PWC Python Course - GUI programming with *Tkinter*

Erik Spence

SciNet HPC Consortium

11 December 2014

An introduction to *Tkinter*

The purpose of this segment of the class is to introduce you to the basics of GUI programming in Python, using *Tkinter* . There are several GUI interfaces available in Python:

- *Tkinter* is the Python interface to the *Tk* GUI toolkit.
- *wxPython* is an open-source Python interface for wxWindows.
- *JPython* is a Python port for Java which gives Python scripts access to Java class libraries.

Many others are also available. We will use *Tkinter*, due to the fact that it is the *de facto* standard Python GUI library.

What is *Tk?*

If *Tkinter* is the Python interface to the *Tk* GUI toolkit, what is *Tk*?

- *Tk* started life as Tcl extension (1991). It is now written in C.
- *Tk* is a high-level windowing toolkit. You can interface with it directly using C or other languages.
- *Tk* interfaces are also available in Python, Ruby, Perl, Tcl, and probably other languages.
- What *Tk* itself is interfacing with depends on your system:
  - Mac: *Tk* provides interfaces to the MacOS windowing system.
  - Windows: *Tk* provides interfaces to the Microsoft windowing system.
  - Other platforms: *Tk* 8.X attempts to look like the Motif window manager, but without using Motif libraries. Prior to that it interfaced with the X window system.

Let it suffice to say that *Tk* provides a high-level means of accessing your system's windowing infrastructure.

Running code in today's class

All the code for this morning's class will be written as stand-alone scripts. As such we won't be using the interactive prompt. Those using Eclipse should be able to run the code from the editor or the toolbar.

If you're using the supplied Python code, and using Eclipse, be sure to import it into your current Project so that you can run it.

Do NOT use IDLE, or any other standard graphical Python interface. Some of these use *Tk* as a back end, and today's code may break or confuse the interface.

Our first *Tkinter* program

- The *Tkinter* module is needed to build *Tkinter* widgets.
- First thing generated is the parent window, upon which everything else will be placed.
- The Label widget is generated.
- Arrange the Label using the pack command.
- The 'mainloop' command is used to launch the window, and start the event loop.
- The window can be moved, resized, and closed.

```
# firstTkinter.py
from Tkinter import Tk, Label

# Create the window.
top = Tk()

# Create a Label.
l = Label(top, text = "Hello World")

# Arrange the Label.
l.pack()

# Run the parent, and its children.
top.mainloop()
```

The window has the 'look' of whatever system you are running.

Event-driven programming

The previous example was trivial, but it illustrates steps which are seen in most *Tkinter* programs. Some notes:

- The mainloop method puts the label on the window, the window on the screen and enters the program into a *Tkinter* wait state.
- In the wait state, the code waits for user-generated activity, called 'events'.
- This is called *event-driven* (also called asynchronous) programming.
- The programs are essentially a set of event handlers that share information rather than a single linear control flow.

This style of programming is notably different from what most of us are accustomed.

Our first *Tkinter* program, continued

We can tweak the appearance of our main window:

- Use the 'title' option to change the title of the window.
- The 'minsize/maxsize' arguments set the minimum/maximum size of the window.
- The 'configure' argument can be used to set a variety of different window features, such as the background colour.

```
# firstTkinter2.py
from Tkinter import Tk, Label

top = Tk()

l = Label(top, "Hello World")
l.pack()

# Give the window a title.
top.title("My App")

# Change the minimum size.
top.minsize(400, 400)

# Change the background colour.
top.configure(bg = "green")

# Run the widget.
top.mainloop()
```

Our second *Tkinter* program

- The *Tkinter* Button command creates a button.
- The first argument is the parent window.
- The 'pack' command makes the widget visible, and tells the parent to resize to fit the children.
- When the button is pushed, the callback function 'hello_callback' is called.
- If the upper-right-corner 'X' is not visible the window is too small. Resize the window.

```
# secondTkinter.py
from Tkinter import Label, Button, Tk

# The 'callback function'.  Invoked
# when the button is pressed.
def hello_callback(): print "Hello"

top = Tk()

# Make a Label.
l = Label(top, text = "My Button:")
l.pack()

# Make a button.
b = Button(top, text = "Hello",
  command = hello_callback)
b.pack()

top.mainloop()
```

