Slides from INF3331 lectures - basic GUI programming in Python

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2011

Simple GUI programming with Python

Contents

- Introductory GUI programming
- Scientific Hello World examples
- GUI for simviz1.py
- GUI elements: text, input text, buttons, sliders, frames (for controlling layout)
- Customizing fonts and colors
- Event bindings (mouse bindings in particular)

GUI toolkits callable from Python

Python has interfaces to the GUI toolkits

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

Discussion of GUI toolkits

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

More info

- Ch. 6 and Ch. 11.2 in the course book
- “Introduction to Tkinter” by Lundh (see `doc.html`)
- “Python/Tkinter Programming” textbook by Grayson
- Efficient working style: grab GUI code from examples
- Demo programs:

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

Tkinter, Pmw and Tix

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

Scientific Hello World GUI

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

The code (1)


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

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

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

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

The code (2)

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

compute = Button(top, text=' equals ', command=comp_s)
compute.pack(side='left')

s_label = Label(top, textvariable=s, width=18)
s_label.pack(side='left')

root.mainloop()
```

Structure of widget creation

- A widget has a parent widget
- A widget must be packed (placed in the parent widget) before it can appear visually
- Typical structure:

```
widget = Tk_class(parent_widget,
                  arg1=value1, arg2=value2)
widget.pack(side='left')
```

- Variables can be tied to the contents of, e.g., text entries, but only special Tkinter variables are legal: `StringVar`, `DoubleVar`, `IntVar`

The event loop

- No widgets are visible before we call the event loop:

  ```
  root.mainloop()
  ```

- This loop waits for user input (e.g. mouse clicks)
- There is no predefined program flow after the event loop is invoked; the program just responds to events
- The widgets define the event responses

Binding events

- Instead of clicking "equals", pressing return in the entry window computes the sine value

```
# bind a Return in the .r entry to calling comp_s:
r_entry.bind('<Return>', comp_s)
```

- One can bind any keyboard or mouse event to user-defined functions
- We have also replaced the "equals" button by a straight label

Packing widgets

- The pack command determines the placement of the widgets:

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

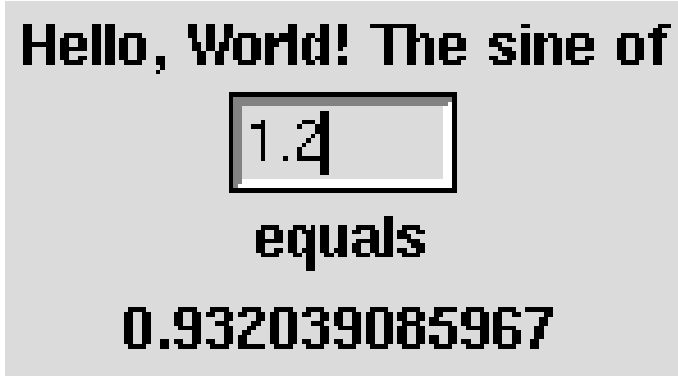
  This results in stacking widgets from left to right

Packing from top to bottom

- Packing from top to bottom:

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

  results in



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

Lining up widgets with frames

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

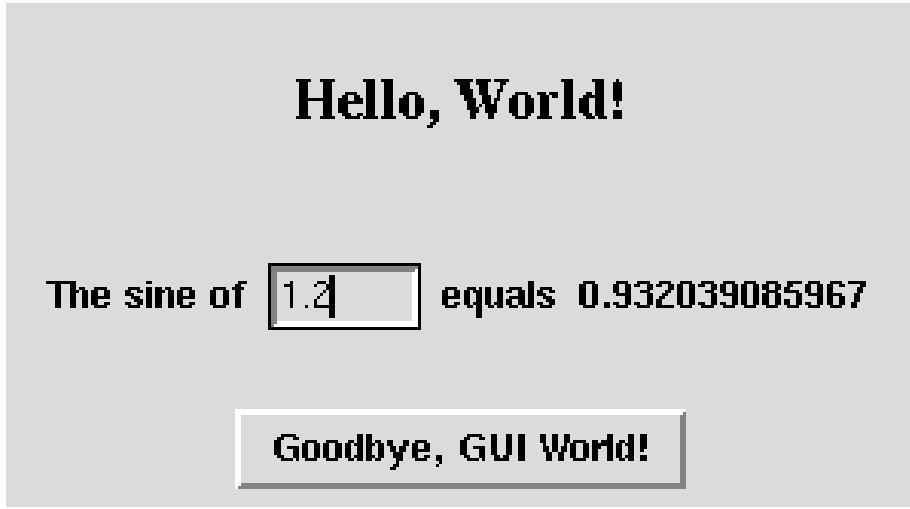
Code for middle frame

