for numbers in scientific notation
- Typical text: `1.27635E+01`, `-1.27635e+1`
- Regular expression:

```
-?\d\.\d+[Ee][+\-]\d\d?
```

- = optional minus, one digit, dot, at least one digit, E or e, plus or minus, one digit, optional digit

Scientific notation regex (2)

- Problem: `1e+00` and `1e1` are not handled
- Remedy: zero or more digits behind the dot, optional e/E, optional sign in exponent, more digits in the exponent (`1e001`):

```
-?\d\.?\d*[Ee][+\-]?\d+
```

Making the regex more compact

- A pattern for integer or decimal notation:

```
-?((\d+\.\d*|\d*\.\d+)|\d+)
```

- Can get rid of an OR by allowing the dot and digits behind the dot be optional:

```
-?(\d+(\.\d*)?|\d*\.\d+)
```

- Such a number, followed by an optional exponent (a la `e+02`), makes up a general real number (!)

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

A more readable regex

- Scientific OR decimal OR integer notation:

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

  or better (modularized):

  ```
  real_in = r'\d+'
  real_dn = r'(\d+\.\d*|\d*\.\d+)'
  real_sn = r'(\d\.?\d*[Ee][+\-]?\d+'
  real = '-?(' + real_sn + '|' + real_dn + '|' + real_in + ')'
  ```

- Note: first test on the most complicated regex in OR expressions

Groups (in introductory example)

- Enclose parts of a regex in () to extract the parts:

```
pattern = r"t=(.*)\s+a:.*\s+(\d+)\s+.*=(.*)"
# groups:    ( )            (   )        ( )
```

  This defines three groups (t, iterations, eps)

- In Python code:

```
match = re.search(pattern, line)
if match:
    time = float(match.group(1))
    iter = int  (match.group(2))
    eps  = float(match.group(3))
```

- The complete match is group 0 (here: the whole line)

Regex for an interval

- Aim: extract lower and upper limits of an interval:

  ```
  [ -3.14E+00, 29.6524]
  ```

- Structure: bracket, real number, comma, real number, bracket, with embedded whitespace

Easy start: integer limits

- Regex for real numbers is a bit complicated
- Simpler: integer limits

```
pattern = r'\[\d+,\d+\]'
```

but this does must be fixed for embedded white space or negative numbers a la

```
[ -3   , 29   ]
```

- Remedy:

```
pattern = r'\[\s*-?\d+\s*,\s*-?\d+\s*\]'
```

- Introduce groups to extract lower and upper limit:

```
pattern = r'\[\s*(-?\d+)\s*,\s*(-?\d+)\s*\]'
```

Testing groups

In an interactive Python shell we write

```
>>> pattern = r'\[\s*(-?\d+)\s*,\s*(-?\d+)\s*\]'
>>> s = "here is an interval: [ -3, 100] ..."
>>> m = re.search(pattern, s)
>>> m.group(0)
[ -3, 100]
>>> m.group(1)
-3
>>> m.group(2)
100
>>> m.groups()    # tuple of all groups
('-3', '100')
```

Named groups

- Many groups? inserting a group in the middle changes other group numbers...
- Groups can be given *logical names* instead
- Standard group notation for interval:

```
# apply integer limits for simplicity: [int,int]
\[\s*(-?\d+)\s*,\s*(-?\d+)\s*\]
```

- Using named groups:

```
\[\s*(?P<lower>-?\d+)\s*,\s*(?P<upper>-?\d+)\s*\]
```

- Extract groups by their names:

```
match.group('lower')
match.group('upper')
```

Regex for an interval; real limits

- Interval with general real numbers:

```
real_short = r'\s*(-?(\d+(\.\d*)?|\d*\.\d+)([eE][+\-]?\d+)?)\s*
interval = r"\[" + real_short + "," + real_short + r"\]"
```

- Example:

```
>>> m = re.search(interval, '[-100,2.0e-1]')
>>> m.groups()
('-100', '100', None, None, '2.0e-1', '2.0', '.0', 'e-1')
```

i.e., lots of (nested) groups; only group 1 and 5 are of interest

Handle nested groups with named groups

- Real limits, previous regex resulted in the groups

```
('-100', '100', None, None, '2.0e-1', '2.0', '.0', 'e-1')
```

- Downside: many groups, difficult to count right
- Remedy 1: use named groups for the outer left and outer right groups:

```
real1 = \
 r"\s*(?P<lower>-?(\d+(\.\d*)?|\d*\.\d+)([eE][+\-]?\d+)?)\s*"
real2 = \
 r"\s*(?P<upper>-?(\d+(\.\d*)?|\d*\.\d+)([eE][+\-]?\d+)?)\s*"
interval = r"\[" + real1 + "," + real2 + r"\]"
...
match = re.search(interval, some_text)
if match:
    lower_limit = float(match.group('lower'))
    upper_limit = float(match.group('upper'))
```

Simplify regex to avoid nested groups

- Remedy 2: reduce the use of groups
- Avoid nested OR expressions (recall our first tries):

```
real_sn = r"-?\d\.?\d*[Ee][+\-]\d+"
real_dn = r"-?\d*\.\d*"
real = r"\s*(" + real_sn + "|" + real_dn + "|" + real_in + r")
interval = r"\[" + real + "," + real + r"\]"
```

- Cost: (slightly) less general and safe regex

Extracting multiple matches (1)

- `re.findall` finds all matches (`re.search` finds the first)

```
>>> r = r"\d+\.\d*"
>>> s = "3.29 is a number, 4.2 and 0.5 too"
>>> re.findall(r,s)
['3.29', '4.2', '0.5']
```

- Application to the interval example:

```
lower, upper = re.findall(real, '[-3, 9.87E+02]')
# real: regex for real number with only one group!
```

Extracting multiple matches (1)

- If the regex contains groups, `re.findall` returns the matches of all groups - this might be confusing!

```
>>> r = r"(\d+)\.\d*"
>>> s = "3.29 is a number, 4.2 and 0.5 too"
>>> re.findall(r,s)
['3', '4', '0']
```

- Application to the interval example:

```
>>> real_short = r"([+\-]?(\d+(\.\d*)?|\d*\.\d+)([eE][+\-]?\d+
>>> # recall: real_short contains many nested groups!
>>> g = re.findall(real_short, '[-3, 9.87E+02]')
>>> g
[('-3', '3', '', ''), ('9.87E+02', '9.87', '.87', 'E+02')]
>>> limits = [ float(g1) for g1, g2, g3, g4 in g ]
>>> limits
[-3.0, 987.0]
```

Making a regex simpler

- Regex is often a question of structure *and context*
- Simpler regex for extracting interval limits:

```
\[(.*),(.*)\]
```

- It works!

```
>>> l = re.search(r'\[(.*),(.*)\]',
                  ' [-3.2E+01,0.11  ]').groups()
>>> l
('-3.2E+01', '0.11  ')

# transform to real numbers:
>>> r = [float(x) for x in l]
>>> r
[-32.0, 0.11]
```

Failure of a simple regex (1)

- Let us test the simple regex on a more complicated text:

```
>>> l = re.search(r'\[(.*),(.*)\]', \
    ' [-3.2E+01,0.11  ] and [-4,8]').groups()
>>> l
('-3.2E+01,0.11  ] and [-4', '8')
```

  Regular expressions can surprise you...!

- Regular expressions are greedy, they attempt to find the longest possible match, here from `[` to the last (!) comma
- We want a shortest possible match, up to the first comma, i.e., a non-greedy match
- Add a `?` to get a non-greedy match:

```
\[(.*?),(.*?)\]
```

- Now `l` becomes

```
('-3.2E+01', '0.11  ')
```

Failure of a simple regex (2)

- Instead of using a non-greedy match, we can use

  `\[([^,]*),([^\]]*)\]`

- Note: only the first group (here first interval) is found by `re.search`, use `re.findall` to find all

Failure of a simple regex (3)

- The simple regexes

```
\[([^,]*),([^\]]*)\]
\[(.*?),(.*?)\]
```

  are not fool-proof:

```
>>> l = re.search(r'\[([^,]*),([^\]]*)\]',
                  ' [e.g., exception]').groups()
>>> l
('e.g.', ' exception')
```

- 100 percent reliable fix: use the detailed real number regex inside the parenthesis
- The simple regex is ok for personal code

Application example

- Suppose we, in an input file to a simulator, can specify a grid using this syntax:

```
domain=[0,1]x[0,2] indices=[1:21]x[0:100]
domain=[0,15] indices=[1:61]
domain=[0,1]x[0,1]x[0,1] indices=[0:10]x[0:10]x[0:20]
```

- Can we easily extract domain and indices limits and store them in variables?

Extracting the limits

- Specify a regex for an interval with real number limits
- Use re.findall to extract multiple intervals
- Problems: many nested groups due to complicated real number specifications
- Various remedies: as in the interval examples, see fdmgrid.py
- The bottom line: a very simple regex, utilizing the surrounding structure, works well

Utilizing the surrounding structure

- We can get away with a simple regex, because of the surrounding structure of the text:

```
indices = r"\[([^:,]*):([^\]]*)\]"  # works
domain  = r"\[([^,]*),([^\]]*)\]"   # works
```

- Note: these ones do not work:

```
indices = r"\[([^:]*):([^\]]*)\]"
indices = r"\[(.*?):(.*?)\]"
```

They match too much:

```
domain=[0,1]x[0,2] indices=[1:21]x[1:101]
       [.....................:
```

we need to exclude commas (i.e. left bracket, anything but comma or colon, colon, anythin but right bracket)

Splitting text

- Split a string into words:

```
line.split(splitstring)
# or
string.split(line, splitstring)
```

- Split wrt a regular expression:

```
>>> files = "case1.ps, case2.ps,     case3.ps"
>>> import re
>>> re.split(r",\s*", files)
['case1.ps', 'case2.ps', 'case3.ps']

>>> files.split(", ")  # a straight string split is undesired
['case1.ps', 'case2.ps', '    case3.ps']
>>> re.split(r"\s+", "some    words    in a text")
['some', 'words', 'in', 'a', 'text']
```

- Notice the effect of this:

```
>>> re.split(r" ", "some    words    in a text")
['some', '', '', '', 'words', '', '', 'in', 'a', 'text']
```

Pattern-matching modifiers (1)

- ...also called flags in Python regex documentation
- Check if a user has written "yes" as answer:

```
if re.search('yes', answer):
```

- Problem: "YES" is not recognized; try a fix

```
if re.search(r'(yes|YES)', answer):
```

- Should allow "Yes" and "YEs" too...

```
if re.search(r'[yY][eE][sS]', answer):
```

- This is hard to read and case-insensitive matches occur frequently - there must be a better way!

Pattern-matching modifiers (2)

```
if re.search('yes', answer,  re.IGNORECASE):
# pattern-matching modifier: re.IGNORECASE
# now we get a match for 'yes', 'YES', 'Yes' ...

# ignore case:
re.I  or  re.IGNORECASE

# let ^ and $ match at the beginning and
# end of every line:
re.M  or  re.MULTILINE

# allow comments and white space:
re.X  or  re.VERBOSE

# let . (dot) match newline too:
re.S  or  re.DOTALL

# let e.g. \w match special chars (?, ?, ...):
re.L  or  re.LOCALE
```

Comments in a regex

- The `re.X` or `re.VERBOSE` modifier is very useful for inserting comments explaning various parts of a regular expression
- Example:

```
# real number in scientific notation:
real_sn = r"""
-?                # optional minus
\d\.\d+            # a number like 1.4098
[Ee][+\-]\d\d?  # exponent, E-03, e-3, E+12
"""

match = re.search(real_sn, 'text with a=1.92E-04 ',
                  re.VERBOSE)

# or when using compile:
c = re.compile(real_sn, re.VERBOSE)
match = c.search('text with a=1.9672E-04 ')
```

Substitution

- Substitute `float` by `double`:

```
# filestr contains a file as a string
filestr = re.sub('float', 'double', filestr)
```

- In general:

```
re.sub(pattern, replacement, str)
```

- If there are groups in pattern, these are accessed by

```
\1      \2      \3      ...
\g<1>   \g<2>   \g<3>   ...

\g<lower>  \g<upper> ...
```

  in `replacement`

Example: strip away C-style comments

- C-style comments could be nice to have in scripts for commenting out large portions of the code:

```
/*
while 1:
    line = file.readline()
    ...
...
*/
```

- Write a script that strips C-style comments away
- Idea: match comment, substitute by an empty string

Trying to do something simple

- Suggested regex for C-style comments:

```
comment = r'/\*.*\*/'

# read file into string filestr
filestr = re.sub(comment, '', filestr)
```

  i.e., match everything between `/*` and `*/`

- Bad: `.` does not match newline
- Fix: `re.S` or `re.DOTALL` modifier makes `.` match newline:

```
comment = r'/\*.*\*/'
c_comment = re.compile(comment, re.DOTALL)
filestr = c_comment.sub(comment, '', filestr)
```

- OK? No!

Testing the C-comment regex (1)

Test file:

```
/*********************************************/
/* File myheader.h                           */
/*********************************************/

#include <stuff.h>  // useful stuff

class MyClass
{
  /* int r; */  float q;
  // here goes the rest class declaration
}

/* LOG HISTORY of this file:
 * $ Log: somefile,v $
 * Revision 1.2  2000/07/25 09:01:40  hpl
 * update
 *
 * Revision 1.1.1.1  2000/03/29 07:46:07  hpl
 * register new files
 *
*/
```

Testing the C-comment regex (2)

- The regex

  ```
  /\*.*\*/  with re.DOTALL (re.S)
  ```

  matches the whole file (i.e., the whole file is stripped away!)
- Why? a regex is by default greedy, it tries the longest possible match, here the whole file
- A question mark makes the regex non-greedy:

  ```
  /\*.*?\*/
  ```

Testing the C-comment regex (3)

- The non-greedy version works
- OK? Yes - the job is done, almost...

```
const char* str ="/* this is a comment */"
```

gets stripped away to an empty string...

Substitution example

- Suppose you have written a C library which has many users
- One day you decide that the function

  ```
  void superLibFunc(char* method, float x)
  ```

  would be more natural to use if its arguments were swapped:

  ```
  void superLibFunc(float x, char* method)
  ```

- All users of your library must then update their application codes - can you automate?

Substitution with backreferences

- You want locate all strings on the form

  ```
  superLibFunc(arg1, arg2)
  ```

  and transform them to

  ```
  superLibFunc(arg2, arg1)
  ```

- Let `arg1` and `arg2` be groups in the regex for the superLibFunc calls
- Write out

  ```
  superLibFunc(\2, \1)

  # recall: \1 is group 1, \2 is group 2 in a re.sub command
  ```

Regex for the function calls (1)

- Basic structure of the regex of calls:

  ```
  superLibFunc\s*\(\s*arg1\s*,\s*arg2\s*\)
  ```

  but what should the `arg1` and `arg2` patterns look like?

- Natural start: `arg1` and `arg2` are valid C variable names

  ```
  arg = r"[A-Za-z_0-9]+"
  ```

- Fix; digits are not allowed as the first character:

  ```
  arg = "[A-Za-z_][A-Za-z_0-9]*"
  ```

Regex for the function calls (2)

- The regex

  ```
  arg = "[A-Za-z_][A-Za-z_0-9]*"
  ```

  works well for calls with variables, but we can call `superLibFunc` with numbers too:

  ```
  superLibFunc ("relaxation", 1.432E-02);
  ```

- Possible fix:

  ```
  arg = r"[A-Za-z0-9_.\-+\"]+"
  ```

  but the disadvantage is that `arg` now also matches

  ```
  .+-32skj  3.ejks
  ```

Constructing a precise regex (1)

- Since `arg2` is a float we can make a precise regex: legal C variable name OR legal real variable format

```
arg2 = r"([A-Za-z_][A-Za-z_0-9]*|" + real + \
       "|float\s+[A-Za-z_][A-Za-z_0-9]*" + ")"
```

where `real` is our regex for formatted real numbers:

```
real_in = r"-?\d+"
real_sn = r"-?\d\.\d+[Ee][+\-]\d\d?"
real_dn = r"-?\d*\.\d+"
real = r"\s*("+ real_sn +"|"+ real_dn +"|"+ real_in +r")\s*"
```

Constructing a precise regex (2)

- We can now treat variables and numbers in calls
- Another problem: should swap arguments in a user's definition of the function:

  ```
  void superLibFunc(char* method, float x)
  ```

  `to`

  ```
  void superLibFunc(float x, char* method)
  ```

  Note: the argument names (`x` and `method`) can also be omitted!
- Calls and declarations of superLibFunc can be written on more than one line and with embedded C comments!
- Giving up?

A simple regex may be sufficient

- Instead of trying to make a precise regex, let us make a very simple one:

```
arg = '.+'    # any text
```

- "Any text" may be precise enough since we have the surrounding structure,

```
superLibFunc\s*(\s*arg\s*,\s*arg\s*)
```

  and assume that a C compiler has checked that `arg` is a valid C code text in this context

Refining the simple regex

- A problem with `.+` appears in lines with more than one calls:

  ```
  superLibFunc(a,x);  superLibFunc(ppp,qqq);
  ```

- We get a match for the first argument equal to

  ```
  a,x);  superLibFunc(ppp
  ```

- Remedy: non-greedy regex (see later) or

  ```
  arg = r"[^,]+"
  ```

  This one matches multi-line calls/declarations, also with embedded comments (`.+` does not match newline unless the `re.S` modifier is used)

Swapping of the arguments

- Central code statements:

```
arg = r"[^,]+"
call = r"superLibFunc\s*\(\s*(%s),\s*(%s)\)" % (arg,arg)

# load file into filestr

# substutite:
filestr = re.sub(call, r"superLibFunc(\2, \1)", filestr)

# write out file again
fileobject.write(filestr)
```

Files: src/py/intro/swap1.py

Testing the code

- Test text:

```
superLibFunc(a,x);  superLibFunc(qqq,ppp);
superLibFunc ( method1, method2 );
superLibFunc(3method /* illegal name! */, method2 ) ;
superLibFunc(  _method1,method_2) ;
superLibFunc (
              method1 /* the first method we have */ ,
          super_method4 /* a special method that
                                       deserves a two-line comment...
             ) ;
```

- The simple regex successfully transforms this into

```
superLibFunc(x, a);  superLibFunc(ppp, qqq);
superLibFunc(method2 , method1);
superLibFunc(method2 , 3method /* illegal name! */) ;
superLibFunc(method_2, _method1) ;
superLibFunc(super_method4 /* a special method that
                                       deserves a two-line comment...
             , method1 /* the first method we have */ ) ;
```

- Notice how powerful a small regex can be!!
- Downside: cannot handle a function call as argument

Shortcomings

- The simple regex

  ```
  [^,]+
  ```

  breaks down for comments with comma(s) and function calls as arguments, e.g.,

  ```
  superLibFunc(m1, a /* large, random number */);
  superLibFunc(m1, generate(c, q2));
  ```

  The regex will match the longest possible string ending with a comma, in the first line

  ```
  m1, a /* large,
  ```

  but then there are no more commas ...

- A complete solution should *parse* the C code

More easy-to-read regex

- The `superLibFunc` call with comments and named groups:

```
call = re.compile(r"""
     superLibFunc   # name of function to match
     \s*            # possible whitespace
     \(             # parenthesis before argument list
     \s*            # possible whitespace
     (?P<arg1>%s)   # first argument plus optional whitespace
     ,              # comma between the arguments
     \s*            # possible whitespace
     (?P<arg2>%s)   # second argument plus optional whitespace
     \)             # closing parenthesis
     """ % (arg,arg), re.VERBOSE)

# the substitution command:
filestr = call.sub(r"superLibFunc(\g<arg2>,
                   \g<arg1>)",filestr)
```

Files: src/py/intro/swap2.py

Example

- Goal: remove C++/Java comments from source codes
- Load a source code file into a string:

```
filestr = open(somefile, 'r').read()

# note: newlines are a part of filestr
```

- Substitute comments *// some text...* by an empty string:

```
filestr = re.sub(r'//.*', '', filestr)
```

- Note: . (dot) does not match newline; if it did, we would need to say

```
filestr = re.sub(r'//[^\n]*', '', filestr)
```

Failure of a simple regex

- How will the substitution

  ```
  filestr = re.sub(r'//[^\n]*', '', filestr)
  ```

  treat a line like

  ```
  const char* heading = "------------//------------";
  ```

  ???

