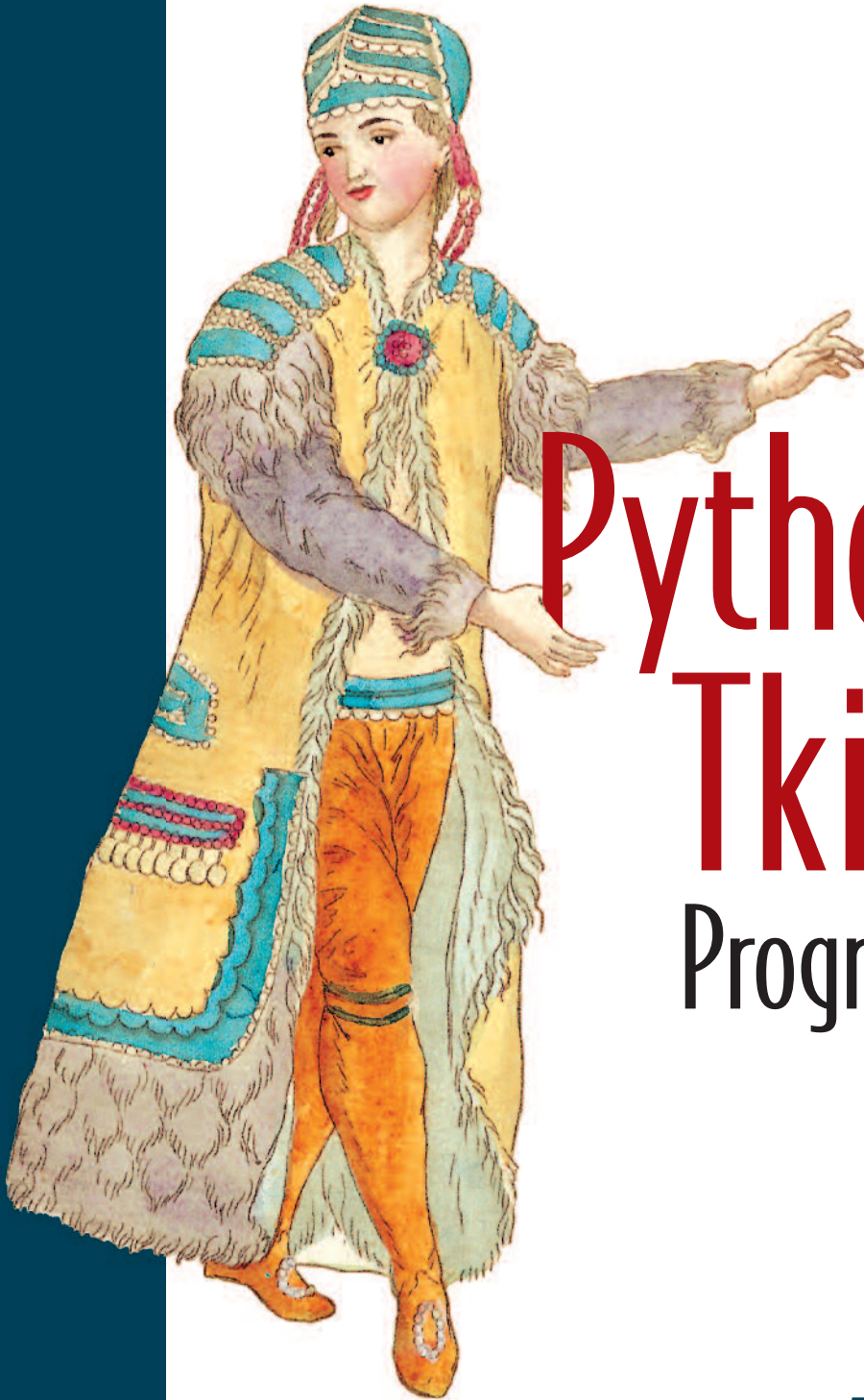


Graphical user interfaces for Python programs

John E. Grayson



Python and Tkinter Programming

Python and Tkinter Programming

Python and Tkinter Programming

JOHN E. GRAYSON



MANNING

Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, go to www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
32 Lafayette Place
Greenwich, CT 06830

Fax: (203) 661-9018
email: orders@manning.com

©2000 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end.