A better second *Tkinter* program

Widgets are usually created as objects, so let's recast our example as such.

```
# secondTkinter2.py
import Tkinter
from MyApp import MyApp

top = Tkinter.Tk()

# Note that the constructor takes the
# parent window as an argument.
app = MyApp(top)

top.mainloop()
```

```
# MyApp.py
from Tkinter import Label, Button

class MyApp:

  def __init__(self, master):
    self.l = Label(master,
      text = "My Button:")
    self.l.pack()

    self.b = Button(master,
      text = "Hello",
      command = self.hello)
    self.b.pack()

  # Function called when the button
  # is pressed.
  def hello(self): print "Hello"
```

An even better second *Tkinter* program

Generally speaking, objects should be invokable on their own.

```
# MyApp2.py
from Tkinter import Label, Button,
  Frame

# Extend the Frame class, to inherit
# the mainloop function.
class MyApp(Frame):

  def __init__(self, master = None):

    # Construct the Frame object.
    Frame.__init__(self, master)
    self.pack()
```

```
# MyApp2.py, continued

    self.l = Label(self,
      text = "My Button:")
    self.l.pack()

    self.b = Button(self,
      text = "Hello",
      command = self.hello)
    self.b.pack()

  # Function called when the button
  # is pressed.
  def hello(self): print "Hello"

# Allow the class to run stand-alone.
if __name__ == "__main__":
  MyApp().mainloop()
```

Callback functions

'Callback functions' are invoked by widgets due to an 'event', such as the pressing of a button. These functions need to be handled carefully:

- Notice that we used "command = self.hello" rather than "command = self.hello()" in the code for the Button in the last example.
- If you do "command = func()" in the widget declaration, func() will be run upon the widget creation, not when it is needed.
- But without the brackets there is no way to pass arguments to the function! If arguments are needed you must use lambda, or another indirection layer.
- (Global variables may also work, but are not recommended.)
- Functions invoked using lambda are only called at runtime, not when the widget is created.

Many different *Tkinter* widgets are available

The list of widgets which can be added to a *Tkinter* window is extensive:

- Buttons, Checkbuttons, Radiobuttons, Menubuttons
- Canvas (for drawing shapes)
- Entry (for text field entries)
- Message (for displaying text messages to the user)
- Labels (text captions, images)
- Frames (a container for other widgets)
- Scale, Scrollbar
- Text (for displaying and editting text)
- and others...

Tkinter control variables

More often then not, we want to tie the value of variables to the specific states of widgets, or to specific events. This is called *tracing*.

- Native Python variables don't track events as they occur.
- However *Tkinter* contains wrapper objects for variables which change value with changing events. These are called *Tkinter* variables.
- Because they are objects, *Tkinter* variables are invoked using a constructor: var = IntVar().

These variables have a number of important functions:

- Checkbuttons use a control variable to hold the status of the button.
- Radiobuttons use a single control variable to indicate which button has been set.
- Control variables hold text strings for several different widgets (Entry, Label, Text).

How to use *Tkinter* control variables

There are four types of control variables: StringVar (string), IntVar (integers), DoubleVar (floats), BooleanVar (booleans). Each variable has a default value. How do these variables tend to manifest themselves?

- Button: set its 'textvariable' to a StringVar. When the StringVar is changed the Button's text will change.
- Checkbutton: set the 'variable' option to an IntVar. Note that you can also use other values for a Checkbutton (string, boolean).
- Entry: set the 'textvariable' option to a StringVar.
- Radiobutton: the 'variable' option must be set to either an IntVar or StringVar.
- Scale: set the 'variable' option to any control variable type. Then set the 'from_' and 'to' values to set the range.

Tkinter control variables example

```
# MyCheckbutton.py
from Tkinter import IntVar, BOTH
  Checkbutton, Frame

class MyCheckbutton(Frame):
  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack(expand = True,
      fill = BOTH)
    self.master.title("")
    self.master.minsize(200, 100)

    # Object variables.
    self.var = IntVar()

    # Create a checkbutton.
    cb = Checkbutton(self, text =
      "Show title", variable =
      self.var, command = self.click)
    cb.place(x = 50, y = 50)
```

```
# MyCheckbutton.py, continued
  def click(self):
    if (self.var.get() == 1):
      self.master.title("Checkbutton")
    else: self.master.title("")

if __name__ == "__main__":
  MyCheckbutton().mainloop()
```

- The IntVar object tracks the checkbox value (0 or 1).
- .get()/.set('x') returns/sets the value of the control variable.
- The 'place' function locates the checkbox in the window, from the upper-right corner.

