GUI Programming with Python

AY250 Fall 2013

authors: J. Bloom

Overview of Today's Lecture

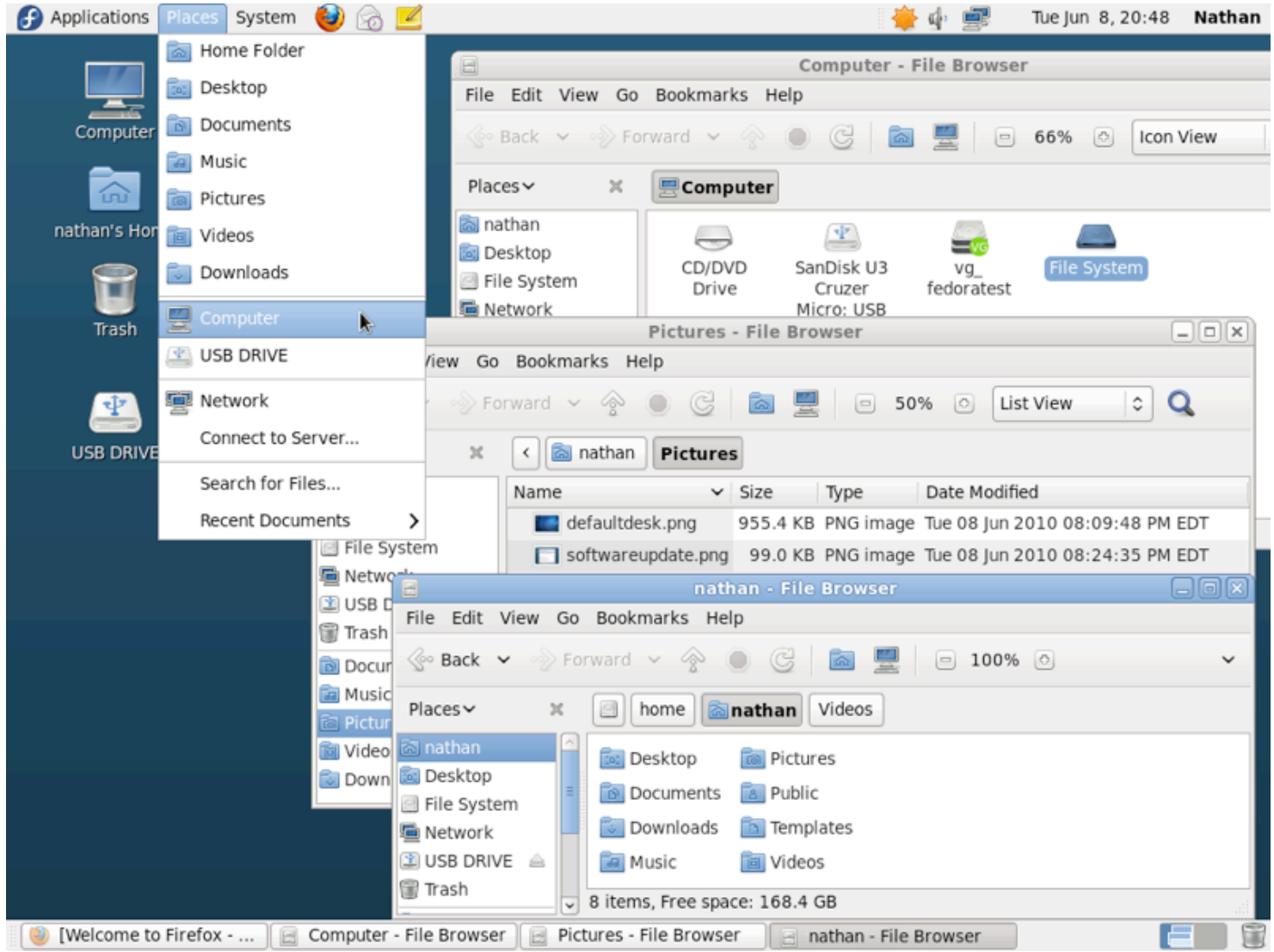
Why do we need GUIs?

GUI programming paradigms

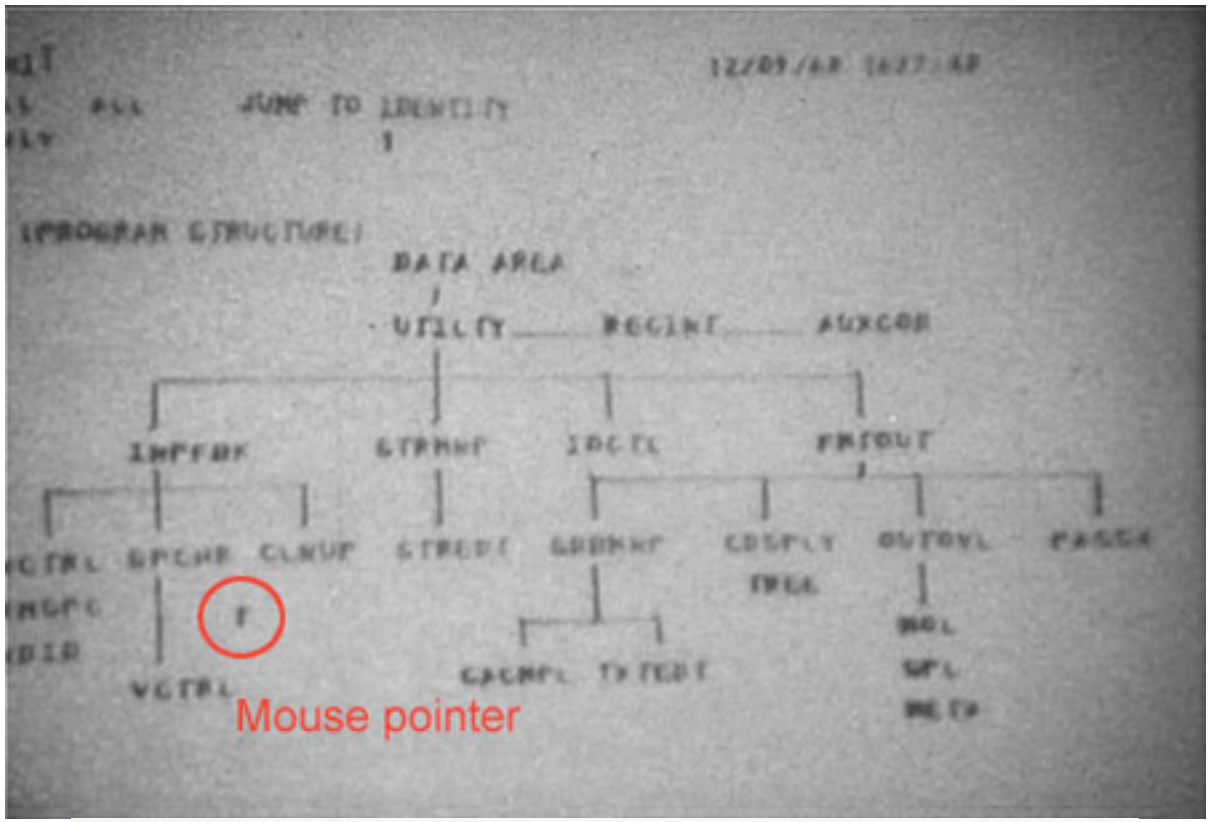
- Tkinter
- Traits
- matplotlib widgets

What are GUIs?

Graphical User Interface. Visual-based (as opposed to text-based) interaction (I/O) experience