 Manning Publications Co.
32 Lafayette Place
Greenwich, CT 06830

Copyeditor: Kristen Black
Typesetter: Dottie Marsico
Cover designer: Leslie Haimes

Second corrected printing 2000
Printed in the United States of America
2 3 4 5 6 7 8 9 10 – CM – 03 02 01 00

To the memory of Backy, who taught me the value of language.

brief contents

preface xv
special thanks xvii
about the reader xix
about the author xx
conventions xxi
about the cover xxii
author online xxiii

Part I Basic concepts 1

- 1 Python 3*
- 2 Tkinter 12*
- 3 Building an application 18*

Part 2 Displays 29

- 4 Tkinter widgets 31*
- 5 Screen layout 77*
- 6 Events, bindings and callbacks 95*
- 7 Using classes, composites and special widgets 120*
- 8 Dialogs and forms 140*
- 9 Panels and machines 199*
- 10 Drawing blobs and rubber lines 237*
- 11 Graphs and charts 276*
- 12 Navigation 300*
- 13 The window manager 306*

Part 3 Putting it all together... 311

- 14 Extending Python 313*
- 15 Debugging applications 329*
- 16 Designing effective graphics applications 338*
- 17 Programming for performance 348*
- 18 Threads and asynchronous techniques 361*
- 19 Distributing Tkinter applications 374*

Part 4 Appendices 381

- appendix A Mapping Tk to Tkinter 383*
- appendix B Tkinter reference 425*
- appendix C Pmw reference: Python megawidgets 542*
- appendix D Building and installing Python, Tkinter 610*
- appendix E Events and keysyms 617*
- appendix F Cursors 621*
- appendix G References 625*
- index 629*

contents

preface xv
special thanks xvii
about the reader xix
about the author xx
conventions xxi
about the cover xxii
author online xxiii

Part I Basic concepts 1

1 Python 3

- 1.1 Introduction to Python programming and a feature review 3
Why Python? 4, Where can Python be used? 5
- 1.2 Key data types: lists, tuples and dictionaries 5
Lists 5, Tuples 7, Dictionaries 8
- 1.3 Classes 9
How do classes describe objects? 9, Defining classes 9, Neat Python trick #10 9,
Initializing an instance 10, Methods 10, Private and public variables and
methods 11, Inheritance 11, Multiple inheritance 11, Mixin classes 11

2 Tkinter 12

- 2.1 The Tkinter module 12
What is Tkinter? 12, What about performance? 13, How do I use
Tkinter? 13, Tkinter features 14
- 2.2 Mapping Tcl/Tk to Tkinter 14
- 2.3 Win32 and UNIX GUIs 15

- 2.4 Tkinter class hierarchy 16
- 2.5 Tkinter widget appearance 17

3 *Building an application 18*

- 3.1 Calculator example: key features 21
- 3.2 Calculator example: source code 21
- 3.3 Examining the application structure 27
- 3.4 Extending the application 28

Part 2 Displays 29

4 *Tkinter widgets 31*

- 4.1 Tkinter widget tour 31
Toplevel 32, Frame 33, Label 35, Button 36, Entry 37,
Radiobutton 37, Checkbutton 38, Menu 39, Message 42, Text 43,
Canvas 44, Scrollbar 45, Listbox 45, Scale 46
- 4.2 Fonts and colors 47
Font descriptors 47, X Window System font descriptors 47, Colors 48,
Setting application-wide default fonts and colors 49
- 4.3 Pmw Megawidget tour 49
AboutDialog 50, Balloon 50, ButtonBox 51, ComboBox 52,
ComboBoxDialog 53, Counter 54, CounterDialog 55, Dialog 56,
EntryField 56, Group 57, LabeledWidget 58, MenuBar 59,
MessageBar 59, MessageDialog 61, NoteBookR 61, NoteBookS 62,
NoteBook 63, OptionMenu 64, PanedWidget 65, PromptDialog 66,
RadioSelect 66, ScrolledCanvas 67, ScrolledField 68, ScrolledFrame 69,
ScrolledListbox 70, ScrolledText 70, SelectionDialog 71, TextDialog 72,
TimeCounter 73
- 4.4 Creating new megawidgets 73
Description of the megawidget 73, Options 74, Creating the
megawidget class 74

5 *Screen layout 77*

- 5.1 Introduction to layout 77
Geometry management 78
- 5.2 Packer 79
Using the expand option 82, Using the fill option 82, Using the padx and
pady options 84, Using the anchor option 84, Using hierarchical packing 84
- 5.3 Grid 86
- 5.4 Placer 90
- 5.5 Summary 94