Exercise 1

Create an application which

- Accepts a numeric entry from the user, using the Entry widget.
- The Entry widget has a Label widget beside it, which says "lbs".
- Has a 'Calculate' button. When the 'Calculate' button is pressed:
  - The application calculates the number of kilograms, assuming that the value given in the Entry widget is numeric, and is in pounds (1 pound = 0.453592 kilograms).
  - Prints the value to the command line.

Hint: create a StringVar() for the entry widget. When the button is pressed grab the value and go.

Exercise one

```
# lbs2kgs.py
from Tkinter import *
class lbs2kgs(Frame):
  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack()

    self.lbs = StringVar()
    lbs_entry = Entry(self, width = 7,
      textvariable = self.lbs)
    lbs_entry.pack(side = LEFT)

    Label(self, text = "lbs").pack(
      side = LEFT)
    Button(self, text = "Calculate",
      command = self.calc).pack(
      side = LEFT)
    lbs_entry.focus()
    for c in self.master.winfo_children():
      c.pack_configure(padx = 5, pady = 5)
```

```
# lbs2kgs.py, continued
  def calc(self):
    try:
      value = float(self.lbs.get())
      print "The number of
        kgs is", 0.453592 * value
    except ValueError: pass

if __name__ == "__main__":
  lbs2kgs().mainloop()
```

- .focus() moves the window focus: type without clicking
- .winfo_children() returns a list of all child widgets.
- .pack_configure() adjusts the packing of the widgets.

Widgets and assignments

Did you notice the strange lines of code in the previous example? What's wrong here?

```
# lbs2kgs.py
from Tkinter import *
class lbs2kgs(Frame):
 def __init__(self, master = None):
  Frame.__init__(self, master)
  self.pack()

  self.lbs = StringVar()
  lbs_entry = Entry(master, width = 7,
   textvariable = self.lbs)
  lbs_entry.pack(side = LEFT)

  Label(self, text = "lbs").pack(
     side = LEFT)
  Button(self, text = "Calculate",
     command = self.calc).pack(
     side = LEFT)

  lbs_entry.focus()
  for c in master.winfo_children():
   c.pack_configure(padx = 5, pady = 5)
```

Widgets and assignments

Did you notice the strange lines of code in the previous example? What's wrong here?

Because these widgets were not assigned to a name, they should be garbage collected as soon as the pack() command is finished.

```
# lbs2kgs.py
from Tkinter import *
class lbs2kgs(Frame):
  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack()

    self.lbs = StringVar()
    lbs_entry = Entry(master, width = 7,
      textvariable = self.lbs)
    lbs_entry.pack(side = LEFT)

    Label(self, text = "lbs").pack(
          side = LEFT)
    Button(self, text = "Calculate",
          command = self.calc).pack(
          side = LEFT)

    lbs_entry.focus()
    for c in master.winfo_children():
      c.pack_configure(padx = 5, pady = 5)
```

Widgets and assignments

Did you notice the strange lines of code in the previous example? What's wrong here?

Because these widgets were not assigned to a name, they should be garbage collected as soon as the pack() command is finished.

Tkinter emits *Tk* calls when objects are constructed. *Tkinter* internally cross-links widget objects into a long-lived tree used to build the display. As such the widgets are retained, even if not in the code itself.

```
# lbs2kgs.py
from Tkinter import *
class lbs2kgs(Frame):
  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack()

    self.lbs = StringVar()
    lbs_entry = Entry(master, width = 7,
      textvariable = self.lbs)
    lbs_entry.pack(side = LEFT)

    Label(self, text = "lbs").pack(
          side = LEFT)
    Button(self, text = "Calculate",
          command = self.calc).pack(
          side = LEFT)

    lbs_entry.focus()
    for c in master.winfo_children():
      c.pack_configure(padx = 5, pady = 5)
```

Using images with *Tkinter*

There are several packages available for using images in *Tkinter* :

- The PhotoImage class can read GIF and PGM/PPM images, as well as base64-encoded GIF files from strings.
- The Python Imaging Library (PIL) contains classes that can handle over 30 file formats. This package is no longer being maintained, and has been succeeded by the Pillow package.
- Important: you must keep a reference to your PhotoImage object (of either the PhotoImage of PIL class). This is in direct contrast to what was shown on the last slide.
- If you do not keep a reference the object will be garbage collected even if the widget is still operating!

