

FontSelect: A Tkinter widget to select fonts



John W. Shipman

2009-10-10 16:42

Abstract

Describes a graphical user interface widget that displays the fonts available with the Python programming language's Tkinter widget set.

This publication is available in Web form¹ and also as a PDF document². Please forward any comments to tcc-doc@nmt.edu.

Table of Contents

1. Introduction	2
2. Controls in the <i>FontSelect</i> widget	2
2.1. Design notes	4
3. Using the <i>FontSelect</i> widget in your Tkinter application	4
4. fontselector: A small test driver	5
4.1. The Application class	5
4.2. Application.__init__(): Constructor	5
4.3. Application.__createWidgets(): Widget layout	6
4.4. Application.__callback(): Observer function	6
4.5. Main program	6
5. fontselect.py: The module	6
5.1. class FontSelect	7
5.2. FontSelect.__init__(): Constructor	9
5.3. FontSelect.__createWidgets(): Widget placement	10
5.4. FontSelect.__familyHandler(): Callback for family changes	11
5.5. FontSelect.__controlHandler(): Callback for any font change	11
5.6. FontSelect.get(): Return the current font	11
5.7. FontSelect.getName(): Return the current font name	12
5.8. FontSelect.addObserver(): Register a callback	12
6. class FamilyPicker: Widgets for selecting the font family	12
6.1. FamilyPicker.__init__(): Constructor	13
6.2. FamilyPicker.__listboxHandler()	14
7. class Controls: Controls frame	14
7.1. Controls.__init__(): Constructor	16
7.2. Controls.__createWidgets()	16
7.3. Controls.__createButtons(): Lay out the small controls frame	18
7.4. Controls.__setFont(): Font change handler	20
7.5. Controls.setFamily(): Change font family name	22
7.6. Controls.getName(): Get the current font name	22

¹ <http://www.nmt.edu/tcc/help/lang/python/examples/fontselect/>

² <http://www.nmt.edu/tcc/help/lang/python/examples/fontselect/fontselect.pdf>

1. Introduction

This document describes a graphical user interface widget that allows the user to select a particular text font. This widget works with the Tkinter graphical user interface for the Python programming language.

The document also contains the actual code of the widget, explained in a lightweight literate programming style.

Relevant documents:

- *Python 2.2 quick reference*³ describes the Python language.
- *Tkinter reference: a GUI for Python*⁴ describes the Tkinter widget set.
- See the author's *Lightweight Literate Programming* page⁵ for an explanation of the technique of embedding the program in its internal documentation.
- See also the author's page on the *Cleanroom software methodology*⁶ for a discussion of the author's preferred style of implementation.

Files extracted from this document:

- `fontselect.py`⁷: Module containing *FontSelect*.
- `fontselector`⁸: Test driver.

2. Controls in the *FontSelect* widget

Here is a screen shot of the widget as it appears in Section 4, “fontselector: A small test driver” (p. 5).

³ <http://www.nmt.edu/tcc/help/pubs/python22/>

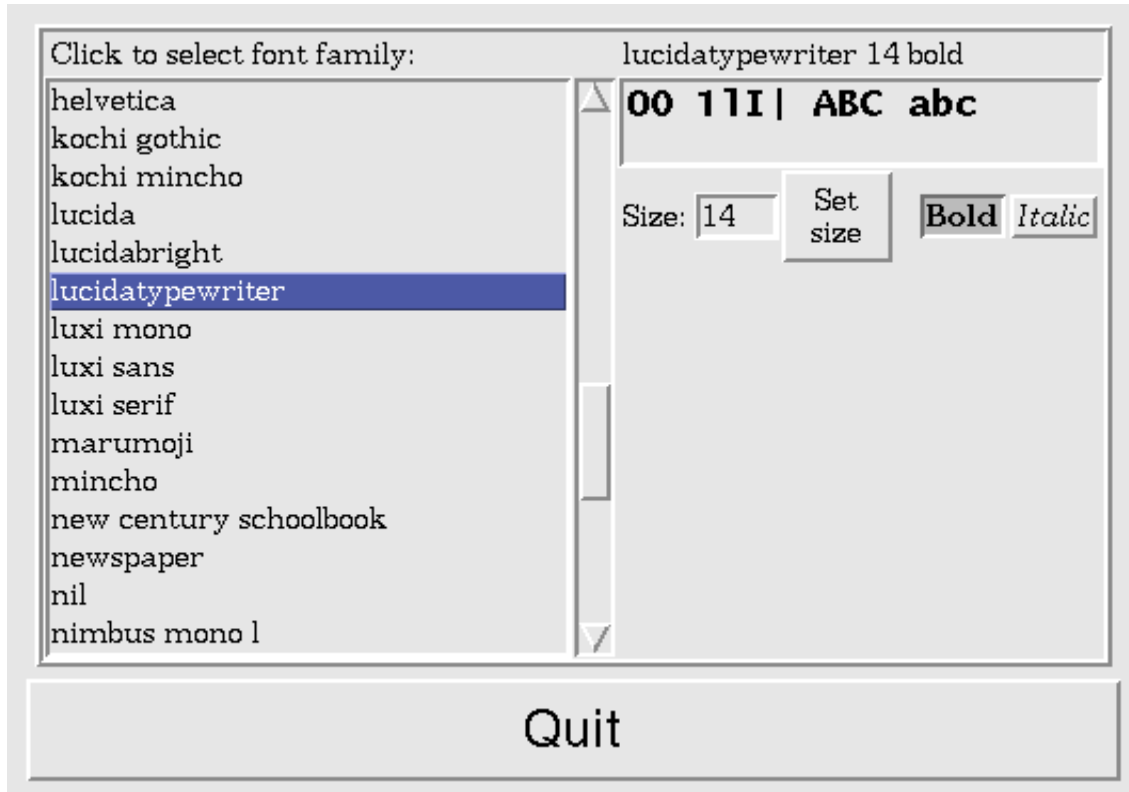
⁴ <http://www.nmt.edu/tcc/help/pubs/tkinter/>