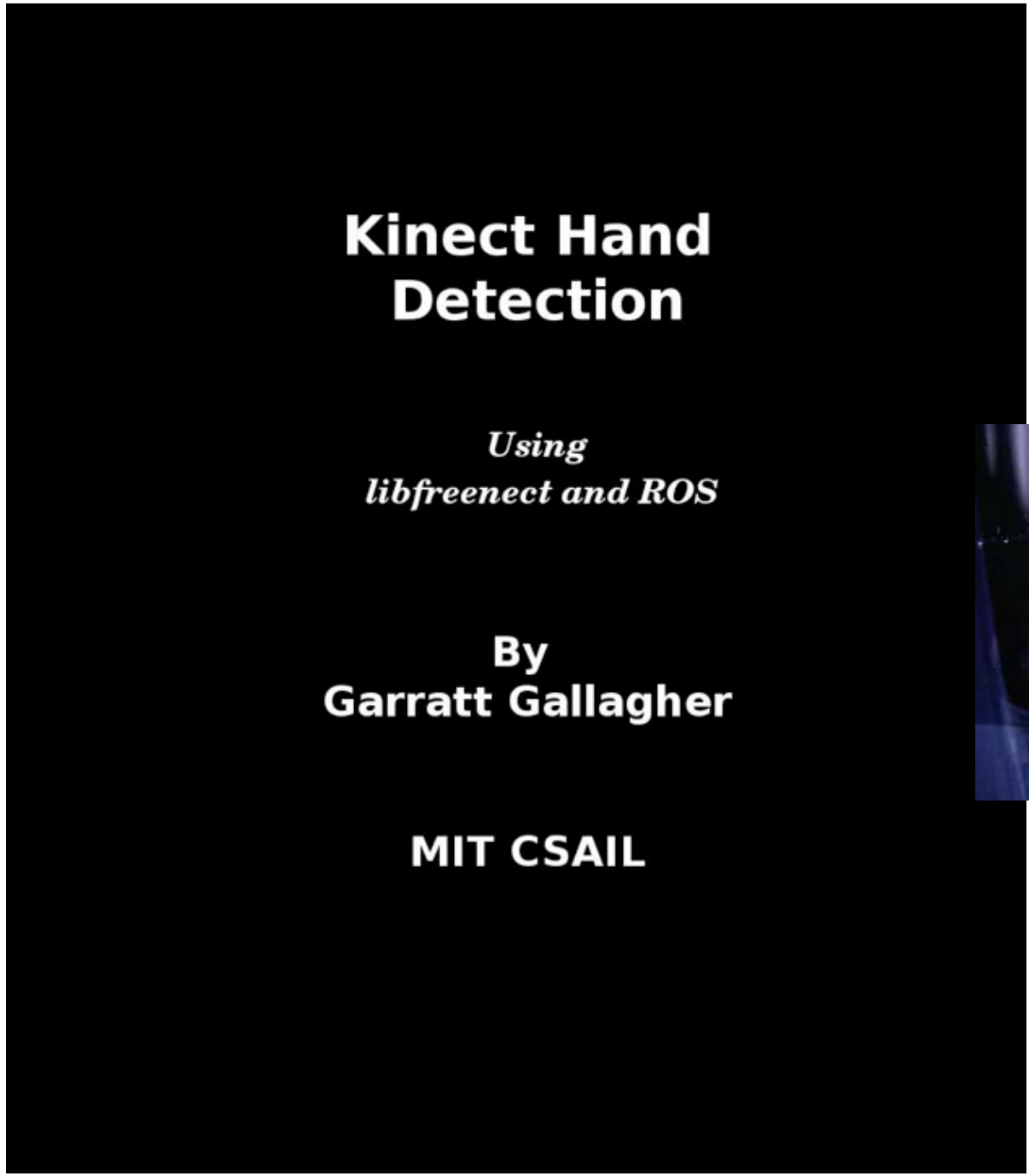
Prehistoric Times



Douglas Engelbart in 1968

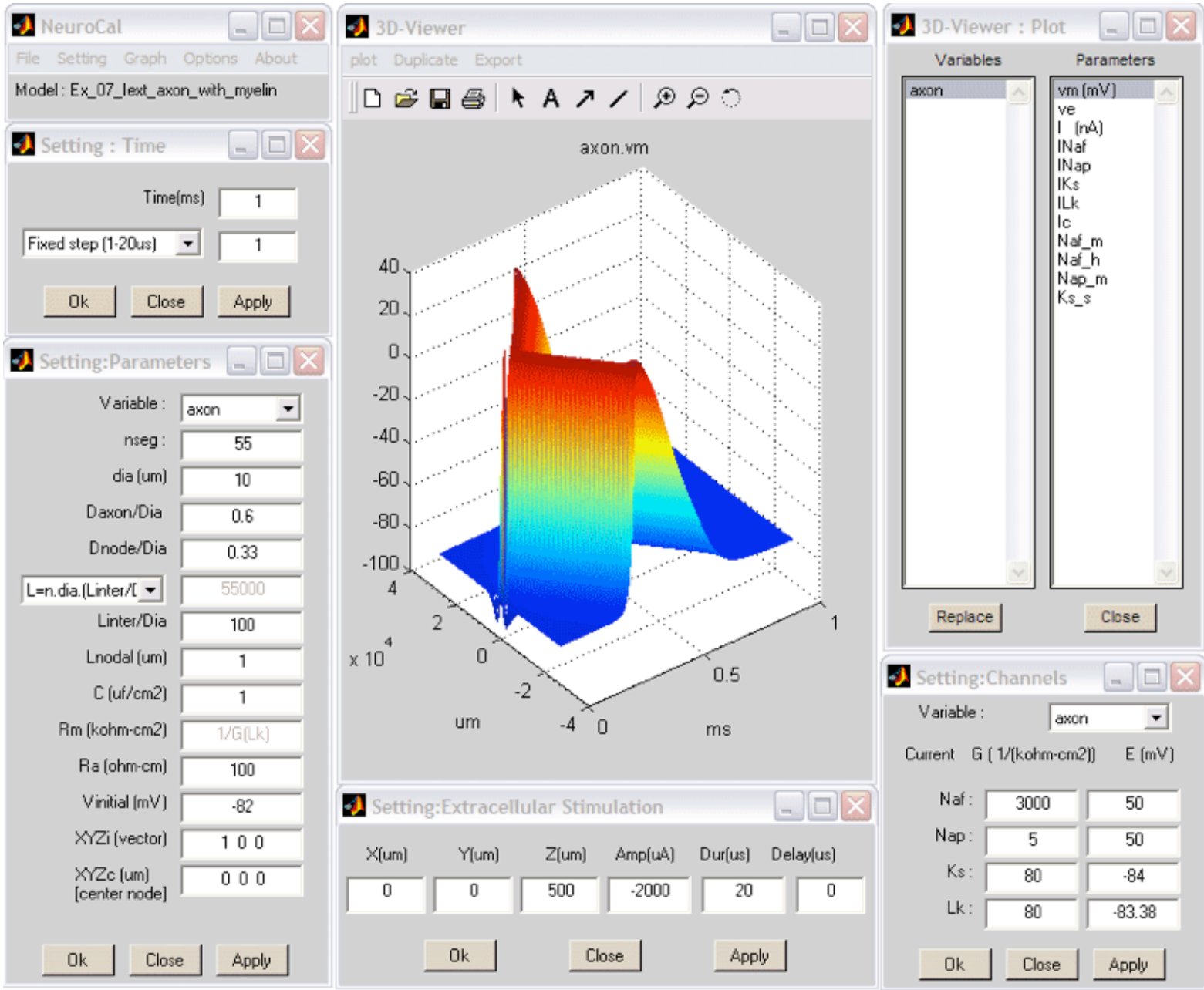
http://arstechnica.com/old/content/2005/05/gui.ars

Future



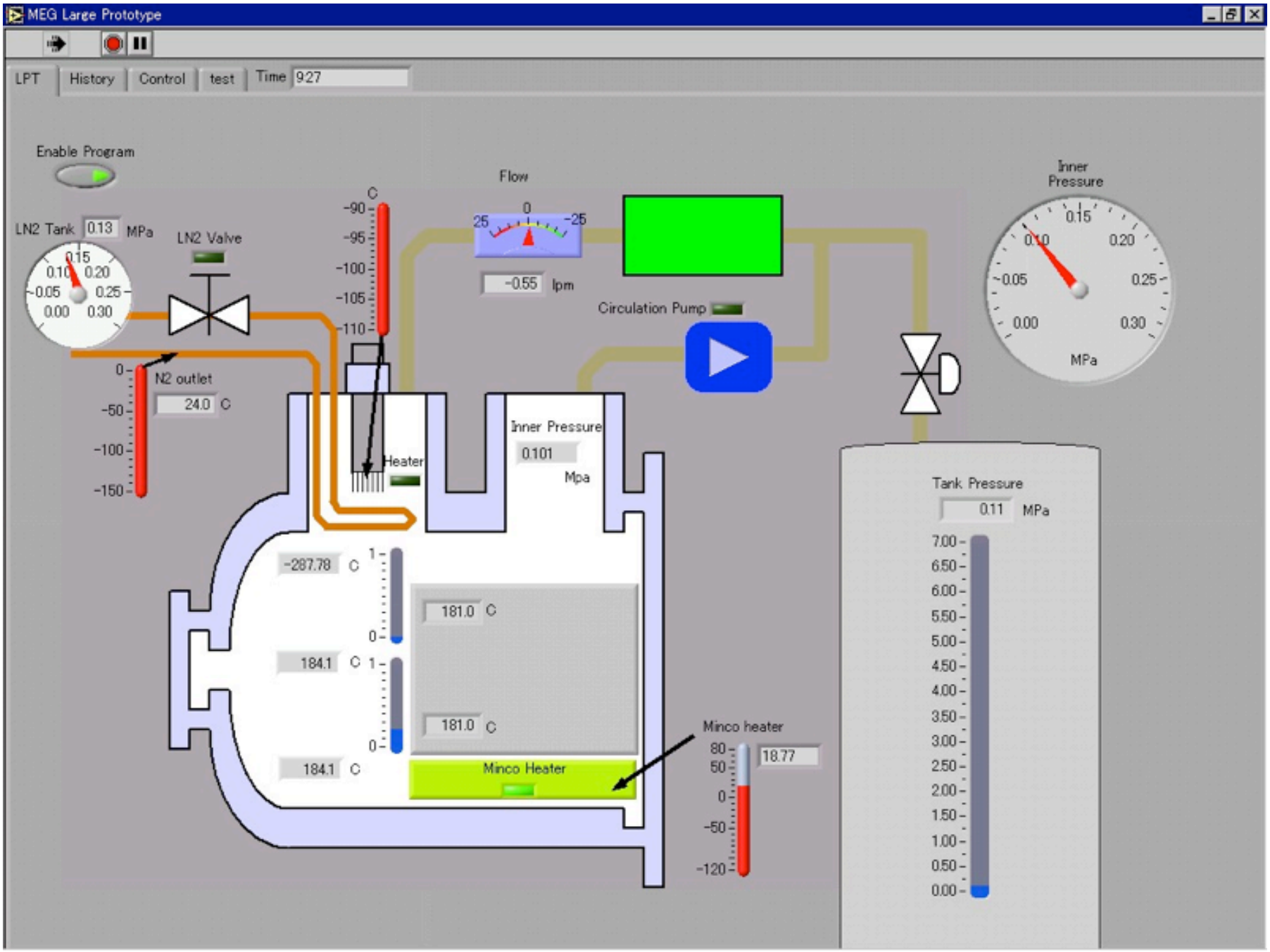
http://www.youtube.com/watch?v=tlLschoMhuE