Using images within widgets

To embed an image in a widget, there are several steps:

- First open the image file in question.
- Then convert it to a *Tkinter* -compatible image object.
- Then embed it into a widget.
- Again: you must keep a reference to your PhotoImage object, otherwise the object will be garbage collected.

```
# MyImage.py
from Tkinter import Label, Frame
from PIL import Image, ImageTk

class MyImage(Frame):
  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack()

    img = Image.open("tatras.jpg")
    pic = ImageTk.PhotoImage(img)
    label = Label(self, image = pic)

    # Keep a reference!
    # (or don't and see what happens)
    label.image = pic
    label.pack()

if __name__ == "__main__":
  MyImage().mainloop()
```

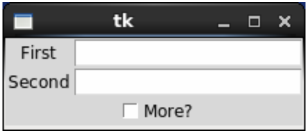
Arranging your widgets

There are three Geometry Managers in *Tkinter* for arranging widgets: 'grid', 'pack' and 'place'. We've already used the latter two. Do not try to mix grid and pack in the same window ('container').

- pack
  - Lays widgets out along the sides of a box.
  - Works best when everything is in one row or one column.
  - Can be tricky to make more-complicated layouts until you understand the packing algorithm, which we won't cover here. It's best not to try.
- grid
  - Lays out widgets in a grid (along row and column boundaries)
  - Good for creating tables and other structured types of layouts.
- place
  - Can place a widget at an absolute position, a given x and y
  - Can place a widget relatively, such as at the edge of another widget.

Grid

The grid Geometry Manager puts widgets in a 2-D table. The 'container' widget is split into rows and columns, and each cell in the table can hold a widget.



The column defaults to 0 if not specified. The row defaults to the first unused row in the grid.

```
# MyGrid.py
from Tkinter import Label, Entry,
  Checkbutton, Frame

class MyGrid(Frame):
  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack()

    Label(self, text = "First").grid()
    Label(self, text = "Second").grid()

    Entry(self).grid(row = 0, column = 1)
    Entry(self).grid(row = 1, column = 1)

    Checkbutton(self, text = "More?").grid(
      columnspan = 2)

if __name__ == "__main__":
  MyGrid().mainloop()
```

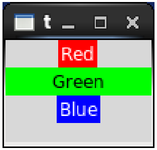
Grid keywords

Grid takes a number of useful keywords:

- column/row: the column or row into which the widget will be placed.
  - If no column is specified, column = 0 is used.
  - If no row is specified, the next unused row is used.
  - If you put two widgets in the same cell, both will be visible, with potentially odd results.
- columnspan/rowspan, number of columns/rows to span, to the right/down.
- sticky, defines how to expand the widget if the cell is larger than the widget.
  - Can be any combination of S, N, E, W, NE, NW, SW, SE.
  - Default is to be centred.
- padx/pady, optional horizontal/vertical padding to place around the widget, within the cell.

Pack, vertical example

The pack Geometry Manager lays widgets on the side of a box, in this example on the top side. Pack can allow the widget to change size if the window is resized.

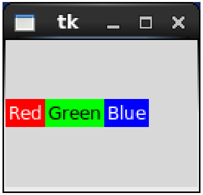


Using 'fill = X' will cause the widget to fill in the horizontal direction if the window is resized.

```
# MyPack1.py
from Tkinter import Label, X, Frame, BOTH

class MyPack1(Frame):
  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack(expand = True, fill = BOTH)

    self.master.minsize(100, 70)

    Label(self, text = "Red", bg = "red",
      fg = "white").pack()
    Label(self, text = "Green",
      bg = "green").pack(fill = X)
    Label(self, text = "Blue", bg = "blue",
      fg = "white").pack()

if __name__ == "__main__":
  MyPack1().mainloop()
```

Pack, horizontal example

Use the 'side' argument to indicate which side of the box pack should pack against.