⁵ <http://www.nmt.edu/~shipman/soft/litprog/>

⁶ <http://www.nmt.edu/~shipman/soft/clean/>

⁷ <http://www.nmt.edu/tcc/help/lang/python/examples/fontselect/fontselect.py>

⁸ <http://www.nmt.edu/tcc/help/lang/python/examples/fontselect/fontselector>



These are the components of a *FontSelect* widget:

family picker

A scrollable *Listbox* widget containing the names of all the font families such as Times, Courier, and so forth. The label above this widget advises the user to click to select a font family.

annunciator

A *Label* field that displays the full name of the currently selected font, e.g., “times 16 bold italic”. Because not all fonts are available in all sizes, this field will reflect the actual font closest to the selected values, not the selected values themselves.

sample text

A *Label* field that displays a few characters in the selected font. The user can add or edit text in this field. The initial contents are: zero, letter O, digit 1, lowercase letter l, capital letter I, and the vertical bar symbol “|”. **Beware of fonts that make it difficult or impossible to tell all these characters apart!**

size field and button

A text *Entry* field for the font size in pixels, and an accompanying *Button* labeled “Set” that applies the font size value. The user can also apply the font size by pressing *Enter* or *Tab* in that field.

bold button

A push-push *Checkbutton* to specify whether the font is boldfaced or not.

italic button

A push-push *Checkbutton* to specify whether the font is italic or not.

2.1. Design notes

In the author's opinion, the most obvious design tradeoff is in the size of the listbox of family names. At this writing, there were over sixty names.

- Making the listbox larger makes life easier for the user, who can see more names at once, and needs to do less scrolling to see all the names.
- On the other hand, this widget may be included in a larger application where space is at a premium.

The design shown above can be adapted to smaller or larger screen sizes by changing the size of the listbox. Accordingly, in the widget constructor, we will allow the user to specify that size (overriding some reasonable default).

3. Using the *FontSelect* widget in your Tkinter application

If you are writing a Tkinter application and you want to let the user change one or more fonts, you can include a *FontSelect* widget in your application. Simply create it with this constructor:

```
w = FontSelect ( master, font=f, listCount=L, observer=F )
```

w

The constructor returns a new, ungridded *FontSelect* widget.

master

The parent *Frame* widget in which this widget will be mastered.

font=f

If you don't like the font used inside this widget, supply this optional keyword argument with *f* set to your preferred font as a `tkFont.Font` object.

listCount=L

By default, the number of families displayed in the family *Listbox* is 15. If you want a different number, pass that number as an integer *L* to this optional keyword argument. This *Listbox* pretty much determines the overall height of the *FontSelect* widget on the screen.

observer=F

An "observer" function to be called whenever any of the currently displayed font options change. The calling sequence is:

```
F ( newFont )
```

where *newFont* is a Python `tkFont.Font` object representing the currently selected font.

Here are the public attributes and methods of a *FontSelect* object:

.scrollList

A *ScrollList* widget containing the family names and associated scrollbar.

.get()

Returns the currently selected font as a Python `tkFont.Font` object. If no font has ever been set, returns `None`.

.getName()

Returns a string describing all the current font options. This string starts with the family and size, followed optionally by the strings "bold" and/or "italic". Example: "luxi mono 14 bold italic". If no font has ever been set, returns `None`.

.addObserver (f)

Adds an observer function *f* that will be called whenever any of the font options change. The calling sequence of the *f* function is as described under the `observer=f` argument to the *FontSelect* constructor.

4. fontselector: A small test driver

Here is a small Tkinter application that you can use to try out the *FontSelect* widget.