Regex debugging (1)

- The following useful function demonstrate how to extract matches, groups etc. for examination:

```
def debugregex(pattern, str):
    s = "does '" + pattern + "' match '" + str + "'?\n"
    match = re.search(pattern, str)
    if match:
        s += str[:match.start()] + "[" + \
             str[match.start():match.end()] + \
             "]" + str[match.end():]
        if len(match.groups()) > 0:
            for i in range(len(match.groups())):
                s += "\ngroup %d: [%s]" % \
                     (i+1,match.groups()[i])
    else:
        s += "No match"
    return s
```

Regex debugging (2)

- Example on usage:

```
>>> print debugregex(r"(\d+\.\d*)",
                     "a= 51.243 and b =1.45")

does '(\d+\.\d*)' match 'a= 51.243 and b =1.45'?
a= [51.243] and b =1.45
group 1: [51.243]
```

Python modules

Contents

More info

- Appendix B.1 in the course book
- Python electronic documentation:
  Distributing Python Modules, Installing Python Modules

Make your own Python modules!

- Reuse scripts by wrapping them in classes or functions
- Collect classes and functions in library modules
- How? just put classes and functions in a file MyMod.py
- Put MyMod.py in one of the directories where Python can find it (see next slide)
- Say

  ```
  import MyMod
  # or
  import MyMod as M   # M is a short form
  # or
  from MyMod import *
  # or
  from MyMod import myspecialfunction, myotherspecialfunction
  ```

  in any script

How Python can find your modules

- Python has some 'official' module directories, typically

  ```
  /usr/lib/python2.3
  /usr/lib/python2.3/site-packages
  ```

  \+ current working directory

- The environment variable `PYTHONPATH` may contain additional directories with modules

  ```
  unix> echo $PYTHONPATH
  /home/me/python/mymodules:/usr/lib/python2.2:/home/you/yourlib:
  ```

- Python's `sys.path` list contains the directories where Python searches for modules

- `sys.path` contains 'official' directories, plus those in `PYTHONPATH`)

Setting PYTHONPATH

- In a Unix Bash environment environment variables are normally set in `.bashrc`:

  ```
  export PYTHONTPATH=$HOME/pylib:$scripting/src/tools
  ```

- Check the contents:

  ```
  unix> echo $PYTHONPATH
  ```

- In a Windows environment one can do the same in `autoexec.bat`:

  ```
  set PYTHONPATH=C:\pylib;%scripting%\src\tools
  ```

- Check the contents:

  ```
  dos> echo %PYTHONPATH%
  ```

- Note: it is easy to make mistakes; `PYTHONPATH` may be different from what you think, so check `sys.path`

Summary of finding modules

- Copy your module file(s) to a directory already contained in `sys.path`

```
unix or dos> python -c 'import sys; print sys.path'
```

- Can extend `PYTHONPATH`

```
# Bash syntax:
export PYTHONPATH=$PYTHONPATH:/home/me/python/mymodules
```

- Can extend `sys.path` in the script:

```
sys.path.insert(0, '/home/me/python/mynewmodules')
```

(insert first in the list)

Packages (1)

- A class of modules can be collected in a *package*
- Normally, a package is organized as module files in a directory tree
- Each subdirectory has a file `__init__.py` (can be empty)
- Packages allow “dotted modules names” like

  `MyMod.numerics.pde.grids`

  reflecting a file `MyMod/numerics/pde/grids.py`

Packages (2)

- Can import modules in the tree like this:

```
from MyMod.numerics.pde.grids import fdm_grids

grid = fdm_grids()
grid.domain(xmin=0, xmax=1, ymin=0, ymax=1)
...
```

  Here, class `fdm_grids` is in module `grids` (file `grids.py`) in the directory `MyMod/numerics/pde`

- Or

```
import MyMod.numerics.pde.grids
grid = MyMod.numerics.pde.grids.fdm_grids()
grid.domain(xmin=0, xmax=1, ymin=0, ymax=1)
#or
import MyMod.numerics.pde.grids as Grid
grid = Grid.fdm_grids()
grid.domain(xmin=0, xmax=1, ymin=0, ymax=1)
```

- See ch. 6 of the Python Tutorial (part of the electronic doc)

Test/doc part of a module

- Module files can have a test/demo script at the end:

```
if __name__ == '__main__':
    infile = sys.argv[1]; outfile = sys.argv[2]
    for i in sys.argv[3:]:
        create(infile, outfile, i)
```

- The block is executed if the module file is run as a script
- The tests at the end of a module often serve as good examples on the usage of the module

Public/non-public module variables

- Python convention: add a leading underscore to non-public functions and (module) variables

```
_counter = 0

def _filename():
    """Generate a random filename."""
    ...
```

- After a standard import `import MyMod`, we may access

```
MyMod._counter
n = MyMod._filename()
```

  but after a `from MyMod import *` the names with leading underscore are *not* available

- Use the underscore to tell users what is public and what is not
- Note: non-public parts can be changed in future releases

Installation of modules/packages

- Python has its own build/installation system: Distutils
- Build: compile (Fortran, C, C++) into module
  (only needed when modules employ compiled code)
- Installation: copy module files to “install” directories
- Publish: make module available for others through PyPi
- Default installation directory:

```
os.path.join(sys.prefix, 'lib', 'python' + sys.version[0:3],
             'site-packages')
# e.g. /usr/lib/python2.3/site-packages
```

- Distutils relies on a `setup.py` script

A simple setup.py script

- Say we want to distribute two modules in two files

```
MyMod.py   mymodcore.py
```

- Typical `setup.py` script for this case:

```
#!/usr/bin/env python
from distutils.core import setup

setup(name='MyMod',
      version='1.0',
      description='Python module example',
      author='Hans Petter Langtangen',
      author_email='hpl@ifi.uio.no',
      url='http://www.simula.no/pymod/MyMod',
      py_modules=['MyMod', 'mymodcore'],
     )
```

setup.py with compiled code

- Modules can also make use of Fortran, C, C++ code
- `setup.py` can also list C and C++ files; these will be compiled with the same options/compiler as used for Python itself
- SciPy has an extension of Distutils for “intelligent” compilation of Fortran files
- Note: `setup.py` eliminates the need for makefiles
- Examples of such `setup.py` files are provided in the section on mixing Python with Fortran, C and C++

Installing modules

- Standard command:

```
python setup.py install
```

- If the module contains files to be compiled, a two-step procedure can be invoked

```
python setup.py build
# compiled files and modules are made in subdir. build/
python setup.py install
```

Controlling the installation destination

- `setup.py` has many options
- Control the destination directory for installation:

```
python setup.py install --home=$HOME/install
# copies modules to /home/hpl/install/lib/python
```

- Make sure that `/home/hpl/install/lib/python` is registered in your `PYTHONPATH`

How to learn more about Distutils

- Go to the official electronic Python documentation
- Look up “Distributing Python Modules”
  (for packing modules in `setup.py` scripts)
- Look up “Installing Python Modules”
  (for running `setup.py` with various options)

Doc strings

Contents

- How to document *usage* of Python functions, classes, modules
- Automatic testing of code (through doc strings)

More info

- App. B.1/B.2 in the course book
- HappyDoc, Pydoc, Epydoc manuals
- Style guide for doc strings (see `doc.html`)

Doc strings (1)

- Doc strings = first string in functions, classes, files
- Put user information in doc strings:

```
def ignorecase_sort(a, b):
    """Compare strings a and b, ignoring case."""
    ...
```

- The doc string is available at run time and explains the purpose and usage of the function:

```
>>> print ignorecase_sort.__doc__
'Compare strings a and b, ignoring case.'
```

Doc strings (2)

- Doc string in a class:

```
class MyClass:
    """Fake class just for exemplifying doc strings."""

    def __init__(self):
        ...
```

- Doc strings in modules are a (often multi-line) string starting in the top of the file

```
"""
This module is a fake module
for exemplifying multi-line
doc strings.
"""
```

Doc strings (3)

- The doc string serves two purposes:
  - documentation in the source code
  - on-line documentation through the attribute `__doc__`
  - documentation generated by, e.g., HappyDoc
- HappyDoc: Tool that can extract doc strings and automatically produce overview of Python classes, functions etc.
- Doc strings can, e.g., be used as balloon help in sophisticated GUIs (cf. IDLE)
- Providing doc strings is a good habit!

Doc strings (4)

There is an official style guide for doc strings:

- PEP 257 "Docstring Conventions" from http://www.python.org/dev/peps/
- Use triple double quoted strings as doc strings
- Use complete sentences, ending in a period

```
def somefunc(a, b):
    """Compare a and b."""
```

Automatic doc string testing (1)

- The `doctest` module enables automatic testing of interactive Python sessions embedded in doc strings

```
class StringFunction:
    """
    Make a string expression behave as a Python function
    of one variable.
    Examples on usage:
    >>> from StringFunction import StringFunction
    >>> f = StringFunction('sin(3*x) + log(1+x)')
    >>> p = 2.0; v = f(p)   # evaluate function
    >>> p, v
    (2.0, 0.81919679046918392)
    >>> f = StringFunction('1+t', independent_variables='t')
    >>> v = f(1.2)   # evaluate function of t=1.2
    >>> print "%.2f" % v
    2.20
    >>> f = StringFunction('sin(t)')
    >>> v = f(1.2)   # evaluate function of t=1.2
    Traceback (most recent call last):
        v = f(1.2)
    NameError: name 't' is not defined
    """
```

Automatic doc string testing (2)

- Class `StringFunction` is contained in the module `StringFunction`
- Let `StringFunction.py` execute two statements when run as a script:

```
def _test():
    import doctest
    return doctest.testmod(StringFunction)

if __name__ == '__main__':
    _test()
```

- Run the test:

```
python StringFunction.py        # no output: all tests passed
python StringFunction.py  -v    # verbose output
```

Numerical Python

Contents

- Efficient array computing in Python
- Creating arrays
- Indexing/slicing arrays
- Random numbers
- Linear algebra
- Plotting

More info

- Ch. 4 in the course book
- www.scipy.org
- The NumPy manual
- The SciPy tutorial

Numerical Python (NumPy)

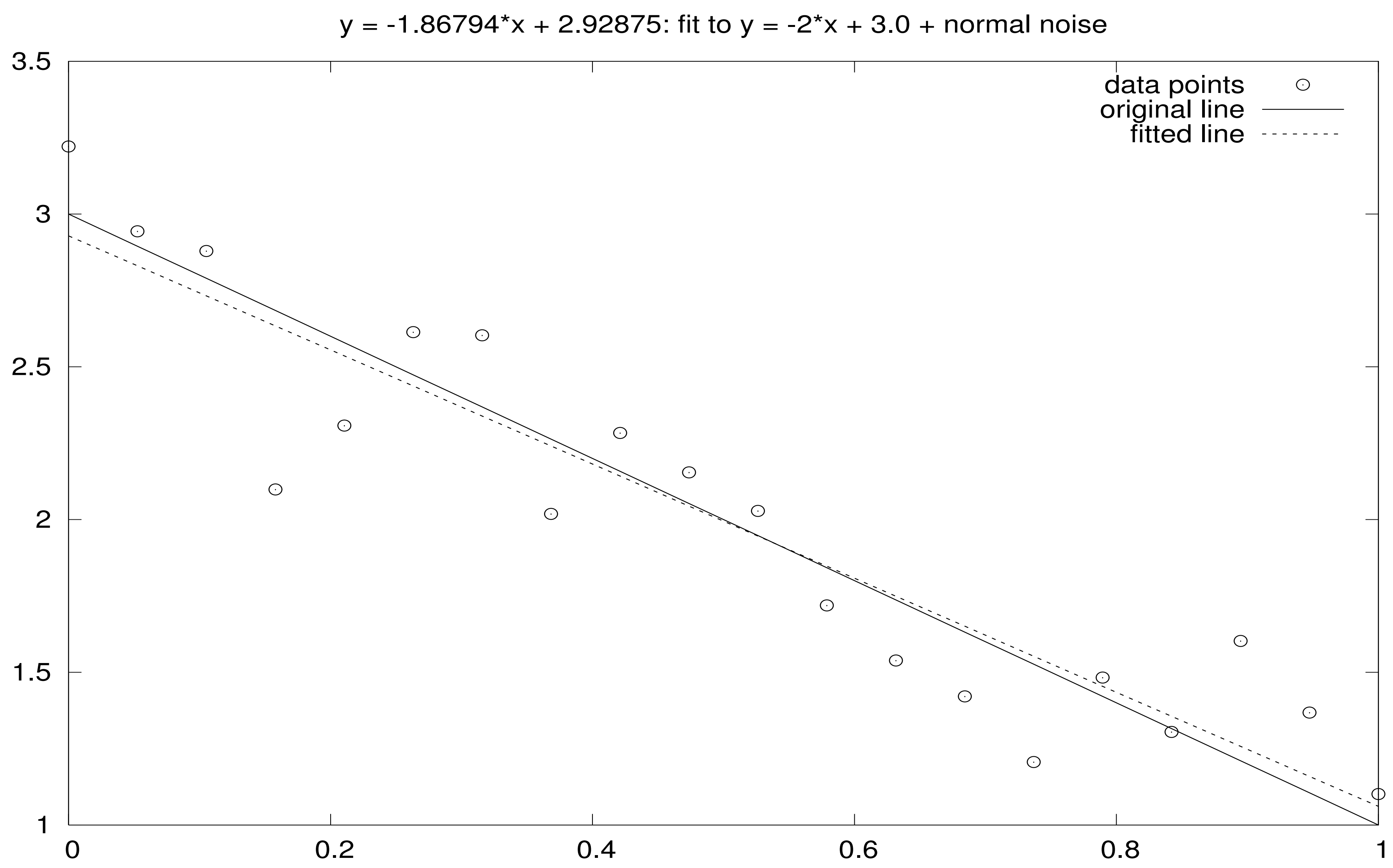
- NumPy enables efficient numerical computing in Python
- NumPy is a package of modules, which offers efficient arrays (contiguous storage) with associated array operations coded in C or Fortran
- There are three implementations of Numerical Python
  - Numeric from the mid 90s (still widely used)
  - numarray from about 2000
  - numpy from 2006
- We recommend to use numpy (by Travis Oliphant)

```
from numpy import *
```

A taste of NumPy: a least-squares procedure

```
x = linspace(0.0, 1.0, n)                  # coordinates
y_line = -2*x + 3
y = y_line + random.normal(0, 0.25, n)   # line with noise

# goal: fit a line to the data points x, y

# create and solve least squares system:
A = array([x, ones(n)])
A = A.transpose()

result = linalg.lstsq(A, y)
# result is a 4-tuple, the solution (a,b) is the 1st entry:
a, b = result[0]

plot(x, y, 'o',          # data points w/noise
     x, y_line, 'r',     # original line
     x, a*x + b, 'b')    # fitted lines
legend('data points', 'original line', 'fitted line')
hardcopy('myplot.png')
```

Resulting plot



Making arrays

```
>>> from numpy import *
>>> n = 4
>>> a = zeros(n)          # one-dim. array of length n
>>> print a
[ 0.  0.  0.  0.]
>>> a
array([ 0.,  0.,  0.,  0.])
>>> p = q = 2
>>> a = zeros((p,q,3))       # p*q*3 three-dim. array
>>> print a
[[[ 0.  0.  0.]
  [ 0.  0.  0.]]

 [[ 0.  0.  0.]
  [ 0.  0.  0.]]]
>>> a.shape                  # a's dimension
(2, 2, 3)
```

Making float, int, complex arrays

```
>>> a = zeros(3)
>>> print a.dtype # a's data type
float64
>>> a = zeros(3, int)
>>> print a
[0 0 0]
>>> print a.dtype
int32
>>> a = zeros(3, float32)    # single precision
>>> print a
[ 0.  0.  0.]
>>> print a.dtype
float32
>>> a = zeros(3, complex)
>>> a
array([ 0.+0.j,  0.+0.j,  0.+0.j])
>>> a.dtype
dtype('complex128')

>>> given an array a, make a new array of same dimension
>>> and data type:
>>> x = zeros(a.shape, a.dtype)
```

Array with a sequence of numbers

- `linspace(a, b, n)` generates `n` uniformly spaced coordinates, starting with `a` and ending with `b`

```
>>> x = linspace(-5, 5, 11)
>>> print x
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

- A special compact syntax is also available:

```
>>> a = r_[-5:5:11j]  # same as linspace(-5, 5, 11)
>>> print a
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

- `arange` works like `range` (`xrange`)

```
>>> x = arange(-5, 5, 1, float)
>>> print x  # upper limit 5 is not included!!
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.]
```

Warning: arange is dangerous

- `arange`'s upper limit may or may not be included (due to round-off errors)
- Better to use a safer method: `seq(start, stop, increment)`

```
>>> from scitools.numpyutils import seq
>>> x = seq(-5, 5, 1)
>>> print x    # upper limit always included
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

- The package `scitools` is available at `http://code.google.com/p/scitools/`

Array construction from a Python list

- `array(list, [datatype])` generates an array from a list:

```
>>> pl = [0, 1.2, 4, -9.1, 5, 8]
>>> a = array(pl)
```

- The array elements are of the simplest possible type:

```
>>> z = array([1, 2, 3])
>>> print z                         # array of integers
[1 2 3]
>>> z = array([1, 2, 3], float)
>>> print z
[ 1.  2.  3.]
```

- A two-dim. array from two one-dim. lists:

```
>>> x = [0, 0.5, 1]; y = [-6.1, -2, 1.2]  # Python lists
>>> a = array([x, y])  # form array with x and y as rows
```

- From array to list: `alist = a.tolist()`

From “anything” to a NumPy array

- Given an object `a`,

```
a = asarray(a)
```

  converts `a` to a NumPy array (if possible/necessary)

- Arrays can be ordered as in C (default) or Fortran:

```
a = asarray(a, order='Fortran')
isfortran(a)   # returns True if a's order is Fortran
```

- Use `asarray` to, e.g., allow flexible arguments in functions:

```
def myfunc(some_sequence):
    a = asarray(some_sequence)
    return 3*a - 5

myfunc([1,2,3])          # list argument
myfunc((-1,1))           # tuple argument
myfunc(zeros(10))        # array argument
myfunc(-4.5)             # float argument
myfunc(6)                # int argument
```

Changing array dimensions

```
>>> a = array([0, 1.2, 4, -9.1, 5, 8])
>>> a.shape = (2,3)        # turn a into a 2x3 matrix
>>> print a
[[ 0.   1.2  4. ]
 [-9.1  5.   8. ]]
>>> a.size
6
>>> a.shape = (a.size,)    # turn a into a vector of length 6 again
>>> a.shape
(6,)
>>> print a
[ 0.   1.2  4.  -9.1  5.   8. ]
>>> a = a.reshape(2,3)     # same effect as setting a.shape
>>> a.shape
(2, 3)
```

Array initialization from a Python function

```
>>> def myfunc(i, j):
...     return (i+1)*(j+4-i)
...
>>> # make 3x6 array where a[i,j] = myfunc(i,j):
>>> a = fromfunction(myfunc, (3,6))
>>> a
array([[  4.,   5.,   6.,   7.,   8.,   9.],
       [  6.,   8.,  10.,  12.,  14.,  16.],
       [  6.,   9.,  12.,  15.,  18.,  21.]])
```

Basic array indexing

Note: all integer indices in Python start at 0!