Resizing the window will not cause the widgets to grow in this case, the way that 'fill' does, though they will stay centered on the left side.

```
# MyPack2.py
from Tkinter import Label, Frame,
  BOTH, LEFT

class MyPack2(Frame):
  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack(expand = True, fill = BOTH)

    self.master.minsize(130, 100)

    Label(self, text = "Red", bg = "red",
      fg = "white").pack(side = LEFT)
    Label(self, text = "Green",
      bg = "green").pack(side = LEFT)
    Label(self, text = "Blue", bg = "blue",
      fg = "white").pack(side = LEFT)

if __name__ == "__main__":
  MyPack2().mainloop()
```

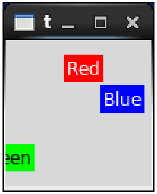
Pack keywords

Pack takes several useful keywords:

- side, which side to pack against. Options are LEFT, TOP (default), RIGHT, BOTTOM. You can mix them within the same parent widget, but you'll likely get unexpected results.
- fill, specifies whether the widget should occupy all the space provided to it by the parent widget. Options are NONE (default), X (horizontal fill), Y (vertical fill), or BOTH.
- expand, specifies whether the widget should be expanded to fill extra space inside the parent widget. Options are False (default) and True.

Place

The place Geometry Manager is the simplest of the three to use. It places the widget either in absolute or relative terms. However, it is a pain to use for general placement of widgets, though can be useful in special cases.



```
# MyPlace.py
from Tkinter import Label, NW, E, CENTER,
 Frame, BOTH
class MyPlace(Frame):
  def __init__(self, master = None, **options):
    Frame.__init__(self, master, **options)
    self.pack(expand = True, fill = BOTH)

    self.config(width = 100, height = 100)
    Label(self, text = "Red", bg = "red",
      fg = "white").place(anchor = NW,
      relx = 0.4, y = 10)
    Label(self, text = "Green",
      bg = "green").place(anchor = E,
      relx = 0.2, rely = 0.8)
    Label(self, text = "Blue", bg = "blue",
      fg = "white").place(anchor = CENTER,
      x = 80, rely = 0.4)

if __name__ == "__main__": MyPlace().mainloop()
```

Place keywords

Place takes a number of useful keywords:

- relx/rely: between 0 and 1, the position of the widget, in the x/y direction, relative to the parent window in which its embedded.
- x/y: in pixels, the absolute position of the widget, in the window in which the widget is embedded.
- If both relx and x are specified then the relative position is calculated first, and the absolute position is added after.
- anchor: the point on the widget that you are actually positioning. Options are the eight points of the compass (E, S, NW, ...) and CENTER.

Centering your widget

The default location of your widget depends on the Window Manager. Generally it's in the upper-left corner.

- Use winfo_screenwidth() to get the window width. Similarly for height.
- The geometry command is used to set the location of the window's upper-left corner.

```
# MyCentre.py
from Tkinter import Frame

class MyCentre(Frame):
  def __init__(self, master = None):

    Frame.__init__(self, master)
    self.pack()
    # Width and height of the window.
    w = 200; h = 50

    # Upper-left corner of the window.
    x = (self.master.winfo_screenwidth() - w) / 2
    y = (self.master.winfo_screenheight() - h) / 2

    # Set the height and location.
    master.geometry("%dx%d+%d+%d" % (w, h, x, y))

if __name__ == "__main__": MyCentre().mainloop()
```

Bindings

Any user action (keyboard or mouse), is called an 'event'. Events can be captured by the application, and specific actions taken. This is accomplished using the 'bind' function.

Event actions can be bound to any widget, not just the main window.

```
# MyBindings.py
from Tkinter import Frame
from tkMessageBox import showerror,
  askyesno

class MyBindings(Frame):
  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack()
```

```
# MyBindings.py, continued

    self.master.minsize(100, 100)
    self.master.bind("a",
      self.a_callback)
    self.master.bind("<Button-1>",
      self.b_callback)

  # Called when the 'a' is pressed.
  def a_callback(self, event):
    if not askyesno("A query",
      "Did you press the 'a' button?"):
      showerror("I am aghast!", "Liar!")

  def b_callback(self, event):
    print "clicked", event.x, event.y

if __name__ == "__main__":
  MyBindings().mainloop()
```

Event formats

A partial list of possible bindings:

- "⟨Button-1⟩": a mouse button is pressed over the widget. Button 2 is the middle, 3 is the right. "⟨Button-1⟩" and "⟨1⟩" are synonyms.
- "⟨Enter⟩"/"⟨Leave⟩": the mouse pointer entered/left the widget.
- "⟨Return⟩": the user pressed the Enter key.
- "⟨key⟩": the user pressed the any key.
- "⟨Control-p⟩": the user pressed Ctrl-p.