The driver starts with a prologue making it self-executing (under Unix), and a comment that points back to this documentation.

```
fontselector
#!/usr/bin/env python
#=====
# fontselector: Test driver for FontSelect
#
# For documentation, see:
#   http://www.nmt.edu/tcc/help/lang/python/examples/fontselect/
#-----
from Tkinter import *
import tkFont
import fontselect
```

4.1. The Application class

Next we start declaring the *Application* class, which embodies the entire graphical user interface.

```
fontselector
class Application(Frame):
    """GUI for fontselector

    State/Invariants:
        .fontSelect: [ self's FontSelect widget ]
    """
```

4.2. Application.__init__(): Constructor

The constructor is quite pro forma: call the parent class constructor, register the application with its `.grid()` method, create a font, create the widgets, and wait in the main loop.

```
fontselector
def __init__ ( self ):
    """Constructor for the Application class.
    """
    Frame.__init__ ( self, None )
    self.buttonFont = tkFont.Font ( family="Helvetica", size=20 )
    self.grid ( padx=5, pady=5 )
    self.__createWidgets()
```

4.3. Application. `__createWidgets()`: Widget layout

There are only two widgets: the *FontSelect* widget we are testing, and a quit button below it.

fontselector

```
def __createWidgets ( self ):  
    """Lay out the widgets.  
    """  
    self.fontSelect = fontselect.FontSelect ( self,  
        listCount=10,  
        observer=self.__callback )  
    self.fontSelect.grid ( row=0, column=0, padx=5, pady=5 )  
  
    self.quitButton = Button ( self, text="Quit",  
        font=self.buttonFont,  
        command=self.quit )  
    self.quitButton.grid ( row=1, column=0, columnspan=99,  
        sticky=E+W, padx=5, pady=5 )
```

4.4. Application. `__callback()`: Observer function

This is the observer function that is called whenever any font option is changed.

fontselector

```
def __callback ( self, font ):  
    """Observer for font changes.  
  
    [ sys.stdout += current font name from self.fontSelect ]  
    """  
    print "Change to:", self.fontSelect.getName()
```

4.5. Main program

This is the usual main program for a Tkinter application: instantiate an *Application* object, set up the application's title in the window frame, and wait for events.

fontselector

```
#===== #  
# Main program #  
#----- #  
  
app = Application()  
app.master.title ( "fontselector" )  
app.mainloop()
```

5. fontselect.py: The module

Here we begin the literate exposition of the actual code. First is a prologue directing the reader back to this documentation.

```

"""fontselect.py: FontSelect, a font selector widget for Tkinter.

   For documentation, see:
   http://www.nmt.edu/tcc/help/lang/python/examples/fontselect/
   """

```

Next, the imported modules: the Tkinter widget set, and its related tkFont module containing the font-related functions. The Dialog module is another Tkinter accessory module that makes it easy to construct a pop-up dialog window.

```

#=====
# Imports
#-----

from Tkinter import *
import tkFont
import Dialog

```

We also use a ScrolledList widget, described elsewhere, to hold the list of families and its associated scrollbar. See the separate document, *ScrolledList: A Tkinter scrollable list widget*⁹.

```
import scrolledlist
```

The manifest constants LISTBOX_HEIGHT and LISTBOX_WIDTH are the default number of lines and line widths in the font family listbox. SAMPLE_TEXT is the initial sample text, and DEFAULT_SIZE is the starting text size.

```

#=====
# Manifest constants
#-----

LISTBOX_HEIGHT = 15
LISTBOX_WIDTH  = 30
SAMPLE_TEXT    = "00 1lI| ABC abc"
DEFAULT_SIZE   = 14
DEFAULT_FAMILY = "helvetica"

```

5.1. class FontSelect

There are a lot of widgets inside a *FontSelect* widget. To simplify the spatial and logical organization, we won't deal with all the widgets at the top level. Instead, we'll use a "frames within frames within frames" technique so that the layout and control linkages at each level are relatively simple.

Hence, at this level, our grid plan has only two cells: the family listbox (and associated label) in row 0, column 0, and everything else in row 0, column 1.

⁹ <http://www.nmt.edu/tcc/help/lang/python/examples/scrolledlist/>

To make this division of function crystal-clear, we'll move the logic for each of those two groups into separate classes:

- class `FamilyPicker` contains the family listbox and associated label. It will have a callback linkage, a function that will be called each time the user clicks on a family name.
- class `Controls` contains everything else. It will have a method called `.setFont()` that changes the current family.

So the work of this class is very simple. It creates and grids the two sub-widgets. It provides the function that the `FamilyPicker` needs when the family is changed, and when that function is called, it just passes the newly chosen family down to the `Controls` sub-widget.

In order to make class `FontSelect` act like any other widget, we declare the class to inherit from Tkinter's `Frame` widget.

fontselect.py

```
class FontSelect(Frame):
    """A compound widget for selecting fonts.
```

Here are the Cleanroom intended functions for the constructor and methods, and declarations of exported attributes. The calling sequence for the `observer` function is discussed in Section 3, "Using the `FontSelect` widget in your Tkinter application" (p. 4).

fontselect.py

```
Exports:
FontSelect ( master, font=None, listCount=None, observer=None ):
    [ (master is a Frame or None) and
      (font is a tkFont.Font or None) and
      (listCount is an integer, defaulting to LISTBOX_HEIGHT) and
      (observer is a function to be called when the font
       changes) ->
        master := master with a new FontSelect widget added
                  but not gridded, with that font, list count,
                  and observer function
        return that widget ]
.scrollList:    [ a ScrolledList containing the family names ]
.get():
    [ if a font has been selected ->
      return that font as a tkFont.Font object
      else -> return None ]
.getName():
    [ if a font has been selected ->
      return a string describing the actual font
      else -> return None ]
.addObserver ( f ):
    [ self := self with a new observer function added ]
```