```
a = linspace(-1, 1, 6)
a[2:4] = -1        # set a[2] and a[3] equal to -1
a[-1]  = a[0]      # set last element equal to first one
a[:]   = 0         # set all elements of a equal to 0
a.fill(0)          # set all elements of a equal to 0

a.shape = (2,3)    # turn a into a 2x3 matrix
print a[0,1]       # print element (0,1)
a[i,j] = 10        # assignment to element (i,j)
a[i][j] = 10       # equivalent syntax (slower)
print a[:,k]       # print column with index k
print a[1,:]       # print second row
a[:,:] = 0         # set all elements of a equal to 0
```

More advanced array indexing

```
>>> a = linspace(0, 29, 30)
>>> a.shape = (5,6)
>>> a
array([[  0.,   1.,   2.,   3.,   4.,   5.,]
       [  6.,   7.,   8.,   9.,  10.,  11.,]
       [ 12.,  13.,  14.,  15.,  16.,  17.,]
       [ 18.,  19.,  20.,  21.,  22.,  23.,]
       [ 24.,  25.,  26.,  27.,  28.,  29.,]])

>>> a[1:3,::2]    # a[i,j] for i=1,2 and j=0,2,4
array([[  6.,    8.,   10.],
       [ 12.,   14.,   16.]])

>>> a[::3,2::2]    # a[i,j] for i=0,3 and j=2,4
array([[  2.,    4.],
       [ 20.,   22.]])

>>> i = slice(None, None, 3);   j = slice(2, None, 2)
>>> a[i,j]
array([[  2.,    4.],
       [ 20.,   22.]])
```

Slices refer the array data

- With `a` as list, `a[:]` makes a copy of the data
- With `a` as array, `a[:]` is a reference to the data

```
>>> b = a[2,:]          # extract 2nd row of a
>>> print a[2,0]
12.0
>>> b[0] = 2
>>> print a[2,0]
2.0                     # change in b is reflected in a!
```

- Take a copy to avoid referencing via slices:

```
>>> b = a[2,:].copy()
>>> print a[2,0]
12.0
>>> b[0] = 2        # b and a are two different arrays now
>>> print a[2,0]
12.0                # a is not affected by change in b
```

Loops over arrays (1)

- Standard loop over each element:

```
for i in xrange(a.shape[0]):
    for j in xrange(a.shape[1]):
        a[i,j] = (i+1)*(j+1)*(j+2)
        print 'a[%d,%d]=%g ' % (i,j,a[i,j]),
    print  # newline after each row
```

- A standard for loop iterates over the first index:

```
>>> print a
[[  2.   6.  12.]
 [  4.  12.  24.]]
>>> for e in a:
...     print e
...
[  2.   6.  12.]
[  4.  12.  24.]
```

Loops over arrays (2)

- View array as one-dimensional and iterate over all elements:

```
for e in a.ravel():
    print e
```

Use `ravel()` only when reading elements, for assigning it is better to use `shape` or `reshape` first!

- For loop over all index tuples and values:

```
>>> for index, value in ndenumerate(a):
...     print index, value
...
(0, 0) 2.0
(0, 1) 6.0
(0, 2) 12.0
(1, 0) 4.0
(1, 1) 12.0
(1, 2) 24.0
```

Array computations

- Arithmetic operations can be used with arrays:

  ```
  b = 3*a - 1    # a is array, b becomes array
  ```

  1) compute `t1 = 3*a`, 2) compute `t2= t1 - 1`, 3) set `b = t2`

- Array operations are much faster than element-wise operations:

  ```
  >>> import time  # module for measuring CPU time
  >>> a = linspace(0, 1, 1E+07)  # create some array
  >>> t0 = time.clock()
  >>> b = 3*a -1
  >>> t1 = time.clock()   # t1-t0 is the CPU time of 3*a-1

  >>> for i in xrange(a.size): b[i] = 3*a[i] - 1
  >>> t2 = time.clock()
  >>> print '3*a-1: %g sec, loop: %g sec' % (t1-t0, t2-t1)
  3*a-1: 2.09 sec, loop: 31.27 sec
  ```

Standard math functions can take array arguments

```
# let b be an array

c = sin(b)
c = arcsin(c)
c = sinh(b)
# same functions for the cos and tan families

c = b**2.5  # power function
c = log(b)
c = exp(b)
c = sqrt(b)
```

Other useful array operations

```
# a is an array

a.clip(min=3, max=12)   # clip elements
a.mean(); mean(a)       # mean value
a.var();  var(a)        # variance
a.std();  std(a)        # standard deviation
median(a)
cov(x,y)                # covariance
trapz(a)                # Trapezoidal integration
diff(a)                 # finite differences (da/dx)

# more Matlab-like functions:
corrcoeff, cumprod, diag, eig, eye, fliplr, flipud, max, min,
prod, ptp, rot90, squeeze, sum, svd, tri, tril, triu
```

More useful array methods and attributes

```
>>> a = zeros(4) + 3
>>> a
array([ 3.,  3.,  3.,  3.])  # float data
>>> a.item(2)                # more efficient than a[2]
3.0
>>> a.itemset(3,-4.5)        # more efficient than a[3]=-4.5
>>> a
array([ 3. ,  3. ,  3. , -4.5])
>>> a.shape = (2,2)
>>> a
array([[ 3. ,  3. ],
       [ 3. , -4.5]])
>>> a.ravel()                # from multi-dim to one-dim
array([ 3. ,  3. ,  3. , -4.5])
>>> a.ndim                   # no of dimensions
2
>>> len(a.shape)             # no of dimensions
2
>>> rank(a)                  # no of dimensions
2
>>> a.size                   # total no of elements
4
>>> b = a.astype(int)        # change data type
>>> b
array([3, 3, 3, 3])
```

Modules for curve plotting and 2D/3D visualization

- Matplotlib (curve plotting, 2D scalar and vector fields)
- PyX (PostScript/TeX-like drawing)
- Interface to Gnuplot
- Interface to Vtk
- Interface to OpenDX
- Interface to IDL
- Interface to Grace
- Interface to Matlab
- Interface to R
- Interface to Blender

Curve plotting with Easyviz

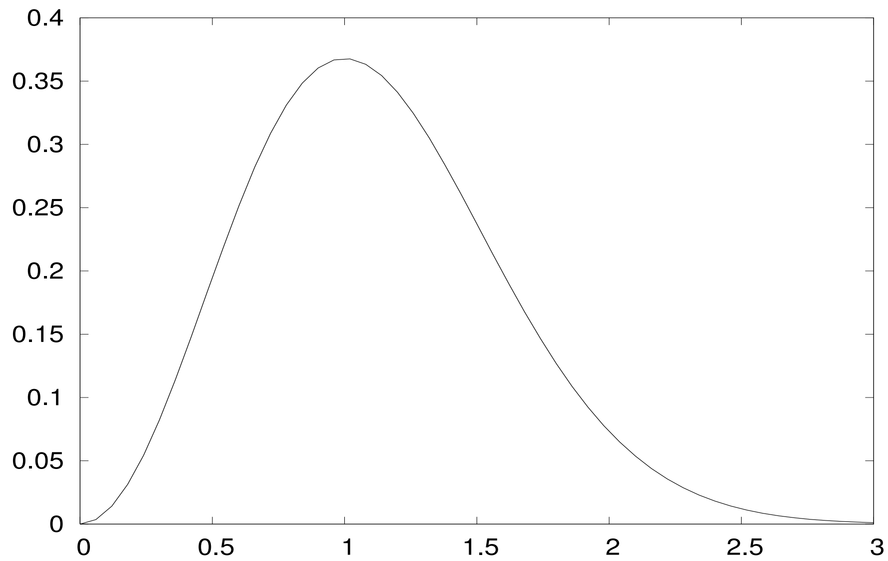
- Easyviz is a light-weight interface to many plotting packages, using a Matlab-like syntax
- Goal: write your program using Easyviz (“Matlab”) syntax and postpone your choice of plotting package
- Note: some powerful plotting packages (Vtk, R, matplotlib, ...) may be troublesome to install, while Gnuplot is easily installed on all platforms
- Easyviz supports (only) the most common plotting commands
- Easyviz is part of SciTools (Simula development)

```
from scitools.all import *
```

(imports all of `numpy`, all of `easyviz`, plus `scitools`)

Basic Easyviz example

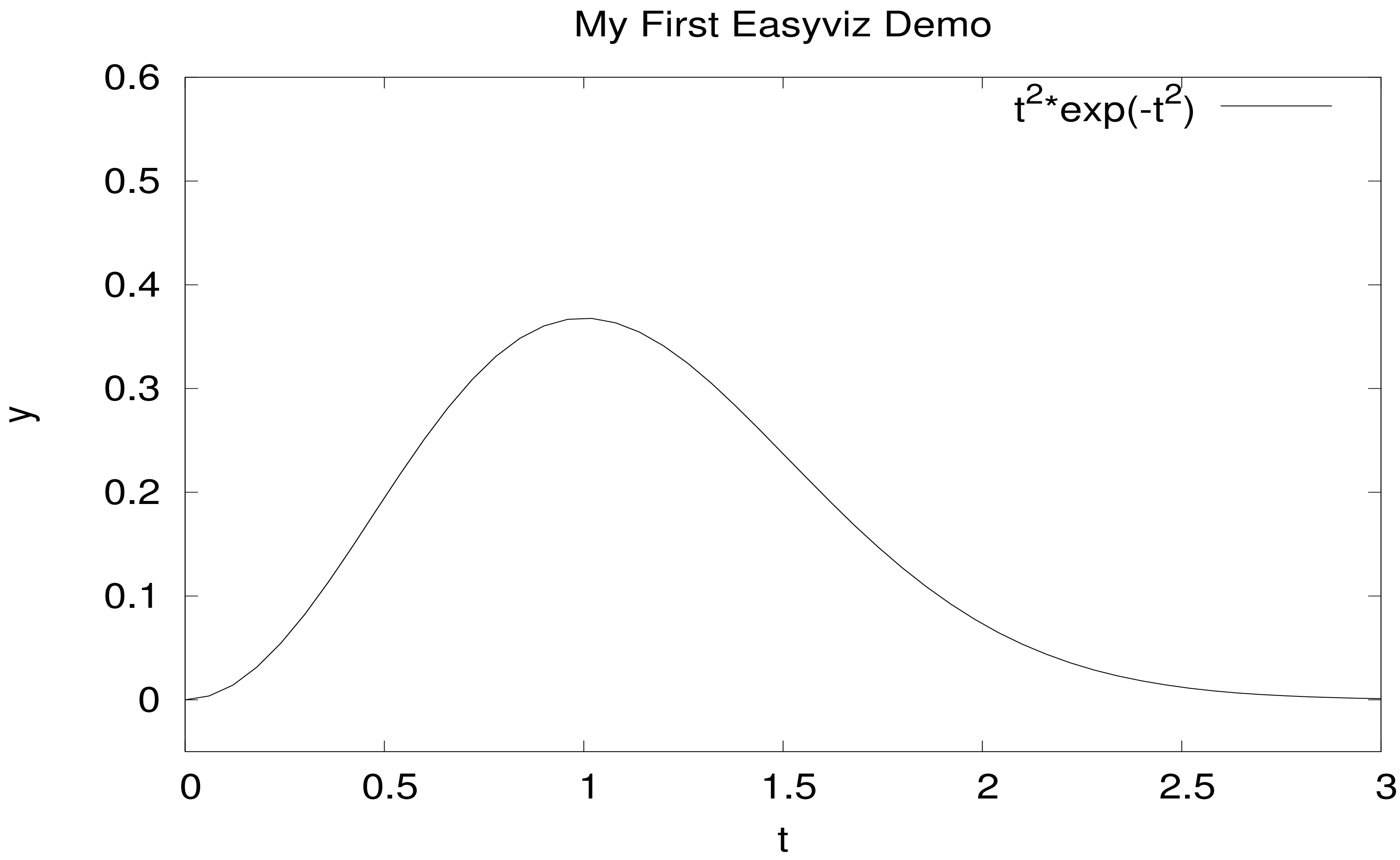
```
from scitools.all import *   # import numpy and plotting
t = linspace(0, 3, 51)       # 51 points between 0 and 3
y = t**2*exp(-t**2)          # vectorized expression
plot(t, y)
hardcopy('tmp1.eps')   # make PostScript image for reports
hardcopy('tmp1.png')   # make PNG image for web pages
```



Decorating the plot

```
plot(t, y)

xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)')
axis([0, 3, -0.05, 0.6])   # [tmin, tmax, ymin, ymax]
title('My First Easyviz Demo')

# or
plot(t, y, xlabel='t', ylabel='y',
     legend='t^2*exp(-t^2)',
     axis=[0, 3, -0.05, 0.6],
     title='My First Easyviz Demo',
     hardcopy='tmp1.eps',
     show=True)   # display on the screen (default)
```

The resulting plot

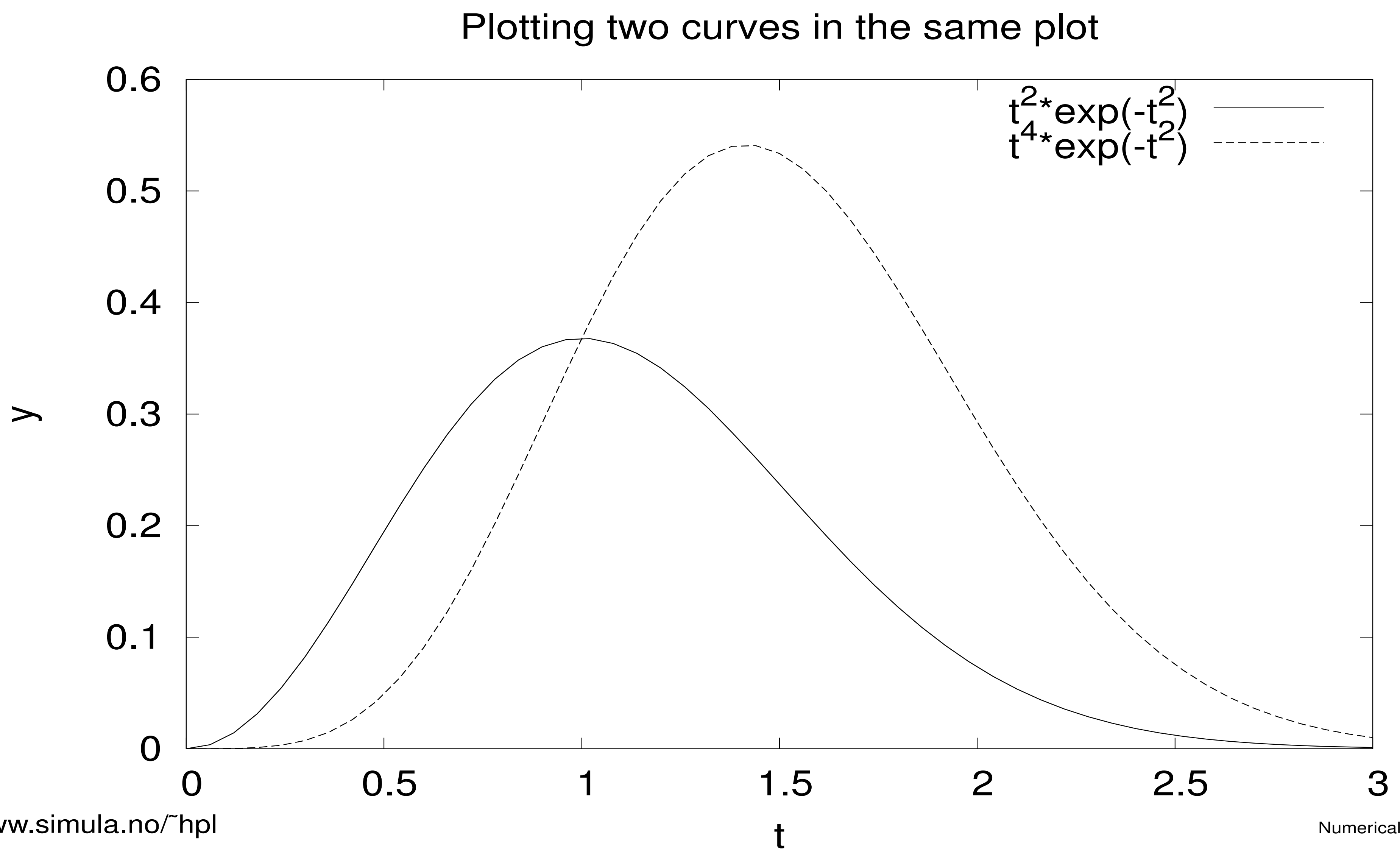


Plotting several curves in one plot

Compare $f_1(t) = t^2 e^{-t^2}$ and $f_2(t) = t^4 e^{-t^2}$ for $t \in [0, 3]$

```
from scitools.all import *    # for curve plotting

def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

plot(t, y1)
hold('on')    # continue plotting in the same plot
plot(t, y2)

xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)')
title('Plotting two curves in the same plot')
hardcopy('tmp2.eps')
```

The resulting plot



Example: plot a function given on the command line

- Task: plot (e.g.) $f(x) = e^{-0.2x}\sin(2\pi x)$ for $x \in [0, 4\pi]$
- Specify $f(x)$ and x interval as text on the command line:

```
Unix/DOS> python plotf.py "exp(-0.2*x)*sin(2*pi*x)" 0 4*pi
```

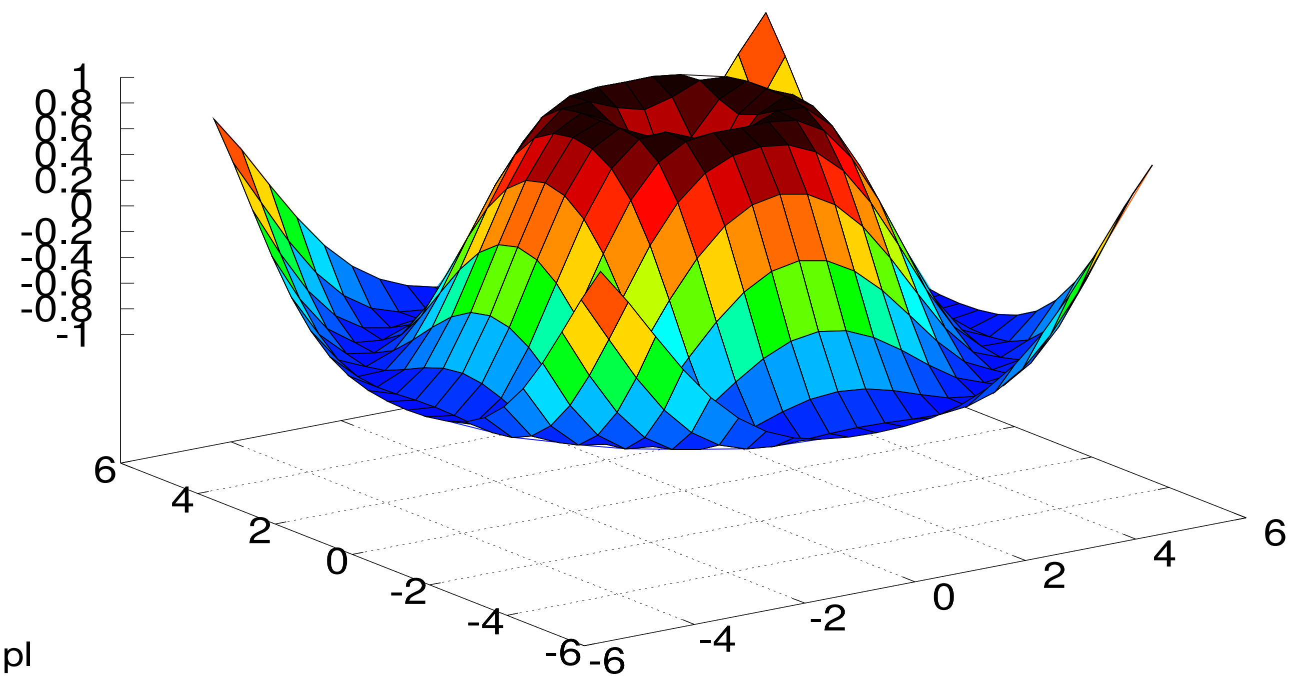
- Program:

```
from scitools.all import *
formula = sys.argv[1]
xmin = eval(sys.argv[2])
xmax = eval(sys.argv[3])

x = linspace(xmin, xmax, 101)
y = eval(formula)
plot(x, y, title=formula)
```

- Thanks to `eval`, input (text) with correct Python syntax can be turned to running code on the fly

Plotting 2D scalar fields

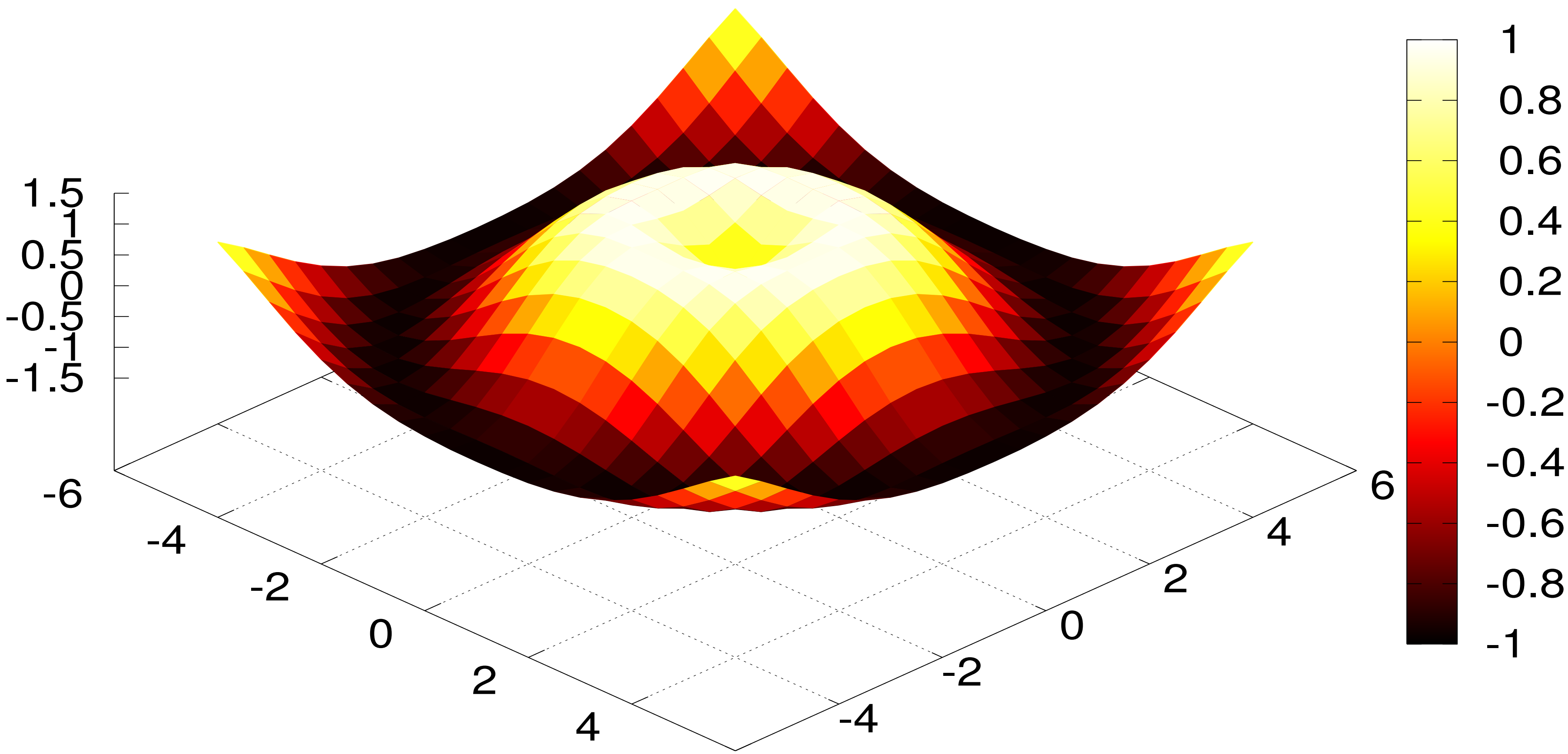
```
from scitools.all import *

x = y = linspace(-5, 5, 21)
xv, yv = ndgrid(x, y)
values = sin(sqrt(xv**2 + yv**2))
surf(xv, yv, values)
```



Adding plot features

```
# Matlab style commands:
setp(interactive=False)
surf(xv, yv, values)
shading('flat')
colorbar()
colormap(hot())
axis([-6,6,-6,6,-1.5,1.5])
view(35,45)
show()

# Optional Easyviz (Pythonic) short cut:
surf(xv, yv, values,
     shading='flat',
     colorbar='on',
     colormap=hot(),
     axis=[-6,6,-6,6,-1.5,1.5],
     view=[35,45])
```

The resulting plot



Other commands for visualizing 2D scalar fields

- `contour` (standard contours)), `contourf` (filled contours), `contour3` (elevated contours)
- `mesh` (elevated mesh), `meshc` (elevated mesh with contours in the xy plane)
- `surf` (colored surface), `surfc` (colored surface with contours in the xy plane)
- `pcolor` (colored cells in a 2D mesh)

Commands for visualizing 3D fields

Scalar fields:

- `isosurface`
- `slice_` (colors in slice plane),
  `contourslice` (contours in slice plane)

Vector fields:

- `quiver3` (arrows), (`quiver` for 2D vector fields)
- `streamline`, `streamtube`, `streamribbon` (flow sheets)

More info about Easyviz

- A plain text version of the Easyviz manual:

  `pydoc scitools.easyviz`

- The HTML version:

  `http://code.google.com/p/scitools/wiki/EasyvizDocumentation`

- Download SciTools (incl. Easyviz):

  `http://code.google.com/p/scitools/`

Class programming in Python

Contents

- Intro to the class syntax
- Special attributes
- Special methods
- Classic classes, new-style classes
- Static data, static functions
- Properties
- About scope

More info

- Ch. 8.6 in the course book
- Python Tutorial
- Python Reference Manual (special methods in 3.3)
- Python in a Nutshell (OOP chapter - recommended!)

Classes in Python

- Similar class concept as in Java and C++
- All functions are virtual
- No private/protected variables
  (the effect can be "simulated")
- Single and multiple inheritance
- Everything in Python is a class and works with classes
- Class programming is easier and faster than in C++ and Java (?)

The basics of Python classes

- Declare a base class `MyBase`:

```
class MyBase:

    def __init__(self,i,j):   # constructor
        self.i = i; self.j = j

    def write(self):          # member function
        print 'MyBase: i=',self.i,'j=',self.j
```

- `self` is a reference to this object
- Data members are prefixed by self:
  `self.i`, `self.j`
- All functions take `self` as first argument in the declaration, but not in the call

```
inst1 = MyBase(6,9); inst1.write()
```

Implementing a subclass

- Class `MySub` is a subclass of `MyBase`:

```
class MySub(MyBase):

    def __init__(self,i,j,k):  # constructor
        MyBase.__init__(self,i,j)
        self.k = k;

    def write(self):
        print 'MySub: i=',self.i,'j=',self.j,'k=',self.k
```

- Example:

```
# this function works with any object that has a write func:
def write(v): v.write()

# make a MySub instance
i = MySub(7,8,9)

write(i)    # will call MySub's write
```

Comment on object-orientation

- Consider

```
def write(v):
    v.write()

write(i)    # i is MySub instance
```

- In C++/Java we would declare `v` as a `MyBase` reference and rely on `i.write()` as calling the virtual function `write` in `MySub`
- The same works in Python, but we do not need inheritance and virtual functions here: `v.write()` will work for *any* object `v` that has a callable attribute `write` that takes no arguments
- Object-orientation in C++/Java for parameterizing types is not needed in Python since variables are not declared with types

Private/non-public data

- There is no technical way of preventing users from manipulating data and methods in an object
- Convention: attributes and methods starting with an underscore are treated as non-public (“protected”)
- Names starting with a double underscore are considered strictly private (Python mangles class name with method name in this case: `obj.__some` has actually the name `_obj__some`)

```
class MyClass:
    def __init__(self):
        self._a = False      # non-public
        self.b = 0           # public
        self.__c = 0         # private
```

Special attributes

`i1` is `MyBase`, `i2` is `MySub`

- Dictionary of user-defined attributes:

```
>>> i1.__dict__   # dictionary of user-defined attributes
{'i': 5, 'j': 7}
>>> i2.__dict__
{'i': 7, 'k': 9, 'j': 8}
```

- Name of class, name of method:

```
>>> i2.__class__.__name__ # name of class
'MySub'
>>> i2.write.__name__     # name of method
'write'
```

- List names of all methods and attributes:

```
>>> dir(i2)
['__doc__', '__init__', '__module__', 'i', 'j', 'k', 'write']
```

Testing on the class type

- Use `isinstance` for testing class type:

```
if isinstance(i2, MySub):
    # treat i2 as a MySub instance
```

- Can test if a class is a subclass of another:

```
if issubclass(MySub, MyBase):
    ...
```

- Can test if two objects are of the same class:

```
if inst1.__class__ is inst2.__class__
```

  (`is` checks object identity, == checks for equal contents)

- `a.__class__` refers the class object of instance `a`

Creating attributes on the fly

- Attributes can be added at run time (!)

```
>>> class G: pass

>>> g = G()
>>> dir(g)
['__doc__', '__module__']  # no user-defined attributes

>>> # add instance attributes:
>>> g.xmin=0; g.xmax=4; g.ymin=0; g.ymax=1
>>> dir(g)
['__doc__', '__module__', 'xmax', 'xmin', 'ymax', 'ymin']
>>> g.xmin, g.xmax, g.ymin, g.ymax
(0, 4, 0, 1)

>>> # add static variables:
>>> G.xmin=0; G.xmax=2; G.ymin=-1; G.ymax=1
>>> g2 = G()
>>> g2.xmin, g2.xmax, g2.ymin, g2.ymax  # static variables
(0, 2, -1, 1)
```

Another way of adding new attributes

- Can work with `__dict__` directly:

```
>>> i2.__dict__['q'] = 'some string'
>>> i2.q
'some string'
>>> dir(i2)
['__doc__', '__init__', '__module__',
 'i', 'j', 'k', 'q', 'write']
```

Special methods

- Special methods have leading and trailing double underscores (e.g. `__str__`)
- Here are some operations defined by special methods:

```
len(a)                  # a.__len__()
c = a*b                 # c = a.__mul__(b)
a = a+b                 # a = a.__add__(b)
a += c                  # a.__iadd__(c)
d = a[3]                # d = a.__getitem__(3)
a[3] = 0                # a.__setitem__(3, 0)
f = a(1.2, True)        # f = a.__call__(1.2, True)
if a:                   # if a.__len__()>0: or if a.__nonzero():
```

Example: functions with extra parameters

- Suppose we need a function of `x` and `y` with three additional parameters `a`, `b`, and `c`:

```
def f(x, y, a, b, c):
    return a + b*x + c*y*y
```

- Suppose we need to send this function to another function

```
def gridvalues(func, xcoor, ycoor, file):
    for i in range(len(xcoor)):
        for j in range(len(ycoor)):
            f = func(xcoor[i], ycoor[j])
            file.write('%g %g %g\n' % (xcoor[i], ycoor[j], f)
```

  `func` is expected to be a function of `x` and `y` only (many libraries need to make such assumptions!)

- How can we send our `f` function to `gridvalues`?

Possible (inferior) solutions

- Bad solution 1: global parameters

```
global a, b, c
...
def f(x, y):
    return a + b*x + c*y*y

...
a = 0.5;  b = 1;  c = 0.01
gridvalues(f, xcoor, ycoor, somefile)
```

  Global variables are usually considered evil

- Bad solution 2: keyword arguments for parameters

```
def f(x, y, a=0.5, b=1, c=0.01):
    return a + b*x + c*y*y

...
gridvalues(f, xcoor, ycoor, somefile)
```

  useless for other values of `a`, `b`, `c`

Solution: class with call operator

- Make a class with function behavior instead of a pure function
- The parameters are class attributes
- Class instances can be called as ordinary functions, now with `x` and `y` as the only formal arguments

```
class F:
    def __init__(self, a=1, b=1, c=1):
        self.a = a;  self.b = b;  self.c = c

    def __call__(self, x, y):    # special method!
        return self.a + self.b*x + self.c*y*y

f = F(a=0.5, c=0.01)
# can now call f as
v = f(0.1, 2)
...
gridvalues(f, xcoor, ycoor, somefile)
```

Alternative solution: Closure

- Make a function that locks the namespace and constructs and returns a tailor made function

```
def F(a=1,b=1,c=1):

    def f(x, y):
        return a + b*x + c*y*y

    return f

f = F(a=0.5, c=0.01)
# can now call f as
v = f(0.1, 2)
...
gridvalues(f, xcoor, ycoor, somefile)
```

Some special methods

- `__init__(self [, args])`: constructor
- `__del__(self)`: destructor (seldom needed since Python offers automatic garbage collection)
- `__str__(self)`: string representation for pretty printing of the object (called by `print` or `str`)
- `__repr__(self)`: string representation for initialization (`a==eval(repr(a))` is true)

Comparison, length, call

- `__eq__(self, x)`: for equality (`a==b`), should return `True` or `False`
- `__cmp__(self, x)`: for comparison (`<, <=, >, >=, ==, !=`); return negative integer, zero or positive integer if `self` is less than, equal or greater than `x` (resp.)
- `__len__(self)`: length of object (called by `len(x)`)
- `__call__(self [, args])`: calls like `a(x,y)` implies `a.__call__(x,y)`

Indexing and slicing

- `__getitem__(self, i)`: used for subscripting:
  `b = a[i]`
- `__setitem__(self, i, v)`: used for subscripting: `a[i] = v`
- `__delitem__(self, i)`: used for deleting: `del a[i]`
- These three functions are also used for slices:
  `a[p:q:r]` implies that `i` is a `slice object` with attributes `start` (p), `stop` (q) and `step` (r)

```
b = a[:-1]
# implies
b = a.__getitem__(i)
isinstance(i, slice) is True
i.start is None
i.stop  is -1
i.step  is None
```

Arithmetic operations

- `__add__(self, b)`: used for `self+b`, i.e., `x+y` implies `x.__add__(y)`
- `__sub__(self, b)`: `self-b`
- `__mul__(self, b)`: `self*b`
- `__div__(self, b)`: `self/b`
- `__pow__(self, b)`: `self**b` or `pow(self,b)`

In-place arithmetic operations

- `__iadd__(self, b): self += b`
- `__isub__(self, b): self -= b`
- `__imul__(self, b): self *= b`
- `__idiv__(self, b): self /= b`

Right-operand arithmetics

- `__radd__(self, b)`: This method defines `b+self`, while `__add__(self, b)` defines `self+b`. If `a+b` is encountered and `a` does not have an `__add__` method, `b.__radd__(a)` is called if it exists (otherwise `a+b` is not defined).
- Similar methods: `__rsub__`, `__rmul__`, `__rdiv__`

Type conversions

- `__int__(self)`: conversion to integer (`int(a)` makes an `a.__int__()` call)
- `__float__(self)`: conversion to float
- `__hex__(self)`: conversion to hexadecimal number

Documentation of special methods: see the *Python Reference Manual* (not the Python Library Reference!), follow link from index “overloading - operator”

Boolean evaluations

- `if a:`
  when is `a` evaluated as true?
- If `a` has `__len__` or `__nonzero__` and the return value is 0 or `False`, `a` evaluates to false
- Otherwise: `a` evaluates to true
- Implication: no implementation of `__len__` or `__nonzero__` implies that `a` evaluates to true!!
- `while a` follows (naturally) the same set-up

Example on call operator: StringFunction

- Matlab has a nice feature: mathematical formulas, written as text, can be turned into callable functions
- A similar feature in Python would be like

```
f = StringFunction_v1('1+sin(2*x)')
print f(1.2)   # evaluates f(x) for x=1.2
```

- `f(x)` implies `f.__call__(x)`
- Implementation of class `StringFunction_v1` is compact! (see next slide)

Implementation of StringFunction classes

- Simple implementation:

```
class StringFunction_v1:
    def __init__(self, expression):
        self._f = expression

    def __call__(self, x):
        return eval(self._f)   # evaluate function expression
```

- Problem: `eval(string)` is slow; should pre-compile expression

```
class StringFunction_v2:
    def __init__(self, expression):
        self._f_compiled = compile(expression,
                                   '<string>', 'eval')

    def __call__(self, x):
        return eval(self._f_compiled)
```

New-style classes

- The class concept was redesigned in Python v2.2
- We have *new-style* (v2.2) and *classic* classes
- New-style classes add some convenient functionality to classic classes
- New-style classes must be derived from the `object` base class:

```
class MyBase(object):
    # the rest of MyBase is as before
```

Static data

- Static data (or class variables) are common to all instances

```
>>> class Point:
        counter = 0 # static variable, counts no of instances
        def __init__(self, x, y):
                self.x = x;  self.y = y;
                Point.counter += 1

>>> for i in range(1000):
        p = Point(i*0.01, i*0.001)

>>> Point.counter        # access without instance
1000
>>> p.counter            # access through instance
1000
```

Static methods

- New-style classes allow static methods
  (methods that can be called without having an instance)

```
class Point(object):
    _counter = 0
    def __init__(self, x, y):
        self.x = x;  self.y = y;  Point._counter += 1
    def ncopies(): return Point._counter
    ncopies = staticmethod(ncopies)
```

- Calls:

```
>>> Point.ncopies()
0
>>> p = Point(0, 0)
>>> p.ncopies()
1
>>> Point.ncopies()
1
```

- Cannot access `self` or class attributes in static methods

Properties

- Python 2.3 introduced “intelligent” assignment operators, known as *properties*
- That is, assignment may imply a function call:

```
x.data = mydata;      yourdata = x.data
# can be made equivalent to
x.set_data(mydata);  yourdata = x.get_data()
```

- Construction:

```
class MyClass(object):    # new-style class required!
    ...
    def set_data(self, d):
        self._data = d
        <update other data structures if necessary...>

    def get_data(self):
        <perform actions if necessary...>
        return self._data

    data = property(fget=get_data, fset=set_data)
```

Attribute access; traditional

- Direct access:

```
my_object.attr1 = True
a = my_object.attr1
```

- get/set functions:

```
class A:
    def set_attr1(attr1):
        self._attr1 = attr # underscore => non-public variable
        self._update(self._attr1)   # update internal data too
    ...

my_object.set_attr1(True)

a = my_object.get_attr1()
```

Tedious to write! Properties are simpler...

Attribute access; recommended style

- Use direct access if user is allowed to read *and* assign values to the attribute
- Use properties to restrict access, with a corresponding underlying non-public class attribute
- Use properties when assignment or reading requires a set of associated operations
- Never use get/set functions explicitly
- Attributes and functions are somewhat interchanged in this scheme ⇒ that's why we use the same naming convention

```
myobj.compute_something()
myobj.my_special_variable = yourobj.find_values(x,y)
```

More about scope

- Example: `a` is global, local, and class attribute

```
a = 1                      # global variable

def f(x):
    a = 2                  # local variable

class B:
    def __init__(self):
        self.a = 3         # class attribute

    def scopes(self):
        a = 4              # local (method) variable
```

- Dictionaries with variable names as keys and variables as values:

```
locals()     : local variables
globals()    : global variables
vars()       : local variables
vars(self)   : class attributes
```

Demonstration of scopes (1)

- Function scope:

```
>>> a = 1
>>> def f(x):
        a = 2                   # local variable
        print 'locals:', locals(), 'local a:', a
        print 'global a:', globals()['a']

>>> f(10)
locals: {'a': 2, 'x': 10} local a: 2
global a: 1
```

`a` refers to local variable

Demonstration of scopes (2)

- Class:

```
class B:
    def __init__(self):
        self.a = 3      # class attribute

    def scopes(self):
        a = 4           # local (method) variable
        print 'locals:', locals()
        print 'vars(self):', vars(self)
        print 'self.a:', self.a
        print 'local a:', a, 'global a:', globals()['a']
```

- Interactive test:

```
>>> b=B()
>>> b.scopes()
locals: {'a': 4, 'self': <scope.B instance at 0x4076fb4c>}
vars(self): {'a': 3}
self.a: 3
local a: 4 global a: 1
```

Demonstration of scopes (3)

- Variable interpolation with `vars`:

```
class C(B):
    def write(self):
        local_var = -1
        s = '%(local_var)d %(global_var)d %(a)s' % vars()
```

- Problem: `vars()` returns dict with local variables and the string needs global, local, and class variables
- Primary solution: use printf-like formatting:

```
s = '%d %d %d' % (local_var, global_var, self.a)
```

- More exotic solution:

```
all = {}
for scope in (locals(), globals(), vars(self)):
    all.update(scope)
s = '%(local_var)d %(global_var)d %(a)s' % all
```

(but now we overwrite `a`...)

Namespaces for exec and eval

- `exec` and `eval` may take dictionaries for the global and local namespace:

```
exec code in globals, locals
eval(expr, globals, locals)
```

- Example:

```
a = 8;  b = 9
d = {'a':1, 'b':2}
eval('a + b', d)   # yields 3
```

and

```
from math import *
d['b'] = pi
eval('a+sin(b)', globals(), d)   # yields 1
```

- Creating such dictionaries can be handy

Generalized StringFunction class (1)

- Recall the StringFunction-classes for turning string formulas into callable objects

```
f = StringFunction('1+sin(2*x)')
print f(1.2)
```

- We would like:
  - an arbitrary name of the independent variable
  - parameters in the formula

```
f = StringFunction_v3('1+A*sin(w*t)',
                      independent_variable='t',
                      set_parameters='A=0.1; w=3.14159')
print f(1.2)
f.set_parameters('A=0.2; w=3.14159')
print f(1.2)
```

First implementation

- Idea: hold independent variable and “set parameters” code as strings
- Exec these strings (to bring the variables into play) right before the formula is evaluated

```
class StringFunction_v3:
    def __init__(self, expression, independent_variable='x',
                 set_parameters=''):
        self._f_compiled = compile(expression,
                                   '<string>', 'eval')
        self._var = independent_variable  # 'x', 't' etc.
        self._code = set_parameters

    def set_parameters(self, code):
        self._code = code

    def __call__(self, x):
        exec '%s = %g' % (self._var, x)  # assign indep. var.
        if self._code:  exec(self._code) # parameters?
        return eval(self._f_compiled)
```

Efficiency tests

- The exec used in the `__call__` method is slow!
- Think of a hardcoded function,

```
def f1(x):
    return sin(x) + x**3 + 2*x
```

  and the corresponding `StringFunction`-like objects
- Efficiency test (time units to the right):

```
f1                  :   1
StringFunction_v1: 13
StringFunction_v2:  2.3
StringFunction_v3: 22
```

  Why?
- eval w/compile is important; exec is very slow

A more efficient StringFunction (1)

- Ideas: hold parameters in a dictionary, set the independent variable into this dictionary, run eval with this dictionary as local namespace
- Usage:

```
f = StringFunction_v4('1+A*sin(w*t)', A=0.1, w=3.14159)
f.set_parameters(A=2)    # can be done later
```

A more efficient StringFunction (2)

- Code:

```
class StringFunction_v4:
    def __init__(self, expression, **kwargs):
        self._f_compiled = compile(expression,
                                   '<string>', 'eval')
        self._var = kwargs.get('independent_variable', 'x')
        self._prms = kwargs
        try:    del self._prms['independent_variable']
        except: pass

    def set_parameters(self, **kwargs):
        self._prms.update(kwargs)

    def __call__(self, x):
        self._prms[self._var] = x
        return eval(self._f_compiled, globals(), self._prms)
```

Extension to many independent variables

- We would like arbitrary functions of arbitrary parameters and independent variables:

```
f = StringFunction_v5('A*sin(x)*exp(-b*t)', A=0.1, b=1,
                      independent_variables=('x','t'))
print f(1.5, 0.01)  # x=1.5, t=0.01
```

- Idea: add functionality in subclass

```
class StringFunction_v5(StringFunction_v4):
    def __init__(self, expression, **kwargs):
        StringFunction_v4.__init__(self, expression, **kwargs)
        self._var = tuple(kwargs.get('independent_variables',
                                     'x'))
        try:    del self._prms['independent_variables']
        except: pass

    def __call__(self, *args):
        for name, value in zip(self._var, args):
            self._prms[name] = value  # add indep. variable
        return eval(self._f_compiled,
                    globals(), self._prms)
```

Efficiency tests

- Test function: `sin(x) + x**3 + 2*x`

```
f1                  :  1
StringFunction_v1: 13       (because of uncompiled eval)
StringFunction_v2:  2.3
StringFunction_v3: 22       (because of exec in __call__)
StringFunction_v4:  2.3
StringFunction_v5:  3.1     (because of loop in __call__)
```

Removing all overhead

- Instead of `eval` in `__call__` we may build a (lambda) function

```
class StringFunction:
    def _build_lambda(self):
        s = 'lambda ' + ', '.join(self._var)
        # add parameters as keyword arguments:
        if self._prms:
            s += ', ' + ', '.join(['%s=%s' % (k, self._prms[k]
                                    for k in self._prms])
        s += ': ' + self._f
        self.__call__ = eval(s, globals())
```

- For a call

```
f = StringFunction('A*sin(x)*exp(-b*t)', A=0.1, b=1,
                   independent_variables=('x','t'))
```

the `s` looks like

```
lambda x, t, A=0.1, b=1: return A*sin(x)*exp(-b*t)
```

Final efficiency test

- `StringFunction` objects are as efficient as similar hardcoded objects, i.e.,

  ```
  class F:
      def __call__(self, x, y):
          return sin(x)*cos(y)
  ```

  but there is some overhead associated with the `__call__` op.

- Trick: extract the underlying method and call it directly

  ```
  f1 = F()
  f2 = f1.__call__
  # f2(x,y) is faster than f1(x,y)
  ```

  Can typically reduce CPU time from 1.3 to 1.0

- Conclusion: now we can grab formulas from command-line, GUI, Web, anywhere, and turn them into callable Python functions *without any overhead*

Adding pretty print and reconstruction

- "Pretty print":

```
class StringFunction:
    ...
    def __str__(self):
        return self._f  # just the string formula
```

- Reconstruction: `a = eval(repr(a))`

```
    # StringFunction('1+x+a*y',
                     independent_variables=('x','y'),
                     a=1)

    def __repr__(self):
        kwargs = ', '.join(['%s=%s' % (key, repr(value)) \
                     for key, value in self._prms.items()])
        return "StringFunction1(%s, independent_variable=%s"
          ", %s)" % (repr(self._f), repr(self._var), kwargs)
```

Examples on StringFunction functionality (1)

```
>>> from scitools.StringFunction import StringFunction
>>> f = StringFunction('1+sin(2*x)')
>>> f(1.2)
1.675463180551151l

>>> f = StringFunction('1+sin(2*t)', independent_variables='t')
>>> f(1.2)
1.6754631805511511

>>> f = StringFunction('1+A*sin(w*t)', independent_variables='t', \
                       A=0.1, w=3.14159)
>>> f(1.2)
0.94122173238695939
>>> f.set_parameters(A=1, w=1)
>>> f(1.2)
1.9320390859672263

>>> f(1.2, A=2, w=1)    # can also set parameters in the call
2.8640781719344526
```

Examples on StringFunction functionality (2)

```
>>> # function of two variables:
>>> f = StringFunction('1+sin(2*x)*cos(y)', \
                       independent_variables=('x','y'))
>>> f(1.2,-1.1)
1.3063874788637866

>>> f = StringFunction('1+V*sin(w*x)*exp(-b*t)', \
                       independent_variables=('x','t'))
>>> f.set_parameters(V=0.1, w=1, b=0.1)
>>> f(1.0,0.1)
1.0833098208613807
>>> str(f)  # print formula with parameters substituted by values
'1+0.1*sin(1*x)*exp(-0.1*t)'
>>> repr(f)
"StringFunction('1+V*sin(w*x)*exp(-b*t)',
independent_variables=('x', 't'), b=0.10000000000000001,
w=1, V=0.10000000000000001)"

>>> # vector field of x and y:
>>> f = StringFunction('[a+b*x,y]', \
                           independent_variables=('x','y'))
>>> f.set_parameters(a=1, b=2)
>>> f(2,1)  # [1+2*2, 1]
[5, 1]
```

Exercise

- Implement a class for vectors in 3D
- Application example:

```
>>> from Vec3D import Vec3D
>>> u = Vec3D(1, 0, 0)  # (1,0,0) vector
>>> v = Vec3D(0, 1, 0)
>>> print u**v # cross product
(0, 0, 1)
>>> len(u)      # Eucledian norm
1.0
>>> u[1]        # subscripting
0
>>> v[2]=2.5    # subscripting w/assignment
>>> u+v         # vector addition
(1, 1, 2.5)
>>> u-v         # vector subtraction
(1, -1, -2.5)
>>> u*v         # inner (scalar, dot) product
0
>>> str(u)      # pretty print
'(1, 0, 0)'
>>> repr(u)     # u = eval(repr(u))
'Vec3D(1, 0, 0)'
```

Exercise, 2nd part

- Make the arithmetic operators $+$, $-$ and $*$ more intelligent:

```
u = Vec3D(1, 0, 0)
v = Vec3D(0, -0.2, 8)
a = 1.2
u+v  # vector addition
a+v  # scalar plus vector, yields (1.2, 1, 9.2)
v+a  # vector plus scalar, yields (1.2, 1, 9.2)
a-v  # scalar minus vector
v-a  # scalar minus vector
a*v  # scalar times vector
v*a  # vector times scalar
```

Python optimalization

Optimization of C, C++, and Fortran

- Compilers do a good job for C, C++, and Fortran.
- The type system makes agressive optimization possible.
- Examples: code inlining, loop unrolling, and memory prefetching.

Python optimization

- No compiler.
- No type declaration of variables.
- No inlining and no loop unrolling.
- Probably inefficient in Python:

```
def f(a, b):
    return a + b
```

Manual timing

- Use `time.time()`.
- Simple statements should be placed in a loop.
- Make sure constant machine load.
- Run the tests several times, choose the fastest.

The `timeit` module (1)

- Usage:

```
import timeit
timer =
timeit.Timer(stmt="a+=1",setup="a=0")
time = timer.timeit(number=10000) #or
times = timer.repeat(repeat=5,
number=10000)
```

The `timeit` module (2)

- Isolates the global namespace.
- Automatically wraps the code in a for–loop.
- Users can provide their own timer (callback).
- Time a user defined function:

```
from __main__ import my_func
```

Profiling modules

- Prior to code optimization, hotspots and bottlenecks must be located.
  ”First make it work. Then make it right. Then make it fast.”
  - Kent Beck
- Two modules: `profile` and `hotshot`.
- `profile` works for all Python versions.
- `hotshot` introduced in Python version 2.2.

The `profile` module (1)

- As a script: `profile.py script.py`
- As a module:

```
import profile
pr = profile.Profile()
res = pr.run("function()", "filename")
res.print\_stats()
```

- Profile data saved to "filename" can be viewed with the `pstats` module.

The `profile` module (2)

- `profile.calibrate(number)` finds the profiling overhead.
- Remove profiling overhead:
  `pr = profile.Profile(bias=overhead)`
- Profile a single function call:

```
pr = profile.Profile()
pr.runcall(func, *args, **kwargs)
```

The `hotshot` module

- Similar to `profile`, but mostly implemented in C.
- Smaller performance impact than `profile`.
- Useage:

```
import hotshot
pr = hotshot.Profile("filename")
pr.run(cmd)
pr.close() # Close log-file and end profiler
```

- Read profile data:

```
import hotshot.stats
data = hotshot.stats.load("filename")# profile.Stats instance
data.print_stats()
```

The `pstats` module

- There are many ways to view profiling data.
- The module `pstats` provides the class `Stats` for creating profiling reports:

```
import pstats
data = pstats.Stats("filename")
data.print_stats()
```

- The method `sort_stats(key, *keys)` is used to sort future output.
- Common used keys: `'calls'`, `'cumulative'`, `'time'`.

Pure Python performance tips

- Place references to functions in the local namespace.

```
from math import *
def f(x):
    for i in xrange(len(x)):
        x[i] = sin(x[i]) # Slow
    return x

def g(x):
    loc_sin = sin # Local reference
    for i in xrange(len(x)):
        x[i] = loc_sin(x[i]) # Faster
    return x
```

- Reason: Local namespace is searched first.

More local references

- Local references to instance methods of global objects are even more important, as we need only one dictionary look–up to find the method instead of three (local, global, instance–dictionary).

```
class Dummy(object):
    def f(self): pass

d = Dummy()

def f():
    loc_f=d.f
    for i in xrange(10000): loc_f()
```

- Calling `loc_f()` instead of `d.f()` is 40% faster in this example.

Exceptions should never happen

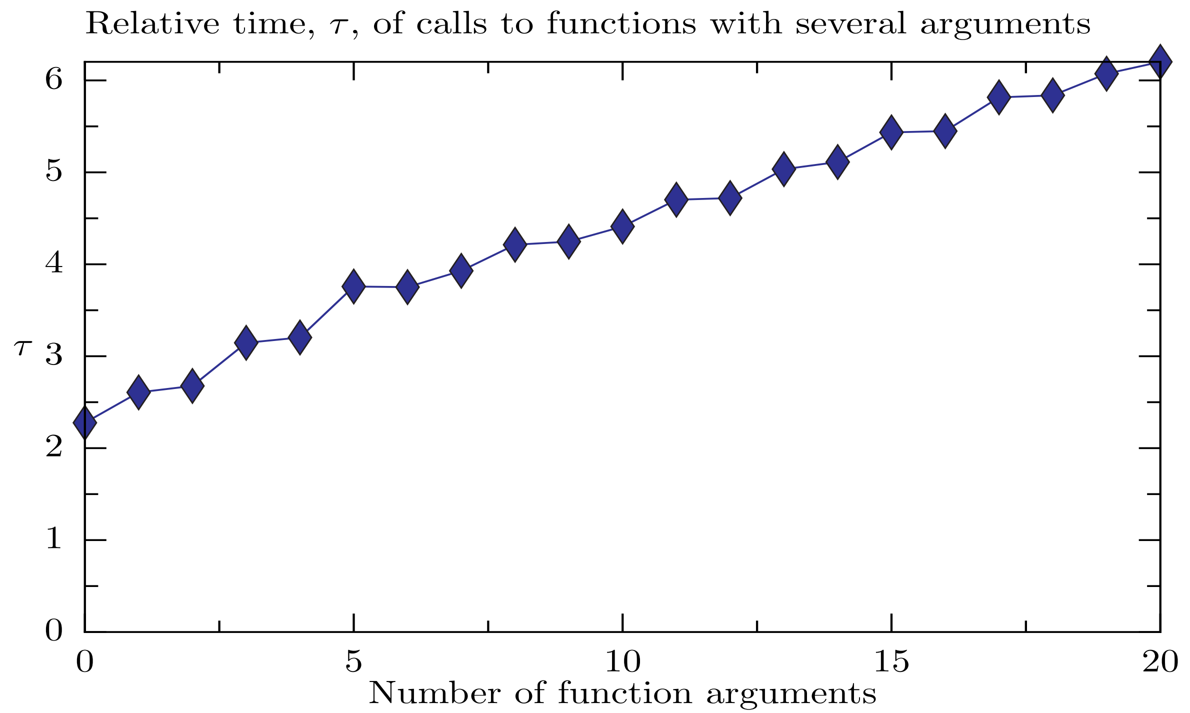
- Use `if/else` instead of `try/except`
- Example:

```
x = 0
try: 1.0/x
except: 0

if not (x==0): 1.0/x
else: 0
```

- `if/else` is more than 20 times faster.

Function calls

- The time of calling a function grows linearly with the number of arguments:



Numerical Python

- Vectorized computations are fast:

```
import numarray # Array functions
x = numarray.arange(-1,1,0.01)
y = numarray.sin(x)

import math # Scalar functions
y = numarray.zeros(len(x), type=numarray.Float)
for i in xrange(len(x)):
    y[i] = math.sin(x[i])
```

- The speedup is a factor of 20.

Resizing arrays

- The `resize` method of arrays is very slow.
- Increasing the array size by one in a loop is about 300-350 times slower than appending elements to a Python list.
- Best approach; allocate the memory once, and assign values later.

`Numeric` vs. `numarray`

- `Numeric` is the old array module in Python
- Still very popular, and will probably live for many years
- The difference between pointwise and array evaluation of a vector is about 13 for `Numeric` (20 for `numarray`)
- Vectorized functions work on scalars as well, but at a high price
- Using `numarray.sin` instead of `math.sin` on a scalar value is slower by a factor of 12. `Numeric.sin` only slower by a factor of 4

Conclusions

- Python scripts can often be heavily optimized.
- The results given here may vary on different architectures and Python versions
- Be extremely careful about the `from numarray import *`.

Mixed language programming

Contents

- Why Python and C are two different worlds
- Wrapper code
- Wrapper tools
- F2PY: wrapping Fortran (and C) code
- SWIG: wrapping C and C++ code

More info

- Ch. 5 in the course book
- F2PY manual
- SWIG manual
- Examples coming with the SWIG source code
- Ch. 9 and 10 in the course book

Optimizing slow Python code

- Identify bottlenecks (via profiling)
- Migrate slow functions to Fortran, C, or C++
- Tools make it easy to combine Python with Fortran, C, or C++

Getting started: Scientific Hello World

- Python-F77 via F2PY
- Python-C via SWIG
- Python-C++ via SWIG

Later: Python interface to `oscillator` code for interactive computational steering of simulations (using F2PY)

The nature of Python vs. C

- A Python variable can hold different objects:

```
d = 3.2      # d holds a float
d = 'txt'    # d holds a string
d = Button(frame, text='push')   # instance of class Button
```

- In C, C++ and Fortran, a variable is declared of a specific type:

```
double d; d = 4.2;
d = "some string";  /* illegal, compiler error */
```

- This difference makes it quite complicated to call C, C++ or Fortran from Python

Calling C from Python

- Suppose we have a C function

```
extern double hw1(double r1, double r2);
```

- We want to call this from Python as

```
from hw import hw1
r1 = 1.2; r2 = -1.2
s = hw1(r1, r2)
```

- The Python variables `r1` and `r2` hold numbers (`float`), we need to extract these in the C code, convert to `double` variables, then call `hw1`, and finally convert the `double` result to a Python `float`
- All this conversion is done in *wrapper code*

Wrapper code

- Every object in Python is represented by C struct `PyObject`
- Wrapper code converts between `PyObject` variables and plain C variables (from `PyObject r1` and `r2` to `double`, and `double` result to `PyObject`):

```
static PyObject *_wrap_hw1(PyObject *self, PyObject *args) {
    PyObject *resultobj;
    double arg1, arg2, result;

