Python\_Scripting\_Computational\_Science\_C06

Chapter 6

Introduction to GUI Programming

Python codes can quickly be altered and re-run, a property that encourages direct editing of the source code to change parameters and program behavior. This type of hardcoded changes is usually limited to the developer of the code. However, the edit-and-run strategy may soon be error-prone and introduce bugs. Most users, and even the developer, of a script will benefit from some kind of user interface. In Chapter 2 we have defined user interfaces through command-line options, which are very convenient if a script is to be called from other scripts. A stand-alone application, at least as seen from an end-user, is often simpler to apply if it is equipped with a self-explanatory graphical user interface (GUI). This chapter explains how easy it is to add a small-size GUI to Python scripts.

To construct a GUI, one needs to call up functionality in a GUI toolkit. There are many GUI toolkits available for Python programmers. The simplest one is Tkinter [10], while PyGtk, PyQt [32,35], and wxPython [31] constitute more sophisticated toolkits that are gaining increased popularity. All of these toolkits require underlying C or C++ libraries to be installed on your computer: Tkinter, PyGtk, PyQt, and wxPython require the Tk, Gtk, Qt, and wxWindows libraries, respectively. Most Python installations have Tk incorporated, a fact that makes Tkinter the default GUI toolkit. Unless you are experienced with GUI programming, I recommend to start with Tkinter, since it is easier to use than PyGtk, PyQt, and wxPython. As soon as you find yourself working a significant amount of time with GUI development in Python, it is time to reconsider the choice of toolkit and your working style.

There are two ways of creating a GUI. Either you write a Python program calling up functionality in the GUI toolkit, or you apply a graphical designer tool to compose the GUI interactively on the screen followed by automatic generation of the necessary code. The doc.html file contains links to software and tutorials for some popular designer tools: Page for Tkinter, Qt Designer for PyQt, Glade for PyGtk, and wxGlade for wxPython. Even if you end up using a designer tool, you will need some knowledge of basic GUI programming, typically the topics covered in the present chapter. When you know how to program with a GUI toolkit, you are well prepared to address some important topics for computational scientists: embedding plotting areas in GUIs (Chapter 11.1), making animated graphics (Chapter 11.3), and developing custom tools for automatically generating frequently needed GUIs (Chapter 11.4).

Chapter 6.1 provides an example-oriented first introduction to GUI programming. How to wrap GUIs around command-line oriented scripts, like simviz1.py from Chapter 2.3, is the topic of Chapter 6.2. Thereafter we list how to use the most common Tkinter and Pmw widgets in Chapter 6.3. After this introduction, I encourage you to take a look at a designer tool such as Glade, which works with PyGtk. There are links to several introductions to Glade in doc.html. A particular advantage of Glade is that the GUI code is completely separated from the application since the GUI specification is stored in an XML file. It is wise to pick up this separation principle and use it for GUI programming in general.

6.1 Scientific Hello World GUI

After some remarks in Chapter 6.1.1, regarding Tkinter programming in general, we start in Chapters 6.1.2–6.1.9, with coding a graphical version of the Scientific Hello World script from Chapter 2.1. A slight extension of this GUI may function as a graphical calculator, as shown in Chapter 6.1.10.

6.1.1 Introductory Topics

Basic Terms. GUI programming deals with graphical objects called widgets. Looking at a window in a typical GUI, the window may consist of buttons, text fields, sliders, and other graphical elements. Each button, slider, text field, etc. is referred to as a widget1. There are also “invisible” widgets, called frames, for just holding a set of smaller widgets. A full GUI is a hierarchy of widgets, with a toplevel widget representing the complete window of the GUI. The geometric arrangement of widgets in parent widgets is performed by a geometry manager.

All scripts we have met in this book so far have a single and obvious program flow. GUI applications are fundamentally different in this regard. First one builds the hierarchy of widgets and then the program enters an event loop. This loop records events, such as keyboard input or a mouse click somewhere in the GUI, and executes procedures in the widgets to respond to each event. Hence, there is no predefined program flow: the user controls the series of actions in the program at run time by performing a set of events.

Megawidgets. Simple widgets like labels and buttons are easy to create in Tkinter, but as soon as you encounter more comprehensive GUIs, several Tkinter elements must be combined to create the desired widgets. For example, user-friendly list widgets will typically be build as a composition of a basic list widget, a label widget, and two scrollbars widgets. One soon

1 In some of the literature, window and widget are used as interchangeable terms. Here we shall stick to the term widget for GUI building blocks.

ends up constructing the same composite widgets over and over again. Fortunately, there are extensions of Tkinter that offer easy-to-use, sophisticated, composite widgets, normally referred to as megawidgets. The Pmw (Python megawidgets) library, implemented in pure Python, provides a collection of very useful megawidgets that we will apply extensively in this book.

**Documentation of Python/Tkinter Programming**. Tkinter programming is documented in an excellent way through the book by Grayson [10]. This book explains advanced GUI programming through complete examples and demonstrates that Python, Tkinter, and **Pmw** can be used for highly complex professional applications. The book also contains the original Tk man pages (written for Tcl/Tk programmers) translated to the actual Python/Tkinter syntax.

The exposition in the present chapter aims at getting novice Python and GUI programmers started with Tkinter and Pmw. The information given is sufficient for equipping smaller scripts with buttons, images, text fields, and so on. Some more advanced use of Tkinter and Pmw is exemplified in Chap- ter 11, and with this information you probably have enough basic knowledge to easily navigate in more detailed and advanced documentation like [10]. If you plan to do some serious projects with Tkinter/Pmw programming, you should definitely get your hands on Grayson’s book [10].

There is a convenient online Python/Tkinter documentation, “Introduction to Tkinter”, by Fredrik Lundh, to which there is a link in the doc.html page. The Python FAQ is also a good place to look up useful Tkinter information. The Pmw module comes with very good documentation in HTML format.

**Demo Programs**. GUI programming is greatly simplified if you can find examples on working constructions that can be adapted to your own applications. Some examples of interest for the computational scientist or engineer are found in this book, but only a limited set of the available GUI features are exemplified. Hence, you may need to make use of other sources as well.

The Python source comes with several example scripts on Tkinter programming. Go to the Demo/tkinter subdirectory of the source distribution. The guido and matt directories contain numerous basic and useful exam- ples on GUI programming with Python and Tkinter. These demo scripts are small and to-the-point – an attractive feature for novice GUI programmers. Grayson’s book [10] has numerous (more advanced) examples, and the source code can be obtained over the Internet.

The **Pmw** package contains a very useful demo facility. The All.py script in the demos subdirectory of the Pmw source offers a GUI where you can examine the layout, functionality, and source code of all the Python megaw- idgets. The electronic Pmw documentation also contains many instructive examples.

There are three main GUI demos in this chapter and Chapter 11:   
– the **demoGUI.py** script in Chapter 6.3, which may act as some kind of a quick-reference for the most common widgets,  
   
– the simvizGUI\*.py family of scripts in Chapter 6.2, which equip the simulation and visualization script from Chapter 2.3 with a GUI, and  
   
– the planet\*.py family of scripts in Chapter 11.3 for introducing animated graphics.

6.1.2 The First Python/Tkinter Encounter

GUI toolkits are often introduced by making a trivial Hello World example, usually a button with “Hello, World!”, which upon a user click destroys the window. Our counterpart to such an introductory GUI example is a graphical version of the Scientific Hello World script described in Chapter 2.1. For pedagogical reasons it will be convenient to define a series of Scientific Hello World GUIs with increasing complexity to demonstrate basic features of GUI programming. The layout of the first version of this GUI is displayed in Figure 6.1. The GUI has a label with “Hello, World!”, but in addition the

Fig. 6.1. Scientific Hello World GUI, version 1 (hwGUI1.py).

user can specify a number in a field, and when clicking the equals button, the GUI can display the sine of the number.

A Python/Tkinter implementation of the GUI in Figure 6.1 can take the following form.

**The Complete Code.**

#!/usr/bin/env python

from Tkinter import \*

import math

root = Tk()

top = Frame(root)

top.pack(side=’top’)

# root (main) window

# create frame

# pack frame in main window

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

r = StringVar() # variable to be attached to r\_entry r.set(’1.2’) # default value

r\_entry = Entry(top, width=6, textvariable=r) r\_entry.pack(side=’left’)

s = StringVar() # variable to be attached to s\_label

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()

The script is available as the file **hwGUI1.py** in **src/py/gui**.

**Dissection**. We need to load the Tkinter module to get access to the Python bindings to Tk widgets. Writing

**from Tkinter import \***

means that we can access the Tkinter variables, functions, and classes without prefixing the names with Tkinter. Later, when we also use the Pmw library, we will sometimes write import Tkinter, which requires us to use the Tkinter prefix. This can be convenient to distinguish Tkinter and Pmw functionality.

The GUI script starts with creating a root (or main) window and then a frame widget to hold all other widgets:

root = Tk()

top = Frame(root)

top.pack(side=’top’)

# root (main) window

# create frame

# pack frame in main window

When creating a widget, such as the frame top, we always need to assign a parent widget, here root. This is the way we define the widget hierarchy in our GUI application. Widgets must be packed before they can appear on the screen, accomplished by calling the pack method. The keyword argument side lets you control how the widgets are packed: vertically (side=’top’ or side=’bottom’) or horizontally (side=’left’ or side=’right’). How we pack the top frame in the root window is of no importance since we only have one widget, the frame, in the root window. The frame is not a requirement, but it is a good habit to group GUI elements in frames – it tends to make extensions easier.

Inside the top frame we start with defining a label containing the text ’Hello, World! The sine of’:

**hwtext = Label(top, text=’Hello, World! The sine of’)**

**hwtext.pack(side=’left’)**

All widgets inside the top frame are to be packed from left to right, specified by the side=’left’ argument to pack.

The next widget is a text entry where the user is supposed to write a num- ber. A Python variable r is tied to this widget such that r always contains the text in the widget. Tkinter cannot tie ordinary Python variables to the contents of a widget: one must use special Tkinter variables. Here we apply a string variable, represented by the class StringVar. We could also have used DoubleVar, which holds floating-point numbers. Declaring a StringVar vari- able, setting its default value, and binding it to a text entry widget translate to

**r = StringVar() # variable to be attached to widgets**

**r.set(’1.2’); # default value**

**r\_entry = Entry(top, width=6, textvariable=r);**

**r\_entry.pack(side=’left’);**

A similar construction is needed for the s variable, which will be tied to the label containing the result of the sine computation:

s = StringVar() # variable to be attached to widgets

s\_label = Label(top, textvariable=s, width=18)

s\_label.pack(side=’left’)

Provided we do not need to access the widget after packing, we can merge creation and packing, e.g.,

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

The equals button, placed between the text entry and the result label, is supposed to call a function comp\_s when being pressed. The function must be declared before we can tie it to the button widget:

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’);

Observe that we have to convert the string r.get to a float prior to computing the sine and then convert the result to a string again before calling s.set. The global s is not required here, but it is a good habit to explicitly declare global variables that are altered in a function.

The last statement in a GUI script is a call to the event loop:

**root.mainloop()**

Without this call nothing is shown on the screen.