So that our subwidgets have access to the fonts created at this level, we make these fonts public attributes.

fontselect.py

```
.regularFont: [ a tkFont.Font for general purposes ]
.listFont:
    [ if a font option was passed to the constructor ->
      that option's value
      else -> self.regularFont ]
```

Attributes inside the class include the sub-widgets `.familyPicker` and `.controls`, and also three fonts we'll need.

fontselect.py

```
Internal widgets:
    .familyPicker:
        [ a FamilyPicker for picking the font family ]
    .controls:
        [ a Controls widget containing other controls ]

State/Invariants:
    .__listCount:
        [ if a listCount option was passed to the constructor ->
          that option's value
          else -> LISTBOX_HEIGHT ]
    .__observerList:
        [ a list containing all observer callback functions
          for self ]
"""
```

5.2. FontSelect.__init__(): Constructor

The constructor for a class that inherits from `Frame` is pretty stereotyped: call the parent class's constructor, then create the widgets.

fontselect.py

```
def __init__ ( self, master=None, font=None, listCount=None,
              observer=None ):
    """Constructor for the FontSelect widget.
    """

    #-- 1 --
    # [ master := master with a new Frame widget added
    #   self   := that Frame ]
    Frame.__init__ ( self, master, relief=RIDGE,
                    borderwidth=4 )
```

We export the usual font for controls as the `.regularFont` attribute.

fontselect.py

```
#-- 2 --
self.regularFont = tkFont.Font (
    family="new century schoolbook", size="13" )
```

Next, we process the various constructor arguments. If a font was passed in, we store it in `self.listFont`; the default value is the regular font.

fontselect.py

```
#-- 3 --
# [ if font is None ->
#   self.listFont := self.regularFont
#   else ->
#   self.listFont := font ]
self.listFont = font or self.regularFont
```

```

#-- 4 --
# [ if listCount is None ->
#     self.__listCount := LISTBOX_HEIGHT
# else ->
#     self.__listCount := listCount ]
self.__listCount = listCount or LISTBOX_HEIGHT

```

If an observer function was passed in, put it in `self.__observerList`.

fontselect.py

```

#-- 5 --
# [ if observer is not None ->
#     self.__observerList := [ observer ]
# else ->
#     self.__observerList := an empty list ]
self.__observerList = []
if observer:
    self.__observerList.append ( observer )

```

For widget creation, see Section 5.3, “FontSelect.__createWidgets(): Widget placement” (p. 10).

fontselect.py

```

#-- 6 --
# [ self := self with all widgets and control linkages ]
self.__createWidgets()

```

5.3. FontSelect.__createWidgets(): Widget placement

This method creates and grids the sub-widgets. It also sets up the control linkage that allows selection of a font family in the FamilyPicker to change the font displayed in the Controls.

fontselect.py

```

def __createWidgets ( self ):
    """Create sub-widgets.

    [ self is a Frame ->
      self := self with all widgets and control linkages ]
    """

```

The FamilyPicker constructor needs two fonts (regular and list) and our handler for change of family.

fontselect.py

```

#-- 1 --
# [ self := self with a new FamilyPicker widget added
#     and gridded that calls self.__familyHandler when
#     a family is selected
# self.familyPicker := that widget
# self.scrollList := the .scrollList attribute of
#                     that widget ]
self.familyPicker = FamilyPicker ( self, self.__familyHandler )
self.familyPicker.grid ( row=0, column=0, sticky=N )
self.scrollList = self.familyPicker.scrollList

#-- 2 --

```

```

# [ self := self with a new Controls widget added and
#       gridded, that calls self.__controlHandler
#       when the font changes
# self.controls := that widget
self.controls = Controls ( self, self.__controlHandler )
self.controls.grid ( row=0, column=1, sticky=N )

```

5.4. FontSelect.__familyHandler(): Callback for family changes

This method is called by the FamilyPicker widget whenever a new family is selected. Its argument is the new family name. All it does is pass that family name to the Controls.setFamily() widget.

fontselect.py

```

def __familyHandler ( self, newFamily ):
    """Handler for change of family name.

    [ newFamily is a string ->
      self.controls := self.controls with its font
        family set to newFamily ]
    """
    self.controls.setFamily ( newFamily )

```

5.5. FontSelect.__controlHandler(): Callback for any font change

This method is called by the Controls widget whenever any font option is changed. Its purpose is to notify the user's observer callback.

fontselect.py

```

def __controlHandler ( self, newFont ):
    """Handler for a changed font.

    [ newFont is a tkFont.Font object ->
      call each function in self.__observerList, passing
        newFont to each one ]
    """
    for observer in self.__observerList:
        observer ( newFont )

```

5.6. FontSelect.get(): Return the current font

This public method returns the current font as a tkFont.Font object.

fontselect.py

```

def get ( self ):
    """Return the current font.
    """
    return self.controls.getFont()