Interacting with (and controlling) Simulations

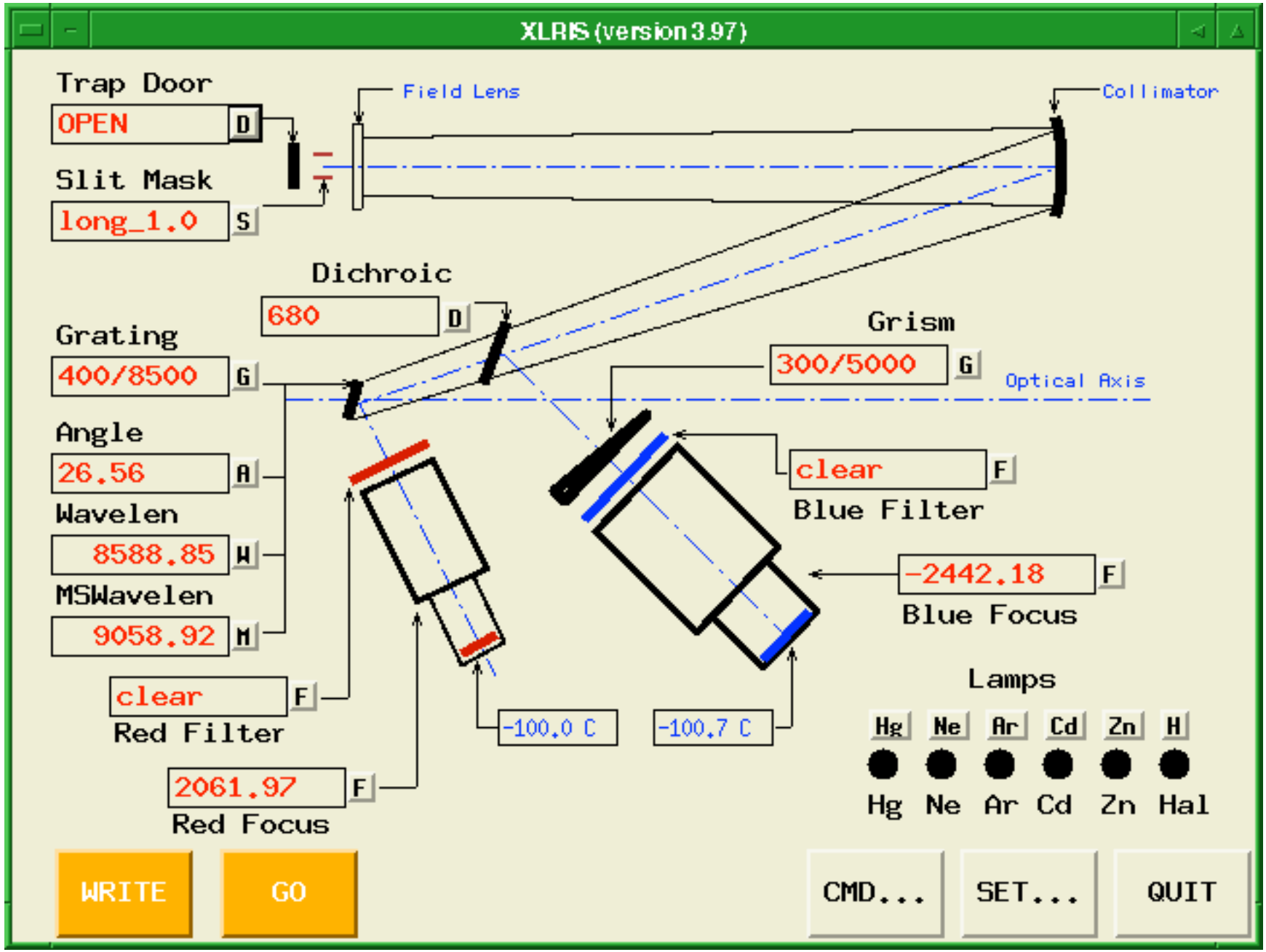


http://www.mathworks.com/matlabcentral/fileexchange/1588-neurocal

Control & Visualize Complex Systems

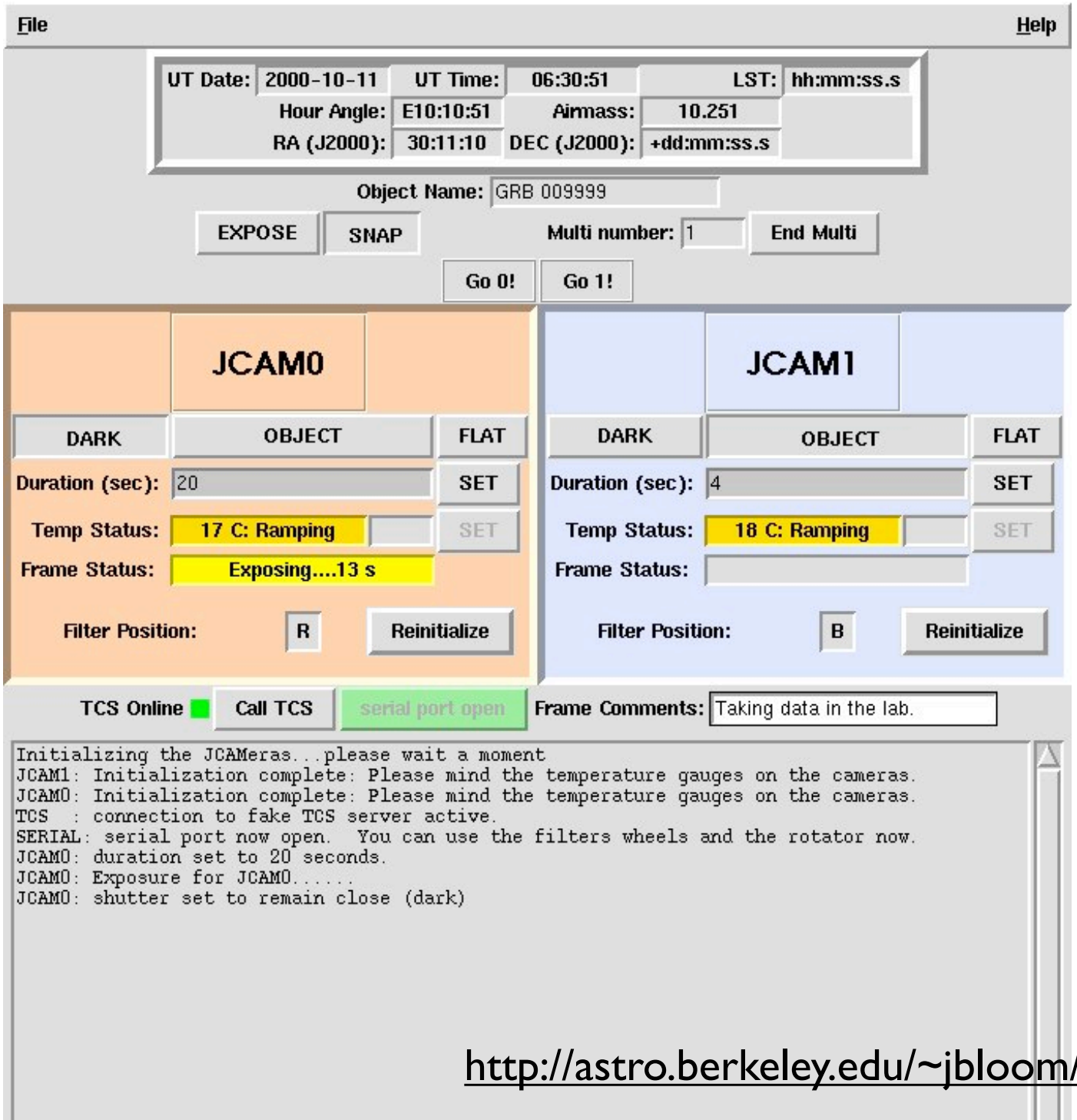


Front end for data taking



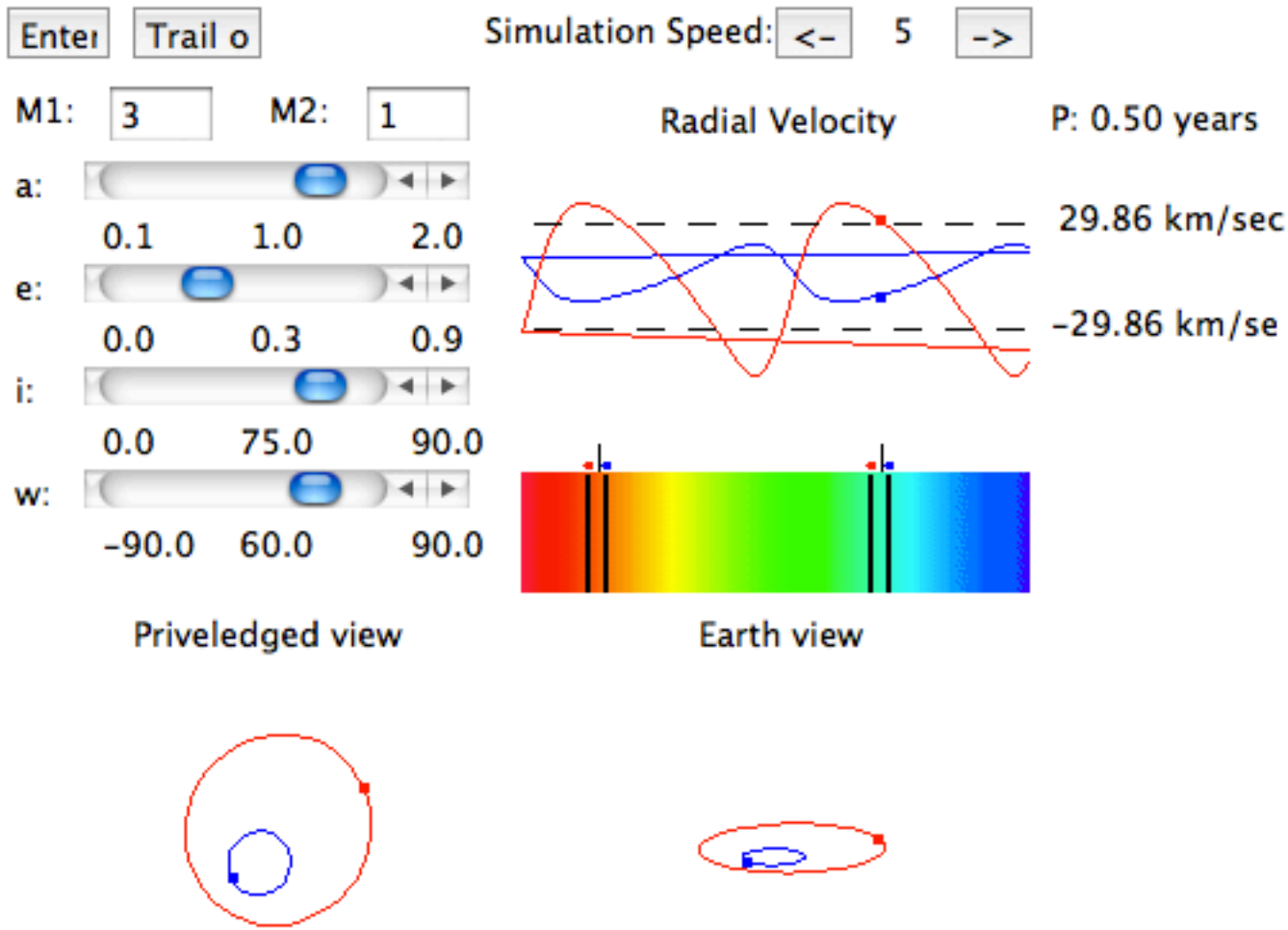
http://www2.keck.hawaii.edu/inst/lris/xlris.html

Front end for data taking



http://astro.berkeley.edu/~jbloom/Jcam/

Teaching Tools



http://www.phy.duke.edu/~kolena/binary/binary.html

Anatomy of a GUI

root window

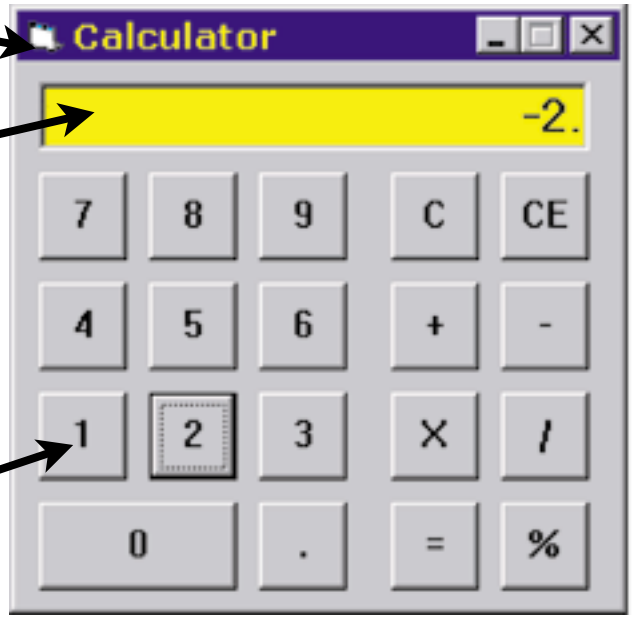
Calculator

Text Entry + Display Widget

Calculator.TxtE

Button

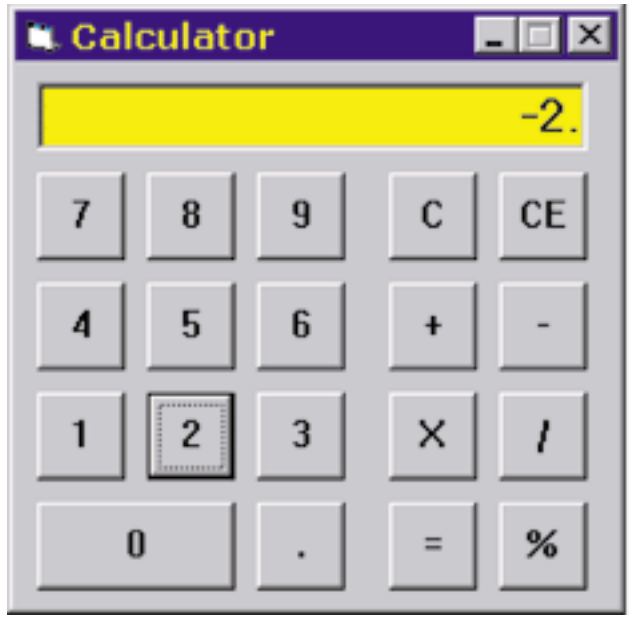
Calculator.Button1



- Objects/Widgets are instantiated and associated with their parent window

- Each object has special properties & capabilities (buttons have different behavior from text entries)