    PyArg_ParseTuple(args,(char *)"dd:hw1",&arg1,&arg2)

    result = hw1(arg1,arg2);

    resultobj = PyFloat_FromDouble(result);
    return resultobj;
}
```

Extension modules

- The wrapper function and `hw1` must be compiled and linked to a shared library file
- This file can be loaded in Python as module
- Such modules written in other languages are called *extension modules*

Writing wrapper code

- A wrapper function is needed for each C function we want to call from Python
- Wrapper codes are tedious to write
- There are tools for automating wrapper code development
- We shall use SWIG (for C/C++) and F2PY (for Fortran)

Integration issues

- Direct calls through wrapper code enables efficient data transfer; large arrays can be sent by pointers
- COM, CORBA, ILU, .NET are different technologies; more complex, less efficient, but safer (data are copied)
- Jython provides a seamless integration of Python and Java

Scientific Hello World example

- Consider this Scientific Hello World module (`hw`):

```
import math

def hw1(r1, r2):
    s = math.sin(r1 + r2)
    return s

def hw2(r1, r2):
    s = math.sin(r1 + r2)
    print 'Hello, World! sin(%g+%g)=%g' % (r1,r2,s)
```

Usage:

```
from hw import hw1, hw2
print hw1(1.0, 0)
hw2(1.0, 0)
```

- We want to implement the module in Fortran 77, C and C++, and use it as if it were a pure Python module

Fortran 77 implementation

- We start with Fortran (F77)
- F77 code in a file `hw.f`:

```
      real*8 function hw1(r1, r2)
      real*8 r1, r2
      hw1 = sin(r1 + r2)
      return
      end

      subroutine hw2(r1, r2)
      real*8 r1, r2, s
      s = sin(r1 + r2)
      write(*,1000) 'Hello, World! sin(',r1+r2,')=',s
 1000 format(A,F6.3,A,F8.6)
      return
      end
```

One-slide F77 course

- Fortran is case insensitive (`reAL` is as good as `real`)
- One statement per line, must start in column 7 or later
- Comments on separate lines
- All function arguments are input and output
  (as pointers in C, or references in C++)
- A function returning one value is called `function`
- A function returning no value is called `subroutine`
- Types: `real`, `double precision`, `real*4`, `real*8`, `integer`, `character` (array)
- Arrays: just add dimension, as in
  `real*8 a(0:m, 0:n)`
- Format control of output requires `FORMAT` statements

Using F2PY

- F2PY automates integration of Python and Fortran
- Say the F77 code is in the file `hw.f`
- Run F2PY (-m module name, -c for compile+link):

```
f2py -m hw -c hw.f
```

- Load module into Python and test:

```
from hw import hw1, hw2
print hw1(1.0, 0)
hw2(1.0, 0)
```

- In Python, `hw` appears as a module with Python code...
- It cannot be simpler!

Call by reference issues

- In Fortran (and C/C++) functions often modify arguments; here the result `s` is an output *argument*:

```
      subroutine hw3(r1, r2, s)
      real*8 r1, r2, s
      s = sin(r1 + r2)
      return
      end
```

- Running F2PY results in a module with wrong behavior:

```
>>> from hw import hw3
>>> r1 = 1;  r2 = -1;  s = 10
>>> hw3(r1, r2, s)
>>> print s
10   # should be 0
```

- Why? F2PY assumes that all arguments are input arguments
- Output arguments must be explicitly specified!

General adjustment of interfaces to Fortran

- Function with multiple input and output variables

  ```
  subroutine somef(i1, i2, o1, o2, o3, o4, io1)
  ```

- input: `i1`, `i2`
- output: `o1`, ..., `o4`
- input *and* output: `io1`
- Pythonic interface, as generated by F2PY:

  ```
  o1, o2, o3, o4, io1 = somef(i1, i2, io1)
  ```

Check F2PY-generated doc strings

- What happened to our `hw3` subroutine?
- F2PY generates doc strings that document the interface:

```
>>> import hw
>>> print hw.__doc__          # brief module doc string
Functions:
  hw1 = hw1(r1,r2)
  hw2(r1,r2)
  hw3(r1,r2,s)

>>> print hw.hw3.__doc__   # more detailed function doc string
hw3 - Function signature:
  hw3(r1,r2,s)
Required arguments:
  r1 : input float
  r2 : input float
  s : input float
```

- We see that `hw3` assumes `s` is *input* argument!
- Remedy: adjust the interface

Interface files

- We can tailor the interface by editing an F2PY-generated *interface file*
- Run F2PY in two steps: (i) generate interface file, (ii) generate wrapper code, compile and link
- Generate interface file `hw.pyf` (`-h` option):

```
f2py -m hw -h hw.pyf hw.f
```

Outline of the interface file

- The interface applies a Fortran 90 module (class) syntax
- Each function/subroutine, its arguments and its return value is specified:

```
python module hw ! in
    interface  ! in :hw
        ...
        subroutine hw3(r1,r2,s) ! in :hw:hw.f
            real*8 :: r1
            real*8 :: r2
            real*8 :: s
        end subroutine hw3
    end interface
end python module hw
```

(Fortran 90 syntax)

Adjustment of the interface

- We may edit `hw.pyf` and specify `s` in `hw3` as an output argument, using F90's `intent(out)` keyword:

```
python module hw ! in
    interface  ! in :hw
        ...
        subroutine hw3(r1,r2,s) ! in :hw:hw.f
            real*8 :: r1
            real*8 :: r2
            real*8, intent(out) :: s
        end subroutine hw3
    end interface
end python module hw
```

- Next step: run F2PY with the edited interface file:

```
f2py -c hw.pyf hw.f
```

Output arguments are always returned

- Load the module and print its doc string:

  ```
  >>> import hw
  >>> print hw.__doc__
  Functions:
    hw1 = hw1(r1,r2)
    hw2(r1,r2)
    s = hw3(r1,r2)
  ```

  Oops! `hw3` takes only two arguments and *returns* `s`!
- This is the “Pythonic” function style; input data are arguments, output data are returned
- By default, F2PY treats all arguments as input
- F2PY generates Pythonic interfaces, different from the original Fortran interfaces, so check out the module’s doc string!

General adjustment of interfaces

- Function with multiple input and output variables

  ```
  subroutine somef(i1, i2, o1, o2, o3, o4, io1)
  ```

- input: `i1`, `i2`
- output: `o1`, ..., `o4`
- input *and* output: `io1`
- Pythonic interface (as generated by F2PY):

  ```
  o1, o2, o3, o4, io1 = somef(i1, i2, io1)
  ```

Specification of input/output arguments; .pyf file

- In the interface file:

```
python module somemodule
    interface
        ...
        subroutine somef(i1, i2, o1, o2, o3, o4, io1)
            real*8, intent(in) :: i1
            real*8, intent(in) :: i2
            real*8, intent(out) :: o1
            real*8, intent(out) :: o2
            real*8, intent(out) :: o3
            real*8, intent(out) :: o4
            real*8, intent(in,out) :: io1
        end subroutine somef
        ...
    end interface
end python module somemodule
```

- Note: no `intent` implies `intent(in)`

Specification of input/output arguments; .f file

- Instead of editing the interface file, we can add special F2PY comments in the Fortran source code:

```
      subroutine somef(i1, i2, o1, o2, o3, o4, io1)
      real*8 i1, i2, o1, o2, o3, o4, io1
Cf2py intent(in) i1
Cf2py intent(in) i2
Cf2py intent(out) o1
Cf2py intent(out) o2
Cf2py intent(out) o3
Cf2py intent(out) o4
Cf2py intent(in,out) io1
```

- Now a single F2PY command generates correct interface:

```
f2py -m hw -c hw.f
```

Specification of input/output arguments; .f90 file

- With Fortran 90:

```
subroutine somef(i1, i2, o1, o2, o3, o4, io1)
real*8 i1, i2, o1, o2, o3, o4, io1
!f2py  intent(in) i1
!f2py  intent(in) i2
!f2py  intent(out) o1
!f2py  intent(out) o2
!f2py  intent(out) o3
!f2py  intent(out) o4
!f2py  intent(in,out) io1
```

- Now a single F2PY command generates correct interface:

```
f2py -m hw -c hw.f
```

Integration of Python and C

- Let us implement the `hw` module in C:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

double hw1(double r1, double r2)
{
  double s;  s = sin(r1 + r2);  return s;
}

void hw2(double r1, double r2)
{
  double s;  s = sin(r1 + r2);
  printf("Hello, World! sin(%g+%g)=%g\n", r1, r2, s);
}

/* special version of hw1 where the result is an argument: */
void hw3(double r1, double r2, double *s)
{
  *s = sin(r1 + r2);
}
```

Using F2PY

- F2PY can also wrap C code if we specify the function signatures as Fortran 90 modules
- My procedure:
  - write the C functions as empty Fortran 77 functions or subroutines
  - run F2PY on the Fortran specification to generate an interface file
  - run F2PY with the interface file and the C source code

Step 1: Write Fortran 77 signatures

```
C file signatures.f

      real*8 function hw1(r1, r2)
Cf2py intent(c) hw1
      real*8 r1, r2
Cf2py intent(c) r1, r2
      end

      subroutine hw2(r1, r2)
Cf2py intent(c)  hw2
      real*8 r1, r2
Cf2py intent(c) r1, r2
      end

      subroutine hw3(r1, r2, s)
Cf2py intent(c) hw3
      real*8 r1, r2, s
Cf2py intent(c) r1, r2
Cf2py intent(out) s
      end
```

Step 2: Generate interface file

- Run

```
Unix/DOS> f2py -m hw -h hw.pyf signatures.f
```

- Result: `hw.pyf`

```
python module hw ! in
    interface  ! in :hw
        function hw1(r1,r2) ! in :hw:signatures.f
            intent(c) hw1
            real*8 intent(c) :: r1
            real*8 intent(c) :: r2
            real*8 intent(c) :: hw1
        end function hw1
        ...
        subroutine hw3(r1,r2,s) ! in :hw:signatures.f
            intent(c) hw3
            real*8 intent(c) :: r1
            real*8 intent(c) :: r2
            real*8 intent(out) :: s
        end subroutine hw3
    end interface
end python module hw
```

Step 3: compile C code into extension module

- Run

```
Unix/DOS> f2py -c hw.pyf hw.c
```

- Test:

```
import hw
print hw.hw3(1.0,-1.0)
print hw.__doc__
```

- One can either write the interface file by hand or write F77 code to generate, but for every C function the Fortran signature must be specified

Using SWIG

- Wrappers to C and C++ codes can be automatically generated by SWIG
- SWIG is more complicated to use than F2PY
- First make a SWIG interface file
- Then run SWIG to generate wrapper code
- Then compile and link the C code and the wrapper code

SWIG interface file

- The interface file contains C preprocessor directives and special SWIG directives:

```
/* file: hw.i */
%module hw
%{
/* include C header files necessary to compile the interface */
#include "hw.h"
%}

/* list functions to be interfaced: */
double hw1(double r1, double r2);
void   hw2(double r1, double r2);
void   hw3(double r1, double r2, double *s);
// or
// %include "hw.h"  /* make interface to all funcs in hw.h */
```

Making the module

- Run SWIG (preferably in a subdirectory):

```
swig -python -I.. hw.i
```

- SWIG generates wrapper code in

```
hw_wrap.c
```

- Compile and link a shared library module:

```
gcc -I.. -fPIC -I/some/path/include/python2.5 \
    -c ../hw.c hw_wrap.c
gcc -shared -fPIC -o _hw.so hw.o hw_wrap.o
```

  Note the underscore prefix in `_hw.so`

A build script

- Can automate the compile+link process
- Can use Python to extract where `Python.h` resides (needed by any wrapper code)

```
swig -python -I.. hw.i

root=`python -c 'import sys; print sys.prefix'`
ver=`python -c 'import sys; print sys.version[:3]'`
gcc -fPIC -I.. -I$root/include/python$ver -c ../hw.c hw_wrap.c
gcc -shared -fPIC -o _hw.so hw.o hw_wrap.o

python -c "import hw" # test
```

this script `make_module_1.sh` is found here:

`http://www.ifi.uio.no/~inf3331/scripting/src/py/mixed/hw/C/swig-hw/`

- The module consists of two files: `hw.py` (which loads) `_hw.so`

Building modules with Distutils (1)

- Python has a tool, Distutils, for compiling and linking extension modules
- First write a script `setup.py`:

```
import os
from distutils.core import setup, Extension

name = 'hw'                # name of the module
version = 1.0              # the module's version number

swig_cmd = 'swig -python -I.. %s.i' % name
print 'running SWIG:', swig_cmd
os.system(swig_cmd)

sources = ['../hw.c', 'hw_wrap.c']

setup(name = name, version = version,
      ext_modules = [Extension('_' + name,  # SWIG requires _
                               sources,
                               include_dirs=[os.pardir])
                     ])
```

Building modules with Distutils (2)

- Now run

```
python setup.py build_ext
python setup.py install --install-platlib=.
python -c 'import hw'   # test
```

- Can install resulting module files in any directory
- Use Distutils for professional distribution!

Testing the hw3 function

- Recall `hw3`:

```
void hw3(double r1, double r2, double *s)
{
  *s = sin(r1 + r2);
}
```

- Test:

```
>>> from hw import hw3
>>> r1 = 1;  r2 = -1;  s = 10
>>> hw3(r1, r2, s)
>>> print s
10   # should be 0 (sin(1-1)=0)
```

Major problem - as in the Fortran case

Specifying input/output arguments

- We need to adjust the SWIG interface file:

```
/* typemaps.i allows input and output pointer arguments to be
   specified using the names INPUT, OUTPUT, or INOUT */
%include "typemaps.i"

void   hw3(double r1, double r2, double *OUTPUT);
```

- Now the usage from Python is

```
s = hw3(r1, r2)
```

- Unfortunately, SWIG does not document this in doc strings

Other tools

- SIP: tool for wrapping C++ libraries
- Boost.Python: tool for wrapping C++ libraries
- CXX: C++ interface to Python (Boost is a replacement)
- Note: SWIG can generate interfaces to most scripting languages (Perl, Ruby, Tcl, Java, Guile, Mzscheme, ...)

Integrating Python with C++

- SWIG supports C++
- The only difference is when we run SWIG (`-c++` option):

```
swig -python -c++ -I.. hw.i
# generates wrapper code in hw_wrap.cxx
```

- Use a C++ compiler to compile and link:

```
root=`python -c 'import sys; print sys.prefix'`
ver=`python -c 'import sys; print sys.version[:3]'`
g++ -fPIC -I.. -I$root/include/python$ver \
    -c ../hw.cpp hw_wrap.cxx
g++ -shared -fPIC -o _hw.so hw.o hw_wrap.o
```

Interfacing C++ functions (1)

- This is like interfacing C functions, except that pointers are usual replaced by references

```
void hw3(double r1, double r2, double *s)   // C style
{ *s = sin(r1 + r2); }

void hw4(double r1, double r2, double& s)   // C++ style
{ s = sin(r1 + r2); }
```

Interfacing C++ functions (2)

- Interface file (`hw.i`):

```
%module hw
%{
#include "hw.h"
%}
%include "typemaps.i"
%apply double *OUTPUT { double* s }
%apply double *OUTPUT { double& s }
%include "hw.h"
```

- That's it!

Interfacing C++ classes

- C++ classes add more to the SWIG-C story
- Consider a class version of our Hello World module:

```
class HelloWorld
{
 protected:
  double r1, r2, s;
  void compute();    // compute s=sin(r1+r2)
 public:
  HelloWorld();
  ~HelloWorld();

  void set(double r1, double r2);
  double get() const { return s; }
  void message(std::ostream& out) const;
};
```

- Goal: use this class as a Python class

Function bodies and usage

- Function bodies:

```
void HelloWorld:: set(double r1_, double r2_)
{
  r1 = r1_;  r2 = r2_;
  compute();  // compute s
}
void HelloWorld:: compute()
{ s = sin(r1 + r2); }
```

  etc.

- Usage:

```
HelloWorld hw;
hw.set(r1, r2);
hw.message(std::cout);  // write "Hello, World!" message
```

- Files: HelloWorld.h, HelloWorld.cpp

Adding a subclass

- To illustrate how to handle class hierarchies, we add a subclass:

```
class HelloWorld2 : public HelloWorld
{
 public:
  void gets(double& s_) const;
};
```

```
void HelloWorld2:: gets(double& s_) const { s_ = s; }
```

  i.e., we have a function with an output argument
- Note: `gets` should return the value when called from Python
- Files: `HelloWorld2.h`, `HelloWorld2.cpp`

SWIG interface file

```
/* file: hw.i */
%module hw
%{
/* include C++ header files necessary to compile the interface */
#include "HelloWorld.h"
#include "HelloWorld2.h"
%}

%include "HelloWorld.h"

%include "typemaps.i"
%apply double* OUTPUT { double& s }
%include "HelloWorld2.h"
```

Adding a class method

- SWIG allows us to add class methods
- Calling `message` with standard output (`std::cout`) is tricky from Python so we add a `print` method for printing to std.output
- `print` coincides with Python's keyword `print` so we follow the convention of adding an underscore:

```
%extend HelloWorld {
    void print_() { self->message(std::cout); }
}
```

- This is basically C++ syntax, but `self` is used instead of `this` and `%extend HelloWorld` is a SWIG directive
- Make extension module:

```
swig -python -c++ -I.. hw.i
# compile HelloWorld.cpp HelloWorld2.cpp hw_wrap.cxx
# link HelloWorld.o HelloWorld2.o hw_wrap.o to _hw.so
```

Using the module

```
from hw import HelloWorld

hw = HelloWorld()   # make class instance
r1 = float(sys.argv[1]);  r2 = float(sys.argv[2])
hw.set(r1, r2)      # call instance method
s = hw.get()
print "Hello, World! sin(%g + %g)=%g" % (r1, r2, s)
hw.print_()

hw2 = HelloWorld2()   # make subclass instance
hw2.set(r1, r2)
s = hw.gets()         # original output arg. is now return value
print "Hello, World2! sin(%g + %g)=%g" % (r1, r2, s)
```

Remark

- It looks that the C++ class hierarchy is mirrored in Python
- Actually, SWIG wraps a *function* interface to any class:

```
import _hw    # use _hw.so directly
hw = _hw.new_HelloWorld()
_hw.HelloWorld_set(hw, r1, r2)
```

- SWIG also makes a proxy class in `hw.py`, mirroring the original C++ class:

```
import hw     # use hw.py interface to _hw.so
c = hw.HelloWorld()
c.set(r1, r2)    # calls _hw.HelloWorld_set(r1, r2)
```

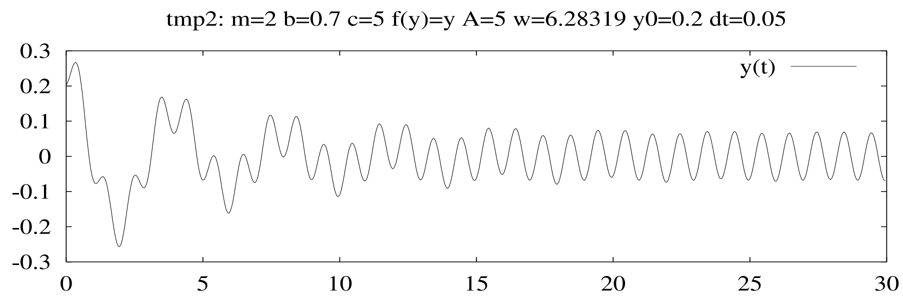
- The proxy class introduces overhead

Computational steering

- Consider a simulator written in F77, C or C++
- Aim: write the administering code and run-time visualization in Python
- Use a Python interface to Gnuplot
- Use NumPy arrays in Python
- F77/C and NumPy arrays share the same data
- Result:
  - steer simulations through scripts
  - do low-level numerics efficiently in C/F77
  - send simulation data to plotting a program

  The best of all worlds?