The StringVar variable is continuously updated as the user writes characters in the text entry field. We can make a very simple GUI illustrating this point, where a label displays the contents of a StringVar variable bound to a text entry field:

#!/usr/bin/env python

from Tkinter import \*

root = Tk()

r = StringVar()

Entry(root, textvariable=r).pack() Label(root, textvariable=r).pack() root.mainloop()

Start this GUI (the code is in the file stringvar.py), write some text in the entry field, and observe how the label is updated for each character you write. Also observe that the label and window expand when more space is needed.

The reason why we need to use special StringVar variables and not a plain Python string is easy to explain. When sending a string as the textvariable argument in Entry or Label constructors, the widget can only work on a copy of the string, whereas an instance of a StringVar class is transferred as a reference and the widget can make in-place changes of the contents of the instance (see Chapter 3.3.4).

6.1.3 Binding Events

Let us modify the previous GUI such that pressing the return key in the text entry field performs the sine computation. The look of the GUI hardly changes, but it is natural to replace the equals button by a text (label), as depicted in Figure 6.2. Replacing a button with a label is easy:

Fig. 6.2. Scientific Hello World GUI, version 2 (hwGUI2.py).

equals = Label(top, text=’ equals ’)

equals.pack(side=’left’)

Binding the event “pressing return in the text entry r\_entry” to calling the comp\_s subroutine is accomplished by the widget’s bind method:

r\_entry.bind(’<Return>’, comp\_s)

To be able to call the bind method, it is important that we have a vari- able holding the text entry (here r\_entry). It is also of importance that the function called by an event (here comp\_s) takes an event object as argument:

def comp\_s(event):

global s

s.set(’%g’ % math.sin(float(r.get()))) # construct string

You can find the complete script in the file hwGUI2.py.

Another useful binding is to destroy the GUI by pressing ’q’ on the keyboard anywhere in the window:

def quit(event):

root.destroy()

root.bind(’<q>’, quit)

For the fun of it, we can pop up a dialog box to confirm the quit:

import tkMessageBox

def quit(event):

if tkMessageBox.askokcancel(’Quit’,’Do you really want to quit?’): root.destroy()

root.bind(’<q>’, quit)

The corresponding script is found in hwGUI3.py. Try it! The look of the GUI is identical to what is shown in Figure 6.2.

6.1.4 Changing the Layout

**Alternative Widget Packing**. Instead of packing the GUI elements from left to right we could pack them vertically (i.e. from top to bottom), as shown in Figure 6.3. Vertical packing is simply a matter of calling the pack method with the argument side=’top’:

hwtext. pack(side=’top’) r\_entry.pack(side=’top’) compute.pack(side=’top’) s\_label.pack(side=’top’)

The corresponding script has the name hwGUI4.py.

Fig. 6.3. Scientific Hello World GUI, version 4 (hwGUI4.py).

**Controlling the Layout**. The layout of the previous GUI can be manipulated in various ways. We can, for instance, add a quit button and arrange the widgets as shown in Figure 6.4. To obtain this result, we need to do a more

Fig. 6.4. Scientific Hello World GUI, version 5 (hwGUI5.py).

sophisticated packing of the widgets. We already know that widgets can be packed from top to bottom (or vice versa) or from left to right (or vice versa). From Figure 6.4 we see that the window contains three rows of widgets packed from top to bottom. The middle row contains several widgets packed horizontally from left to right. The idea is that a collection of widgets can be packed into a frame, while the frames or single widgets can then be packed into the main window or another frame.

As an example of how to pack widgets inside a frame, we wrap a frame around the label “Hello, World!”:

**# 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:**

**hwtext = Label(hwframe, text=’Hello, World!’) hwtext.pack(side=’top’) # side is irrelevant (one widget!)**

Our next task is to declare a set of widgets for the sine computations, pack them horizontally, and then pack this frame in the vacant space from the top in the top frame:

**# create frame to hold the middle row of widgets:**

**rframe = Frame(top)**

**# this frame (row) is packed from top to bottom (in the top frame): 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 r\_entry r.set(’1.2’) # default value**

**r\_entry = Entry(rframe, width=6, textvariable=r) r\_entry.pack(side=’left’)**

s = StringVar() # variable to be attached to s\_label def comp\_s(event):

global s

s.set(’%g’ % math.sin(float(r.get()))) # construct string

r\_entry.bind(’<Return>’, comp\_s)

compute = Label(rframe, text=’ equals ’) compute.pack(side=’left’)

s\_label = Label(rframe, textvariable=s, width=12) s\_label.pack(side=’left’)

Notice that the widget hierarchy is reflected in the way we create children of widgets. For example, we create the compute label as a child of rframe. The complete script is found in the file hwGUI5.py. We remark that only the middle row of the GUI requires a frame: both the “Hello, World!” label and the quit button can be packed with side=’top’ directly into the top frame. In the hwGUI5.py code we use a frame for the “Hello, World!” label, just for illustration, but not for the quit button.

The hwGUI5.py script also offers a quit button bound to a quit function in addition to binding ’q’ on the keyboard to the quit function. Unfortunately, Python demands that a function called from a button (using command=quit) takes no arguments while a function called from an event binding, such as the statement root.bind(’<q>’,quit), must take one argument event, cf. our previous example on a quit function. This potential inconvenience is elegantly resolved by defining a quit function with an optional argument:

def quit(event=None):

root.destroy()

**Controlling the Widgets’ Appearance**. The GUI shown in Figure 6.5 dis- plays the text “Hello, World!” in a larger boldface font. Changing the font is performed with an optional argument when constructing the label:

**hwtext = Label(hwframe, text=’Hello, World!’, font=’times 18 bold’)**

Fonts can be specified in various ways:

**font = ’times 18 bold’ # cross-platform font description**

**font = (’Times’, 18, ’bold’) # tuple (font family, size, style)**

**# X11 font specification:**

**font = ’-adobe-times-bold-r-normal-\*-18-\*-\*-\*-\*-\*-\*-\*’**

**hwtext = Label(hwframe, text=’Hello, World!’, font=font)**

Enlarging the font leads to a squeezed appearance of the widgets in the GUI. We therefore add some space around the widget as part of the pack command:

hwtext.pack(side=’top’, pady=20)

Here, pady=20 means that we add a space of 20 pixels in the vertical direction. Padding in the horizontal direction is specified by the padx keyword. The complete script is found in the file hwGUI6.py.

Changing the colors of the foreground text or the background of a widget is straightforward:

**quit\_button = Button(**

**top, text=’Goodbye, GUI World!’, command=quit, background=’yellow’, foreground=’blue’)**

Making this quit button fill the entire horizontal space in the GUI, as shown in Figure 6.6, is enabled by the fill option to pack:

quit\_button.pack(side=’top’, pady=5, fill=’x’)

Fig. 6.5. Scientific Hello World GUI, version 6 (hwGUI6.py).

Fig. 6.6. Scientific Hello World GUI, version 7 (hwGUI7.py).

The fill value ’x’ means expanding the widget in horizontal direction, ’y’ indicates expansion in vertical direction (no space left here in that direc- tion), or ’both’, meaning both horizontal and vertical fill. You can play with hwGUI7.py to see the effect of using fill and setting colors.

The anchor option to pack controls how the widgets are placed in the available space. By default, pack inserts the widget in a centered position (anchor=’center’). Figure 6.7 shows an example where the widgets appear left-adjusted. This packing employs the option anchor=’w’ ( ’w’ means west, and other anchor values are ’s’ for south, ’n’ for north, ’nw’ for north west, etc.). There is also more space around the text inside the quit wid- get in this GUI, specified by the ipadx and ipady options. For example, ipadx=30,ipady=30 adds a space of 30 pixels around the text:

**quit\_button.pack(side=’top’,pady=5,ipadx=30,ipady=30,anchor=’w’)**

The complete script appears in the file hwGUI8.py.

Chapter 6.1.7 guides the reader through an interactive session for increasing the understanding of how the pack method and its many options work. Chapter 6.1.8 describes an alternative to pack, called grid, which applies a table format for controlling the layout of the widgets in a GUI.

Fig. 6.7. Scientific Hello World GUI, version 8 (hwGUI8.py).

6.1.5 The Final Scientific Hello World GUI

In our final version of our introductory GUI we replace the equals label by a button with a flat relief2 such that it looks like a label but performs compu- tations when being pressed:

compute = Button(rframe, text=’ equals ’, command=comp\_s, relief=’flat’)

compute.pack(side=’left’)

Figure 6.16a on page 261 demonstrates various values and effects of the relief keyword.

When the computation function comp\_s is bound to pressing the return key in the text entry widget,

r\_entry.bind(’<Return>’, comp\_s)

an event object is passed as the first argument to the function, while when bound to a button, no event argument is present (cf. our previous discussion of calling the quit function through a button or an event binding). The comp\_s function must therefore take an optional event argument:

**def comp\_s(event=None):**

**global s**

**s.set(’%g’ % math.sin(float(r.get()))) # construct string**

The GUI has the same appearance as in Figure 6.6. The complete code is found in the file hwGUI9.py and is listed next.

2 Relief is the three-dimensional effect that makes a button appear as raised and an entry field as sunken.

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

root = Tk()

top = Frame(root)

top.pack(side=’top’)

# root (main) window

# create frame

# pack frame in main window

# create frame to hold the first widget row:

hwframe = Frame(top)

# this frame (row) is packed from top to bottom (in the top frame): 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)

# 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’)

r = StringVar() # variable to be attached to r\_entry r.set(’1.2’) # default value

r\_entry = Entry(rframe, width=6, textvariable=r) r\_entry.pack(side=’left’)

s = StringVar() # variable to be attached to s\_label def comp\_s(event=None):

global s

s.set(’%g’ % math.sin(float(r.get()))) # construct string

r\_entry.bind(’<Return>’, comp\_s)

compute = Button(rframe, text=’ equals ’, command=comp\_s, relief=’flat’)

compute.pack(side=’left’)

s\_label = Label(rframe, textvariable=s, width=12) s\_label.pack(side=’left’)

# finally, make a quit button: def quit(event=None):

root.destroy()

quit\_button = Button(top, text=’Goodbye, GUI World!’, command=quit,

background=’yellow’, foreground=’blue’) quit\_button.pack(side=’top’, pady=5, fill=’x’)

root.bind(’<q>’, quit) root.mainloop()

6.1.6 An Alternative to Tkinter Variables

The Scientific Hello World scripts with a GUI presented so far, use special Tkinter variables for holding the input from the text entry widget and the result to be displayed in a label widget. Instead of using variables tied to the widgets, one can simply read the contents of a widget or update widgets, when needed. In fact, all the widget properties that can be set at construction time, can also be updated when desired, using the configure or config methods (the names are equivalent). The cget method is used to extract a widget property. If w is a Label widget, we can run

>>> w.configure(text=’new text’)

>>> w.config(text=’new text’)

>>> w[’text’] = ’new text’ #

>>> print w.cget(’text’)

’new text’

>>> print w[’text’] # ’new text’

equiv. to w.configure or w.config

equiv. to w.cget

Consider the script hwGUI9.py. We now modify the script and create the entry widget without any textvariable option: r\_entry = Entry(rframe, width=6)

r\_entry.pack(side=’left’)

A default value can be inserted directly in the widget:

**r\_entry.insert(’end’, ’1.2’) # insert default text ’1.2’**

Inserting text requires a specification of where to start the text: here we specify ’end’, which means the end of the current text (but there is no text at the present stage).