Anatomy of a GUI



Anatomy of a GUI

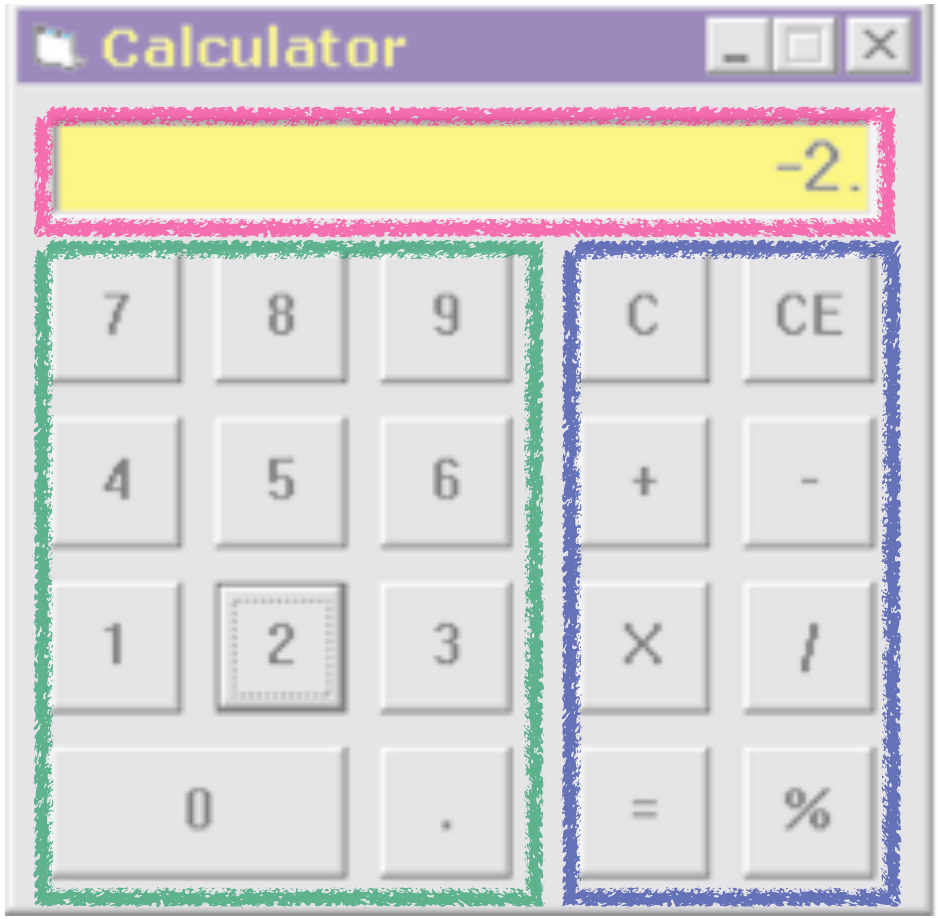
Layout Widgets

(perhaps within their own subcontainers called Frames)

<u>Different notions of geometry layout</u>

packing, gridding, placing, ...

these are handled by geometry managers



GUIs in Python

Tkinter: object-oriented binding for Tk, an interpreted GUI framework, ported to most platforms (Tcl is the standard interpreter). built-in & now gives native OS look & feel.

wxPython: OO wrapper for cross-platform wxWidgets. Popular, but not a built-in

Traits: different (interesting) GUI paradigm, part of the EPD installation

pyQt, pyGTK:

http://docs.python.org/faq/gui.html
https://wiki.python.org/moin/GuiProgramming

4 Basics of GUI Programming

1) You must specify how you want the UI to **look**.

2) You must decide what you want the UI to **do**.

3) You must associate the "looking" with the "doing". That is, you must write code that associates the things that the user sees on the screen with the routines that you have written to perform the program's tasks.

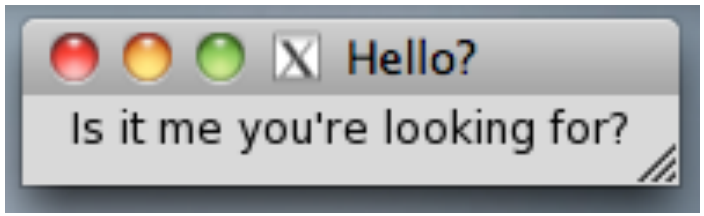
4) finally, you must write code that sits and waits for input from the user.

http://www.ferg.org/thinking_in_tkinter/all_programs.html

GUI Hello World

file: ghello.py

```
>>> from Tkinter import *
>>> root = Tk() ; root.title("Hello?") #make a root window
>>> # stick a label in it and pack it in.
>>> Label(root, text="Is it me you're looking for?").pack()
>>> root.mainloop() # run the main loop
```



Pretty good reference:

http://effbot.org/tkinterbook/

Event loop Manager

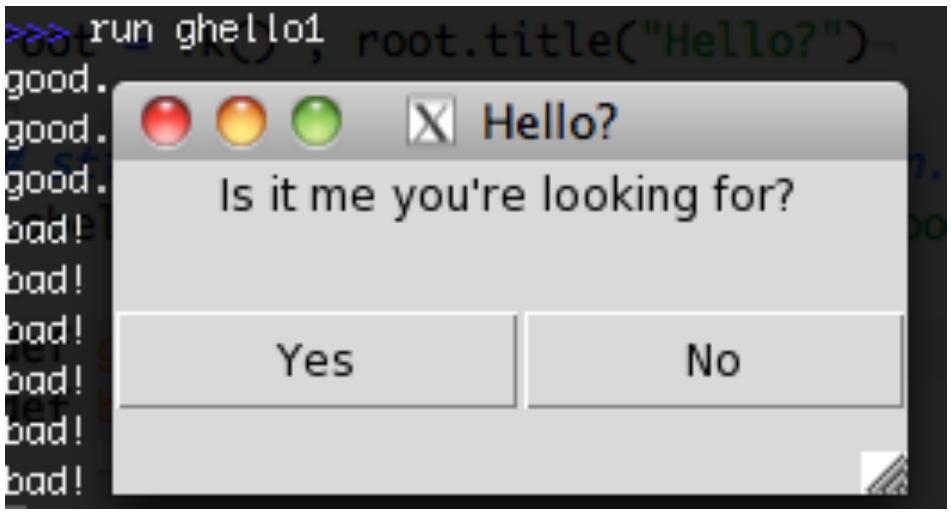
- in charge of making windows, resizing them, etc.

- lords over geometry managers (grid, pack)

- watches events & takes actions as appropriate

file: ghello1.py

```
>>> from Tkinter import *
>>> root = Tk() ; root.title("Hello?") #make a root window
>>> # stick a label in it and pack it in.
>>> Label(root, text="Is it me you're looking for?").pack()
>>> # make some buttons
>>> def good(): print "good."
>>> def bad(): print "bad!"
>>> b1 = Button(root,text="Yes",command=good)
>>> b2 = Button(root,text="No",command=bad)
>>> b1.pack(side=LEFT,expand=1,fill=X) ; b2.pack(side=LEFT,fill=X,expand=1)
>>> root.mainloop() # run the main loop
```

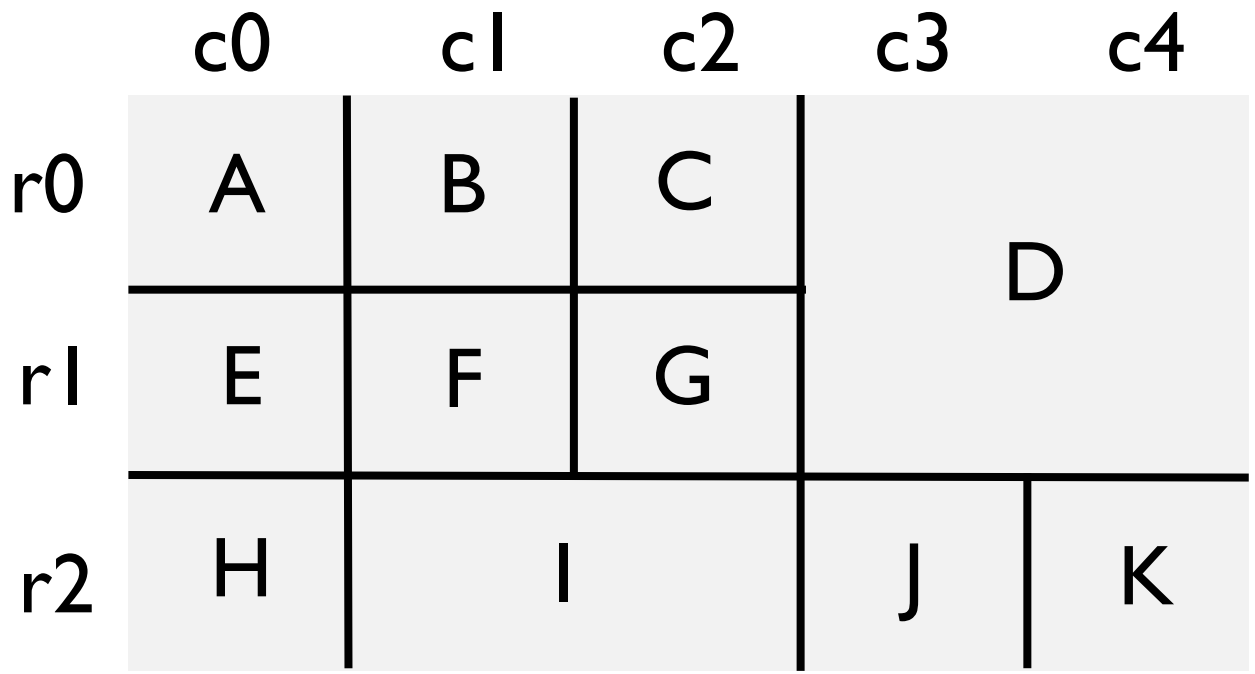


`Button(command=func)`

`func` gets called whenever `Button` is invoked

Geometry Managers

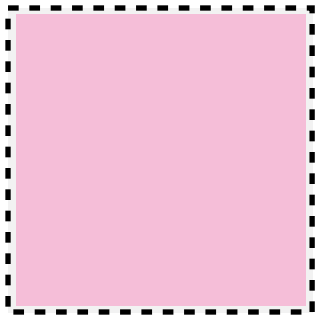
Grid: matrix layout with columns and rows




```
A: row=0, column=0

G: row=1, column=2

I: row=2, column=1,
    columnspan=2

D: row=0, column=3,
columnspan=2, rowspan=2
```