```
# create frame to hold the middle row of widgets:
rframe = Frame(top)
# this frame (row) is packed from top to bottom:
rframe.pack(side='top')

# create label and entry in the frame and pack from left:
r_label = Label(rframe, text='The sine of')
r_label.pack(side='left')

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

Change fonts

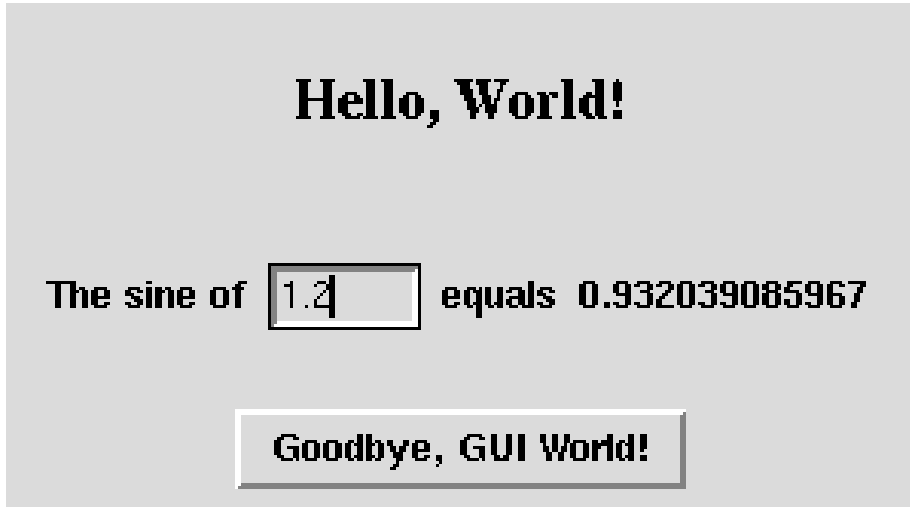



```
# platform-independent font name:
font = 'times 18 bold'

# or X11-style:
font = '-adobe-times-bold-r-normal-*-18-*-*-*-*-*-*-*'

hwtext = Label(hwframe, text='Hello, World!',
               font=font)
```

Add space around widgets



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

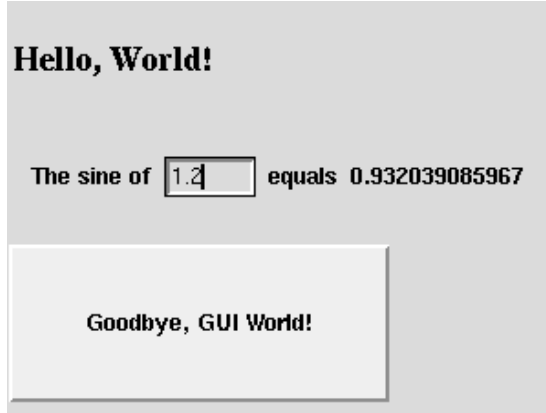
```
hwtext.pack(side='top', pady=20)
rframe.pack(side='top', padx=10, pady=20)
```

Changing colors and widget size


```
quit_button = Button(top,
                     text='Goodbye, GUI World!',
                     command=quit,
                     background='yellow',
                     foreground='blue')
quit_button.pack(side='top', pady=5, fill='x')

# fill='x' expands the widget throughout the available
# space in the horizontal direction
```

Translating widgets



- The anchor option can move widgets:

```
quit_button.pack(anchor='w')
# or 'center', 'nw', 's' and so on
# default: 'center'
```