When we need to extract the contents of the entry widget, we call its get method (many widgets provide such type of function for extracting the user’s input):

r = float(r\_entry.get())

s = math.sin(r)

The label widget s\_label, which is supposed to hold the result of the sine computation, can at any time be updated by a configure method. For exam- ple, right after s is assigned the sine value, we can say

s\_label.configure(text=str(s))

or use a printf-like string if format control is desired:

s\_label.configure(text=’%g’ % s)

The complete code is found in hwGUI9\_novar.

Whether to bind variables to the contents of widgets or use the get and configure methods, is up to the programmer. We apply both techniques in this book.

**6.1.7 About the Pack Command**

Below is a summary of common options to the pack command. Most of the options are exemplified in Chapter 6.1.4.

* – The side option controls the way the widgets are stacked. The various values are: ’left’ for placing the widget as far left as possible in the frame, ’right’ for stacking from the right instead, ’top’ (default) for stacking the widgets from top to bottom, and ’bottom’ for stacking the widgets from bottom to top.
* – The padx and pady options add space to the widget in the horizontal and vertical directions, respectively. For example, the space around a button can be made larger.
* – The ipadx and ipady options add space inside the widget. For example, a button can be made larger.
* – The anchor option controls the placement of the text inside the widget. The options are ’center’ for center, ’w’ for west, ’n’w for north west, ’s’ for south, and so on.
* – The fill option with value ’x’ lets the widget fill all available horizontal space. The value ’y’ implies filling all available vertical space, and ’both’ is the combination of ’x’ and ’y’.
* – The expand option with a true value (1, True, or ’yes’) creates a frame around the widget that extends as much as possible, in the directions specified by fill, when the main window is resized by the user (see Chapter 6.3.21).  
     
  Getting an understanding of the pack command takes some time. A very good tool for developing a feel for how the pack options work is a demo program src/tools/packdemo.tcl, written by Ryan McCormack. With this script you can interactively see the effect of padding, filling, anchoring, and packing left-right versus top-bottom. Figure 6.8 shows the GUI of the script.  
     
  The reader is strongly encouraged to start the packdemo.tcl script and perform the steps listed below to achieve an understanding of how the various options to pack influence the placement of widgets.  
  1. Start with pressing Spawn R to place a widget in the right part of the white frame.
  2. A widget is placed in the available space of its parent widget. In the demo script packdemo.tcl the available space is recognized by its white color. Placing a new widget in the left part of the available space, corresponding to pack(side=’left’), is performed by clicking on Spawn L. The widget itself is just big enough to hold its text Object 2, but it has a larger geometrical area available, marked with a gray color.

Fig. 6.8. The GUI of the packdemo.tcl script for illustrating the effect of various options to the pack command for placing widgets.

1. Clicking on Fill: y corresponds to pack(side=’left’,fill=’y’). The effect is that the widget fills the entire gray space. Click Fill: none to reset the fill option.
2. Pressing the check button Expand illustrates the expand=True option: the available area for the widget is now the complete available space in the parent widget. The widget can expand into all of this area if we request a fill in both directions (Fill: both).
3. Reset the expand and fill options. Try different anchoring: n, s, e, and so on. These actions move the widget around in the available gray space. Turn on Expand and see the effect of anchoring in this case.
4. Turn off the expand option and reset the anchoring to the center position. Change the padx and pady parameters to 30 and 50, respectively. You will see that the space around the widget, the gray area, is enlarged.
5. Try different side parameters: top, bottom, and right by choosing Spawn T, Spawn B, Spawn R. Observe how the values of the padx and pady parameters influence the size of the gray area.

8. Click on Shrink Wrap. The space in the parent of the spawned widgets is now made as small as possible. This is the normal layout when creating a GUI.

Playing with packdemo.tcl as outlined in the previous list hopefully estab- lishes an understanding that makes it easier to construct the correct pack commands for a desired layout.

More information on how the pack method and its options work is found in [10, Ch. 5] and [26, Ch. 17].

6.1.8 An Introduction to the Grid Geometry Manager

The grid geometry manager, grid, is an alternative to the pack method. Wid- gets are placed in a grid of m × n cells, like a spreadsheet. In some cases this gives simpler control of the GUI layout than using the pack command. How- ever, in most cases pack is the simplest choice and clearly the most widespread tool among Tk programmers for placing widgets.

We have rewritten the Hello World GUI script hwGUI9.py to make use of the grid geometry manager. Figure 6.6 on page 237 displays the layout of this GUI. There are three rows of widgets, one widget in the first row, four widgets in the second row, and one widget in the last row. This makes up 3 × 4 cells in the GUI layout. The widget in the first row should be centered in the space of all four columns, and the widget in the last row should expand across all columns. The version of the Python script hwGUI9.py utilizing the grid geometry manager is called hwGUI9\_grid.py and is explained after the complete source code listing.

The Complete Code.

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

root = Tk()

top = Frame(root)

top.pack(side=’top’)

# root (main) window

# create frame

# pack frame in main window

# use grid to place widgets in 3x4 cells:

font = ’times 18 bold’

hwtext = Label(top, text=’Hello, World!’, font=font) hwtext.grid(row=0, column=0, columnspan=4, pady=20)

r\_label = Label(top, text=’The sine of’) r\_label.grid(row=1, column=0)

r = StringVar() # variable to be attached to r\_entry r.set(’1.2’) # default value

r\_entry = Entry(top, width=6, textvariable=r) r\_entry.grid(row=1, column=1)

s = StringVar() # variable to be attached to s\_label def comp\_s(event=None):

global s

s.set(’%g’ % math.sin(float(r.get()))) # construct string

r\_entry.bind(’<Return>’, comp\_s)

compute = Button(top, text=’ equals ’, command=comp\_s, relief=’flat’) compute.grid(row=1, column=2)

s\_label = Label(top, textvariable=s, width=12) s\_label.grid(row=1, column=3)

# finally, make a quit button: def quit(event=None):

root.destroy()

quit\_button = Button(top, text=’Goodbye, GUI World!’, command=quit,

background=’yellow’, foreground=’blue’) quit\_button.grid(row=2, column=0, columnspan=4, pady=5, sticky=’ew’)

root.bind(’<q>’, quit) root.mainloop()

Dissection. The only difference from hwGUI9.py is that we do not use sub- frames to pack widgets. Instead, we lay out all widgets in a 3×4 cell structure within a top frame. For example, the text entry widget is placed in the second row and column (row and column indices start at 0):

r\_entry.grid(row=1, column=1)

The “Hello, World!” label is placed in the first row and first column, allowing it to span the whole row of four columns:

hwtext.grid(row=0, column=0, columnspan=4, pady=20)

A corresponding rowspan option enables spanning a specified number of rows. The quit button should also span four columns, but in addition we want it to fill all the available space in that row. This is achieved with the sticky op- tion: sticky=’ew’. In the case a cell is larger than the widget inside it, sticky controls the size and position of the widget. The parameters ’n’ (north), ’s’ (south), ’e’ (east), and ’w’ (west), and any combinations of them, let you justify the widget to the top, bottom, right, or left. The quit button has sticky=’ew’, which means that the button is placed towards left and right at

the same time, i.e., it expands the whole row.

The GUI in Figure 6.7 on page 238 can be realized with the grid geometry

manager by using the sticky option. The “Hello, World!” label and the quit button are simply placed with sticky=’w’.

More detailed information about the grid geometry manager is found in [10] and [41]. One can use pack and grid in the same application, as we do in the simvizGUI2.py script in Chapter 6.2.

.1.9 Implementing a GUI as a Class

GUI scripts often assemble some primitive Tk widgets into a more compre- hensive interface, which occasionally can be reused as a part of another GUI. The class concept is very well suited for encapsulating the details of a GUI component and makes it simple to reuse the GUI in other GUIs. We shall therefore in this book implement Python GUIs in terms of classes to promote reuse. To illustrate this technique, we consider the final version of the Hello World GUI, in the file hwGUI9.py, and reorganize that code using classes. The basic ideas are sketched below.

* – Send in a parent (also called master) widget to the constructor of the class. All widgets in the class are then children of the parent widget. This makes it easy to embed the GUI in this class in other GUIs: just construct the GUI instance with a different parent widget. In many cases, including this introductory example, the supplied parent widget is the main (root) window of the GUI.
* – Let the constructor make all permanent widgets. If the code in the con- structor becomes comprehensive, we can divide it into smaller pieces im- plemented as methods.
* – The variables r and s, which are tied to an entry widget and a label widget, respectively, must be class attributes such that they are accessible in all class methods.

– The comp\_s and quit functions are methods in the class.  
   
The rest of this chapter only assumes that the reader has grasped the very basics of Python classes, e.g., as described in Chapter 3.2.9.  
   
Before we present the complete code, we outline the contents of the class:  
   
 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()

Here is the specific hwGUI10.py script implementing all Python details in the previous sketch of the program.

#!/usr/bin/env python

"""Class version of hwGUI9.py."""

from Tkinter import \* import math

class HelloWorld:

def \_\_init\_\_(self, parent):

self.master = parent top = Frame(parent) top.pack(side=’top’)

# create frame to hold

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)

# 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’)

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.master.bind(’<q>’, self.quit)

def quit(self, event=None): self.master.quit()

def comp\_s(self, event=None):

self.s.set(’%g’ % math.sin(float(self.r.get())))

root = Tk() # root (main) window hello = HelloWorld(root)

root.mainloop()

With the previous outline of the organization of the class and the fact that all statements in the functions are copied from the non-class versions of the hwGUI\*.py codes, there is hopefully no need for dissecting the hwGUI10.py script. From now on we will put all our GUIs in classes.

6.1.10 A Simple Graphical Function Evaluator

Consider the GUI shown in Figure 6.9. The user can type in the formula of a mathematical function f (x) and evaluate the function at a particular value of x. The GUI elements are familiar, consisting of labels and entry fields. How much code do you think is required by such a GUI? In compiled languages, like C and C++, the code has a considerable size as you probably need to parse mathematical expressions. Just a few Python statements are necessary to build this GUI, thanks to the possibility in interpreted, dynamically typed languages for evaluating an arbitrary string as program code.

Fig. 6.9. GUI for evaluating user-defined functions.

The labels and text entries are straightforward to create if one has un- derstood the introductory Hello World GUI scripts from Chapters 6.1.2 and 6.1.3. The contents in the text entry fields and the result label are set and extracted using insert/configure and get commands as explained in Chap- ter 6.1.6 (we could, alternatively, tie Tkinter variables to the entry fields).

We build a label, a text entry field f\_entry for the f(x) expression, a new label, a text entry field x\_entry for the x value, a button “f=” (with flat relief) for computing f(x), and finally a label s\_label for the result of f applied to x. The button is bound to a function calc, which must grab the expression for f(x), grab the x value, compute the f(x) value, and update s\_label with the result. We want to call calc by either pressing the button or typing return in the x\_entry field. In the former case, no arguments are transferred to calc, while in the latter case, calc receives an event argument. We can create calc as follows:

def calc(event=None):

f\_txt = f\_entry.get() # get function expression as string x = float(x\_entry.get()) # define x

res = eval(f\_txt) # the magic line calculating f(x)

global s\_label

s\_label.configure(text=’%g’ % res) # display f(x) value

Note that since s\_label is changed, we need to declare it as a global variable in the function.