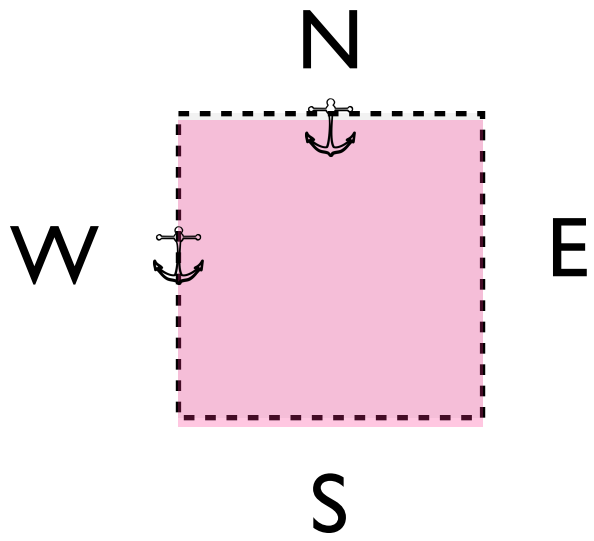
- x & y sizes of the cells are equal
- empty (unfilled) cells are ignored
- no fixed grid size; grid manager figures out what you want, making a minimal spanning grid

Placement within a cell is controlled too

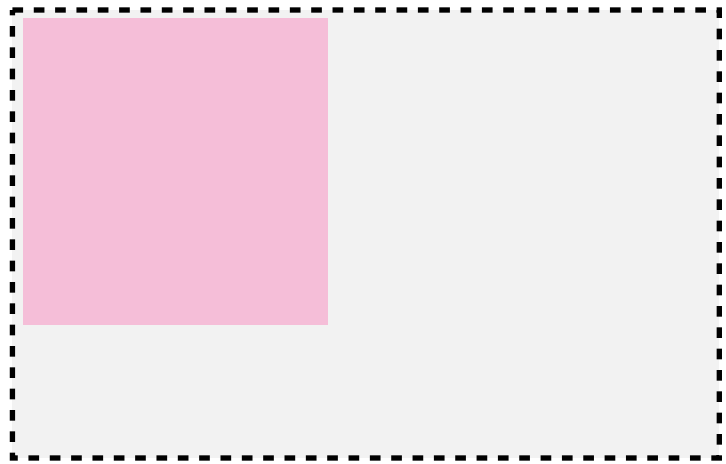


default is for the widget to
float at the center of the cell

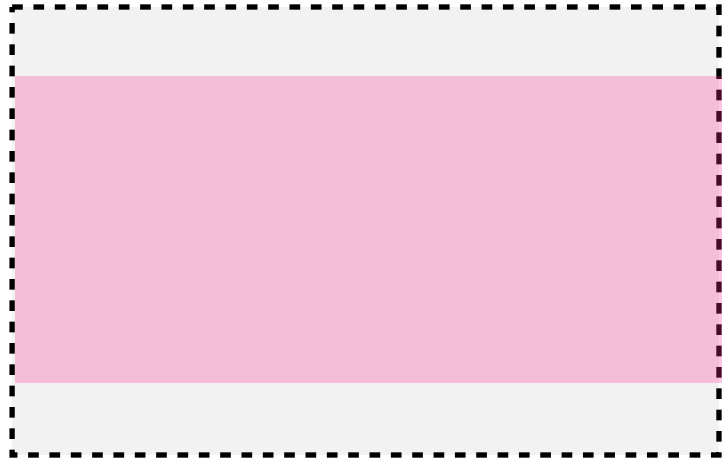
we can anchor the widget to
any combination of N,S,E,W
(e.g., `sticky=(N,W)`)



when expanded, sticky edges,
keep the widget anchored



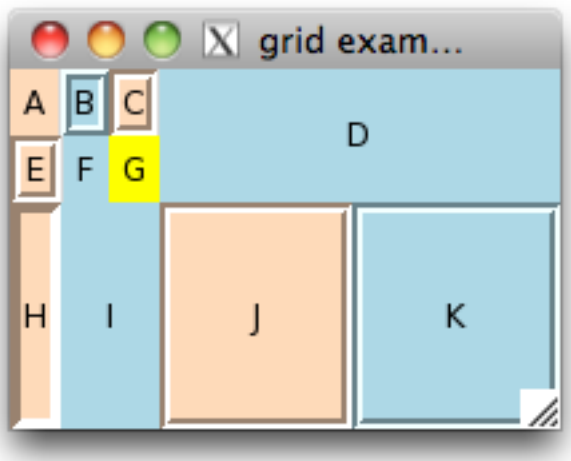
when expanded, sticky edges, keep the widget anchored, even stretching out the widget...

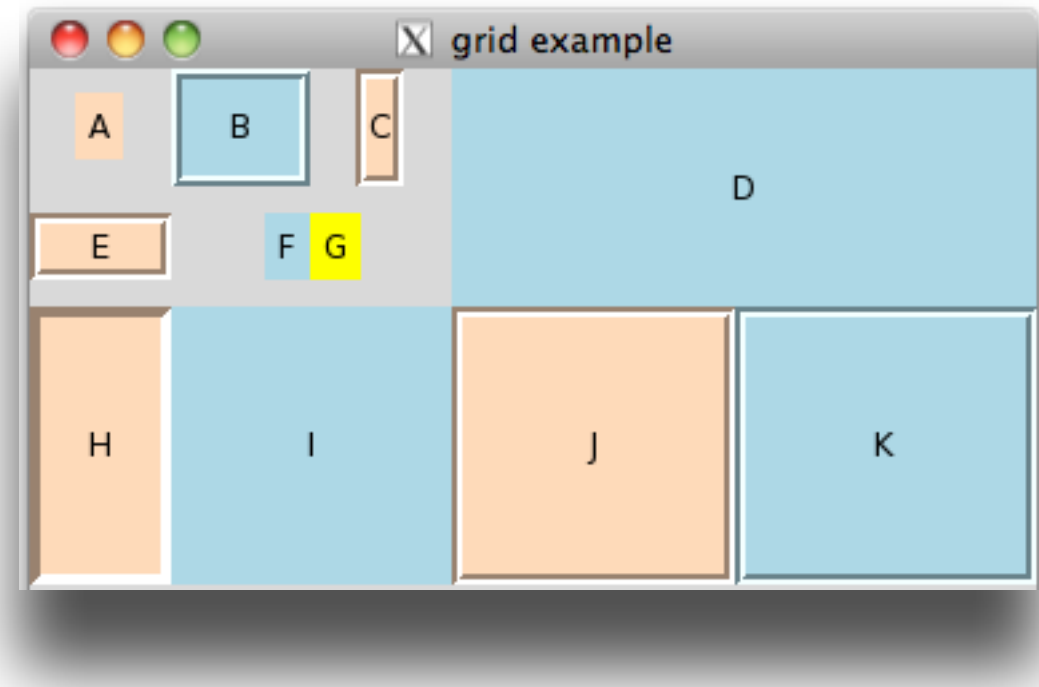


```
sticky=(E,W)
```

Control what cells can expand and by how much with:

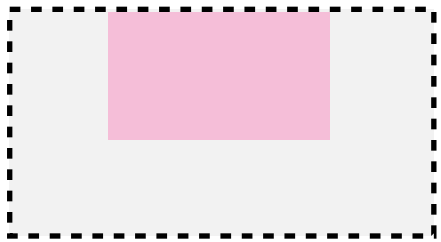
```
parent.rowconfigure(row_index, weight=1)
parent.columnconfigure(col_index, weight=1)
```



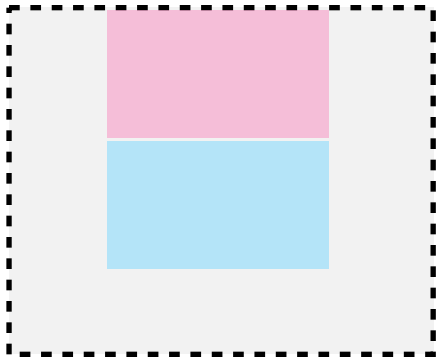


file: grid.py

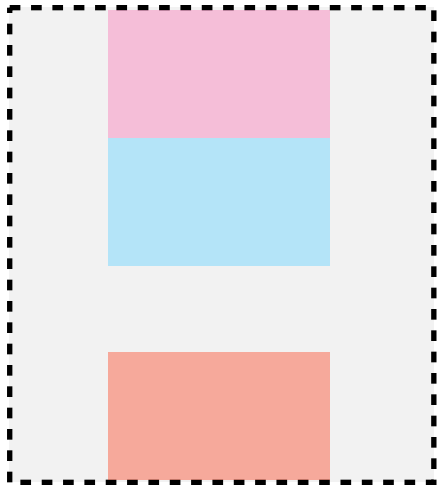
Pack: places new child widgets in order they are packed, with new widgets filling in available space left over