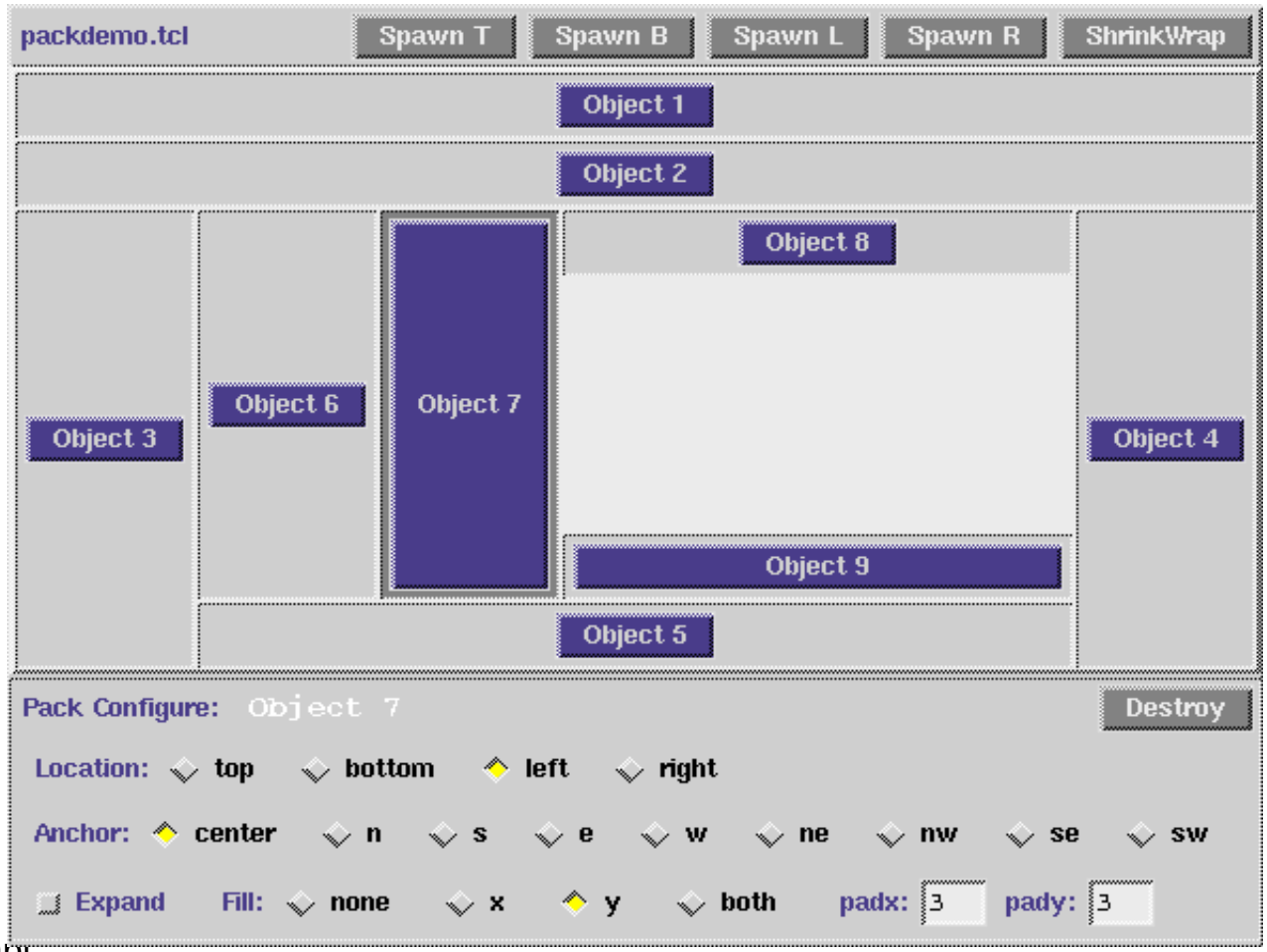
- `ipadx`/`ipady`: more space inside the widget

```
quit_button.pack(side='top', pady=5,
                 ipadx=30, ipady=30, anchor='w')
```

Learning about pack

Pack is best demonstrated through packdemo.tcl:

```
$scripting/src/tools/packdemo.tcl
```