Example on computational steering



Consider the `oscillator` code. The following interactive features would be nice:

- set parameter values
- run the simulator for a number of steps and visualize
- change a parameter
- option: rewind a number of steps
- continue simulation and visualization

Example on what we can do

- Here is an interactive session:

```
>>> from simviz_f77 import *
>>> A=1; w=4*math.pi  # change parameters
>>> setprm()    # send parameters to oscillator code
>>> run(60)     # run 60 steps and plot solution
>>> w=math.pi   # change frequency
>>> setprm()    # update prms in oscillator code
>>> rewind(30)  # rewind 30 steps
>>> run(120)    # run 120 steps and plot
>>> A=10; setprm()
>>> rewind()    # rewind to t=0
>>> run(400)
```

Principles

- The F77 code performs the numerics
- Python is used for the interface (`setprm`, `run`, `rewind`, plotting)
- F2PY was used to make an interface to the F77 code (fully automated process)
- Arrays (NumPy) are created in Python and transferred to/from the F77 code
- Python communicates with both the simulator and the plotting program ("sends pointers around")

About the F77 code

- Physical and numerical parameters are in a common block
- `scan2` sets parameters in this common block:

```
subroutine scan2(m_, b_, c_, A_, w_, y0_, tstop_, dt_, func_)
real*8 m_, b_, c_, A_, w_, y0_, tstop_, dt_
character func_*(*)
```

  can use `scan2` to send parameters from Python to F77
- `timeloop2` performs `nsteps` time steps:

```
subroutine timeloop2(y, n, maxsteps, step, time, nsteps)

integer n, step, nsteps, maxsteps
real*8 time, y(n,0:maxsteps-1)
```

  solution available in `y`

Creating a Python interface w/F2PY

- `scan2`: trivial (only input arguments)
- `timestep2`: need to be careful with
  - output and input/output arguments
  - multi-dimensional arrays (`y`)
- Note: multi-dimensional arrays are stored differently in Python (i.e. C) and Fortran!

Using timeloop2 from Python

- This is how we would like to write the Python code:

```
maxsteps = 10000; n = 2
y = zeros((n,maxsteps), order='Fortran')
step = 0; time = 0.0

def run(nsteps):
    global step, time, y

    y, step, time = \
       oscillator.timeloop2(y, step, time, nsteps)

    y1 = y[0,0:step+1]
    g.plot(Gnuplot.Data(t, y1, with='lines'))
```

Arguments to timeloop2

- Subroutine signature:

```
subroutine timeloop2(y, n, maxsteps, step, time, nsteps)

integer n, step, nsteps, maxsteps
real*8 time, y(n,0:maxsteps-1)
```

- Arguments:

```
y   : solution (all time steps), input and output
n   : no of solution components (2 in our example), input
maxsteps : max no of time steps, input
step    : no of current time step, input and output
time    : current value of time, input and output
nsteps : no of time steps to advance the solution
```

Interfacing the timeloop2 routine

- Use `Cf2py` comments to specify argument type:

```
Cf2py intent(in,out) step
Cf2py intent(in,out) time
Cf2py intent(in,out) y
Cf2py intent(in)     nsteps
```

- Run F2PY:

```
f2py -m oscillator -c --build-dir tmp1 --fcompiler='Gnu' \
   ../timeloop2.f \
   $scripting/src/app/oscillator/F77/oscillator.f \
   only: scan2 timeloop2 :
```

Testing the extension module

- Import and print documentation:

```
>>> import oscillator
>>> print oscillator.__doc__
This module 'oscillator' is auto-generated with f2py
Functions:
  y,step,time = timeloop2(y,step,time,nsteps,
                          n=shape(y,0),maxsteps=shape(y,1))
  scan2(m_,b_,c_,a_,w_,y0_,tstop_,dt_,func_)
COMMON blocks:
  /data/ m,b,c,a,w,y0,tstop,dt,func(20)
```

- Note: array dimensions (`n`, `maxsteps`) are moved to the end of the argument list and given default values!
- Rule: always print and study the doc string since F2PY perturbs the argument list

More info on the current example

- Directory with Python interface to the oscillator code:

  ```
  src/py/mixed/simviz/f2py/
  ```

- Files:

  ```
  simviz_steering.py    :  complete script running oscillator
                           from Python by calling F77 routines
  simvizGUI_steering.py :  as simviz_steering.py, but with a GUI
  make_module.sh        :  build extension module
  ```

Comparison with Matlab

- The demonstrated functionality can be coded in Matlab
- Why Python + F77?
- We can define our own interface in a much more powerful language (Python) than Matlab
- We can much more easily transfer data to and from or own F77 or C or C++ libraries
- We can use any appropriate visualization tool
- We can call up Matlab if we want
- Python + F77 gives tailored interfaces and maximum flexibility

Mixed language numerical Python

Contents

- Migrating slow for loops over NumPy arrays to Fortran, C and C++
- F2PY handling of arrays
- Handwritten C and C++ modules
- C++ class for wrapping NumPy arrays
- C++ modules using SCXX
- Pointer communication and SWIG
- Efficiency considerations

More info

- Ch. 5, 9 and 10 in the course book
- F2PY manual
- SWIG manual
- Examples coming with the SWIG source code
- Electronic Python documentation:
  Extending and Embedding..., Python/C API
- Python in a Nutshell
- Python Essential Reference (Beazley)

Is Python slow for numerical computing?

- Fill a NumPy array with function values:

```
n = 2000
a = zeros((n,n))
xcoor = arange(0,1,1/float(n))
ycoor = arange(0,1,1/float(n))

for i in range(n):
    for j in range(n):
        a[i,j] = f(xcoor[i], ycoor[j])  # f(x,y) = sin(x*y) + 8
```

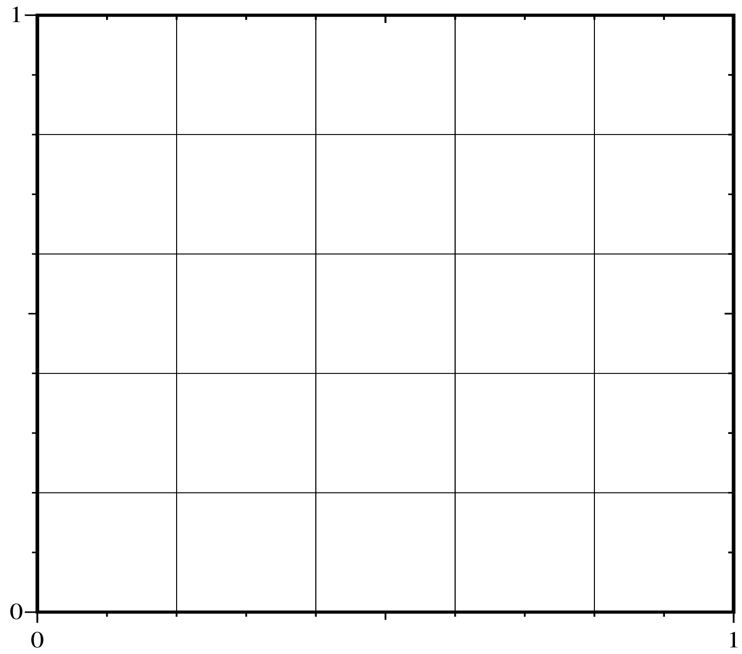
- Fortran/C/C++ version: (normalized) time 1.0
- NumPy vectorized evaluation of `f`: time 3.0
- Python loop version (version): time 140 (`math.sin`)
- Python loop version (version): time 350 (`numarray.sin`)

Comments

- Python loops over arrays are extremely slow
- NumPy vectorization may be sufficient
- However, NumPy vectorization may be inconvenient
  - plain loops in Fortran/C/C++ are much easier
- Write administering code in Python
- Identify bottlenecks (via profiling)
- Migrate slow Python code to Fortran, C, or C++
- Python-Fortran w/NumPy arrays via F2PY: easy
- Python-C/C++ w/NumPy arrays via SWIG: not that easy

Case: filling a grid with point values

- Consider a rectangular 2D grid



- A NumPy array `a[i,j]` holds values at the grid points

Python object for grid data

- Python class:

```
class Grid2D:
    def __init__(self,
                 xmin=0, xmax=1, dx=0.5,
                 ymin=0, ymax=1, dy=0.5):
        self.xcoor = sequence(xmin, xmax, dx)
        self.ycoor = sequence(ymin, ymax, dy)

        # make two-dim. versions of these arrays:
        # (needed for vectorization in __call__)
        self.xcoorv = self.xcoor[:,newaxis]
        self.ycoorv = self.ycoor[newaxis,:]

    def __call__(self, f):
        # vectorized code:
        return f(self.xcoorv, self.ycoorv)
```

Slow loop

- Include a straight Python loop also:

```
class Grid2D:
    ....
    def gridloop(self, f):
        lx = size(self.xcoor); ly = size(self.ycoor)
        a = zeros((lx,ly))

        for i in xrange(lx):
            x = self.xcoor[i]
            for j in xrange(ly):
                y = self.ycoor[j]
                a[i,j] = f(x, y)
        return a
```

- Usage:

```
g = Grid2D(dx=0.01, dy=0.2)
def myfunc(x, y):
    return sin(x*y) + y
a = g(myfunc)
i=4; j=10;
print 'value at (%g,%g) is %g' % (g.xcoor[i],g.ycoor[j],a[i,j]
```

Migrate gridloop to F77

```
class Grid2Deff(Grid2D):
    def __init__(self,
                 xmin=0, xmax=1, dx=0.5,
                 ymin=0, ymax=1, dy=0.5):
        Grid2D.__init__(self, xmin, xmax, dx, ymin, ymax, dy)

    def ext_gridloop1(self, f):
        """compute a[i,j] = f(xi,yj) in an external routine."""
        lx = size(self.xcoor);  ly = size(self.ycoor)
        a = zeros((lx,ly))
        ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, f)
        return a
```

We can also migrate to C and C++ (done later)

F77 function

- First try (typical attempt by a Fortran/C programmer):

```
      subroutine gridloop1(a, xcoor, ycoor, nx, ny, func1)
      integer nx, ny
      real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1)
      real*8 func1
      external func1

      integer i,j
      real*8 x, y
      do j = 0, ny-1
         y = ycoor(j)
         do i = 0, nx-1
            x = xcoor(i)
            a(i,j) = func1(x, y)
         end do
      end do
      return
      end
```

- Note: `float` type in NumPy array *must* match `real*8` or `double precision` in Fortran! (Otherwise F2PY will take a copy of the array `a` so the type matches that in the F77 code)

Making the extension module

- Run F2PY:

```
f2py -m ext_gridloop -c gridloop.f
```

- Try it from Python:

```
import ext_gridloop
ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, myfunc,
                       size(self.xcoor), size(self.ycoor))
```

  wrong results; `a` is not modified!

- Reason: the `gridloop1` function works on a copy a (because higher-dimensional arrays are stored differently in C/Python and Fortran)

Array storage in Fortran and C/C++

- C and C++ has row-major storage
(two-dimensional arrays are stored row by row)
- Fortran has column-major storage
(two-dimensional arrays are stored column by column)
- Multi-dimensional arrays: first index has fastest variation in Fortran,
last index has fastest variation in C and C++

Example: storing a 2x3 array

1	2	3	4	5	6	C storage
1	4	2	5	3	6	Fortran storage

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

F2PY and multi-dimensional arrays

- F2PY-generated modules treat storage schemes transparently
- If input array has C storage, a copy is taken, calculated with, and returned as output
- F2PY needs to know whether arguments are input, output or both
- To monitor (hidden) array copying, turn on the flag

```
f2py ... -DF2PY_REPORT_ON_ARRAY_COPY=1
```

- In-place operations on NumPy arrays are possible in Fortran, but the default is to work on a copy, that is why our `gridloop1` function does not work

Always specify input/output data

- Insert Cf2py comments to tell that `a` is an output variable:

```
      subroutine gridloop2(a, xcoor, ycoor, nx, ny, func1)
      integer nx, ny
      real*8 a(0:nx-1,ny-1), xcoor(0:nx-1), ycoor(0:ny-1), func
      external func1
Cf2py intent(out) a
Cf2py intent(in) xcoor
Cf2py intent(in) ycoor
Cf2py depend(nx,ny) a
```

gridloop2 seen from Python

- F2PY generates this Python interface:

```
>>> import ext_gridloop
>>> print ext_gridloop.gridloop2.__doc__

gridloop2 - Function signature:
  a = gridloop2(xcoor,ycoor,func1,[nx,ny,func1_extra_args])
Required arguments:
  xcoor : input rank-1 array('d') with bounds (nx)
  ycoor : input rank-1 array('d') with bounds (ny)
  func1 : call-back function
Optional arguments:
  nx := len(xcoor) input int
  ny := len(ycoor) input int
  func1_extra_args := () input tuple
Return objects:
  a : rank-2 array('d') with bounds (nx,ny)
```

- `nx` and `ny` are optional (!)

Handling of arrays with F2PY

- Output arrays are returned and are not part of the argument list, as seen from Python
- Need `depend(nx,ny) a` to specify that `a` is to be created with size `nx`, `ny` in the wrapper
- Array dimensions are optional arguments (!)

```
class Grid2Deff(Grid2D):
    ...
    def ext_gridloop2(self, f):
        a = ext_gridloop.gridloop2(self.xcoor, self.ycoor, f)
        return a
```

- The modified interface is well documented in the doc strings generated by F2PY

Input/output arrays (1)

- What if we really want to send `a` as argument and let F77 modify it?

```
def ext_gridloop1(self, f):
    lx = size(self.xcoor);  ly = size(self.ycoor)
    a = zeros((lx,ly))
    ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, f)
    return a
```

- This is not Pythonic code, but it can be realized
- 1. the array must have Fortran storage
- 2. the array argument must be `intent(inout)` (in general not recommended)

Input/output arrays (2)

- F2PY generated modules has a function for checking if an array has column major storage (i.e., Fortran storage):

```
>>> a = zeros((n,n), order='Fortran')
>>> isfortran(a)
True
>>> a = asarray(a, order='C')   # back to C storage
>>> isfortran(a)
False
```

Input/output arrays (3)

- Fortran function:

```
      subroutine gridloop1(a, xcoor, ycoor, nx, ny, func1)
      integer nx, ny
      real*8 a(0:nx-1,ny-1), xcoor(0:nx-1), ycoor(0:ny-1), func
C     call this function with an array a that has
C     column major storage!
Cf2py intent(inout) a
Cf2py intent(in) xcoor
Cf2py intent(in) ycoor
Cf2py depend(nx, ny) a
```

- Python call:

```
def ext_gridloop1(self, f):
    lx = size(self.xcoor);  ly = size(self.ycoor)
    a = asarray(a, order='Fortran')
    ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, f)
    return a
```

Storage compatibility requirements

- Only when `a` has Fortran (column major) storage, the Fortran function works on `a` itself
- If we provide a plain NumPy array, it has C (row major) storage, and the wrapper sends a copy to the Fortran function and transparently transposes the result
- Hence, F2PY is very user-friendly, at a cost of some extra memory
- The array returned from F2PY has Fortran (column major) storage

F2PY and storage issues

- `intent(out) a` is the right specification; `a` should not be an argument in the Python call
- F2PY wrappers will work on copies, if needed, and hide problems with different storage scheme in Fortran and C/Python
- Python call:

```
a = ext_gridloop.gridloop2(self.xcoor, self.ycoor, f)
```

Caution

- Find problems with this code (`comp` is a Fortran function in the extension module `pde`):

```
x = arange(0, 1, 0.01)
b = myfunc1(x)  # compute b array of size (n,n)
u = myfunc2(x)  # compute u array of size (n,n)
c = myfunc3(x)  # compute c array of size (n,n)

dt = 0.05
for i in range(n)
    u = pde.comp(u, b, c, i*dt)
```

About Python callbacks

- It is convenient to specify the `myfunc` in Python
- However, a callback to Python is costly, especially when done a large number of times (for every grid point)
- Avoid such callbacks; vectorize callbacks
- The Fortran routine should actually direct `a` back to Python (i.e., do nothing...) for a vectorized operation
- Let's do this for illustration

Vectorized callback seen from Python

```
class Grid2Deff(Grid2D):
    ...
    def ext_gridloop_vec(self, f):
        """Call extension, then do a vectorized callback to Python."""
        lx = size(self.xcoor);  ly = size(self.ycoor)
        a = zeros((lx,ly))
        a = ext_gridloop.gridloop_vec(a, self.xcoor, self.ycoor, f)
        return a

def myfunc(x, y):
    return sin(x*y) + 8*x

def vectorize(func):

    def vec77(a, xcoor, ycoor, nx, ny):
        """Vectorized function to be called from extension module."""
        x = xcoor[:,NewAxis];  y = ycoor[NewAxis,:]
        a[:,:] = func(x, y)  # in-place modification of a

    return vec77

g = Grid2Deff(dx=0.2, dy=0.1)
a = g.ext_gridloop_vec(vectorize(myfunc))
```

Vectorized callback from Fortran

```
      subroutine gridloop_vec(a, xcoor, ycoor, nx, ny, func1)
      integer nx, ny
      real*8 a(0:nx-1,ny-1), xcoor(0:nx-1), ycoor(0:ny-1)
Cf2py intent(in,out) a
Cf2py intent(in) xcoor
Cf2py intent(in) ycoor
      external func1

C     fill array a with values taken from a Python function,
C     do that without loop and point-wise callback, do a
C     vectorized callback instead:
      call func1(a, xcoor, ycoor, nx, ny)

C     could work further with array a here...

      return
      end
```

Caution

- What about this Python callback:

```
def vectorize(func):

    def vec77(a, xcoor, ycoor, nx, ny):
        """Vectorized function to be called from extension module."""
        x = xcoor[:,NewAxis];   y = ycoor[NewAxis,:]
        a = func(x, y)

    return vec77
```

- `a` now refers to a new NumPy array; no in-place modification of the input argument

Avoiding callback by string-based if-else wrapper

- Callbacks are expensive
- Even vectorized callback functions degrades performace a bit
- Alternative: implement “callback” in F77
- Flexibility from the Python side: use a string to switch between the “callback” (F77) functions

```
a = ext_gridloop.gridloop2_str(self.xcoor, self.ycoor, 'myfunc'
```

F77 wrapper:

```
      subroutine gridloop2_str(xcoor, ycoor, func_str)
      character*(*) func_str
      ...
     if (func_str .eq. 'myfunc') then
        call gridloop2(a, xcoor, ycoor, nx, ny, myfunc)
     else if (func_str .eq. 'f2') then
        call gridloop2(a, xcoor, ycoor, nx, ny, f2)
     ...
```

Compiled callback function

- Idea: if callback formula is a string, we could embed it in a Fortran function and call Fortran instead of Python
- F2PY has a module for “inline” Fortran code specification and building

```
source = """
      real*8 function fcb(x, y)
      real*8 x, y
      fcb = %s
      return
      end
""" % fstr
import f2py2e
f2py_args = "--fcompiler='Gnu' --build-dir tmp2 etc..."
f2py2e.compile(source, modulename='callback',
               extra_args=f2py_args, verbose=True,
               source_fn='sourcecodefile.f')
import callback
<work with the new extension module>
```

gridloop2 wrapper

- To glue F77 `gridloop2` and the F77 callback function, we make a `gridloop2` wrapper:

```
      subroutine gridloop2_fcb(a, xcoor, ycoor, nx, ny)
      integer nx, ny
      real*8 a(0:nx-1,ny-1), xcoor(0:nx-1), ycoor(0:ny-1)
Cf2py intent(out) a
Cf2py depend(nx,ny) a
      real*8 fcb
      external fcb

      call gridloop2(a, xcoor, ycoor, nx, ny, fcb)
      return
      end
```

- This wrapper and the callback function `fcb` constitute the F77 source code, stored in `source`
- The source calls `gridloop2` so the module must be linked with the module containing `gridloop2` (`ext_gridloop.so`)

Building the module on the fly

```
source = """
      real*8 function fcb(x, y)
      ...
      subroutine gridloop2_fcb(a, xcoor, ycoor, nx, ny)
      ...
""" % fstr

f2py_args = "--fcompiler='Gnu' --build-dir tmp2"\
            " -DF2PY_REPORT_ON_ARRAY_COPY=1 "\
            " ./ext_gridloop.so"
f2py2e.compile(source, modulename='callback',
               extra_args=f2py_args, verbose=True,
               source_fn='_cb.f')

import callback
a = callback.gridloop2_fcb(self.xcoor, self.ycoor)
```

gridloop2 could be generated on the fly

```
def ext_gridloop2_compile(self, fstr):
    if not isinstance(fstr, str):
        <error>
    # generate Fortran source for gridloop2:
    import f2py2e
    source = """
      subroutine gridloop2(a, xcoor, ycoor, nx, ny)
      ...
      do j = 0, ny-1
         y = ycoor(j)
         do i = 0, nx-1
            x = xcoor(i)
            a(i,j) = %s
      ...
""" % fstr  # no callback, the expression is hardcoded
    f2py2e.compile(source, modulename='ext_gridloop2', ...)

def ext_gridloop2_v2(self):
    import ext_gridloop2
    return ext_gridloop2.gridloop2(self.xcoor, self.ycoor)
```

Extracting a pointer to the callback function

- We can implement the callback function in Fortran, grab an F2PY-generated pointer to this function and feed that as the `func1` argument such that Fortran calls Fortran and not Python
- For a module `m`, the pointer to a function/subroutine `f` is reached as `m.f._cpointer`

```
def ext_gridloop2_fcb_ptr(self):
    from callback import fcb
    a = ext_gridloop.gridloop2(self.xcoor, self.ycoor,
                               fcb._cpointer)
    return a
```

`fcb` is a Fortran implementation of the callback in an F2PY-generated extension module `callback`

C implementation of the loop

- Let us write the `gridloop1` and `gridloop2` functions in C
- Typical C code:

```
void gridloop1(double** a, double* xcoor, double* ycoor,
               int nx, int ny, Fxy func1)
{
  int i, j;
  for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
       a[i][j] = func1(xcoor[i], ycoor[j])
}
```

- Problem: NumPy arrays use single pointers to data
- The above function represents `a` as a double pointer (common in C for two-dimensional arrays)

Manual writing of extension modules

- SWIG needs some non-trivial tweaking to handle NumPy arrays (i.e., the use of SWIG is much more complicated for array arguments than running F2PY)
- We shall write a complete extension module by hand
- We will need documentation of the Python C API (from Python's electronic doc.) and the NumPy C API (from the NumPy book)
- Source code files in

  `src/mixed/py/Grid2D/C/plain`

- Warning: manual writing of extension modules is very much more complicated than using F2PY on Fortran code! You need to know C quite well...