The only non-trivial part of the calc code is the evaluation of f(x). We have a string expression for f(x) available as f\_txt, and we have the value of x available as a floating point number x. Python offers the function eval(s) to evaluate an arbitrary expression s as Python code (see Chapter 8.1.3). Hence, eval(f\_txt) can now be used to evaluate the f(x) function. Of course, f\_txt must contain a mathematical expression in valid Python syntax. The statement

res = eval(f\_txt)

works well if f\_txt is, e.g., x + sin(x), since x is a variable with a value when res = ... is executed and since x + sin(x) is valid Python syntax. The value of res is the same as if this variable were set as res = x + sin(x). On the other hand, the expression x + sin(x\*a) for f\_txt does not work well, because a is not defined in this script. Observe that in order to write expressions like sin(x), we need to have imported the math module as from math import \*.

The complete code is found in src/py/gui/simplecalc.py. 6.1.11 Exercises

Exercise 6.1. Modify the Scientific Hello World GUI.

Create a GUI as shown in Figure 6.10, where the user can write a number and click on sin, cos, tan, or sqrt to take the sine, cosine, etc. of the number. After the GUI is functioning, adjust the layout such that the computed num- ber appears to the right in the label field. (Hint: Look up the man page for the Label widget. The “Introduction to Tkinter” link in doc.html is a starting point.) ⋄

Fig. 6.10. GUI to be developed in Exercise 6.1. The GUI consists of an entry field, four buttons, and a label (with sunken relief).

Exercise 6.2. Change the layout of the GUI in Exercise 6.1.

Change the GUI in Exercise 6.1 on page 248 such that the layout becomes similar to the one in Figure 6.11. Now there is only one input/output field (and you can work with only one StringVar or DoubleVar variable), just like a calculator. A Courier 18pt bold font is used in the text entry field.

⋄

Exercise 6.3. Control a layout with the grid geometry manager.

Consider the following script, whose result is displayed in Figure 6.12a:

#!/usr/bin/env python

from Tkinter import \*

root = Tk() root.configure(background=’gray’) row = 0

for color in (’red’, ’orange’, ’yellow’, ’blue’, ’green’, ’brown’, ’purple’, ’gray’, ’pink’):

l = Label(root, text=color, background=’white’) l.grid(row=row, column=0)

f = Frame(root, background=color, width=100, height=2) f.grid(row=row, column=1)

row = row + 1

root.mainloop()

Use appropriate grid options (sticky and pady) to obtain the improved layout in Figure 6.12b. The original script is available in src/misc/colorsgrid1.py. ⋄

Fig. 6.12. To the left is the layout produced by the script listed in Exercise 6.3, and to the right is the desired layout.

Exercise 6.4. Make a demo of Newton’s method.

The purpose of this exercise is to make a GUI for demonstrating the steps in Newton’s method for solving equations f(x) = 0. The GUI consists of a text entry for writing the function f(x) (in valid Python syntax), a text entry for giving a start point x0 for the iteration, a button Next step for computing and visualizing the next iteration in the method, and a label containing the current approximation to the root of the equation f(x) = 0. The user fills in the mathematical expression for f(x), clicks repeatedly on Next step, and for each click a Gnuplot window pops up with a graph y = f(x), a graph of the straight line approximation to f(x): y = f′(xp)(x − xp) + f(xp), and a vertical dotted line x = xp indicating where the current approximation xp to the root is located. Recall that Newton’s method uses the straight line approximation to find the next xp. Use a finite difference approximation to evaluate f′(x):

f′(x) ≈ f(x+h)−f(x−h), 2h

for h small (say h ∼ 10−5). Test the GUI with f(x) = x−sinx and f(x) = tanh x.

Hint: see Chapter 4.3.3 for how to make Gnuplot plots directly from a Python script. The range of the x axis must be adjusted according to the current value of the xp point.

⋄

6.2 Adding GUIs to Scripts

Scripts are normally first developed with a command-line based user interface for two reasons: (i) parsing command-line options is easy to code (see Chap- ter 2.3.5 or 8.1.1), and (ii) scripts taking input data from the command line (or file) are easily reused by other scripts (cf. Chapter 2.4). When a desire for having a graphical user interface arises, this can be created as a separate GUI wrapper on top of the command-line oriented script. The main advantage of such a strategy is that we can reuse the hopefully well-tested command-line oriented script.

The forthcoming sections show how to make a GUI wrapper on top of the simviz1.py script from Chapter 2.3. With this example, and a little help from Chapter 6.3, you should be able to wrap your own command-line oriented tools with simple graphical user interfaces. You need to be familiar with Chapter 6.1 before proceeding.

6.2.1 A Simulation and Visualization Script with a GUI

Chapter 2.3 describes a script simviz1.py for automating the execution of a simulation program and the subsequent visualization of the results. The

interface to this script is a set of command-line options. A GUI version of the script will typically replace the command-line options with text entry fields, sliders, and other graphical elements. Our aim now is to make a GUI front-end to simviz1.py, i.e., we collect input data from the GUI, construct the proper simviz1.py command, and run that command by in the operating system.

Our first attempt to create the GUI is found in the file simvizGUI1.py in the directory src/py/gui. The look of this GUI is shown in Figure 6.13. The layout in the middle part of the GUI is far from satisfactory, but we shall improve the placement of the widgets in forthcoming versions of the script.

Fig. 6.13. Snapshot of the simvizGUI1.py GUI. Note the ugly arrangement of the label and text entry widgets in the middle part.

Here is a rough sketch of the class used to realize the GUI:

class SimVizGUI:

def \_\_init\_\_(self, parent):

"""Build the GUI.""" ...

def compute(self):

"""Run simviz1.py."""

...

Clicking on the Compute button makes a call to compute, where the contents of the GUI elements are extracted to form the proper simviz1.py command. The input data to simviz1.py fall in three categories: text, numbers of “arbitrary” value, and numbers in a prescribed interval. An entry widget is

useful for the two first categories, whereas a slider is convenient for the latter. To tie variables to widgets, we may represent all the floating-point numbers by DoubleVar objects and all text variables by StringVar objects. Since there are 10 input parameters in total, we can avoid repetitive construction of sliders and text entry fields by providing functions for these two actions. Text entry fields are created by

def textentry(self, parent, variable, label):

"""Make a textentry field tied to variable."""

# pack a label and entry horizontally in a frame: 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 Scale widget is used to create sliders:

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

We employ the idea from Chapter 3.2.5 of putting all parameters in a script into a common dictionary. This dictionary will now consist of Tkinter vari- ables of type DoubleVar or StringVar tied to widgets. A typical realization of a slider widget follows this pattern:

self.p[’m’] = DoubleVar(); self.p[’m’].set(1.0) self.slider(slider\_frame, self.p[’m’], 0, 5, ’m’)

This creates a slider, with label m, ranging from 0 to 5, packed in the parent frame slider\_frame. The default value of the slider is 1. We have simply dropped to store the widget returned from self.slider, because we do not have a need for this. (If the need should arise later, we can easily store the widgets in a dictionary (say) self.w, typically self.w[’m’] in the present example. See also Exercise 6.7.)

All the slider widgets are placed in a frame in the left part of the GUI (slider\_frame). In the middle part (middle\_frame) we place the text entries, plus two buttons, one for running simviz1.py and one for destroying the GUI. In the right part, we include a sketch of the problem being solved.

The compute function runs through all the keys in the self.p dictionary and builds the simviz1.py using a very compact list comprehension statement:

def compute(self):

"""Run simviz1.py."""

# add simviz1.py’s directory to PATH: os.environ[’PATH’] += os.pathsep + os.path.join(

os.environ[’scripting’], ’src’, ’py’, ’intro’) cmd = ’simviz1.py ’

# join options; -X self.p[’X’].get()

opts = [’-%s %s’ % (prm, str(self.p[prm].get()))

for prm in self.p]

cmd += ’ ’.join(opts)

print cmd

failure = os.system(cmd)

if failure:

tkMessageBox.Message(icon=’error’, type=’ok’, message=’Underlying simviz1.py script failed’, title=’Error’).show()

If simviz1.py fails, we launch a dialog box with an error message. The module tkMessageBox has a ready-made dialog widget Message whose basic use here is hopefully easy to understand. More information on this and other types of message boxes appears in Chapter 6.3.15.

A sketch of the physical problem being solved by the present application is useful, especially if the symbols in the sketch correspond to labels in the GUI. Tk supports inclusion of GIF pictures, and the following lines do the job in our script:

sketch\_frame = Frame(self.master) sketch\_frame.pack(side=’left’, padx=2, pady=2)

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

self.sketch = PhotoImage(file=gifpic)

Label(sketch\_frame, image=self.sketch).pack(side=’top’,pady=20)

We remark that the variable holding the PhotoImage object must be a class attribute (no picture will be displayed if we use a local variable).

6.2.2 Improving the Layout

Improving the Layout Using the Grid Geometry Manager. As already men- tioned, the layout of this GUI (Figure 6.13 on page 251) is not satisfactory: we need to align the text entry widgets in the middle part of the window. One method would be to pack the labels and the entries in a table fashion, as in a spreadsheet. The grid geometry manager from Chapter 6.1.8 is the right tool for this purpose. We introduce a new frame, entry\_frame, inside the middle frame to hold the labels and text entries. The labels are placed

by grid in column 0 and the text entries are put in column 1. A class vari- able row\_counter is used to count the rows in the two-column grid. The new statements in the constructor are the creation of the entry frame and the initialization of the row counter, whereas the call to textentry for setting up the widgets almost remains the same (only the parent frame is changed):

entry\_frame = Frame(middle\_frame, borderwidth=2) entry\_frame.pack(side=’top’, pady=22, padx=12)

self.row\_counter = 0 # updated in self.textentry

self.p[’func’] = StringVar(); self.p[’func’].set(’y’) self.textentry(entry\_frame, self.p[’func’], ’func’)

The textentry method must be changed since it now makes use of the grid geometry manager:

def textentry(self, parent, variable, label):

"""Make a textentry field tied to variable."""

# pack a label and entry horizontally in a frame:

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

The complete code is found in simvizGUI2.py in src/py/gui. A snapshot of the GUI appears in Figure 6.14 (compare with Figure 6.13 to see the layout improvement). The extra space (pady=22, padx=12) in the entry frame is an essential ingredient in the layout.

Improving the Layout Using the Pmw EntryField Widget. Text entry fields are often used in GUIs, and the packing of a Label and an Entry in a Frame is a tedious, repetitive construction. The Pmw package offers a megawidget, Pmw.EntryField, for constructing a text entry field with a label in one state- ment. This will be our first example on working with megawidgets from the Pmw library. A particularly attractive feature of the Pmw.EntryField widget is that a function Pmw.alignlabels can be used to nicely align several entry fields under each other. This means that the nice alignment we obtained in simvizGUI2.py by using the grid geometry manager can be more easily accom- plished using Pmw.EntryField megawidgets. (You are encouraged to modify simvizGUI2.py to use Pmw.EntryField in Exercise 6.6.)