```

5.7. `FontSelect.getName()`: Return the current font name

Calls the method of the same name in the `Controls` widget.

fontselect.py

```
def getName ( self ):  
    """Return the current font name.  
    """  
    return self.controls.getName()
```

5.8. `FontSelect.addObserver()`: Register a callback

This method adds another observer function to `self.__observerList`, the list of functions to be called when the font changes.

fontselect.py

```
def addObserver ( self, observer ):  
    """Add another observer callback.  
    """  
    self.__observerList.append ( observer )
```

6. class `FamilyPicker`: Widgets for selecting the font family

This class is a compound widget that allows the user to select a font family. It inherits from `Frame` so that it can act as a widget.

fontselect.py

```
class FamilyPicker(Frame):  
    """Widget to select a font family.  
  
    Exports:  
    FamilyPicker ( fontSelect, observer=None ):  
    [ (fontSelect is a FontSelect widget) and  
      (observer is a function or None) ->  
      fontSelect := fontSelect with a new FamilyPicker  
      widget added but not gridded, with that  
      observer callback if given  
      return that FamilyPicker widget ]
```

This class has only two sub-widgets: a label for the list, and a `ScrolledList` widget that contains the list of family names.

fontselect.py

```
    Internal widgets:  
    .topLabel: [ a Label above the family list ]  
    .scrollList:  
    [ a ScrolledList containing the family names ]  
    """
```

6.1. FamilyPicker.__init__(): Constructor

As with any compound widget, we start by calling the parent class constructor.

fontselect.py

```
def __init__ ( self, fontSelect, observer ):  
    """Constructor for FamilyPicker  
    """
```

Note that we call the parent frame argument `fontSelect` and not `master` as is conventional. We need the parent to be a `Frame`, but we need it specifically to be a `FontSelect` widget, because we'll need to use its font attributes `.regularFont` and `.listFont`.

fontselect.py

```
#-- 1 --  
# [ fontSelect := fontSelect with a new Frame added  
#   self := that Frame ]  
Frame.__init__ ( self, fontSelect )
```

The grid plan is quite simple: the label is in row 0, and the family list is in row 1.

fontselect.py

```
#-- 2 --  
self.fontSelect = fontSelect  
self.observer   = observer  
  
#-- 3 --  
# [ self := self with a new Label widget added  
#       and gridded  
#   self.topLabel := that widget ]  
self.topLabel = Label ( self,  
                        font=fontSelect.regularFont,  
                        text="Click to select font family:" )  
self.topLabel.grid ( row=0, column=0, sticky=W )  
  
#-- 4 --  
# [ self := self with a new ScrolledList widget added  
#       and gridded  
#   self.scrollList := that widget ]  
self.scrollList = scrolledlist.ScrolledList ( self,  
                                              width=LISTBOX_WIDTH, height=LISTBOX_HEIGHT,  
                                              callback=self.__listboxHandler )  
self.scrollList.grid ( row=1, column=0 )  
self.scrollList.listbox["font"] = fontSelect.listFont
```

Now that we have a listbox for the font families, we need to load in the list of families. The `tkFont.families()` function returns an unsorted tuple of the family names. We'll convert that to a list, sort them, then load them into the listbox.

fontselect.py

```
#-- 5 --  
# [ self.scrollList := self.scrollList with a list of  
#   the font families from tkFont added ]  
familySet = list ( tkFont.families() )  
familySet.sort()
```

```
for name in familySet:
    self.scrollList.append ( name )
```

6.2. FamilyPicker.__listboxHandler()

This method is called whenever the user clicks in the family names list. When called, it calls the **observer** function and passes the font name to it.

fontselect.py

```
def __listboxHandler ( self, lineNo ):
    """Handler for selection in the listbox.
    """
```

First we need to translate the line number in the list box into a family name. Then we pass that to our observer callback.

fontselect.py

```
#-- 1 --
# [ familyName := text from the (lineNo)th line of
#     self.scrollList ]
familyName = self.scrollList[lineNo]

#-- 2 --
self.observer ( familyName )
```

7. class Controls: Controls frame

This compound widget contains all the controls for changing the font except for the family-picking apparatus. It has a method that tells it what font family to use. It calls its observer callback whenever any font option is changed.

fontselect.py

```
class Controls(Frame):
    """Frame for all the small widgets in the application.

    Exports:
    Controls ( fontSelect, observer ):
        [ (fontSelect is a FontSelect widget) and
          (observer is a callback function or None) ->
          fontSelect := fontSelect with a new Controls widget
            added but not gridded, with that observer callback
            if given
          return that new Controls widget ]
    .setFamily ( familyName ):
        [ familyName is the name of a font family in tkFont ->
          self := self with that font family set ]
    .getName():
        [ return a string describing self's current font ]
    .getFont():
        [ returns self's current font as a tkFont.Font ]
```

Here is the grid plan and list of internal widgets.

Internal widgets:

```

    0
    +-----+
0 | .annunciator | Label: Shows the current font name.
    +-----+
1 | .sample      | Text: Some sample text in the current font.
    +-----+
2 | .buttons     | Frame: Remaining small controls.
    +-----+

```

All the rest of the controls live in the `self.buttons` *Frame*.