6 *Events, bindings and callbacks 95*

- 6.1 Event-driven systems: a review 95
What are events? 96, Event propagation 97, Event types 97
- 6.2 Tkinter events 98
Events 98
- 6.3 Callbacks 102
- 6.4 Lambda expressions 103
Avoiding lambdas altogether 103
- 6.5 Binding events and callbacks 104
Bind methods 104, Handling multiple bindings 106
- 6.6 Timers and background procedures 107
- 6.7 Dynamic callback handlers 107
- 6.8 Putting events to work 108
Binding widgets to dynamic data 108, Data verification 111,
Formatted (smart) widgets 117
- 6.9 Summary 119

7 *Using classes, composites and special widgets 120*

- 7.1 Creating a Light Emitting Diode class 120
Let's try that again 126, What has changed? 129
- 7.2 Building a class library 129
Adding a hex nut to our class library 131, Creating a switch class 133,
Building a MegaWidget 136
- 7.3 Summary 139

8 *Dialogs and forms 140*

- 8.1 Dialogs 141
Standard dialogs 141, Data entry dialogs 142, Single-shot forms 146,
Tkinter variables 152
- 8.2 A standard application framework 155
- 8.3 Data dictionaries 165
- 8.4 Notebooks 172
- 8.5 Browsers 175
- 8.6 Wizards 184
- 8.7 Image maps 191
- 8.8 Summary 198

9 *Panels and machines 199*

- 9.1 Building a front panel 199
- 9.2 Modularity 201

9.3	Implementing the front panel	201
9.4	GIF, BMP and overlays	215
9.5	And now for a more complete example	220
9.6	Virtual machines using POV-Ray	232
	And now for something completely different... #10 The Example	233
9.7	Summary	236
10	<i>Drawing blobs and rubber lines</i>	237
10.1	Drawing on a canvas	238
	Moving canvas objects	243
10.2	A more complete drawing program	244
10.3	Scrolled canvases	251
10.4	Ruler-class tools	254
10.5	Stretching canvas objects	258
10.6	Some finishing touches	262
10.7	Speed drawing	271
10.8	Summary	275
11	<i>Graphs and charts</i>	276
11.1	Simple graphs	276
11.2	A graph widget	279
	Adding bargraphs	286,
	Pie charts	289
11.3	3-D graphs	292
11.4	Strip charts	296
11.5	Summary	298
12	<i>Navigation</i>	300
12.1	Introduction: navigation models	300
12.2	Mouse navigation	301
12.3	Keyboard navigation: “mouseless navigation”	301
12.4	Building navigation into an application	302
12.5	Image maps	305
12.6	Summary	305
13	<i>The window manager</i>	306
13.1	What is a window manager?	306
13.2	Geometry methods	307
13.3	Visibility methods	308
13.4	Icon methods	309

- 13.5 Protocol methods 309
- 13.6 Miscellaneous wm methods 310

Part 3 Putting it all together... 311

14 Extending Python 313

- 14.1 Writing a Python extension 313
- 14.2 Building Python extensions 316
 - Linking an extension statically in UNIX 316, Linking an extension statically in Windows 317, Building a dynamic module in UNIX 317, Building a dynamic module in Windows 318, Installing dynamic modules 319, Using dynamic modules 319
- 14.3 Using the Python API in extensions 319
- 14.4 Building extensions in C++ 320
- 14.5 Format strings 321
- 14.6 Reference counts 324
- 14.7 Embedding Python 325
- 14.8 Summary 328

15 Debugging applications 329

- 15.1 Why print statements? 329
- 15.2 A simple example 330
- 15.3 How to debug 333
- 15.4 A Tkinter explorer 334
- 15.5 pdb 336
- 15.6 IDLE 336
- 15.7 DDD 337

16 Designing effective graphics applications 338

- 16.1 The elements of good interface design 339
- 16.2 Human factors 342
 - Choosing fonts 343, Use of color in graphical user interfaces 344, Size considerations 346
- 16.3 Alternative graphical user interfaces 346
- 16.4 Summary 347

17 Programming for performance 348

- 17.1 Everyday speedups 348
 - Program organization 349, Using the Python optimizer 350, Examining code 350