The grid geometry manager

- Alternative to pack: grid
- Widgets are organized in m times n cells, like a spreadsheet
- Widget placement:

```
widget.grid(row=1, column=5)
```

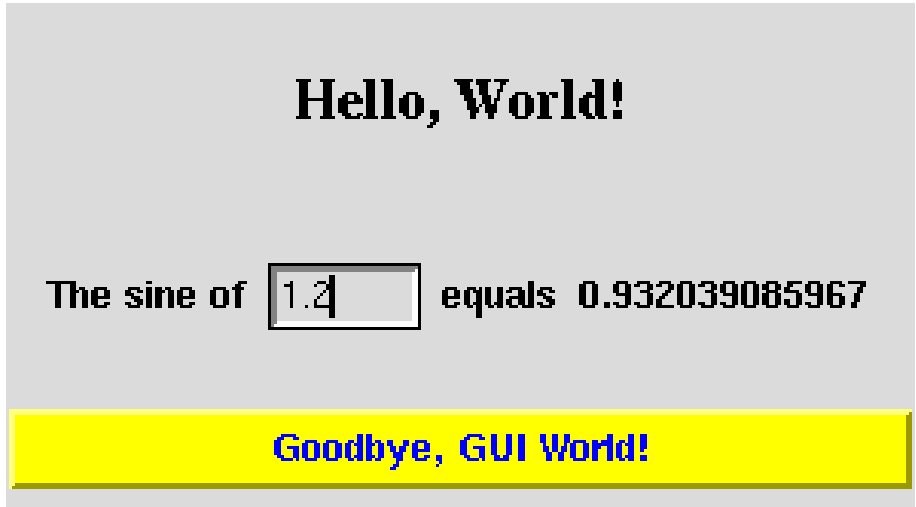
- A widget can span more than one cell

```
widget.grid(row=1, column=2, columnspan=4)
```

Basic grid options

- Padding as with pack (`padx`, `ipadx` etc.)
- `sticky` replaces `anchor` and `fill`

Example: Hello World GUI with grid




```
# use grid to place widgets in 3x4 cells:

hwtext.grid(row=0, column=0, columnspan=4, pady=20)
r_label.grid(row=1, column=0)
r_entry.grid(row=1, column=1)
compute.grid(row=1, column=2)
s_label.grid(row=1, column=3)
quit_button.grid(row=2, column=0, columnspan=4, pady=5,
                 sticky='ew')
```

The sticky option

- `sticky='w'` means `anchor='w'` (move to west)
- `sticky='ew'` means `fill='x'` (move to east and west)
- `sticky='news'` means `fill='both'` (expand in all dirs)

Configuring widgets (1)

- So far: variables tied to text entry and result label
- Another method:
  - ask text entry about its content
  - update result label with `configure`
- Can use `configure` to update any widget property

Configuring widgets (2)

- No variable is tied to the entry:

```
r_entry = Entry(rframe, width=6, relief='sunken')
r_entry.insert('end','1.2')  # insert default value

r = float(r_entry.get())
s = math.sin(r)

s_label.configure(text=str(s))
```

- Other properties can be configured:

```
s_label.configure(background='yellow')
```

- See
  `$scripting/src/py/gui/hwGUI9_novar.py`

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)

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)
```

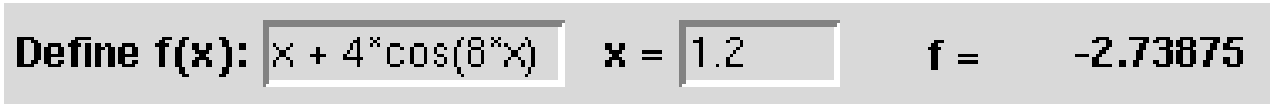
More on event bindings (1)

- 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



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')
```