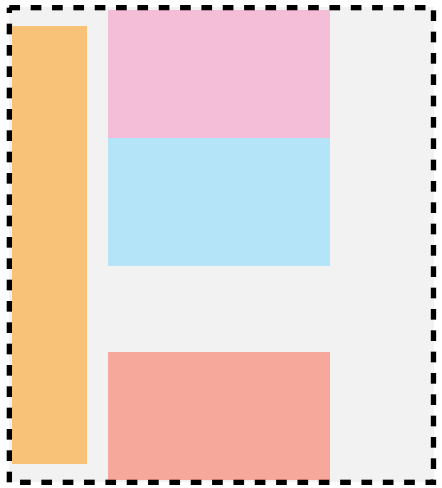
`side=TOP`



`side=TOP`



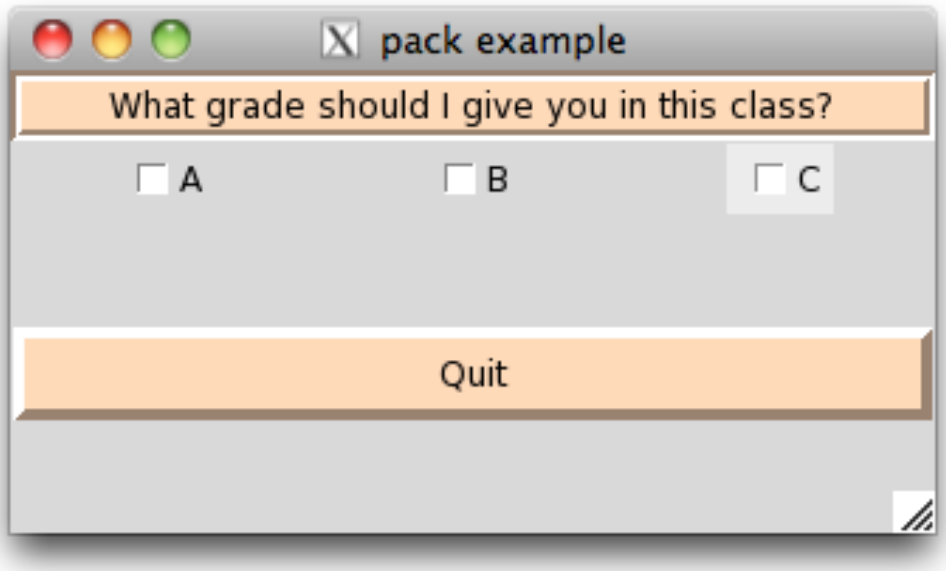
`side=BOTTOM`



`side=LEFT`

```
# stick a label in it and pack it in.
q = Button(root, text="Quit",borderwidth=4,bg="PeachPuff")
a = Checkbutton(root, text="A",borderwidth=4)
b = Checkbutton(root, text="B",borderwidth=4)
c = Checkbutton(root, text="C",borderwidth=4)
e = Label(root, text="What grade should I give you in this
class?",borderwidth=4,bg="PeachPuff",relief=GROOVE)

e.pack(side=TOP,fill=X)
q.pack(side=BOTTOM,expand=1,fill=X)
a.pack(side=LEFT,expand=1)
```

file: pack.py



small exercise: get about the same layout in `pack.py` but using the grid manager instead

Connecting & Dealing with Events

We can do callbacks when:

1. actions on the widget are taken

```
Button(...,command=func_name)
```

2. on Keyboard, Mouse, Focus, or File events

```
frame1.bind("<Button-1>",callback_name)
```

3. When special Tk variables are touched/ changed

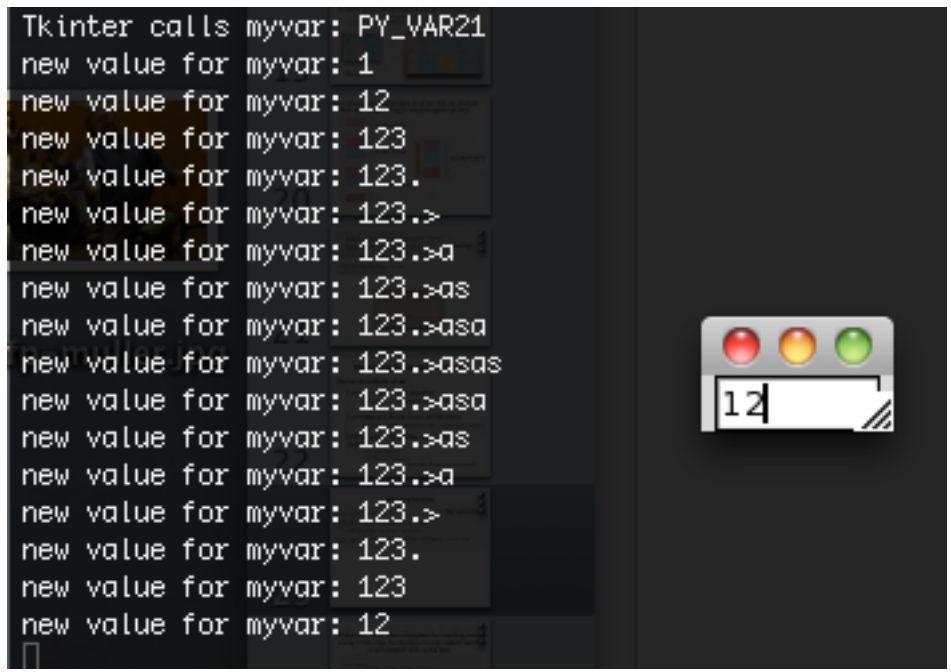
```
var = StringVar()
var.set("hello")
var.trace("w", callback)
```

(BooleanVar, DoubleVar, IntVar, StringVar)

Watching Variables

file: events0.py

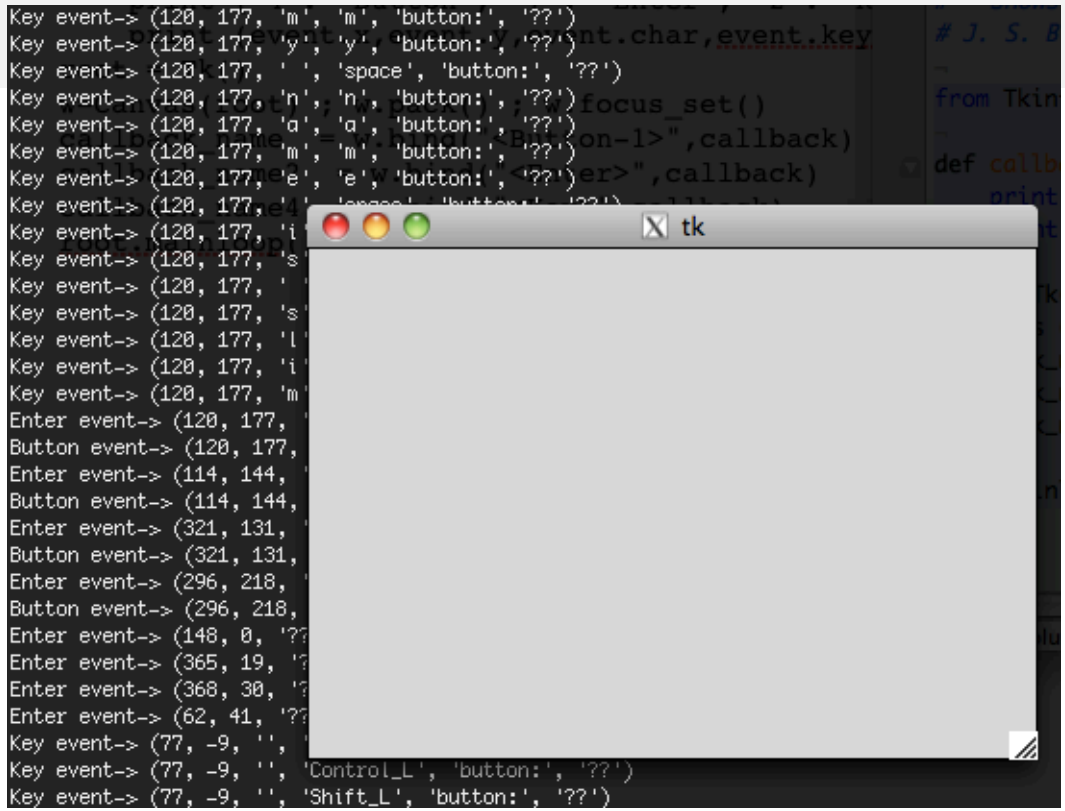
```
from Tkinter import *
root = Tk()
myvar = StringVar()
Entry(root, width=7, textvariable=myvar).pack()
def changing(*args):
    print "new value for myvar:", myvar.get()

myvar.trace("w",changing) ; print 'Tkinter calls myvar:', myvar._name
root.mainloop()
```



Binding Events

file: bind.py

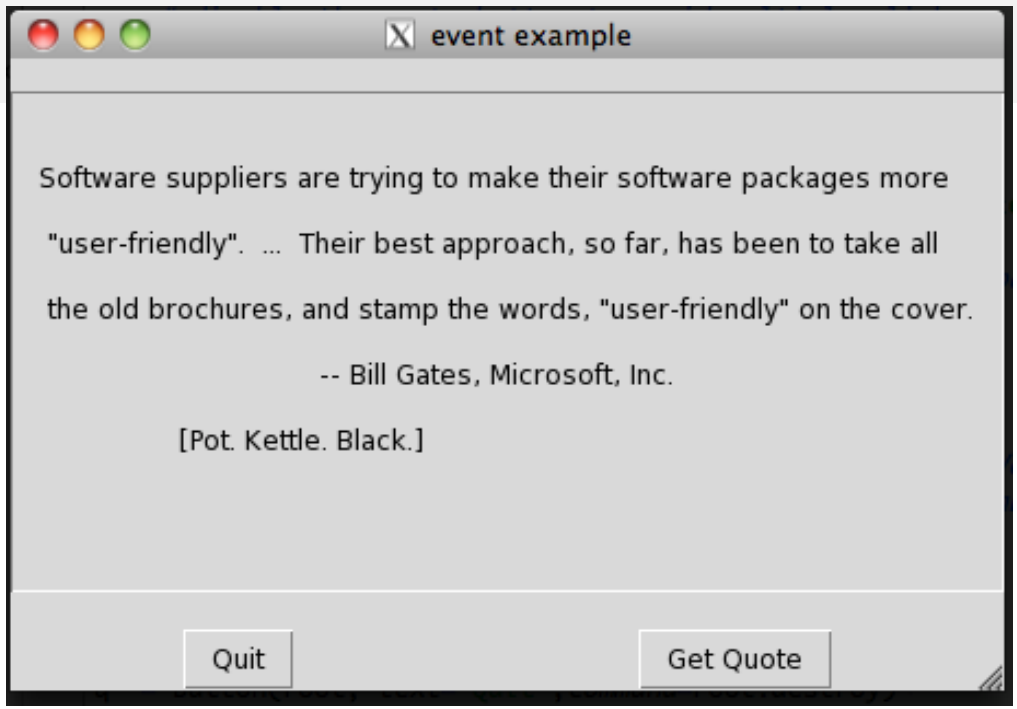
```
from Tkinter import *
def callback(event):
    print {"4": "Button", "7": "Enter", "2": "Key"}[event.type] + " event->",
    print (event.x,event.y,event.char,event.keysym,"button:",event.num)
root = Tk()
w=Canvas(root) ; w.pack() ; w.focus_set()
callback_name  = w.bind("<Button-1>",callback)
callback_name2  = w.bind("<Enter>",callback)
callback_name4  = w.bind("<Key>",callback)
root.mainloop()
```



Commands & Variable Watch Mashup
quote of the day generator

file: events.py

```
...
qt = StringVar()
def show_quote(*args):
    # disable the quote button to avoid multiple clicks
    a.configure(state=DISABLED)
    a.update_idletasks()

    # grab a new quote from the web
    qt.set(" ".join(urllib2.urlopen("http://www.iheartquotes.com/api/v1/random").readlines()[:-2]))

    # set the button state back to normal (enabled) and update all the Event Loop Tasks
    a.configure(state=NORMAL)
    root.update()
...
```