The event object has a number of standard attributes:

- x, y: current mouse position, in pixels.
- char: the character code, as a string.
- type: the event type.
- and others...

Bindings exercise

Create an application which

- Has a label at the top of the frame which says "Are you an idiot?"
- Has a button below the label which contains the text "No".
- Moves to a random location on the screen every time you try to press the "No" button.
- Don't bother making this one a class, since that doesn't really make sense.

Hints:

- Use the "Enter" binding to bind the mouse pointer.
- When the mouse enters the "No" button, move the window (use the random.random() function to get a random number).
- master.winfo_screenwidth() and height might be useful.

Bindings exercise

```
# idiot.py
from Tkinter import Tk, Label, Button
from random import random

def moveme(event):
  x = xsize * random()
  y = ysize * random()

  # Move the window.
  master.geometry( "%dx%d+%d+%d" %
    (w, h, x, y))

master = Tk()
xsize = master.winfo_screenwidth()
ysize = master.winfo_screenheight()
w = 200
h = 50
```

```
# idiot.py, continued

master.minsize(w, h)
master.title("Let me check")

Label(master,
  text = "Are you an idiot?").pack()
b = Button(master, text = "No!")
b.pack()
b.bind("<Enter>", moveme)

master.mainloop()
```

Pop-up windows

Pop-up windows are fun. Every app needs a pop-up window. The easiest package to use for pop-up windows is tkMessageBox.

Like the main window, the pop-up windows have a default look which depends upon the system running the code.

```python
# MyPopup.py
from Tkinter import Button, Frame
from tkMessageBox import showinfo

class MyPopup(Frame):

  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack()

    Button(self, text = "Pop-up!",
      command = self.popup).pack()

  def popup(self):
    showinfo("My Pop-Up", "Hello")

if __name__ == "__main__":
  MyPopup().mainloop()
```

Many pre-made pop-up windows are available

If you're going to use pop-up windows, the defaults that come with tkMessageBox should be sufficient:

- single-button pop-ups:
  - "Ok": showinfo, showwarning, showerror
- double-button pop-ups:
  - "Yes-No": askquestion, returns the strings "yes", "no"
  - "Yes-No": askyesno, returns True/False
  - "Ok-Cancel": askokcancel, returns True/False
  - "Retry-Cancel": askretrycancel, returns True/False

These functions all have the same syntax:
tkMessageBox.function(title, message [, options]).

Toplevel windows

Sometimes the pre-made pop-up windows don't meet your needs, since these are canned pop-ups. For these cases one uses Toplevel windows.

Toplevel windows behave like main windows, but are actually children of whichever window spawned them.

```
# MyToplevel.py
from Tkinter import Button, Frame,
  Toplevel
class MyToplevel(Frame):

  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack()

    Button(self, text = "A new window!",
      command = self.new_window).pack()

  # A new functional window.
  def new_window(self):
    top = Toplevel(master = self)
    Button(top, text = "Quit",
      command = top.quit).pack()

if __name__ == "__main__":
  MyToplevel().mainloop()
```

File manager windows

```
# MyFile.py
from Tkinter import Button, Frame
from tkFileDialog import
  askopenfilename

class MyFile(Frame):
  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack()

    Button(self, text = "Get a file!",
      command = self.getfile).pack()

  def getfile(self):
    filename = askopenfilename(
      parent = self,
      title = "Please select a file")

    if (len(filename) > 0):
      print "You chose %s" % filename
```

```
# MyFile.py, continued

if __name__ == "__main__":
  MyFile().mainloop()
```

Pre-made file manager dialog boxes are available through the tkFileDialog module.