This class manages both its own grid and the grid inside the `.buttons` frame. We could have made yet another compound widget class for this frame, but the separation of controls was made for reasons of layout and not because they are really a separate group of controls.

Internal widgets inside `self.buttons` are stacked sideways.

Column

```

0   .sizeLabel: Label, 'Size:'
1   .sizeField: Entry, text size in pixels
2   .sizeButton: Button, applies .sizeField
3       (spacer column, absorbs remaining space)
4   .boldButton: Checkbutton, turns boldface on and off
5   .italicButton: Checkbutton, turns italics on and off

```

Control variables:

```

.__fontName:
  [ string control variable for .annunciator ]
.__sizeText:
  [ string control variable for .sizeField ]
.__isBold:
  [ int control variable for .boldButton ]
.__isItalic:
  [ int control variable for .italicButton ]

```

Here are the internal attributes of the class.

State/Invariants:

```

.fontSelect: [ self's parent, a FontSelect ]
.__currentFamily:
  [ if a family has been selected ->
    that family's name as a string
    else -> None ]
.__currentFont:
  [ if a font has been selected ->
    a tkFont.Font representing that font
    else ->

```

```

        None ]
    ..__observer: [ as passed to the constructor, read-only ]
    """

```

7.1. Controls.__init__(): Constructor

As we create the controls, we'll need access to a number of fonts: fonts for control labels, but also the special italic and bold fonts for the italic and bold buttons. If we stipulate that our parent widget is a `FontSelect` widget, then we can access these fonts, which are public attributes of that widget.

The constructor starts by calling the parent's constructor. Then we set up the initial invariants for the internal state items, and create the widgets.

fontselect.py

```

def __init__ ( self, fontSelect, observer=None ):
    """Constructor for the Controls widget.
    """

    #-- 1 --
    # [ fontSelect := fontSelect with a new Frame added
    #   self := that Frame ]
    Frame.__init__ ( self, fontSelect )

    #-- 2 --
    self.fontSelect      = fontSelect
    self.__currentFont   = None
    self.__currentFamily = None

    #-- 3 --
    # [ self := self with all widgets added and gridded ]
    self.__createWidgets()

```

It is necessary to select some family initially. However, the usually family selection logic will call the `observer` function, and we don't want to do that for the initial selection. So `self.__observer` is set temporarily to `None` while we select the initial family, and then set correctly afterward.

fontselect.py

```

    #-- 4 --
    # [ self := self with DEFAULT_FAMILY as the selected family ]
    self.__observer = None
    self.setFamily(DEFAULT_FAMILY)

    #-- 5 --
    self.__observer      = observer

```

7.2. Controls.__createWidgets()

For details of the layout and grid plans, see Section 7, "class `Controls: Controls frame`" (p. 14).

fontselect.py

```

def __createWidgets ( self ):

```

```
"""Widget layout.
"""
```

First we set up the three items gridded in self.

fontselect.py

```
## 1 --
# [ self := self with a new Label added and gridded
#       with a string control variable
#   self.annunciator := that Label
#   self.__fontName := that control variable ]
self.__fontName = StringVar()
self.annunciator = Label ( self,
                           font=self.fontSelect.regularFont,
                           textvariable=self.__fontName,
                           text="" )
rowx = 0
self.annunciator.grid ( row=rowx, sticky=W )
```

The width and height options of the Text widget are in lines and characters, respectively. This gives a bit of room for the user to add or change text.

fontselect.py

```
## 2 --
# [ self := self with a new Text widget added and gridded,
#       displaying SAMPLE_TEXT
#   self.sample := that widget ]
self.sample = Text ( self,
                    width=20, height=2,
                    font=self.fontSelect.regularFont )
rowx += 1
self.sample.grid ( row=rowx, sticky=W )
self.sample.insert ( END, SAMPLE_TEXT )
```

The self.buttons frame is made to stick to the left (west) side of the containing column by using sticky=W when it is gridded.

fontselect.py

```
## 3 --
# [ self := self with a new Frame widget added and gridded
#   self.buttons := that widget ]
self.buttons = Frame ( self )
rowx += 1
self.buttons.grid ( row=rowx, sticky=W )
```

The creation and gridding of the remaining controls is done in Section 7.3, “Controls.__createButtons(): Lay out the small controls frame” (p. 18).

fontselect.py

```
## 4 --
# [ self := self with .sizeLabel, .sizeField, .sizeButton,
#   .boldButton, and .italicButton added and gridded ]
self.__createButtons()
```

7.3. Controls. `__createButtons()`: Lay out the small controls frame

This method creates and grids all the widgets inside the `self.buttons` frame, as well as their control variables.

fontselect.py

```
def __createButtons ( self ):  
    """Create all the widgets and control variables in self.buttons.  
    """  
  
    #-- 1 --  
    # [ self.buttons := self.buttons with a new Label added and  
    #           gridded  
    # self.sizeLabel := that Label ]  
    self.sizeLabel = Label ( self.buttons,  
                             font=self.fontSelect.regularFont,  
                             text="Size:" )  
    colx = 0  
    self.sizeLabel.grid ( row=0, column=colx )
```

The `size` field has a control variable attached. It also has event bindings for either *Return* or the `FocusOut` event that occurs when the user tabs out of the field.