NumPy objects as seen from C

NumPy objects are C structs with attributes:

- `int nd`: no of indices (dimensions)
- `int dimensions[nd]`: length of each dimension
- `char *data`: pointer to data
- `int strides[nd]`: no of bytes between two successive data elements for a fixed index
- Access element (i,j) by

```
a->data + i*a->strides[0] + j*a->strides[1]
```

Creating new NumPy array in C

- Allocate a new array:

```
PyObject * PyArray_FromDims(int n_dimensions,
                            int dimensions[n_dimensions],
                            int type_num);

PyArrayObject *a;  int dims[2];
dims[0] = 10;  dims[1] = 21;
a = (PyArrayObject *) PyArray_FromDims(2, dims, PyArray_DOUBLE);
```

Wrapping data in a NumPy array

- Wrap an existing memory segment (with array data) in a NumPy array object:

```
PyObject * PyArray_FromDimsAndData(int n_dimensions,
           int dimensions[n_dimensions],
           int item_type,
           char *data);
```

```
/* vec is a double* with 10*21 double entries */
PyArrayObject *a;  int dims[2];
dims[0] = 10;  dims[1] = 21;
a = (PyArrayObject *) PyArray_FromDimsAndData(2, dims,
    PyArray_DOUBLE, (char *) vec);
```

Note: `vec` is a stream of numbers, now interpreted as a two-dimensional array, stored row by row

From Python sequence to NumPy array

- Turn any relevant Python sequence type (list, type, array) into a NumPy array:

```
PyObject * PyArray_ContiguousFromObject(PyObject *object,
                                        int item_type,
                                        int min_dim,
                                        int max_dim);
```

  Use `min_dim` and `max_dim` as 0 to preserve the original dimensions of `object`

- Application: ensure that an object is a NumPy array,

```
/* a_ is a PyObject pointer, representing a sequence
   (NumPy array or list or tuple) */
PyArrayObject a;
a = (PyArrayObject *) PyArray_ContiguousFromObject(a_,
                      PyArray_DOUBLE, 0, 0);
```

  a list, tuple or NumPy array `a` is now a NumPy array

Python interface

```
class Grid2Deff(Grid2D):
    def __init__(self,
                 xmin=0, xmax=1, dx=0.5,
                 ymin=0, ymax=1, dy=0.5):
        Grid2D.__init__(self, xmin, xmax, dx, ymin, ymax, dy)

    def ext_gridloop1(self, f):
        lx = size(self.xcoor);  ly = size(self.ycoor)
        a = zeros((lx,ly))

        ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, f)

        return a

    def ext_gridloop2(self, f):

        a = ext_gridloop.gridloop2(self.xcoor, self.ycoor, f)

        return a
```

gridloop1 in C; header

- Transform `PyObject` argument tuple to NumPy arrays:

```
static PyObject *gridloop1(PyObject *self, PyObject *args)
{
  PyArrayObject *a, *xcoor, *ycoor;
  PyObject *func1, *arglist, *result;
  int nx, ny, i, j;
  double *a_ij, *x_i, *y_j;

  /* arguments: a, xcoor, ycoor */
  if (!PyArg_ParseTuple(args, "O!O!O!O:gridloop1",
                        &PyArray_Type, &a,
                        &PyArray_Type, &xcoor,
                        &PyArray_Type, &ycoor,
                        &func1)) {
    return NULL; /* PyArg_ParseTuple has raised an exception */
  }
```

gridloop1 in C; safety checks

```
if (a->nd != 2 || a->descr->type_num != PyArray_DOUBLE) {
  PyErr_Format(PyExc_ValueError,
  "a array is %d-dimensional or not of type float", a->nd);
  return NULL;
}
nx = a->dimensions[0];  ny = a->dimensions[1];
if (xcoor->nd != 1 || xcoor->descr->type_num != PyArray_DOUBLE |
    xcoor->dimensions[0] != nx) {
  PyErr_Format(PyExc_ValueError,
  "xcoor array has wrong dimension (%d), type or length (%d)",
               xcoor->nd,xcoor->dimensions[0]);
  return NULL;
}
if (ycoor->nd != 1 || ycoor->descr->type_num != PyArray_DOUBLE |
    ycoor->dimensions[0] != ny) {
  PyErr_Format(PyExc_ValueError,
  "ycoor array has wrong dimension (%d), type or length (%d)",
               ycoor->nd,ycoor->dimensions[0]);
  return NULL;
}
if (!PyCallable_Check(func1)) {
  PyErr_Format(PyExc_TypeError,
  "func1 is not a callable function");
  return NULL;
}
```

Callback to Python from C

- Python functions can be called from C
- Step 1: for each argument, convert C data to Python objects and collect these in a tuple

```
PyObject *arglist; double x, y;
/* double x,y -> tuple with two Python float objects: */
arglist = Py_BuildValue("(dd)", x, y);
```

- Step 2: call the Python function

```
PyObject *result;  /* return value from Python function */
PyObject *func1;   /* Python function object */
result = PyEval_CallObject(func1, arglist);
```

- Step 3: convert result to C data

```
double r;  /* result is a Python float object */
r = PyFloat_AS_DOUBLE(result);
```

gridloop1 in C; the loop

```
  for (i = 0; i < nx; i++) {
    for (j = 0; j < ny; j++) {
      a_ij = (double *)(a->data+i*a->strides[0]+j*a->strides[1]);
      x_i = (double *)(xcoor->data + i*xcoor->strides[0]);
      y_j = (double *)(ycoor->data + j*ycoor->strides[0]);

      /* call Python function pointed to by func1: */
      arglist = Py_BuildValue("(dd)", *x_i, *y_j);
      result = PyEval_CallObject(func1, arglist);
      *a_ij = PyFloat_AS_DOUBLE(result);
    }
  }
  return Py_BuildValue("");  /* return None: */
}
```

Memory management

- There is a major problem with our loop:

```
arglist = Py_BuildValue("(dd)", *x_i, *y_j);
result = PyEval_CallObject(func1, arglist);
*a_ij = PyFloat_AS_DOUBLE(result);
```

- For each pass, `arglist` and `result` are dynamically allocated, but not destroyed
- From the Python side, memory management is automatic
- From the C side, we must do it ourself
- Python applies reference counting
- Each object has a number of references, one for each usage
- The object is destroyed when there are no references

Reference counting

- Increase the reference count:

  `Py_INCREF(myobj);`

  (i.e., I need this object, it cannot be deleted elsewhere)

- Decrease the reference count:

  `Py_DECREF(myobj);`

  (i.e., I don't need this object, it can be deleted)

gridloop1; loop with memory management

```
for (i = 0; i < nx; i++) {
  for (j = 0; j < ny; j++) {
    a_ij = (double *)(a->data + i*a->strides[0] + j*a->strides[1
    x_i = (double *)(xcoor->data + i*xcoor->strides[0]);
    y_j = (double *)(ycoor->data + j*ycoor->strides[0]);

    /* call Python function pointed to by func1: */
    arglist = Py_BuildValue("(dd)", *x_i, *y_j);
    result = PyEval_CallObject(func1, arglist);
    Py_DECREF(arglist);
    if (result == NULL) return NULL; /* exception in func1 */
    *a_ij = PyFloat_AS_DOUBLE(result);
    Py_DECREF(result);
  }
}
```

gridloop1; more testing in the loop

- We should check that allocations work fine:

```
arglist = Py_BuildValue("(dd)", *x_i, *y_j);
if (arglist == NULL) {  /* out of memory */
   PyErr_Format(PyExc_MemoryError,
                "out of memory for 2-tuple);
```

- The C code becomes quite comprehensive; much more testing than “active” statements

gridloop2 in C; header

gridloop2: as gridloop1, but array a is returned

```
static PyObject *gridloop2(PyObject *self, PyObject *args)
{
  PyArrayObject *a, *xcoor, *ycoor;
  int a_dims[2];
  PyObject *func1, *arglist, *result;
  int nx, ny, i, j;
  double *a_ij, *x_i, *y_j;

  /* arguments: xcoor, ycoor, func1 */
  if (!PyArg_ParseTuple(args, "O!O!O:gridloop2",
                        &PyArray_Type, &xcoor,
                        &PyArray_Type, &ycoor,
                        &func1)) {
    return NULL; /* PyArg_ParseTuple has raised an exception */
  }
  nx = xcoor->dimensions[0];  ny = ycoor->dimensions[0];
```

gridloop2 in C; macros

- NumPy array code in C can be simplified using macros
- First, a smart macro wrapping an argument in quotes:

```
#define QUOTE(s) # s   /* turn s into string "s" */
```

- Check the type of the array data:

```
#define TYPECHECK(a, tp) \
  if (a->descr->type_num != tp) { \
    PyErr_Format(PyExc_TypeError, \
    "%s array is not of correct type (%d)", QUOTE(a), tp); \
    return NULL; \
  }
```

- `PyErr_Format` is a flexible way of raising exceptions in C (must return `NULL` afterwards!)

gridloop2 in C; another macro

- Check the length of a specified dimension:

```
#define DIMCHECK(a, dim, expected_length) \
  if (a->dimensions[dim] != expected_length) { \
    PyErr_Format(PyExc_ValueError, \
    "%s array has wrong %d-dimension=%d (expected %d)", \
               QUOTE(a),dim,a->dimensions[dim],expected_length); \
    return NULL; \
  }
```

gridloop2 in C; more macros

- Check the dimensions of a NumPy array:

```
#define NDIMCHECK(a, expected_ndim) \
  if (a->nd != expected_ndim) { \
    PyErr_Format(PyExc_ValueError, \
    "%s array is %d-dimensional, expected to be %d-dimensional",\
                 QUOTE(a), a->nd, expected_ndim); \
    return NULL; \
  }
```

- Application:

```
NDIMCHECK(xcoor, 1); TYPECHECK(xcoor, PyArray_DOUBLE);
```

  If `xcoor` is 2-dimensional, an exceptions is raised by `NDIMCHECK`:

```
exceptions.ValueError
xcoor array is 2-dimensional, but expected to be 1-dimensional
```

gridloop2 in C; indexing macros

- Macros can greatly simplify indexing:

```
#define IND1(a, i) *((double *)(a->data + i*a->strides[0]))
#define IND2(a, i, j) \
 *((double *)(a->data + i*a->strides[0] + j*a->strides[1]))
```

- Application:

```
for (i = 0; i < nx; i++) {
  for (j = 0; j < ny; j++) {
    arglist = Py_BuildValue("(dd)", IND1(xcoor,i), IND1(ycoor,j));
    result = PyEval_CallObject(func1, arglist);
    Py_DECREF(arglist);
    if (result == NULL) return NULL; /* exception in func1 */
    IND2(a,i,j) = PyFloat_AS_DOUBLE(result);
    Py_DECREF(result);
  }
}
```

gridloop2 in C; the return array

- Create return array:

```
a_dims[0] = nx; a_dims[1] = ny;
a = (PyArrayObject *) PyArray_FromDims(2, a_dims,
                                       PyArray_DOUBLE);
if (a == NULL) {
  printf("creating a failed, dims=(%d,%d)\n",
         a_dims[0],a_dims[1]);
  return NULL; /* PyArray_FromDims raises an exception */
}
```

- After the loop, return `a`:

```
return PyArray_Return(a);
```

Registering module functions

- The method table must always be present - it lists the functions that should be callable from Python:

```
static PyMethodDef ext_gridloop_methods[] = {
  {"gridloop1",      /* name of func when called from Python */
   gridloop1,        /* corresponding C function */
   METH_VARARGS,     /* ordinary (not keyword) arguments */
   gridloop1_doc},   /* doc string for gridloop1 function */
  {"gridloop2",      /* name of func when called from Python */
   gridloop2,        /* corresponding C function */
   METH_VARARGS,     /* ordinary (not keyword) arguments */
   gridloop2_doc},   /* doc string for gridloop1 function */
  {NULL, NULL}
};
```

- `METH_KEYWORDS` (instead of `METH_VARARGS`) implies that the function takes 3 arguments (`self`, `args`, `kw`)

Doc strings

```
static char gridloop1_doc[] = \
  "gridloop1(a, xcoor, ycoor, pyfunc)";

static char gridloop2_doc[] = \
  "a = gridloop2(xcoor, ycoor, pyfunc)";

static char module_doc[] = \
  "module ext_gridloop:\n\
   gridloop1(a, xcoor, ycoor, pyfunc)\n\
   a = gridloop2(xcoor, ycoor, pyfunc)";
```

The required init function

```
PyMODINIT_FUNC initext_gridloop()
{
  /* Assign the name of the module and the name of the
     method table and (optionally) a module doc string:
  */
  Py_InitModule3("ext_gridloop", ext_gridloop_methods, module_doc);
  /* without module doc string:
  Py_InitModule ("ext_gridloop", ext_gridloop_methods); */

  import_array();    /* required NumPy initialization */
}
```

Building the module

```
root=`python -c 'import sys; print sys.prefix'`
ver=`python -c 'import sys; print sys.version[:3]'`
gcc -O3 -g -I$root/include/python$ver \
    -I$scripting/src/C \
    -c gridloop.c -o gridloop.o
gcc -shared -o ext_gridloop.so gridloop.o

# test the module:
python -c 'import ext_gridloop; print dir(ext_gridloop)'
```

A setup.py script

- The script:

```
from distutils.core import setup, Extension
import os

name = 'ext_gridloop'
setup(name=name,
      include_dirs=[os.path.join(os.environ['scripting'],
                                 'src', 'C')],
      ext_modules=[Extension(name, ['gridloop.c'])])
```

- Usage:

```
python setup.py build_ext
python setup.py install --install-platlib=.
# test module:
python -c 'import ext_gridloop; print ext_gridloop.__doc__'
```

Using the module

- The usage is the same as in Fortran, when viewed from Python
- No problems with storage formats and unintended copying of `a` in `gridloop1`, or optional arguments; here we have full control of all details
- `gridloop2` is the “right” way to do it
- It is much simpler to use Fortran and F2PY

Debugging

- Things usually go wrong when you program...
- Errors in C normally shows up as “segmentation faults” or “bus error” - no nice exception with traceback
- Simple trick: run `python` under a debugger

```
unix> gdb `which python`
(gdb) run test.py
```

- When the script crashes, issue the gdb command `where` for a traceback (if the extension module is compiled with -g you can see the line number of the line that triggered the error)
- You can only see the traceback, no breakpoints, prints etc., but a tool, PyDebug, allows you to do this

First debugging example

- In src/py/mixed/Grid2D/C/plain/debugdemo there are some C files with errors
- Try

  ```
  ./make_module_1.sh gridloop1
  ```

  This scripts runs

  ```
  ../../../Grid2Deff.py verify1
  ```

  which leads to a segmentation fault, implying that something is wrong in the C code (errors in the Python script shows up as exceptions with traceback)

1st debugging example (1)

- Check that the extension module was compiled with debug mode on (usually the -g option to the C compiler)
- Run `python` under a debugger:

```
unix> gdb `which python`
GNU gdb 6.0-debian
...
(gdb) run ../../../Grid2Deff.py verify1
Starting program: /usr/bin/python ../../../Grid2Deff.py verify1
...
Program received signal SIGSEGV, Segmentation fault.
0x40cdfab3 in gridloop1 (self=0x0, args=0x1) at gridloop1.c:20
20          if (!PyArg_ParseTuple(args, "O!O!O!O:gridloop1",
```

This is the line where something goes wrong...

1st debugging example (2)

```
(gdb) where
#0  0x40cdfab3 in gridloop1 (self=0x0, args=0x1) at gridloop1.c:20
#1  0x080fde1a in PyCFunction_Call ()
#2  0x080ab824 in PyEval_CallObjectWithKeywords ()
#3  0x080a9bde in Py_MakePendingCalls ()
#4  0x080aa76c in PyEval_EvalCodeEx ()
#5  0x080ab8d9 in PyEval_CallObjectWithKeywords ()
#6  0x080ab71c in PyEval_CallObjectWithKeywords ()
#7  0x080a9bde in Py_MakePendingCalls ()
#8  0x080ab95d in PyEval_CallObjectWithKeywords ()
#9  0x080ab71c in PyEval_CallObjectWithKeywords ()
#10 0x080a9bde in Py_MakePendingCalls ()
#11 0x080aa76c in PyEval_EvalCodeEx ()
#12 0x080acf69 in PyEval_EvalCode ()
#13 0x080d90db in PyRun_FileExFlags ()
#14 0x080d9d1f in PyRun_String ()
#15 0x08100c20 in _IO_stdin_used ()
#16 0x401ee79c in ?? ()
#17 0x41096bdc in ?? ()
```

1st debugging example (3)

- What is wrong?
- The `import_array()` call was removed, but the segmentation fault happended in the first call to a Python C function

2nd debugging example

- Try

  ```
  ./make_module_1.sh gridloop2
  ```

  and experience that

  ```
  python -c 'import ext_gridloop; print dir(ext_gridloop); \
             print ext_gridloop.__doc__'
  ```

  ends with an exception

  ```
  Traceback (most recent call last):
    File "<string>", line 1, in ?
  SystemError: dynamic module not initialized properly
  ```

- This signifies that the module misses initialization
- Reason: no `Py_InitModule3` call