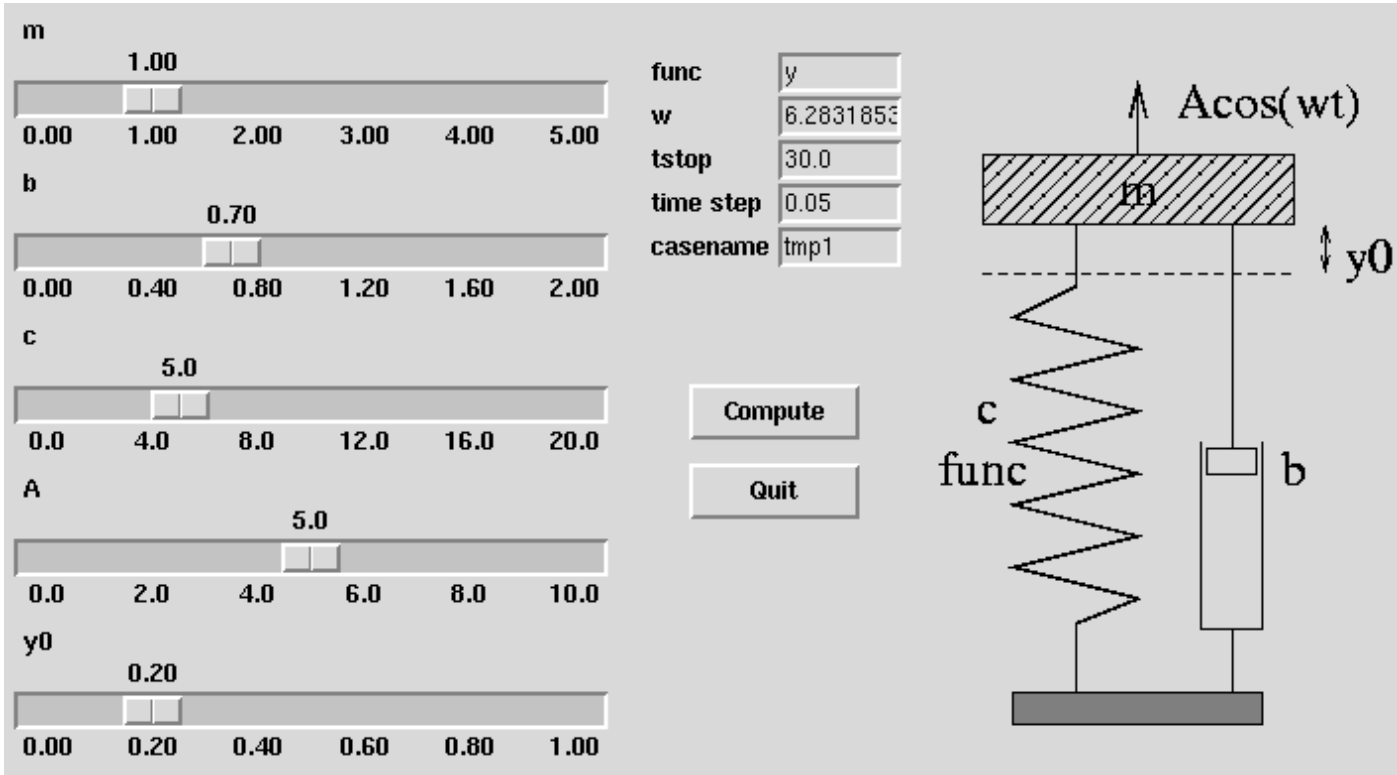
- `exec(s)` executes the string `s` as Python code

```
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:



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 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"""
    ...
```