The textentry method takes the following simple form if we apply the Pmw.EntryField megawidget:

def textentry(self, parent, variable, label): """Make a textentry field tied to variable.""" widget = Pmw.EntryField(parent,

labelpos=’w’,

label\_text=label,

entry\_textvariable=variable, entry\_width=8)

widget.pack(side=’top’) return widget

Pmw megawidgets are built of standard Tk widgets and implemented in pure Python. The Pmw.EntryField widget, for example, consists of a Tk label and a Tk entry widget. Typical options for the label part have the same name as in a standard Label widget, but with a prefix label\_ (for example, label\_text, label\_width). Similarly, Entry widget options are prefixed by entry\_ (for ex- ample, entry\_textvariable and entry\_width). The labelpos option is specific to the megawidget and indicates where the label is to be positioned: ’w’ means west, i.e., to the left of the entry; ’n’ means north, i.e., centered above the entry; ’nw’ means north west, i.e., adjusted to the left above the entry; ’s’ denotes south (below); ’e’ denotes east (to the right), and so on. The labelpos option must be given for the label\_text label to be displayed.

In the calling code, it is smart to store the Pmw.EntryField widgets in a list,

ew = [] # hold Pmw.EntryField widgets

self.p[’func’] = StringVar(); self.p[’func’].set(’y’) ew.append(self.textentry(middle\_frame, self.p[’func’], ’func’)) ...

The list ew allows us to use the Pmw.alignlabels method for nice alignment: Pmw.alignlabels(ew)

The labels and entries are placed in a grid-like fashion as in Figure 6.14.

Scripts using Pmw need an initialization after the root window is created, typically

root = Tk()

Pmw.initialise(root)

The present description of Pmw.EntryField is meant as a first Pmw en- counter. More advanced features of Pmw.EntryField appear in Chapter 6.3.4.

Remark. Gluing simulation, visualization, and perhaps data analysis is one of the major applications of scripting in computational science. Wrapping a command-line based script like simviz1.py with a GUI, as exemplified in simvizGUI2.py, is therefore a frequently encountered task. Our simvizGUI2.py script is a special-purpose script whose statements are tightly connected to the underlying simviz1.py script. By constructing reusable library tools and following a set of coding rules, it is possible to write the GUI wrapper in a few lines. In fact, typical simulation and visualization GUIs can be al- most automatically generated! Chapter 11.4 explains the design and usage of such tools. If you plan to write quite some GUIs similar to simvizGUI2.py, I strongly recommend reading Chapter 11.4.

6.2.3 Exercises

Exercise 6.5. Program with Pmw.EntryField in hwGUI10.py.

Modify the hwGUI10.py script such that the label ”The sine of” and the text entry are replaced by a Pmw.EntryField megawidget. ⋄

Exercise 6.6. Program with Pmw.EntryField in simvizGUI2.py.

Modify the simvizGUI2.py script such that all text entries are implemented with the Pmw.EntryField megawidget. (Use the pack geometry manager ex- clusively.) ⋄

Exercise 6.7. Replace Tkinter variables by set/get-like functions.

Instead of using StringVar and DoubleVar variables tied to widgets in the simvizGUI2.py script, one can call functions in the widgets for setting and getting the slider and text entry values. Use the src/py/gui/hwGUI9\_novar.py script as an example (see Chapter 6.1.6). Implement this approach and discuss pros and cons relative to simvizGUI2.py. (Hint: Now the returned widgets from the textentry and slider functions must be stored, e.g., in a dictionary self.w. The self.p dictionary can be dropped.) ⋄

Exercise 6.8. Use simviz1.py as a module in simvizGUI2.py.

The simvizGUI2.py script runs simviz1.py as a separate operating system process. To avoid starting a separate process, we can use the module version of simviz1.py, developed in Exercise B.1, as a module in simvizGUI2.py. Perform the necessary modifications of simvizGUI2.py. ⋄

Exercise 6.9. Apply Matlab for visualization in simvizGUI2.py.

The purpose of this exercise is to use Matlab as visualization engine in the simvizGUI2.py script from Chapter 6.2. Use two methods for visualiz- ing data with Matlab: (i) a Matlab script (M-file) as in Exercise 2.14 and (ii) the direct Python-Matlab connection offered by the pymat module shown in Chapter 4.4.3. (In the latter case, open the connection to Matlab in the constructor of the GUI and close it in the destructor). Add two extra but- tons Visualize (Mfile) and Visualize (pymat), and corresponding functions, for

visualizing sim.dat by the two Matlab-based methods.

You can issue Matlab commands for reading data from the sim.dat file

or you can load the sim.dat file into NumPy arrays in the script and transfer the arrays to Matlab. ⋄

6.3 A List of Common Widget Operations

A Python script demoGUI.py, in the src/py/gui directory, has been developed to demonstrate the basic usage of many of the most common Tkinter and Pmw widgets. Looking at this GUI and its source code should give you a quick recipe for how to construct widely used GUI elements. Once a widget is up and running, it is quite easy to study its man page for fine-tuning the desired functionality. The purpose of the widget demo script is to help you with quickly getting a basic version of a GUI up and running.

Contents and Layout. Figure 6.15 shows the look of the main window pro- duced by demoGUI.py. The GUI consists of a menu bar with four pulldown menus: File, Dialogs, Demo, and Help, plus a core area with text entries, a slider, a checkbutton, two ordinary buttons, and a status label. Clicking on the Display widgets for list data button launches a window (Figure 6.18 on page 270) with list box widgets, combo boxes, radio and check buttons, and an option menu. The File menu (Figure 6.17a on page 267) demonstrates file dialogs (Figures 6.17d–e on page 267) and how to terminate the application.

Examples on other types of dialogs are provided by the Dialogs menu (Figure 6.17b on page 267). This includes short messages (Figure 6.19 on page 276), arbitrary user-defined dialogs (Figure 6.20 on page 277), and di- alogs for choosing colors (Figure 6.21 on page 279). The File–Open... and Help–Tutorial menus also demonstrate how to load a large piece of text, e.g. a file, into a scrollable text widget in a separate window.

The Demo menu (Figure 6.17c on page 267) shows the effect of the relief and borderwidth widget options as well as a list of pre-defined bitmap images (Figure 6.16 on page 261).

The following text with short widget constructions assumes that you have played around with the demoGUI.py script and noticed its behavior. Observe that when you activate (most of) the widgets, a status label at the bottom of the main window is updated with information about your actions. This

Fig.6.15. GUI for demonstrating basic usage of Tkinter and Pmw widgets (demoGUI.py script).

feature makes it easy to demonstrate, in the demoGUI.py source code, how to extract user input from a widget.

Organization of the Source Code. The script demoGUI.py is organized as a class, named TkinterPmwDemo. The widgets between the menu bar and the two buttons in the main window are managed by a class InputFields, which is reused when creating a user-defined dialog, see Figure 6.20 on page 277. The demo of widgets for list data, launched by pressing the button in the main window, is also realized as a class named InputLists. The InputFields and InputLists classes work much in the same way as megawidgets, as many widgets are put together, but they are not megawidgets in the strict meaning of the term, because there is very limited control of the widgets’ properties from the calling code.

Look at the Source Code! The reader is encouraged to invest some time to get familiar with the demoGUI.py script. A good start is to concentrate on class InputFields only. This class defines nicely aligned Pmw.EntryField widgets, a Pmw.OptionMenu widget, a Tkinter.Scale widget (slider), and a Tkinter.Checkbutton. The following code segment imports demoGUI.py as a module and creates the InputFields GUI:

from demoGUI import InputFields

root = Tk()

Pmw.initialise(root)

status\_line = Label(root)

widget = InputFields(root, status\_line) widget.pack()

status\_line.pack() # put the status line below the widgets

Notice that the InputFields class demands a “status line”, i.e., a Label to which it can send information about user actions. We therefore need to create such a label in the calling code. Also notice that we can explicitly pack the InputFields GUI and place it above the status line. Launch the GUI as described (or simply run demoGUI.py fields, which is a short-cut). Load the demoGUI.py file into an editor and get familiar with the organization of the InputFields class. All the widgets are created in the create function. Most widgets have a command keyword argument which ties user actions in the widget to a function. This function normally retrieves the user-provided contents of the widget and updates the status line (label) accordingly.

When you know how class InputFields roughly works, you can take a look at InputLists, which follows the same pattern. Thereafter it is appropriate to look at the main class, TkinterPmwDemo, to see how to total GUI makes use of basic Tkinter widgets, Pmw, and the InputFields and InputLists classes. An important part of class TkinterPmwDemo is the menu bar with pulldown menus and all the associated dialogs. The widgets here follow the same set-up as in the InputFields and InputLists classes, i.e., most widgets use a command keyword argument to call a function for retrieving widget data and update the status line.

If you want to build a GUI and borrow code from demoGUI.py, you can launch demoGUI.py, find the desired widget, find the creation of that widget in the file demoGUI.py (this is one reason why you need to be a bit familiar with the structure of the source code), copy the source, and edit it to your needs, normally with a visit to the man page of the widget so you can fine-tune details.

On the following pages we shall describe the various widgets encountered in demoGUI.py in more detail. The shown code segments are mostly taken directly from the demoGUI.py script.

6.3.1 Frame

The frame widget is a container used to hold and group other widgets, usually for controlling the layout.

self.topframe = Frame(self.master, borderwidth=2, relief=’groove’) self.topframe.pack(side=’top’)

The border of the frame can be adjusted in various ways. The size of the border (in pixels) is specified by the borderwidth option, which can be com- bined with the relief option to obtain a three-dimensional effect. The effect

is demonstrated in the demoGUI.py main window (relief=’groove’), see Fig- ure 6.15, and in the relief demo in Figure 6.16a. Space around the frame is controlled by the padx and pady options, when packing the frame, or using borderwidth with relief=’flat’ (default).

Occasionally a scrolled frame is needed. That is, we can fix the size of the frame, and if the widgets inside the frame need more space, scrollbars are automatically added such that one can scroll through the frame’s widgets. Pmw offers a megawidget frame with built-in scrollbars:

self.topframe = Pmw.ScrolledFrame(self.master,

usehullsize=1, hull\_height=210, hull\_width=340)

In this case, the size of the frame is 210 × 340 pixels. The Pmw.ScrolledFrame widget is a composite widget, consisting of a standard Frame widget, Tk scrollbars, and an optional label widget. To access the plain Frame widget, we need to call

self.topframe.interior()

This frame widget can act as parent for other widgets. You can start the Pmw user-defined dialog on the Dialog menu to see a Pmw.ScrolledFrame widget in action.

6.3.2 Label

Label widgets typically display a text, such as the headline “Widgets for list data” in Figure 6.18 on page 270. This particular label is constructed by

header = Label(parent, text=’Widgets for list data’, font=’courier 14 bold’, foreground=’blue’,

background=’#%02x%02x%02x’ % (196,196,196)) header.pack(side=’top’, pady=10, ipady=10, fill=’x’)

Fonts can be named (like here) or be X11 font specification strings, as on page 236. Colors are specified either by names or by the hexadecimal code. (Observe how three rgb values (196,196,196) are converted to hexadecimal form using a simple format string: %02x prints an integer in hexadecimal form in a field of width 2 characters, padded with zeroes from the left if necessary.)