Quitting cleanly

We want our program to close cleanly. But we must be careful how we do so:

- All *Tkinter* widgets come with the 'quit' function built in. This will close the entire *Tkinter* program, which may not be what you want.
- Alternatively, you can use the 'destroy' function, which will only close the particular widget which you are referencing.

```
# badquit.py
from Tkinter import Tk, Button

# behaviour 1
t1 = Tk()
t1.b = Button(t1, text = "push me",
  command = lambda:t1.b.destroy())
t1.b.pack()
t1.mainloop()

# behaviour 2
t2 = Tk()
t2.b = Button(t2, text = "me too!",
  command = lambda:t2.b.quit())
t2.b.pack()
t2.mainloop()
```

Quitting cleanly

The lambda command needs to be used in this case because the callback command which is being referenced is self-referential.

```
# badquit.py
from Tkinter import Tk, Button

# behaviour 1
t1 = Tk()
t1.b = Button(t1, text = "push me",
  command = lambda:t1.b.destroy())
t1.b.pack()
t1.mainloop()

# behaviour 2
t2 = Tk()
t2.b = Button(t2, text = "me too!",
  command = lambda:t2.b.quit())
t2.b.pack()
t2.mainloop()
```

A 'Quit' button class

Let's create a class that we can use in future widgets.

```
# MyQuitter.py
from Tkinter import Button, LEFT, YES,
  BOTH, Frame
from tkMessageBox import askokcancel

# Extends the Frame class.
class MyQuitter(Frame):
  def __init__(self, master = None):

    Frame.__init__(self, master)
    self.pack()

    b = Button(self, text = "Quit",
      command = self.myquit)
    b.pack(side = LEFT, expand = YES,
      fill = BOTH)
```

```
# MyQuitter.py, continued

  def myquit(self):
    if askokcancel("Quit",
      "Do you really wish to quit?"):
      Frame.quit(self)
```

- The askokcancel function returns True if 'OK' is pressed.
- The 'LEFT' argument indicates the position of the button.
- The second 'pack' is invoked after the Button is created, and so can go in the same line.

Capturing destroy events

Tkinter lets you manipulate 'protocol handlers'

- These handle the interaction between the application and the window manager
- The most-used way to do this is re-assigning the WM_DELETE_WINDOW protocol (invoked by pressing the 'X' in the upper-right corner).

```
# MyCapture.py
from Tkinter import Frame
from MyQuitter import MyQuitter

class MyCapture(Frame):
  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack()

    q = MyQuitter(self)
    q.pack()

    self.master.protocol(
      "WM_DELETE_WINDOW", q.myquit)

if __name__ == "__main__":
  MyCapture().mainloop()
```

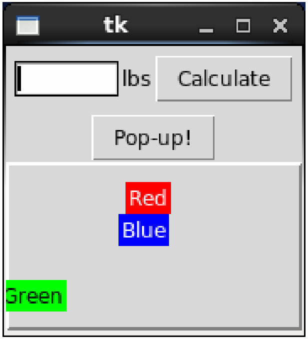
Notes on code re-usability

Modular programming is always to be encouraged, and GUI programming is no exception.

- Throughout this class we have crafted our examples such that they are classes that can be embedded in other widgets.
- Custom GUI classes can be written as extensions of existing classes, the most common choice being Frame.
- Using widgets that are extensions of existing classes allows uniform and consistent modification of the look-and-feel of your widgets.

Code re-usability example

Because we set up our previous code examples as classes, we can just drop them into other widgets.



```python
# ABunchOfWidgets.py
from Tkinter import Frame, RAISED
from lbs2kgs import lbs2kgs
from MyPopup import MyPopup
from MyPlace import MyPlace

class ABunchOfWidgets(Frame):
  def __init__(self, master = None):

    Frame.__init__(self, master)
    self.pack()

    lbs2kgs(self).pack()
    MyPopup(self).pack()
    MyPlace(self, borderwidth = 2,
      relief = RAISED).pack()

if __name__ == "__main__":
  ABunchOfWidgets().mainloop()
```