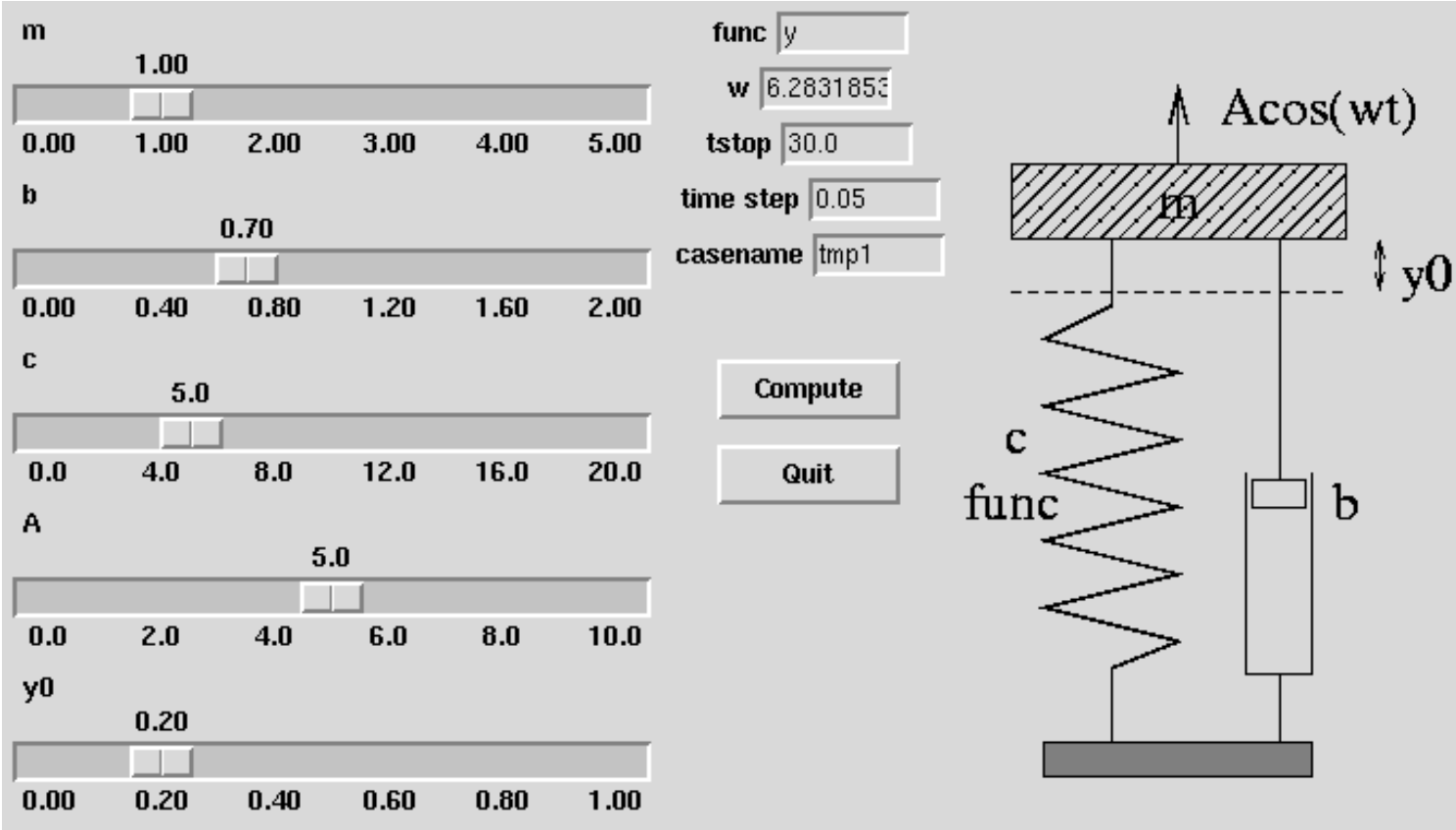
The text entry field

- Version 1 of creating a text field: straightforward packing of labels and entries in frames:

```
def textentry(self, parent, variable, label):
    """make a textentry field tied to variable"""
    f = Frame(parent)
    f.pack(side='top', padx=2, pady=2)
    l = Label(f, text=label)
    l.pack(side='left')
    widget = Entry(f, textvariable=variable, width=8)
    widget.pack(side='left', anchor='w')
    return widget
```

The result is not good...

The text entry frames (f) get centered:



Ugly!

Improved text entry layout

- Use the grid geometry manager to place labels and text entry fields in a spreadsheet-like fashion:

```
def textentry(self, parent, variable, label):
    """make a textentry field tied to variable"""
    l = Label(parent, text=label)
    l.grid(column=0, row=self.row_counter, sticky='w')
    widget = Entry(parent, textvariable=variable, width=8)
    widget.grid(column=1, row=self.row_counter)

    self.row_counter += 1
    return widget
```

- You can mix the use of grid and pack, but not within the same frame

The image

```
sketch_frame = Frame(self.parent)
sketch_frame.pack(side='left', padx=2, pady=2)

gifpic = os.path.join(os.environ['scripting'],
                      'src','gui','figs','simviz2.xfig.t.gif')

self.sketch = PhotoImage(file=gifpic)
# (images must be tied to a global or class variable!)

Label(sketch_frame,image=self.sketch).pack(side='top',pady=20)
```