The relief option (encountered in Chapter 6.3.1) can also be used in labels to obtain, e.g., a sunken or raised effect. The demo script displays the effect of all the relief values, see Figure 6.16a, using the following code to generate widgets in a loop:

# use a frame to align examples on various relief values: frame = Frame(parent); frame.pack(side=’top’,pady=15)

reliefs = (’groove’, ’raised’, ’ridge’, ’sunken’, ’flat’) row = 0

for borderwidth in (0,2,4,6):

Fig.6.16. The Demo menu in Figure 6.15 gives rise to the pulldown menu in Figure 6.17c. The entry Relief/borderwidth lanuches the window displayed in (a), with examples of various relief values and the effect of the borderwidth parameter. Clicking the entry Bitmaps on the Demo menu, results in a list of various pre-defined bitmaps (for labels, buttons, and dialogs), as shown in (b).

label = Label(frame, text=’reliefs with borderwidth=%d: ’ % \ borderwidth)

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=borderwidth)

l.grid(row=row, column=i+1, padx=5, pady=5) row += 1

The individual widgets are here placed in a table fashion, with two rows and six columns, using grid as geometry manager instead of pack. Information about grid is given in Chapter 6.1.8.

Looking at Figure 6.16a, we see that the borderwidth option amplifies the effect of the relief. By default, borderwidth is 2 in labels and buttons, and 0 in frames.

Labels can also hold images, either predefined bitmaps or GIF files. The script simvizGUI1.py exemplifies a label with a GIF image (see page 253), whereas we here show how to include a series of predefined Tk bitmaps:

bitmaps = (’error’, ’gray25’, ’gray50’, ’hourglass’, ’info’, ’questhead’, ’question’, ’warning’)

Label(parent, text="""\

Predefined bitmaps, which can be used to label dialogs (questions, info, etc.)""",

foreground=’red’).pack()

frame = Frame(parent); frame.pack(side=’top’, pady=5) for i in range(len(bitmaps)): # write name of bitmaps

Label(frame, text=bitmaps[i]).grid(row=0, column=i+1) for i in range(len(bitmaps)): # insert bitmaps

Label(frame, bitmap=bitmaps[i]).grid(row=1, column=i+1)

Also here we use the grid geometry manager to place the widgets. Figure 6.16b displays the resulting graphics.

6.3.3 Button

A button executes a command when being pressed.

Button(self.master, text=’Display widgets for list data’, command=self.list\_dialog, width=29).pack(pady=2)

The horizontal size is specified by the width option. When left out, the but- ton’s size is just large enough to display the text. A button can hold an image or bitmap instead of a text.

6.3.4 Text Entry

One-line text entry fields are represented by entry widgets, usually in com- bination with a leading label, packed together in a frame:

frame = Frame(parent); frame.pack()

Label(frame, text=’case name’).pack(side=’left’) self.entry\_var = StringVar(); self.entry\_var.set(’mycase’) e = Entry(frame, textvariable=self.entry\_var, width=15,

command=somefunc)

e.pack(side=’left’)

Since such constructions are frequently needed, it is more convenient to use the Pmw.EntryField megawidget (see also page 254):

self.case\_widget = Pmw.EntryField(parent, labelpos=’w’,

label\_text=’case name’, entry\_width=15, entry\_textvariable=self.case, command=self.status\_entries)

Another convenient feature of Pmw.EntryField is that multiple entries can be nicely aligned below each other. This is exemplified in the main window of the demoGUI.py GUI, see Figure 6.15 on page 258. Having several widgets with labels, here Pmw.EntryField and Pmw.OptionMenu widgets, we can collect the widget instances in a list or tuple and call Pmw.alignlabels to nicely align the labels:

widgets = (self.case\_widget, self.mass\_widget, self.damping\_widget, self.A\_widget, self.func\_widget)

Pmw.alignlabels(widgets)

The various Pmw.EntryField widgets in demoGUI.py demonstrate some use- ful options. Of particular interest is the validate option, which takes a dic- tionary, e.g.,

{’validator’ : ’real’, ’min’: 0, ’max’: 2.5}

as a description of valid user input. In the current example, the input must be a real variable in the interval [0,2.5]. The Pmw.EntryField manual page, which can be reached by links from doc.html, explains the validation features in more detail.

To show the use of a validate argument, consider the entry field mass, where the input must be a positive real number:

self.mass = DoubleVar(); self.mass.set(1.0) self.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=self.mass, command=self.status\_entries)

Try to write a negative number in this field. Writing a minus sign, for instance, disables further writing. It is also impossible to write letters.

The self.status\_entries method, given through the command option, is called when hitting the return key inside the entry field. Here, this method grabs the input data in all four entry fields and displays the result in the status label at the bottom of the GUI:

def status\_entries(self):

"""Read values from entry widgets or variables tied to them.""" s = "entry fields: ’" + self.case.get() + \

"’, " + str(self.mass.get()) + \

", " + self.damping\_widget.get() + \ ", " + str(self.A.get())

self.status\_line.configure(text=s)

The self.status\_line widget is a plain label, constructed like this: self.status\_line = Label(frame, relief=’groove’,

font=’helvetica 8’, anchor=’w’)

Change the contents of some entry fields, hit return, and observe that the status label is updated.

Most entry fields are tied to a Tkinter variable. For example, the mass wid- get has an associated variable self.mass, such that calling self.mass.get() anywhere in the script extracts the value of this particular entry field. How- ever, for demonstration purposes we have included a Pmw.EntryField instance

self.damping\_widget, which is not connected to a Tkinter variable. To get the entry field’s content, we call the widget’s get function: damping\_widget.get() (cf. the status\_entries function).

Setting the value of an entry can either be done through the Tkinter vari- able’s set method or the set method in the Pmw.EntryField widget. Similar get/set functionality is explained in relation to the hwGUI9\_novar.py script or page 240.

6.3.5 Balloon Help

Balloon help means that a small window with an explaining text pops up when the user points at a widget in a user interface. Such a feature can be very helpful for novice users of an application, but quite irritating for more experienced users. Most GUIs therefore have a way of turning the balloon help on and off.

Creating balloon help with Pmw is very easy. First a Balloon object is declared and bound to the parent widget or the top frame of the window:

self.balloon = Pmw.Balloon(self.master) # used for all balloon helps Thereafter we can bind a balloon help text to any widget, e.g., a Pmw.EntryField

widget self.A\_widget: self.balloon.bind(self.A\_widget,

’Pressing return updates the status line’)

If you point with the mouse at the entry field with name amplitude, in the main window of the demoGUI.py application, you will see a balloon help pop- ping up:

The help can be turned on and off with aid of the Balloon help entry on the Help menu in the menu bar.

6.3.6 Option Menu

An option menu is a kind of pulldown menu suitable for selecting one out of n options. The realization of such a menu in Figure 6.15 on page 258 is based on a convenient Pmw widget3 and created by the following code:

3 Tkinter also has an option menu widget, called OptionMenu.

self.func = StringVar(); self.func.set(’y’) self.func\_widget = Pmw.OptionMenu(parent,

labelpos=’w’, # n, nw, ne, e, and so on label\_text=’spring’,

items=[’y’, ’y3’, ’siny’], menubutton\_textvariable=self.func, menubutton\_width=6, command=self.status\_option)

The function being called when selecting an option takes the selected value as a string argument:

def status\_option(self, value): self.status\_line.configure(text=self.func.get())

# or use the value argument instead of a Tkinter variable: self.status\_line.configure(text=value)

6.3.7 Slider

A slider, also called ruler or scale widget, is used to set a real or integer variable inside a specified interval. In Tkinter a slider is represented by the Scale class. The value of the slider is tied to a Tkinter variable (StringVar, DoubleVar, IntVar).

self.y0 = DoubleVar(); self.y0.set(0.2) self.y0\_widget = Scale(parent,

orient=’horizontal’,

from\_=0, to=2, # range of slider

tickinterval=0.5, # tickmarks on the slider "axis" resolution=0.05, # the steps of the counter above the slider

label=’initial value y(0)’, #font=’helvetica 12 italic’, length=300, variable=self.y0, command=self.status\_slider)

# label printed above the slider # optional font

# length of slider in pixels

# value is tied to self.y0

When the mouse is over the slider, the self.status\_slider method is called, and the current value is “continuously” updated in the status line:

def status\_slider(self, value): self.status\_line.configure(text=’slider value: ’ + \

str(self.y0.get())) self.status\_line.configure(text=’slider value: ’ + value)

6.3.8 Check Button

A boolean variable can be turned on or off using a check button widget. The check button is visualized as a “light” marker with an accompanying text. Pressing the button toggles the value of the associated boolean variable (an integer with values 0 or 1):

self.store\_data = IntVar(); self.store\_data.set(1) self.store\_data\_widget = Checkbutton(parent,

text=’store data’, variable=self.store\_data, command=self.status\_checkbutton)

A function can also be called when pressing a check button. In the demoGUI.py script, this function reports the state of the boolean variable:

def status\_checkbutton(self): self.status\_line.configure(text=’store data checkbutton: ’ + \

str(self.store\_data.get()))

6.3.9 Making a Simple Megawidget

The entry fields, the option menu, the slider, and the check button in Fig- ure 6.15 are collected in a separate class InputFields. This class represents a kind of megawidget. Two statements are sufficient for realizing this part of the total GUI:

fields = InputFields(self.master, self.status\_line, balloon=self.balloon, scrolled=False)

fields.pack(side=’top’)

The InputFields class defines a top frame self.topframe, into which all wid-

gets are packed, such that a simple pack method,

def pack(self, \*\*kwargs): # method in class InputFields

self.topframe.pack(kwargs, expand=True, fill=’both’)

enables us to place the composite widget fields wherever we want. Note that the arbitrary set of keyword arguments, \*\*kwargs, is just transferred from the calling code to the pack method of self.topframe, see page 113 for an explanation of variable-length keyword arguments (\*\*kwargs). Also note that after kwargs in the self.topframe.pack call we add expand=True and fill=’both’, meaning that we force the widget to be aware of the user’s window resize actions (see Chapter 6.3.21).

The parameter scrolled in the InputFields constructor allows us to choose between a standard Frame, whose size is determined by the size of the interior widgets, or a scrolled frame (Pmw.ScrolledFrame) with fixed size. The version with scrollbars is used in the user-defined dialog launched by the Dialog– Pmw user-defined dialog menu. The constructor also takes information about an external status label and a balloon help.

The code in class InputFields is simply made up of our examples on Pmw.EntryField widgets, Checkbutton, Scale, and Pmw.OptionMenu from previ- ous sections. We encourage the reader to have a look at class InputFields to

see how easy it is to group a set of widgets as one object and use the object as a simple megawidget4.

6.3.10 Menu Bar

Graphical user interfaces frequently feature a menu bar at the top of the main window. Figure 6.15 on page 258 shows such a menu bar, with four menus: File, Dialog, Demo, and Help. The look of the former three pulldown menus appears in Figure 6.17a–c. These menus can be created by the plain Tk widgets Menu and Menubutton. However, the code becomes shorter if we use the composite widget Pmw.MenuBar.

(a) (b) (c)

(d)

(e)

Fig. 6.17. The GUI in Figure 6.15 on page 258 has a menu bar with File, Dialogs, Demo, and Help menu buttons. The former three menus are displayed in (a), (b), and (c). The entries Open... and Save as... in the File menu in (a) pop up the file dialogs in (d) and (e).