The `self.__setFont()` method is a general-purpose handler. Whenever it is called, it assembles all the font options and, assuming they are valid, changes the current font and calls the observer function. See Section 7.4, “Controls. `__setFont()`: Font change handler” (p. 20).

fontselect.py

```
#-- 2 --  
# [ self.buttons := self.buttons with a new Entry added  
#       and gridded, with a string control variable, that  
#       calls self.__setFont on Return or Tab keypresses  
# self.sizeField := that Entry  
# self.__sizeText := that control variable ]  
self.__sizeText = StringVar()  
self.__sizeText.set ( DEFAULT_SIZE )  
self.sizeField = Entry ( self.buttons,  
                         font=self.fontSelect.regularFont,  
                         width=4,  
                         textvariable=self.__sizeText )  
colx += 1  
self.sizeField.grid ( row=0, column=colx )
```

These two `.bind()` calls set up the widget so that `__setFont()` will be called if the user either hits the *Enter* key (whose keyname is *Return*, *not* *Enter*), or tabs out of the field. The latter event is called `FocusOut` because tabbing out of a field causes it to lose keyboard focus.

fontselect.py

```
self.sizeField.bind ( "<KeyPress-Return>",  
                     self.__setFont )  
self.sizeField.bind ( "<FocusOut>", self.__setFont )  
  
#-- 3 --  
# [ self.buttons := self.buttons with a new Button added
```

```

#         and gridded, that calls self.__setFont when clicked
# self.sizeButton := that Button ]
self.sizeButton = Button ( self.buttons,
    text="Set\nsize",
    font=self.fontSelect.regularFont,
    command=self.__setFont )
colx += 1
self.sizeButton.grid ( row=0, column=colx )

```

So that the user sees the size label, size entry, and size button as a group, we want a little extra space to be allocated to the next column inside this frame. This takes two steps. First, we use `.columnconfigure` to add some extra space within that column. Details are in the section on “Configuring column and row sizes” in the *Tkinter reference*¹⁰.

However, this isn't enough. Tkinter will ignore this column configuration unless there is at least one widget in that column. So we also create an empty *Frame* there.

fontselect.py

```

#-- 4 --
# [ self.buttons := self.buttons with column (colx+1)
#     made stretchable
#   colx += 1 ]
colx += 1
self.buttons.columnconfigure ( colx, pad=10 )
self.spacer = Frame ( self.buttons )
self.spacer.grid ( row=0, column=colx )

```

We want the boldface and italic buttons to look like the regular font, only bolded and italicized. We use the `.copy()` method on `self.fontSelect.regularFont` to get a copy of that font, then use the `.configure` method on the copy to change its weight or slant.

Also, we use a medium gray `selectcolor`, overriding the rather alarming default red color that shows on a selected push-push button.

To make push-push buttons (click to set, click again to reset) for the controls for boldface and italics, we use a `Checkbutton` widget with the attribute `indicatoron=0`.

fontselect.py

```

#-- 5 --
# [ self.buttons := self.buttons with a Checkbutton added
#     and gridded, with an integer control variable
# self.boldButton := that Checkbutton
# self.__isBold := that control variable ]
self.__isBold = IntVar()
self.boldFont = self.fontSelect.regularFont.copy()
self.boldFont.configure ( weight=tkFont.BOLD )
self.boldButton = Checkbutton ( self.buttons,
    command=self.__setFont,
    variable=self.__isBold,
    selectcolor="#bbbbbb",
    indicatoron=0,
    font=self.boldFont,
    text="Bold" )
colx += 1
self.boldButton.grid ( row=0, column=colx )

```

¹⁰ <http://infohost.nmt.edu/tcc/help/pubs/tkinter/grid-config.html>

```

#-- 6 --
# [ self.buttons := self.buttons with a Checkbutton added
#     and gridded, with an integer control variable
#   self.italicButton := that Checkbutton
#   self.__isItalic := that control variable ]
self.__isItalic = IntVar()
self.italicFont = self.fontSelect.regularFont.copy()
self.italicFont.configure ( slant=tkFont.ITALIC )
self.italicButton = Checkbutton ( self.buttons,
    command=self.__setFont,
    variable=self.__isItalic,
    selectcolor="#bbbbbb",
    indicatoron=0,
    font=self.italicFont,
    text="Italic" )
colx += 1
self.italicButton.grid ( row=0, column=colx )

```

7.4. Controls. `__setFont()`: Font change handler

This method is a generic handler that is called when any font option changes. It tries to build a new font with all the currently selected options. It can fail—for example, if the text in the size field is not a valid integer. Assuming it succeeds, it builds a new font with all the current options, stores it in `self.__currentFont`, and calls the observer function.