Simulate and visualize buttons

- Straight buttons calling a function
- Simulate: copy code from simviz1.py
  (create dir, create input file, run simulator)
- Visualize: copy code from simviz1.py
  (create file with Gnuplot commands, run Gnuplot)

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

Resizing widgets (1)

- Example: display a file in a text widget

```
root = Tk()
top = Frame(root); top.pack(side='top')
text = Pmw.ScrolledText(top, ...
text.pack()
# insert file as a string in the text widget:
text.insert('end', open(filename,'r').read())
```

- Problem: the text widget is not resized when the main window is resized

Resizing widgets (2)

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

```
text.pack(expand=1, fill='both')
# all parent widgets as well:
top.pack(side='top', expand=1, fill='both')
```

  `expand` allows the widget to expand, `fill` tells in which directions the widget is allowed to expand

- Try fileshow1.py and fileshow2.py!
- Resizing is important for text, canvas and list widgets

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

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 widg
*Font:                  Helvetica 19 roman
*Foreground:            blue
! then specific properties in specific widgets:
*Label*Font:            Times 10 bold italic
*Listbox*Background:    yellow
*Listbox*Foregrund:     red
*Listbox*Font:          Helvetica 13 italic
```

- Load the file:

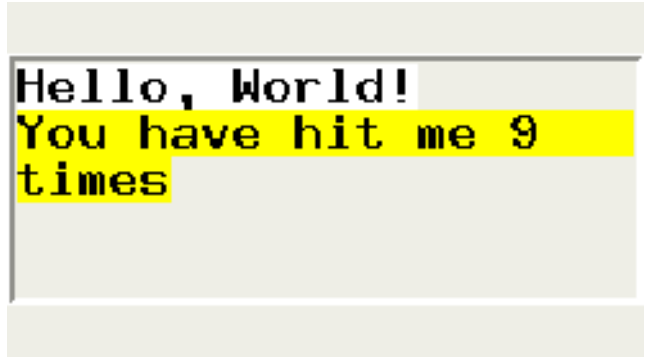
```
root = Tk()
root.option_readfile(filename)
```

Setting widget options in a script

```
general_font = ('Helvetica', 19, 'roman')
label_font   = ('Times', 10, 'bold italic')
listbox_font = ('Helvetica', 13, 'italic')
root.option_add('*Font',                general_font)
root.option_add('*Foreground',          'black')
root.option_add('*Label*Font',          label_font)
root.option_add('*Listbox*Font',        listbox_font)
root.option_add('*Listbox*Background',  'yellow')
root.option_add('*Listbox*Foreground',  'red')
```

Play around with src/py/gui/options.py !

Key bindings in a text widget



- Move mouse over text: change background color, update counter
- Must bind events to text widget operations

Tags

- Mark parts of a text with tags:

```
self.hwtext = Text(parent, wrap='word')
# wrap='word' means break lines between words
self.hwtext.pack(side='top', pady=20)

self.hwtext.insert('end','Hello, World!\n', 'tag1')
self.hwtext.insert('end','More text...\n',  'tag2')
```

- `tag1` now refers to the 'Hello, World!' text
- Can detect if the mouse is over or clicked at a tagged text segment

Problems with function calls with args

- We want to call

```
self.hwtext.tag_configure('tag1', background='blue')
```

  when the mouse is over the text marked with `tag1`

- The statement

```
self.hwtext.tag_bind('tag1','<Enter>',
     self.tag_configure('tag1', background='blue'))
```

  does not work, because function calls with arguments are not allowed as parameters to a function (only the name of the function, i.e., the function object, is allowed)

- Remedy: lambda functions

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

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!

Designer tools/GUI builders

- With the basic knowledge of GUI programming, you may try out a designer tool for interactive automatic generation of a GUI
- Several alternatives exist:
  - Tkinter: Rapyd-Tk, Visual Tkinter, Komodo, PAGE
  - PyGtk: Glade, see `doc.html` for introductions
- Working style: pick a widget, place it in the GUI window, open a properties dialog, set packing parameters, set callbacks etc.