The Pmw.MenuBar widget is instantiated by

self.menu\_bar = Pmw.MenuBar(parent, hull\_relief=’raised’,

hull\_borderwidth=1, balloon=self.balloon,

hotkeys=True) # define accelerators

self.menu\_bar.pack(fill=’x’)

4 Making a real megawidget, according to the Pmw standard, is a more compre- hensive task, but well described in the Pmw manual.

The relief of the menu bar is usually raised, so this is an important parameter for achieving the right look. We may also provide a balloon help. The hotkeys option allows us to define hotkeys or accelerators. If you look at the File menu in Figure 6.15, you see that there is an underscore under the F in File. This means that typing Alt+f on the keyboard5 is equivalent to pointing the cursor to File and clicking the left mouse button. The File menu is pulled down, and with the down-arrow on the keyboard one can move to, e.g., Open... and hit return to invoke the file open menu. Instead of using the arrow, one can type Alt+o to open the file dialog directly, because the letter O is underlined in the menu item Open.... These accelerators are very convenient for quick and mouse-free use of a graphical user interface. With hotkeys=True, the MenuBar widget automatically assigns appropriate accelerators.

The next natural step is to show how we realize the File menu:

self.menu\_bar.addmenu(’File’, None, tearoff=True)

self.menu\_bar.addmenuitem(’File’, ’command’, statusHelp=’Open a file’, label=’Open...’, command=self.file\_read)

self.menu\_bar.addmenuitem(’File’, ’command’, statusHelp=’Save a file’,

label=’Save as...’, command=self.file\_save)

self.menu\_bar.addmenuitem(’File’, ’command’, statusHelp=’Exit this application’, label=’Quit’,

command=self.quit)

The addmenu method adds a new pulldown menu to the menu bar. The None argument is a balloon help, but here we drop the help since the purpose of our File menu needs no further explanation. The tearoff option allows us to “tear off” the pulldown menu. If you click on File, or use the Alt+f accelerator, you see a dashed line at the top of the menu. Clicking on this dashed line tears off the menu so it is permanently available in a separate window. The feature is best understood by testing it out.

An entry in the pulldown menu is added using the addmenuitem function, which takes the name of the parent menu as first argument (here ’File’). The second argument specifies the type of menu item: ’command’ is a simple button/label-like item, ’checkbutton’ results in a check button (see Help– Balloon help), and ’separator’ makes a separating line. We refer as usual to the Pmw manual for explaining the various options of a megawidget. The label keyword argument is used to assign a visible name for this menu item, whereas command specifies the function that carries out the tasks associated with the menu item. The self.file\_read and self.file\_save methods are

5 Hold the Alt key down while pressing f or shift-f (F).

explained later, and self.quit is similar to the quit function in the introduc- tory GUIs in Chapter 6.1.

The statusHelp keyword argument is used to assign a help message. To view this message, the balloon help instance must be tied to a message bar (Pmw.MessageBar) in the main window. We have not included this feature since this is the task of Exercise 6.13.

On the Dialogs menu we have a Color dialogs item that pops up a new pull- down menu. Such nested menus are usually referred to as cascading menus, and the addcascademenu method is used to create them:

self.menu\_bar.addmenu(’Dialogs’,

’Demonstrate various Tk/Pmw dialog boxes’, # balloon help tearoff=True)

...

self.menu\_bar.addcascademenu(’Dialogs’, ’Color dialogs’,

statusHelp=’Exemplify different color dialogs’)

self.menu\_bar.addmenuitem(’Color dialogs’, ’command’, label=’Tk Color Dialog’, command=self.tk\_color\_dialog)

6.3.11 List Data

The Display widgets for list data button in the main window of the demoGUI.py GUI launches a separate window, see Figure 6.18, with various examples of suitable widgets for list-type data. The window is realized as a composite widget, implemented in class InputLists. This implementation follows the ideas of class InputFields described in Chapter 6.3.9.

A list of alternatives can be displayed using many different widgets: list box, combo box, option menu, radio buttons, and check buttons. The choice depends on the number of list items and whether we want to select single or multiple items.

6.3.12 Listbox

The most flexible widget for displaying and selecting list data is the list box. It can handle long lists, if equipped with scrollbars, and it enables single or multiple items to be selected. Pmw offers a basic Tk list box combined with a label and two scrollbars, called Pmw.ScrolledListBox. The code segment from demoGUI.py should explain the basic construction:

self.list1 = Pmw.ScrolledListBox(frame, listbox\_selectmode=’single’, # or ’multiple’ vscrollmode=’static’, hscrollmode=’dynamic’, listbox\_width=12, listbox\_height=6, label\_text=’plain listbox\nsingle selection’, labelpos=’n’,

Fig.6.18. Illustration of various widgets for representing list data: Pmw.ScrolledListBox, Pmw.ComboBox, Pmw.RadioSelect, and Tk Radiobutton. The win- dow is launched either from the Display widgets for list data button in the main menu window in Figure 6.15, or from the List data item on the Demo menu (Figure 6.17c).

selectioncommand=self.status\_list1) self.list1.pack(side=’left’, padx=10, anchor=’n’)

The list box can be configured for selecting a single item only or a collection of items, using the listbox\_selectmode keyword argument. Four values of this argument are possible: single and multiple, requiring the user to click on items, as well as browse and extended for single and multiple choices, respectively, obtained by holding the left mouse button down and moving it over the list. The reader is encouraged to edit the select mode argument in the list box demo and try out the four values.

Vertical and horizontal scrollbars are controlled by the vscrollmode and hscrollmode keywords, respectively, which take on the values static (always include scrollbars), dynamic (include scrollbars only when required, i.e., when the list is longer than the specified or default widget size), and none (no scrollbars). The widget size is here given as 6 lines of maximum 12 characters, assigned through the listbox\_height and listbox\_weight arguments. The list box has an optional label (label\_text) which can be placed above the list, indicated here by labelpos=’n’ (’n’ means north, other values are ’w’ for west, ’nw’ for north-west, and so on). Note that labelpos must be speficied for the list box to work if label\_text is specified.

A function can be called when clicking on an item in the list, here the name of this function is self.status\_list1. The purpose of this function is to extract information about the items that have been marked by the user. These are provided by the getcurselection and curselection list box functions. The former returns the text of the chosen items, whereas the latter returns the indices of the chosen items (first index is 0).

def status\_list1(self):

"""Extract single list selection."""

selected\_item = self.list1.getcurselection()[0] selected\_index = self.list1.curselection()[0]

text = ’selected list item=’ + str(selected\_item) + \

’, index=’ + str(selected\_index) self.status\_line.configure(text=text)

We have also exemplified a list box where the user can select multiple items:

self.list2 = Pmw.ScrolledListBox(frame\_left, listbox\_selectmode=’multiple’, vscrollmode=’static’, hscrollmode=’dynamic’, listbox\_width=12, listbox\_height=6, label\_text=’plain listbox\nmultiple selection’, labelpos=’n’,

items=listitems,

selectioncommand=self.status\_list2) self.list2.pack(side=’left’, anchor=’n’) ...

def status\_list2(self):

"""Extract multiple list selections."""

selected\_items = self.list2.getcurselection() # tuple selected\_indices = self.list2.curselection() # tuple text = ’list items=’ + str(selected\_items) + \

’, indices=’ + str(selected\_indices) self.status\_line.configure(text=text)

Values of list items can be provided at construction time using the items keyword argument and a Python list or tuple as value:

self.list2 = Pmw.ScrolledListBox(frame, ...

items=listitems,

...

)

Alternatively, the list can be filled out item by item after the widget con- struction:

for item in listitems:

self.list1.insert(’end’, item) # insert after end of list

A third alternative is to use submit the whole list at once:

self.list1.setlist(listitems)

# or with configure (using the keyword for the constructor): self.list.configure(items=listitems)

The ScrolledListBox class contains standard Tkinter widgets: a Listbox, a Label, and two Scrollbars. Arguments related to the label have the same name as in the basic Label widget, except that they are prefixed by label\_, as in label\_text. Similarly, one can invoke Listbox arguments by prefixing the arguments to ScrolledListBox by listbox\_, one example being listbox\_width. This naming convention is important to know about, because various options for the Tkinter widget building blocks are not included in the Pmw documen- tation. The programmer actually needs to look up the Tkinter (or Tk) man pages for those details. Hence, to get documentation about the listbox\_width parameter, one must consult the width option in the basic Listbox man page. Appropriate sources for such a man page are the electronic Tkinter man pages or the original Tcl/Tk man pages (see doc.html for relevant links), or the nicely typeset Tkinter man pages in Grayson’s book [10]. Note that the name of the list box widget is listbox in Tk and Listbox in Tkinter.

The underlying Tkinter objects in Pmw widgets can be reached using the component method. Here is an example accessing the Tkinter Listbox object in the ScrolledListBox megawidget (for making a blue background color in the list):

self.list.component(’listbox’).configure(background=’blue’)

The Pmw documentation lists the strings that can be used in the component

call.

6.3.13 Radio Button

A parameter that can take on n distinct values may for small n be represented by n radio buttons. Each radio button represents a possible value and looks like a check button, with a “light” marker and an associated text, but the n radio buttons are bound to the same variable. That is, only one button at a time can be in an active state. Radio buttons are thus an alternative to list boxes with single item selection, provided the list is short.

Plain Tk radio buttons can be constructed as follows.

self.radio\_var = StringVar() # common variable for radio buttons self.radio1 = Frame(frame\_right)

self.radio1.pack(side=’top’, pady=5)

Label(self.radio1,

text=’Tk radio buttons’).pack(side=’left’)

for radio in (’radio1’, ’radio2’, ’radio3’, ’radio4’):

r = Radiobutton(self.radio1, text=radio, variable=self.radio\_var, value=’radiobutton no. ’ + radio[5],

command=self.status\_radio1) r.pack(side=’left’)

The self.status\_radio1 method is called when the user clicks on a radio button, and the value of the associated self.radio\_var variable is written in the status line:

def status\_radio1(self):

text = ’radiobutton variable = ’ + self.radio\_var.get() self.status\_line.configure(text=text)

The values that self.radio\_var can take on are specified through the value keyword argument in the construction of the radio button.

Pmw also offers a set of radio buttons: Pmw.RadioSelect. One advantage with Pmw.RadioSelect is the flexible choice of the type of buttons: one can have radio buttons (single selection), check buttons (multiple selection), or plain buttons in single or multiple selection mode. The user’s selections can only be obtained through the function given as command argument to the constructor. If it is more convenient to tie a Tkinter variable to a set of radio buttons, the previous construction with self.radio1\_var and the Radiobutton widget is preferable.

A set of radio buttons is declared as exemplified below.

self.radio2 = Pmw.RadioSelect(frame\_right, selectmode=’single’,

buttontype=’radiobutton’, # ’button’: plain button layout labelpos=’w’,

label\_text=’Pmw radio buttons\nsingle selection’, orient=’horizontal’,

frame\_relief=’ridge’, # try some decoration...

command=self.status\_radio2) self.radio2.pack(side=’top’, padx=10, anchor=’w’)

# add items; radio buttons are only feasible for a few items: for text in (’item1’, ’item2’, ’item3’, ’item4’):

self.radio2.add(text)

self.radio2.invoke(’item2’) # ’item2’ is pressed by default

...

def status\_radio2(self, value):

text = ’Pmw check buttons: ’ + value self.status\_line.configure(text=text)

Almost the same construction can be used to define a set of check buttons. This is convenient for a list with multiple selections, although check buttons are most commonly associated with boolean variables, with one variable tied to each button. With Pmw.RadioSelect we must extract the selected items in a function and, if desired, convert this information to a set of boolean variables.