Drop-down menus

```
# MyMenus.py
from Tkinter import Menu, Frame
class MyMenus(Frame):

  def __init__(self, master = None):
    Frame.__init__(self, master)
    self.pack()
    self.master.minsize(100,100)

    # Create a menu instance,
    # the menu does not need packing.
    self.mbar = Menu(self)

    # Attach to the root window.
    self.master.config(menu =
      self.mbar)

    # Create a new menu instance...
    self.filemenu = Menu(self.mbar,
      tearoff = 0)
```

```
# MyMenus.py, continued

    # ...and stick into the menubar.
    self.mbar.add_cascade(label =
      "File", menu = self.filemenu)

    # Add entries to filemenu.
    self.filemenu.add_command(label =
      "New", command = self.new_call)
    self.filemenu.add_command(label =
      "Open", command = self.o_call)

  # The callback functions.
  def new_call(self): print "New_call"

  def o_call(self): print "o_call"

if __name__ == "__main__":
  MyMenus().mainloop()
```

Threads and GUIs

In general, we recommend against using Python's threading capabilities:

- Python's Global Interpreter Lock prevents more than one thread from running at a given time.
- Consequently there is no increase in computational performance.

However, these concerns do not apply when dealing with GUIs, since computational performance is not usually at issue. There are some details worth noting:

- The main event loop runs in a single thread.
- If a callback function is invoked, it runs in the same thread.
- If the function takes a long time to complete, you will notice:
  - the windows will not update (resize, redraw, minimize)
  - the windows will not respond to new events.

Threads can be useful to fix this problem.

Illustrating the problem

```
# MySummer.py
from Tkinter import Tk, Button, Label

def button_press():
  total = 0
  for i in xrange(100000000):
    total += i
  label.config(text = str(total))

master = Tk()
Button(master, text = "Add it up",
  command = button_press).pack()
label = Label(master)
label.pack()
master.mainloop()
```

Here we illustrate the problem that GUIs can have. Perform the following steps:

- Run the GUI.
- Press the button.
- While the calculation is being performed, resize the window.

What happens?

Fixing the problem, using threads

```
# MySummer.threaded.py
from Tkinter import Tk, Button, Label
import threading

def button_press():

  # Create a function for the thread.
  def callback():
    total = 0
    for i in xrange(100000000):
      total += i
    label.config(text = str(total))

  # Launch the thread.
  threading.Thread(
    target = callback).start()
```

Because the control of the calculation is in a separate thread, control returns to the event loop.

```
# MySummer.threaded.py, continued

master = Tk()
Button(master, text = "Add it up",
  command = button_press).pack()
label = Label(master)
label.pack()
master.mainloop()
```

If you get an 'infinite loop' error then your Tcl was not compiled with threading support. If so, try using the 'mtTkinter' package.

Making your GUIs launchable

By default if you click on your GUI application, it will bring up a Python console as well as the GUI you're after. It would be nice if that weren't necessary.

To do this, the code must be converted into a stand-alone executable. This requires a package that will bundle all the needed code and libraries into a single file. There are several such packages available:

- py2exe
- pyinstaller
- cx_freeze

We will use py2exe, as it is the most commonly used, at least for Windows.

Using py2exe

To use py2exe to create an executable, perform the following steps:

1. Create a setup.py file in the directory which contains the code.

```
from distutils.core import setup
import py2exe
setup(windows = ['firstTkinter.py'])
```

2. Open the 'cmd' terminal. Change directory to the location of the code.
3. Run the command 'python.exe setup.py py2exe'.

```
C:\Users\Erik>
C:\Users\Erik> cd Desktop\tkinter_code
C:\Users\Erik\tkinter_code> C:\Python27\python.exe setup.py py2exe
```

The executable is in the 'dist' directory.

Using py2exe, continued

Some notes about the py2exe results:

- Two directories were built, 'build' and 'dist'. The build directory may be deleted, it is no longer needed.
- Like most Windows applications, the executable must stay in the dist directory to run correctly. If you want an icon you can click on your desktop, create a shortcut.
- In theory you should be able to zip up the dist directory and distribute it to your friends. Once unzipped the executable should work out of box. This assumes, of course, that the Windows dll files that are needed are on the system in question.

Enough to get started

Using the material here, you should have enough to get started. More information can be found on the web. A few good websites are:

- http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html
- http://www.effbot.org/tkinterbook/tkinter-index.htm
- http://cs.mcgill.ca/ hv/classes/MS/TkinterPres
- http://www.python-course.eu/python_tkinter.php

Other event-driven programming packages:

- Twisted
- asyncio
- asyncore (for handling sockets)
- asynchat (socket command/response handler)