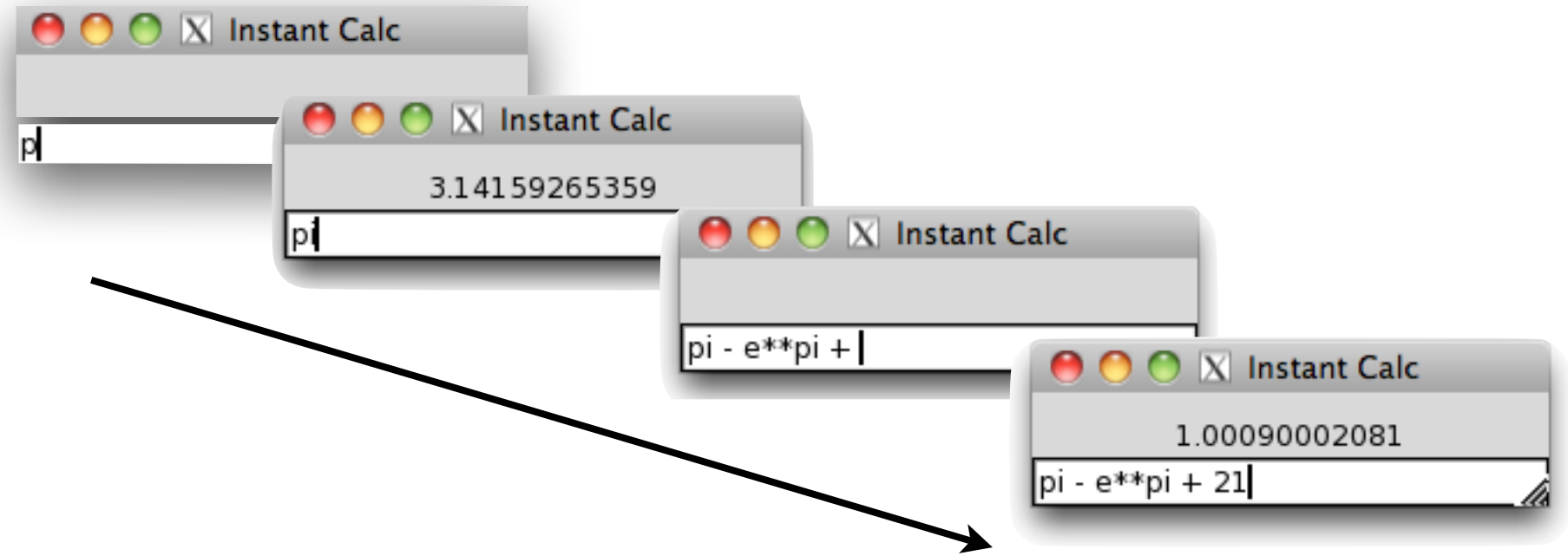


Breakout

create an "Instant Calculator," showing you Google-like response as you type in an expression

a. pack a Label and Entry widget
b. use a callback which does a "safe evaluation" of math expressions:

```
import math
seval = lambda expr: eval(expr, dict(__builtins__=None), vars(math))
```



Traits

http://code.enthought.com/projects/traits/

"The brilliance of Traits UI is that you don't really have to think about the mapping between model and GUI. You can concentrate on the model, and the GUI automatically reflects changes to the model ... and vice versa."
— Danny Shevitz, Los Alamos NM

Model-View-Controller Paradigm

model - manages the data, state, and internal logic of the application
view - format the model data into a graphical display with which the end user can interact
controller - manages the transfer of information between model and view so that neither needs to be directly linked to the other

Traits: allow you to think first about the data, then work on visualizing

Features of Traits

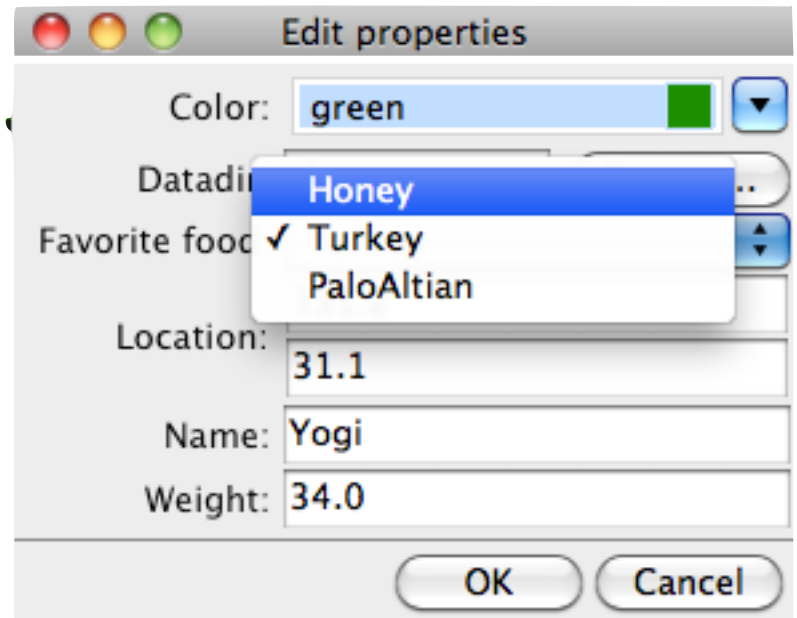
- **Initialization**: All traits have default values
- **Validation**: Trait attribute type is explicitly declared. Only values that meet the trait definition can be assigned to that attribute.
- **Delegation**: The value of a trait attribute can be contained either in the defining object or in another object delegated to by the trait.
- **Notification**: Setting the value of a trait attribute can notify other parts of the program that the value has changed.
- **Visualization**: User interfaces that allow a user to interactively modify the value of a trait attribute can be automatically constructed using the trait's definition.

Data Model as Classes

```
try:
  from enthought.traits.api import HasTraits, Str, Int, Directory, RGBColor, Float, Array
  from enthought.traits.ui.api import View, Item, Group
except:
  from traits.api import HasTraits, Str, Int, Directory, RGBColor, Float, Array
  from traitsui.api import View, Item, Group

class Bear(HasTraits):
    name = Str
    color = RGBColor("gold")
    weight = Float
    location = Array()
    datadir = Directory("./")

yogi = Bear(name="Yogi",weight=34.0,location=(131.2,+31.1))
yogi.configure_traits()
```

- data attributes are strongly typed (*Validation*)
- different data types have different editors in the GUI

file: `traits1.py`



Lots of predefined Trait types

Name
Any
Array
Button
Callable
CArray
Class
Code
Color
CSet
Dict, DictStrAny, DictStrBool, DictStrFloat, DictStrInt, DictStrList, DictStrLong, DictStrStr
Directory
Disallow
Enum
Event
Expression
false
File
Font
Function
Generic
generic_trait
HTML
Instance
List, ListBool, ListClass, ListComplex, ListFloat, ListFunction, ListInstance, ListInt, ListMethod, ListStr, ListThis, ListUnicode
Method
Module
Password
Property
Python
PythonValue
Range
ReadOnly
Regex
RGBColor
self
Set
String
This
ToolbarButton
true
Tuple
Type
undefined
UStr
UUID [3]
WeakRef

_variable_changed, _variable_fired: special methods to watch traits and take action (*Notification*)

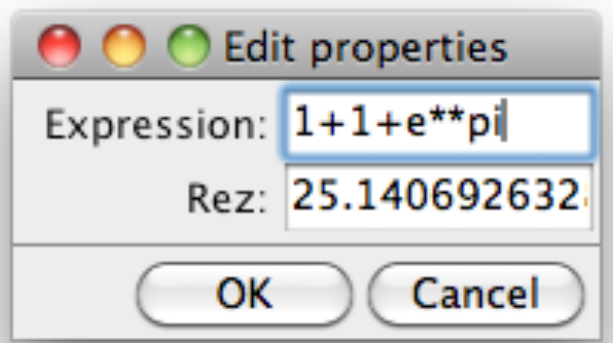
```
try:
  from enthought.traits.api import HasTraits, Int, Button
  from enthought.traits.ui.api import View, Item, ButtonEditor
except:
  from traits.api import HasTraits, Int, Button
  from traitsui.api import View, Item, ButtonEditor

class Counter(HasTraits):
    value =  Int()
    add_one = Button()
    def _add_one_fired(self):
        self.value +=1
    view = View('value', Item('add_one', show_label=False ))

Counter().configure_traits()
```

file: button.py

_variable_changed, _variable_fired: special methods to watch traits and take action (*Notification*)

```
try:
  from enthought.traits.api import HasTraits, Str, Float ; import math
except:
  from traits.api import HasTraits, Str, Float ; import math

seval = lambda expr: eval(expr, dict(__builtins__=None), vars(math))

class Calc(HasTraits):
    expression = Str
    rez        = Str("Type an expression")
    def _expression_changed(self, old, new):
            try:
                self.rez = str(seval(new))
            except:
                pass
```

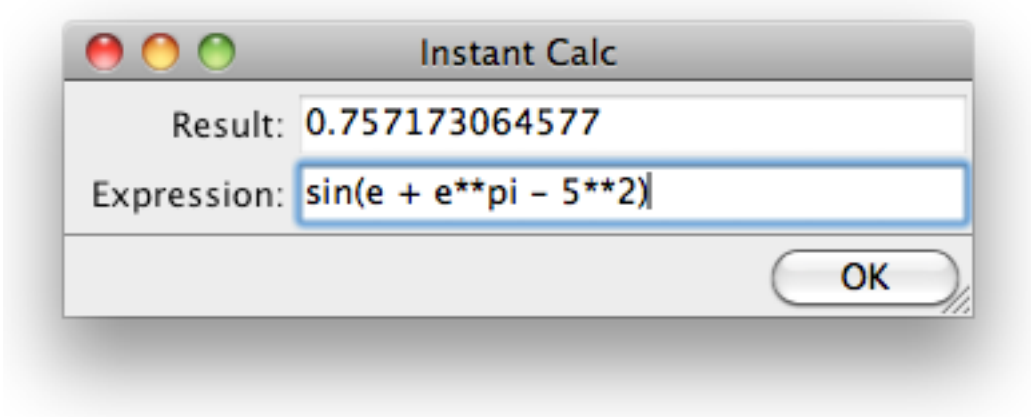
file: icalc.py



View..Item allows us to create stylized views of the models (*Visualization*)

file: icalc1.py

```
view1 = View(Item('rez',width=-250,resizable=True,padding=2,
                  label="Result",enabled_when="False",full_size=True),
             Item('expression', width=-250,resizable=True),
             title='Instant Calc',resizable=True,
                   buttons=['OK'],scrollable=False)

c = Calc() ; c.configure_traits(view=view1)
```