# check button list:

self.radio3 = Pmw.RadioSelect(frame\_right,

selectmode=’multiple’,

buttontype=’checkbutton’,

labelpos=’w’,

label\_text=’Pmw check buttons\nmultiple selection’, orient=’horizontal’,

frame\_relief=’ridge’, # try some decoration...

command=self.status\_radio3) self.radio3.pack(side=’top’, padx=10, anchor=’w’)

# add items; radio xobuttons are only feasible for a few items: for text in (’item1’, ’item2’, ’item3’, ’item4’):

self.radio3.add(text)

# press ’item2’ and ’item4’ by default: self.radio3.invoke(’item2’); self.radio3.invoke(’item4’) ...

def status\_radio3(self, button\_name, pressed):

if pressed:

else:

text = ’Pmw

action = ’pressed’

action = ’released’

radio button ’ + button\_name + ’ was ’ + \

action + ’; pressed buttons: ’ + \

str(self.radio3.getcurselection()) self.status\_line.configure(text=text)

6.3.14 Combo Box

A combo box can be viewed as a list, allowing single selections, where the selected item is displayed in a separate field. In a sense, combo boxes are easier to work with than lists. Figure 6.18 on page 270 displays two types of combo boxes offered by the Pmw ComboBox widget: (i) a simple combo box, where the list is visible all the time, and (ii) a dropdown combo box, where the list becomes visible upon clicking on the arrow. The basic usage is the same for both types:

# having a Python list listitems, put it into a Pmw.ComboBox: self.combo1 = Pmw.ComboBox(frame,

label\_text=’simple combo box’, labelpos=’nw’, scrolledlist\_items=listitems, selectioncommand=self.status\_combobox, listbox\_height=6,

dropdown=False) self.combo1.pack(side=’left’, padx=10, anchor=’n’)

Check out the description of the Pmw list box widget to see the meaning of most of the keyword arguments. The dropdown parameter controls whether we have a simple combo box (false) or a dropdown combo box (true). The value of this parameter is actually the only difference between the two combo boxes in Figure 6.18.

Clicking on items in the combo box forces a call to a function, here self.status\_combobox, which takes the chosen list item value as argument:

def status\_combobox(self, value):

text = ’combo box value = ’ + str(value) self.status\_line.configure(text=text)

6.3.15 Message Box

A message box widget allows a message to pop up in a separate window, Three examples on such boxes are shown in Figure 6.19. These boxes are launched from the Dialog menu in the demoGUI.py application.

The message box in Figure 6.19a is created by the function askokcancel in the tkMessageBox module:

import tkMessageBox

...

def confirmation\_dialog(self):

message = ’This is a demo of a Tk conformation dialog box’ ok = tkMessageBox.askokcancel(’Quit’, message)

if ok:

self.status\_line.configure(text="’OK’ was pressed") else:

self.status\_line.configure(text="’Cancel’ was pressed")

The buttons are labeled OK and Cancel, whereas the argument ’Quit’ spec- ifies the title in the window manager decoration of the dialog box. Another version of this message box is askyesno (also present in the demoGUI.py code), where the buttons have the names Yes and No.

Figure 6.19b shows a plain Tk message box:

def

Tk\_message\_dialog(self):

message = ’This is a demo of a Tk message dialog box’ answer = tkMessageBox.Message(icon=’info’, type=’ok’,

message=message, title=’About’).show() self.status\_line.configure(text="’%s’ was pressed" % answer)

As icon

on page 261). The type argument allows us to control the label of the button that quits the dialog window. Typical values are ok for a button with text OK, okcancel for two buttons with text OK and Cancel, yesno for two buttons with text Yes and No, and yesnocancel for three buttons with text Yes, No, and Cancel. The return value stored in answer can be used to take appropriate actions (values of answer are typically ’ok’, ’yes’, ’no’, ’cancel’). We see that the Message widget is a generalization of the askokcancel and askyesno functions.

Error messages may be displayed by the tkMessageBox.showerror function: tkMessageBox.showerror(title=’Error’, message=’invalid number’)

Run pydoc tkMessageBox to see the documentation of the various functions in that module.

Pmw provides several convenient and flexible dialog widgets. The Pmw message dialog entry of the Dialog pulldown menu in demoGUI.py activates Pmw’s MessageDialog widget shown in Figure 6.19c.

Fig.6.19. The dialog menu in Figure 6.17b on page 267 has three items for demonstrating typical message boxes: (a) Tk confirmation dialog (made by tkMessage.askokcancel); (b) Tk message dialog (made by tkMessage.Message); (c) Pmw message dialog (made by Pmw.MessageBox).

def Pmw\_message\_dialog(self): message = """\

This is a demo of the Pmw.MessageDialog box, which is useful for writing longer text messages to the user."""

Pmw.MessageDialog(self.master, title=’Description’, buttons=(’Quit’,), message\_text=message,

message\_justify=’left’, message\_font=’helvetica 12’, icon\_bitmap=’info’,

# must be present if icon\_bitmap is: iconpos=’w’)

The MessageDialog class is composed of a Tk label widget for showing the message6 and button widgets. The label component’s keyword arguments are the same as for the constructor of class Label, except that they are prefixed by a message\_ string. The justify argument of a Label controls how multiple lines are typeset. By default, all lines are centered, while we here demand them to be justified to the left. The icon\_bitmap values can be one of the names of the predefined bitmaps (see Figure 6.16b on page 261).

6.3.16 User-Defined Dialogs

Pmw offers a Dialog widget for user-defined dialog boxes. The user can insert any set of widgets and specify a set of control buttons. This makes it easy to tailor a dialog to one’s specific needs. Figure 6.20 shows such a dialog box,

6 That is why we need explicit newlines in the message text.

launched from the Pmw user-defined dialog entry of the Dialog menu. Clicking on this menu entry activates the self.userdef\_dialog function, which creates a Pmw Dialog widget and fills it with entries: an option menu, a slider, and a check button. Fortunately, all these widgets are created and packed properly by class InputFields (see Chapter 6.3.9).

Fig.6.20. A user-defined Pmw dialog (made by Pmw.Dialog). The dialog arises from clicking on the Pmw user-defined dialog item in the menu in Figure 6.17b on page 267.

def userdef\_dialog(self):

self.userdef\_d = Pmw.Dialog(self.master,

title=’Programmer-Defined Dialog’, buttons=(’Apply’, ’Cancel’), #defaultbutton=’Apply’, command=self.userdef\_dialog\_action)

self.userdef\_d\_gui = InputFields(self.userdef\_d.interior(), self.status\_line,

self.balloon, scrolled=True)

self.userdef\_d\_gui.pack()

The Pmw.Dialog widget’s interior frame, which we can use as parent widget, is accessed through the interior() method. Upon clicking one of the buttons, in the present case Apply or Cancel, the self.userdef\_dialog\_action method is called. In this method we can extract the user’s input. Here we only present the skeleton of such a method:

def userdef\_dialog\_action(self, result):

# result contains the name of the button that we clicked if result == ’Apply’:

# example on extracting dialog variables:

case = self.userdef\_d\_gui.case.get() else:

text = ’you just canceled the dialog’

self.status\_line.configure(text=text) self.userdef\_d.destroy() # destroy dialog window

.3.17 Color-Picker Dialogs

Full-fledged graphical applications often let the user change background and foreground colors. Picking the right color is most conveniently done in a dialog where one can experiment with color compositions in an interactive way. A basic Tk dialog, accessible through the tkColorChooser module from Python scripts, is launched from the Tk color dialog entry in the Color dialogs submenu of the Dialog pulldown menu. Selecting this entry calls the following function, which runs the dialog and changes the background color:

def tk\_color\_dialog(self):

import tkColorChooser

color = tkColorChooser.Chooser(

initialcolor=’gray’,title=’Choose background color’).show() # or:

* # color = tkColorChooser.askcolor()
* # color[0] is now an (r,g,b) tuple and
* # color[1] is a hexadecimal number; send the latter to
* # tk\_setPalette to change the background color:
* # (when Cancel is pressed, color is (None,None)) if color[0] is not None:  
     
  self.master.tk\_setPalette(color[1])  
   text = ’new background color is ’ + str(color[0]) + \  
     
  ’ (rgb) or ’ + str(color[1]) self.status\_line.configure(text=text)  
     
  A snapshot of the color-picker dialog is shown in Figure 6.21. We mention that the tk\_setPalette method with a more sophisticated argument list can be used to change the whole color scheme for an application (see the man pages for more information).  
     
  Information on tkColorChooser and other modules not included in the Tkinter module can be found in the source files of these modules in the Lib/lib-tk directory of the Python source code distribution.  
     
  There is a more sophisticated color editor that comes with Python, called Pynche and located in the Tools/pynche directory of the Python source. At the time of this writing, you need to install Pynche manually by copying Tools/pynche to some directory where Python can find modules (see Ap- pendix B.1) or include the path of the Tools directory in PYTHONPATH. The README file in the pynche directory describes the nice features of this color- picker tool.

6.3.18 File Selection Dialogs

File dialogs are used to prompt the user for a filename, often combined with browsing of existing filenames and directories, see Figure 6.17d–e. A module tkFileDialog provides access to basic Tk file dialogs for loading and saving files. The class Open is used for asking the user about a filename for loading:

Fig. 6.21. The entry Color dialogs in the Dialogs menu launches a new pulldown menu with an entry Tk color dialog whose resulting dialog box is displayed above. The Tk color dialog is made by the tkColorChooser module.

import tkFileDialog

fname = tkFileDialog.Open(filetypes=[(’anyfile’,’\*’)]).show() if fname:

f = open(fname, ’r’) ...

The filetypes argument allows us to specify a family of relevant files, here called anyfile, and a glob-style (Unix shell-style wildcard) specification of the filenames. The call to show pops up a separate window containing icons of all the files specified by filetypes in the current directory, see Figure 6.17e. In the present example all files and directories are shown. You can click on an icon and then on Open. The window is then closed, and the chosen filename is returned as a string, here stored in fname. It is not possible to return from the file dialog before a valid filename is provided, but pressing Cancel returns an empty string (that is why we make the test if fname). Do not forget the show call, without it no file dialog is shown!

The filetypes list is used to specify the type of files that are to be dis- played in the dialog. For instance,

filetypes=[(’datafiles’,’\*.dat’),(’gridfiles’,’\*.grid’)]

makes the dialog show the names of either all \*.dat files or all \*.grid files. Through an option menu in the dialog the user can choose which of these two classes of files that should be displayed.

The tkFileDialog also contains a SaveAs class for fetching an output file- name. The usage is the same as for the Open class (Figure 6.17f displays the layout of the dialog):

fname = tkFileDialog.SaveAs(

filetypes=[(’temporary files’,’\*.tmp’)],

initialfile=’myfile.tmp’, title=’Save a file’).show()

if fname:

f = open(fname, ’w’) ...