17.2	Tkinter performance 350
	Keep it short! 350, Eliminate local variables 351, Keep it simple 351, Fast initialization 352, Throttling events 352
17.3	Python techniques 352
	Importing modules 353, Concatenating strings 353, Getting nested loops right 354, Eliminate module references 354, Use local variables 355, Using exceptions 356, Using map, filter and reduce 356
17.4	Application profiling 357
17.5	Python extensions 359
17.6	Summary 360
18	<i>Threads and asynchronous techniques 361</i>
18.1	Threading 361
	Non-GUI threads 362, GUI threads 365
18.2	“after” processing 369
18.3	Summary 373
19	<i>Distributing Tkinter applications 374</i>
19.1	General issues in distributing applications 374
19.2	Distributing UNIX applications 375
19.3	Distributing Win32 applications 376
19.4	Python distribution tools 379
Part 4	<i>Appendices 381</i>
	<i>appendix A Mapping Tk to Tkinter 383</i>
	<i>appendix B Tkinter reference 425</i>
	<i>appendix C Pmw reference: Python megawidgets 542</i>
	<i>appendix D Building and installing Python, Tkinter 610</i>
	<i>appendix E Events and keysyms 617</i>
	<i>appendix F Cursors 621</i>
	<i>appendix G References 625</i>
	<i>index 629</i>

Example_8_6.py

```
from Tkinter import *
from tkSimpleDialog import Dialog
import Pmw

class MixedWidgets(Dialog):
    def body(self, master):
        Label(master, text='Select Case:').grid(row=0, sticky=W)
        Label(master, text='Select Type:').grid(row=1, sticky=W)
        Label(master, text='Enter Value:').grid(row=2, sticky=W)

        self.combo1 = Pmw.ComboBox(master,
                                   scrolledlist_items=("Upper", "Lower", "Mixed"),
                                   entry_width=12, entry_state="disabled",
                                   selectioncommand = self.ripple)
        self.combo1.selectitem("Upper")
        self.combo1.component('entry').config(bg='gray80')

        self.combo2 = Pmw.ComboBox(master, scrolledlist_items=(),
                                   entry_width=12, entry_state="disabled")
        self.combo2.component('entry').config(background='gray80')

        self.entry1 = Entry(master, width = 12)

        self.combo1.grid(row=0, column=1, sticky=W)
        self.combo2.grid(row=1, column=1, sticky=W)
        self.entry1.grid(row=2, column=1, sticky=W)

        return self.combo1

    def apply(self):
        c1 = self.combo1.get()
        c2 = self.combo2.get()
        e1 = self.entry1.get()
        print c1, c2, e1

    def ripple(self, value):
        lookup = {'Upper': ("ANIMAL", "VEGETABLE", "MINERAL"),
                  'Lower': ("animal", "vegetable", "mineral"),
                  'Mixed': ("Animal", "Vegetable", "Mineral")}
        items = lookup[value]
        self.combo2.setlist(items)
        self.combo2.selectitem(items[0])

root = Tk()
dialog = MixedWidgets(root)
```

Code comments

- 1 ComboBoxes are important widgets for data entry and selection. One of their most valuable attributes is that they occupy little space, even though they may give the user access to an unlimited number of selectable values.

```
self.combo1 = Pmw.ComboBox(master,
                           scrolledlist_items=("Upper", "Lower", "Mixed"),
```

In this case, we are just loading three values into the combo's list. Typically data may be either loaded from databases or calculated.

- ❷ We do not intend for the values selected in this `ComboBox` to be editable, so we need to disable the entry field component of the widget.

```
entry_width=12, entry_state="disabled",
self.combo1.component('entry').config(bg='gray80')
```

We set the background of the `Entry` widget to be similar to the background to give the user a clear indication that the field is not editable.

- ❸ This one is an unusual one. Frequently, fields on a screen are dependent on the values contained within other fields on the same screen (on other screens in some cases). So, if you change the value in the combobox, you *ripple* the values within other widgets. (Ripple is a term that I invented, but it somewhat conveys the effect you can see as the new values ripple through the interface.)

```
selectioncommand = self.ripple)
```

Note Careless use of the ripple technique can be dangerous! Using ripple must be considered carefully, since it is quite easy to design a system which results in constant value modification if several fields are dependent on each other. Some sort of control flag is necessary to prevent a continuous loop of `selectioncommand` callbacks consuming CPU cycles.

See “Tkinter performance” on page 350 for other important factors you should consider when designing an application.

- ❹ We select default value from the lists or else the entry would be displayed as blank, which is probably not appropriate for a non-editable combobox.

```
self.combo1.selectitem("Upper")
```

- ❺ This is our ripple callback function. The `selectioncommand` callback returns the value of the item selected as an argument. We use this to look up the list to be applied to the second combobox:

```
def ripple(self, value):
    lookup = {'Upper': ("ANIMAL", "VEGETABLE", "MINERAL"),
             'Lower': ("animal", "vegetable", "mineral"),
             'Mixed': ("Animal", "Vegetable", "Mineral")}
    items = lookup[value]
```

- ❻ The list obtained from the lookup replaces the current list.

```
self.combo2.setlist(items)
self.combo2.selectitem(items[0])
```

As before, you need to select one of the values in the lists to be displayed in the widget.