This method may be called when the user clicks the button to set the size; in that case it is called with no arguments. By adding a second argument with a default value of `None`, we can use the same handler for the event bindings on `self.sizeField`; event handlers get an `Event` object as an argument.

fontselect.py

```

def __setFont ( self, event=None ):
    """Handler for size button or size field events.

    [ event is an Event object or None ->
      if self's controls describe a valid font ->
        self := self displaying that new font
        call self's observers with that new font
      else ->
        pop up an error dialog ]
    """

```

The contents of the `self.sizeField` `Entry` widget should be a number (in string form). If the text in the field isn't a valid integer, we'll use a pop-up dialog to alert the user. In the `Dialog` constructor, the `default=0` option tells it to use the first button by default, and the `strings` tuple specifies that there be only one button, labeled "OK".

fontselect.py

```

#-- 1 --
# [ if the text in self.sizeField is an integer ->
#   sizeValue := that text as an integer
#   else ->
#     pop up a dialog

```

```

#     return ]
try:
    sizeValue = int ( self.__sizeText.get() )
except ValueError:
    d = Dialog.Dialog ( self,
                        title="Message", bitmap="info",
                        text="Size must be an integer.",
                        default=0, strings=("OK",) )

    return

```

It's also an error if no family has ever been selected.

fontselect.py

```

#-- 2 --
# [ if self.__currentFamily is None ->
#     pop up a dialog
#     return
# else -> I ]
if self.__currentFamily is None:
    d = Dialog.Dialog ( self,
                        title="Message", bitmap="info",
                        text="No family selected.",
                        default=0, strings=("OK",) )

    return

```

Next we convert the states of the boldface and italic buttons into appropriate values of the weight and slant options for font creation.

fontselect.py

```

#-- 3 --
if self.__isBold.get():    weightValue = tkFont.BOLD
else:                     weightValue = tkFont.NORMAL

```

This next bit brought out a bug in Tkinter itself. The Tk package wants the `slant` attribute to be either "italic" (which is available as the constant `tkFont.ITALIC`) or "roman". However, there is no such constant as `tkFont.ROMAN`.

fontselect.py

```

#-- 4 --
if self.__isItalic.get(): slantValue = tkFont.ITALIC
else:                     slantValue = "roman"

```

We have everything we need now: family name, size, and bold and italic attributes. Build the font, apply it to the `.sample`, and call the observer function if any.

fontselect.py

```

#-- 5 --
# [ self.__currentFont := a new font with
#     family=self.__currentFamily, weight=weightValue,
#     slant=slantValue, and size=sizeValue ]
self.__currentFont = tkFont.Font ( family=self.__currentFamily,
                                   size=sizeValue, weight=weightValue, slant=slantValue )

```

It's a simple matter to change the font of `self.sample`: we just use its `.configure()` method. Also get a string describing all the current font options and place it into the annunciator label.

```

#-- 6 --
# [ self.sample := self.sample with font self.__currentFont
#   self.annunciator := self.annunciator with the actual
#     name of self.__currentFont ]
self.sample.configure ( font=self.__currentFont )
self.__fontName.set ( self.getName() )

#-- 7 --
if self.__observer:
    self.__observer ( self.__currentFont )

```

7.5. Controls.setFamily(): Change font family name

This method changes self's family name.

```

def setFamily ( self, newFamily ):
    """Sets self's font family name.
    """
    #-- 1 --
    self.__currentFamily = newFamily

    #-- 2 --
    # [ if self's controls describe a valid font ->
    #   self := self displaying that new font
    #   call self's observers with that new font
    #   else ->
    #     pop up an error dialog ]
    self.__setFont()

```

7.6. Controls.getName(): Get the current font name

This public method returns a string with up to four components: family name, size, and optional weight and slant. Example: "luxi mono 14 bold italic".

However, the font we have built may not be exactly what the user specified. Not all fonts are available in all sizes, for example. So we use the .actual() method on the font to retrieve the options actually in force.

```

def getName ( self ):
    """Return a string describing the current font's options.

    [ if self.__currentFont is None ->
      return None
      else ->
        return a string describing the current font's options ]
    """

    #-- 1 --

```

```

if self.__currentFont is None:
    return None

#-- 2 --
# [ attrs := a list containing self.__currentFont's
#       actual family name and actual size (as a string) ]
attrs = [ self.__currentFont.actual ( "family" ),
          str ( self.__currentFont.actual ( "size" ) ) ]

#-- 3 --
# [ if self.__currentFont is bold ->
#   attrs += "bold"
#   else -> I ]
if self.__currentFont.actual ( "weight" ) == tkFont.BOLD:
    attrs.append ( "bold" )

#-- 4 --
# [ if self.__currentFont is italic ->
#   attrs += "italic"
#   else -> I ]
if self.__currentFont.actual ( "slant" ) == tkFont.ITALIC:
    attrs.append ( "italic" )

```

The `.join()` string method assembles this list into a string, with the pieces separated by one space.

fontselect.py

```

#-- 5 --
return " ".join ( attrs )

```

7.7. Controls.getFont(): Return the current font

Returns the font currently selected.

fontselect.py

```

# - - -   C o n t r o l s . g e t F o n t

def getFont ( self ):
    """Return the current tkFont.Font instance.
    """
    return self.__currentFont

```