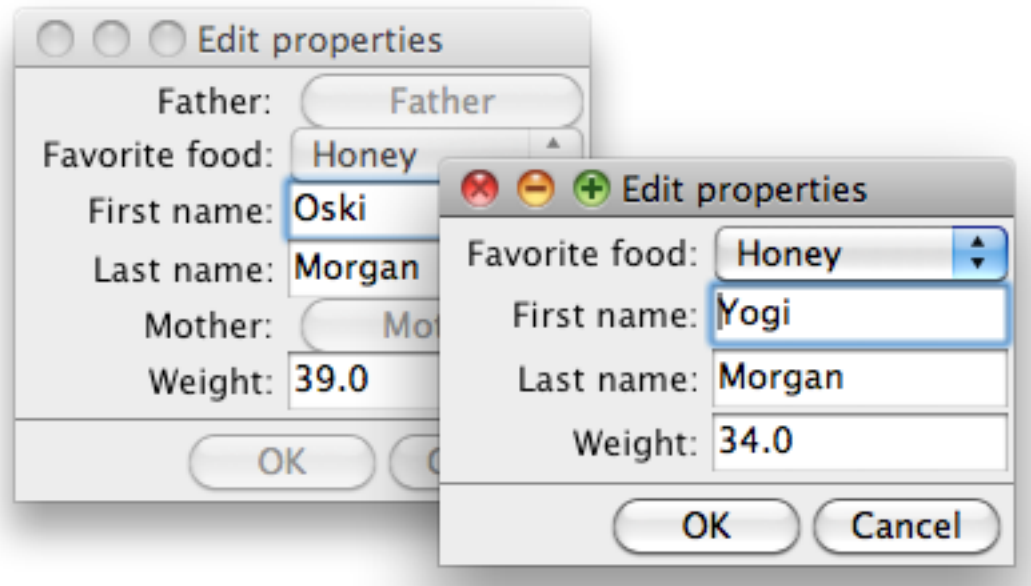


Delegation: PrototypesFrom, DelegatesTo

file: traits2.py

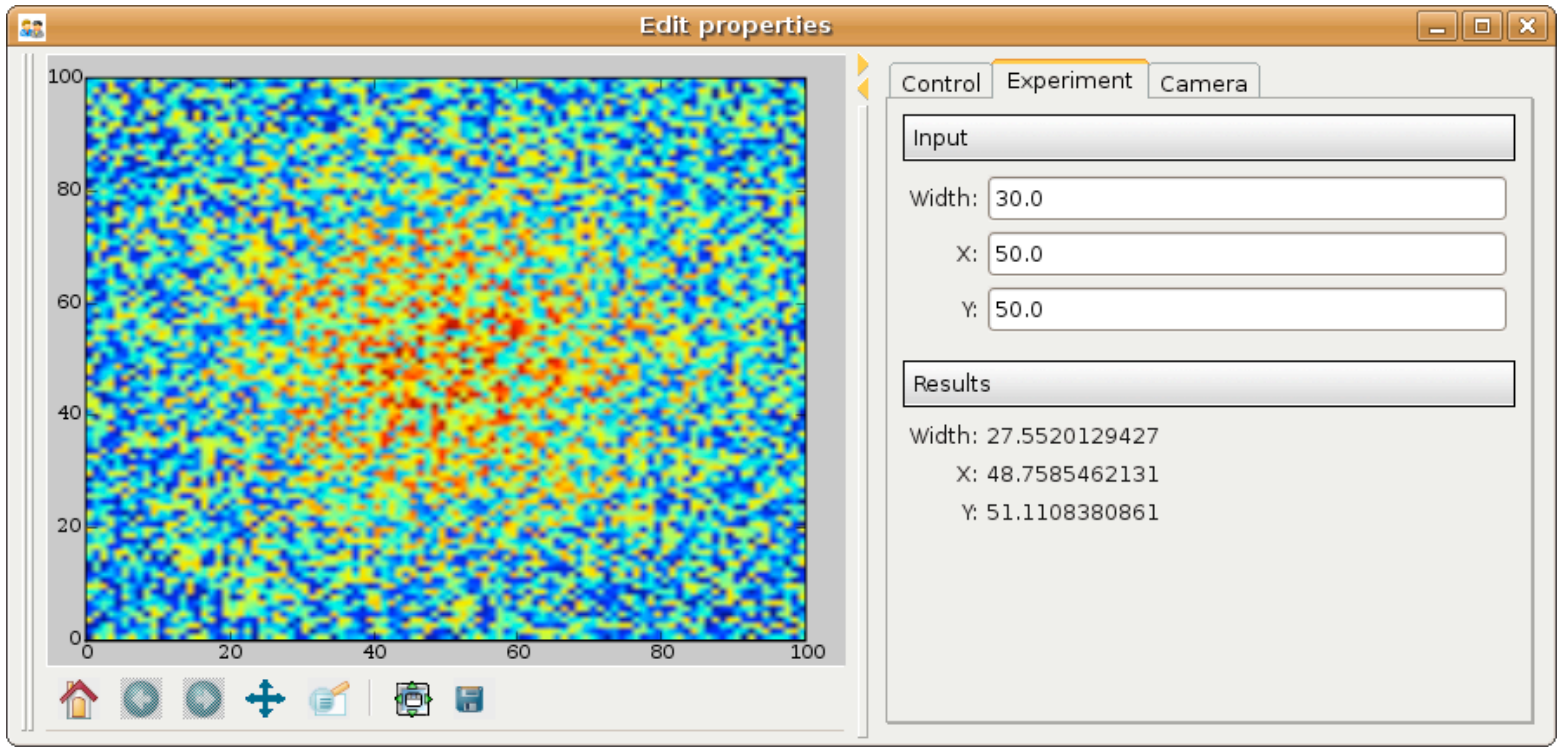
```
class Bear(HasTraits):
    first_name = Str
    weight = Float
    favorite_food = Enum("Honey","Turkey","PaloAltian",default="Honey")

class Parent(Bear):
    last_name = Str
    def _last_name_changed(self, old, new):
            print "Parent's last name changed to %s (was = '%s')." % (new,old)
            sys.stdout.flush()

class Cub(Bear):
    last_name  = DelegatesTo('father')
    father = Instance(Parent)
    mother = Instance(Parent)

yogi  = Parent(first_name="Yogi",weight=34.0,last_name="Morgan")
sally = Parent(first_name="Sally",weight=30.0,last_name="Klein")
oski = Cub(first_name="Oski",weight=39.0,father=yogi,mother=sally)
```

command-line demo



matplotlib + Traits GUIs

An HasTraits model is happy to take an instance of an mpl Figure, but Figure itself does not "HasTraits" so it does not have a native "editor" that knows how to display itself



http://code.enthought.com/projects/traits/docs/html/tutorials/traits_ui_scientific_app.html

matplotlib + Traits GUIs

An HasTraits model is happy to take an instance of an mpl Figure, but Figure itself does not "HasTraits" so it does not have a native "editor" that knows how to display itself

```
from mpl_figure_editor import MPLFigureEditor

class Test(HasTraits):

  figure = Instance(Figure, ())

  view = View(Item('figure', editor=MPLFigureEditor(),
                                show_label=False),
                        width=400,
                        height=300,
                        resizable=True)
  def __init__(self):
     super(Test, self).__init__()
     axes = self.figure.add_subplot(111)
     t = linspace(0, 2*pi, 200)
     axes.plot(sin(t)*(1+0.5*cos(11*t)), cos(t)*(1+0.5*cos(11*t)))
```

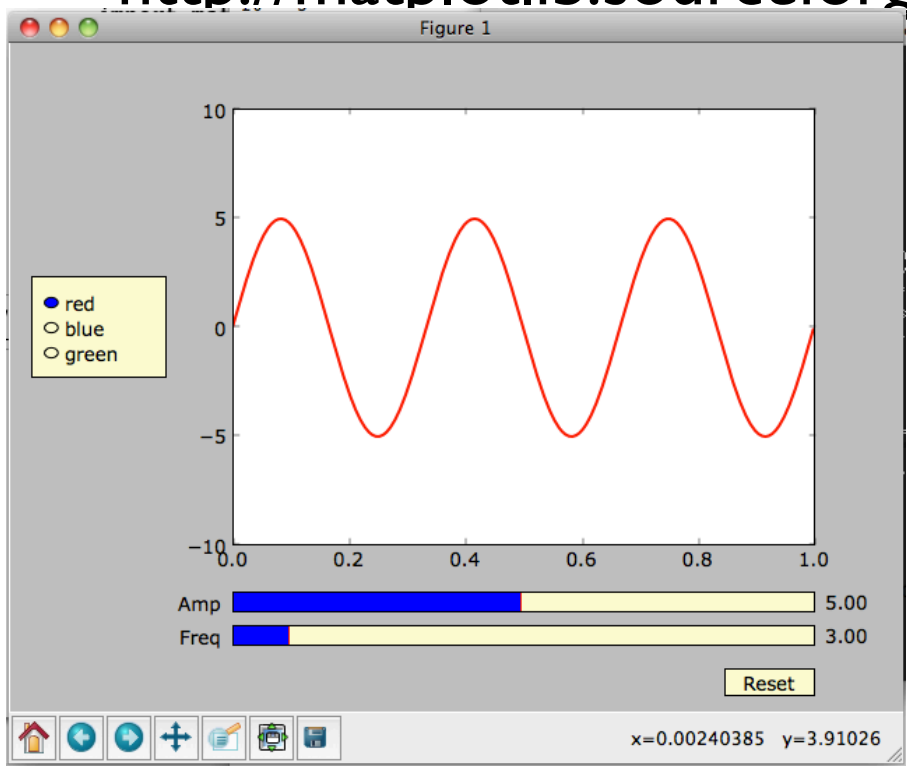
http://code.enthought.com/projects/traits/docs/html/tutorials/traits_ui_scientific_app.html

matplotlib has its own widgets

```
import matplotlib.widgets as wg
```

```
In [10]: wg.
wg.Button                       wg.RadioButtons
wg.CheckButtons                 wg.Rectangle
wg.Circle                       wg.RectangleSelector
wg.Cursor                       wg.Slider
wg.HorizontalSpanSelector       wg.SpanSelector
wg.Lasso                        wg.SubplotTool
wg.Line2D                       wg.Widget
wg.LockDraw                     wg.__builtins__
wg.MultiCursor                  wg.__class__
```

http://matplotlib.sourceforge.net/examples/widgets/



Slider, Checkbox, Button

file: slider_demo.py

Homework 6

Using Traits and TraitsUI create a GUI application that consists of a search box, a text display, an image display, and a series of buttons. The application will accept search strings in the search box and then run an internet image search. The first returned image from the search will be downloaded, the image url displayed in the text display field, and the actual image displayed in the image display field.

The buttons will provide the user interface to run the image search as well as perform manipulations on the image currently stored in the display (for example, blurring or rotation). Provide at least three unique (and interesting) image manipulation functions.

Tips: Feel free to use an image search (or, include support for multiple different searches). Yahoo has a convenient search module. There is also freedom in how you display the image, but matplotlib is recommended.

Yahoo! Image Search

Input

Query string: Monty Python, humour

Run query

Image URL

http://www.ombresetlumieres.com/wp-content/uploads/2008/04/monty_python.jpg

Image Display

Image Manipulations Options

Refresh | Blur | Sharpen | Smooth | Edge | Contrast | Color | Decolor | Brighten | Darken

OK