If you run `Example_8_6.py`, you will see this simple example of rippled widgets. Part of the effect can be seen in figure 8.6.

up an *atomic** operation, the result of another thread executing the same code path can be disastrous. If it is important that threads maintain *atomic* operations, you need to use *locks* (*semaphores* or *mutexes*†) to prevent another thread from gaining access to shared data. You can find some examples of locking in the standard Python distribution in the `Demo/threads` directory.

18.1.2 GUI threads

The decision to use threads with a GUI must be made with some caution. In general, a system that uses threads must ensure that the GUI is updated from within the main thread, which includes the mainloop. The following example is an adaptation of `wpi.py`, of one of the threading examples in the standard Python distribution. The example has a main thread which creates a GUI and starts the mainloop, and a thread which continuously calculates successive digits of `pi`. In addition, the example uses Tkinter's event loop to update the GUI periodically.

thread2.py

```
# Display digits of pi in a window, calculating in a separate thread.
# Compare with wpi.py in Demo/threads/wpi.py.

import sys
import time
import thread
from Tkinter import *

class ThreadExample:
    def __init__(self, master=None):
        self.ok = 1
        self.digits = []
        self.digits_calculated = 0
        self.digits_displayed = 0
        self.master = master

        thread.start_new_thread(self.worker_thread, ()) ❶

        self.frame = Frame(master, relief=RAISED, borderwidth=2)
        self.text = Text(self.frame, height=26, width=50)
        self.scroll = Scrollbar(self.frame, command=self.text.yview)
        self.text.configure(yscrollcommand=self.scroll.set)
        self.text.pack(side=LEFT)
        self.scroll.pack(side=RIGHT, fill=Y)
        self.frame.pack(padx=4, pady=4)
        Button(master, text='Close', command=self.shutdown).pack(side=TOP)

        self.master.after(100, self.check_digits) ❷

    def worker_thread(self):
        k, a, b, al, bl = 21, 41, 11, 121, 41
        while self.ok:
```

* Atomic operations complete in their entirety before any other process or thread can follow the same code path.

† Mutual exclusion locks.

```

        # Next approximation
        p, q, k = k*k, 21*k+11, k+11
        a, b, a1, b1 = a1, b1, p*a+q*a1, p*b+q*b1
        # Print common digits
        d, d1 = a/b, a1/b1
    while d == d1:
        self.digits.append(`int(d)`)
        a, a1 = 101*(a%b), 101*(a1%b1)
        d, d1 = a/b, a1/b1
        time.sleep(0.001)

def shutdown(self):
    self.ok = 0
    self.master.after(100, self.master.quit)

def check_digits(self):
    self.digits_calculated = len(self.digits)
    diff = self.digits_calculated - self.digits_displayed
    ix = self.digits_displayed
    for i in range(diff):
        self.text.insert(END, self.digits[ix+i])
    self.digits_displayed = self.digits_calculated
    self.master.title('%d digits of pi' % self.digits_displayed)

    self.master.after(100, self.check_digits)

root = Tk()
root.option_readfile('optionDB')
example = ThreadExample(root)
root.mainloop()

```

Code comments

- ❶ The `worker_thread`, which continuously updates digits of `pi`, is started up as before, with `start_new_thread`.
- ❷ We are using `after` to place a timed event on the event queue. Using this technique allows a delay without blocking the mainloop.
- ❸ Again, a small `sleep` is added at the end of the `worker_thread`'s loop to ensure that the mainloop gets to spin (see below for more detail).
- ❹ To ensure an orderly shutdown of the application, we set the `ok` flag, which will cause the `worker_thread` to exit and wait for 100 ms before killing the GUI. A more realistic application might need additional methods.
- ❺ `after` events are one-shot events; you must put an event back on the queue if the timed event is to be repetitive.

If you run `thread2.py`, you will see a screen similar to figure 18.4. Digits are appended to the text widget at approximately 100ms intervals. You may wish to experiment and comment out the `sleep` in the `worker_thread`. When you run the code, you will see that the application takes much longer to respond to a click on the Close button—the `worker_thread` gets a complete timeslice to run before getting suspended, so the event queue for GUI events gets drained less frequently.