Writing GUI Wizards in Python

Release 1.4

Rich Salz

November 9, 2001

Zolera Systems, http://www.zolera.com
E-mail: rsalz@zolera.com

Contents

---

1 Introduction

The `wizard` module makes it easy to write typical desktop GUI-style "wizards," where a user is guided through a series of forms and enters configuration data. By design, the `wizard` module tries to shield the developer from all aspects of GUI development. This extends to keeping the `Tkinter` names out of the global namespace.

A *wizard* encapsulates a set of interactions with a user. It contains one or more *sheets*. A sheet can either have text for the user to read, or it can have one or more data entry *fields*. These fields can be text entry, file or directory selection, check-boxes or multiple-choice lists. When the user indicates that they are finished, the *wizard* returns a dictionary containing the data the user entered. Entries can also be *validated* (see section 3) by calling out to application-specific code. Validation can be done when the user tries to move to the next sheet, or at the end after all the data has been entered. The `wizard` module includes several validators.

2 Input

The `wizard` module defines a number of classes. Only the `Wizard` class has methods beyond the constructor.

class `Wizard` (*sheets, title*[*, root=None*[*, defaults=None*]][*, **keywords*])

The main class of this module. Users create an instance of this class, and then `run` it, which creates the display and solicits input from the user.

The *root* parameter is the root of the display into which the wizard should be created, and is usually `None`. (Advanced applications may wish to create their own popup or frame, for example, but must be aware that the `wizard` module uses `Tkinter`'s `grid` layout manager.) The *sheets* parameter is a list of `LicenseSheet`, `DynamicSheet`, and `Sheet` objects, each of which represents one "page" of display. The *title* parameter is a text string that is displayed at the top of the wizard. The `defaults` parameter is a dictionary that contains the default (initial) values of all the entries.

The following keyword parameters can also be given:

Keyword	Default	Description
`entrywidth`	`40`	The desired width of text entry fields, in characters.
`whenvalidate`	`atend`	When to validate user input. Choose from `atend` or `perpage`.
`center`	`0`	If non-zero, then the wizard is centered in the screen.
`geometry`	`None`	A `geometry` string used to specify the on-screen placement. The default is for the window system to control the location.
`help`	`None`	A `Help` object for context-sensitive help, see section 4.
`helpkey`	`'<F1>'`	If a `Help` object is given, then this parameter specifies the help key. For details, see the `Tkinter` documentation.
`titlefont`	16-point bold italic	A font modifier to display the title font.
`sheetfont`	underlined	The font modifier for the sub-titles on each sheet.
`sheetchange`	`None`	A `SheetChange` object, see below.

The `center` and `geometry` parameters are ignored if `root` is not `None`.

`set_defaults` (*defaults*)

Sets the default initial values from the dictionary `defaults`.

`run` (*self*[*, start = 0*])

Start the display and collect the user's input. The optional `start` parameter specifies the starting sheet. This method returns `None` if the user clicked on the `Cancel` button, or a dictionary containing the data the user provided. Note that all values in the dictionary are text strings.

This method can be called multiple times. Unless `set_defalts` is called, subsequent invocations will start with the values from the previous `run`.

class `LicenseSheet` (*title, text*[*, **keywords*])

A `LicenseSheet` object contains text to be displayed to the user. The text can be specified inline, or come from an external file.

The `title` is the text to appear at the top of the sheet. The `text` is the text to display. If the `file` parameter is specified (see the following table), then `text` is used only if the file cannot be read.

The following keyword parameters can also be given:

Keyword	Default	Description
`file`	`None`	Name of the file containing text to display. It is not an error if the file cannot be open – the value of the *text* will be used instead.
`height`	`10`	Height of display area, in lines.
`width`	`40`	Width of display area, in characters.
`wrap`	`Tkinter.WORD`	Whether or not to do word-wrapping on lines. If set to `Tkinter.CHAR` then lines are broken without regard to word boundaries. To avoid wrapping, use `hscrollbar`.
`font`	`None`	Font modifier; see below.
`mustread`	`0`	If non-zero, then the `Next/Done` button is disabled until the end of the text appears on the screen.
`hscrollbar`	`0`	If non-zero, then a horizontal scrollbar is attached to the bottom of the text box.

class `DynamicSheet` (*title, builder*[, ***keywords*])

A `DynamicSheet` object is like a `LicenseSheet` except that the content is generated dynamically each time the sheet is displayed. This class can be used to generate "confirmation" pages that offer a last chance to view the actions that the program is about to perform.

The `title` is the text to appear at the top of the sheet. The `builder` is an object that must implement the following three methods:

`open` (*dict*)

The *dict* parameter is a dictionary with all the current field values. The return value is ignored.

`readline` ()

This method returns text to be displayed. It will be called repeatedly until it returns `None`.

`close` ()

This method will be called when `readline` is finished. The return value is ignored.

The following `height`, `width`, `wrap`, `font`, and `hscrollbar` keyword parameters defined in the `LicenseSheet` class can also be used.

class `Sheet` (*title, fields*[, ***keywords*])

A `Sheet` object contains entry fields for the user to fill-in. The *title* parameter is a text string to be displayed at the top of the sheet. The *fields* parameter is a list of ``` ``xxxField'' ``` objects.

The following keyword parameters can also be given:

Keyword	Default	Description
`longvalidate`	`0`	If non-zero, then validating the fields on this sheet could take a noticeable amount of time, so the `wizard` module will temporarily change the cursor to an hourglass or its equivalent.

The field objects also accept the `longvalidate` parameter.

class `SpacerField` ([***keywords*])

This object is used to leave one or more blank lines, usually used to group related entries.

The following keyword parameters can also be given:

Keyword	Default	Description
`lines`	`1`	Number of blanks lines desired.

class `LabelField`(*prompt*)

This object is used to display a line of text on the sheet. It is usually used to provide a heading for a group of related fields, or to provide a label for subsequent a `RBField` (radiobutton) object.

The following keyword parameters can also be given:

Keyword	Default	Description
`font`	`None`	Font modifier; see below.
`align`	`align_l`	Specifies the alignment of the text; use `align_l`, `align_c`, or `align_r` for flush-left, center, or flush-right alignment.

class `EntryField`(*key, prompt*[, ***keywords*])

This object is used to collect a line of text from the user. The *key* parameter specifies the key in the dictionary of returned values to use for this field. The *prompt* parameter specifies the prompt text that will appear to the left of the text-entry area.

The following keyword parameters can also be given:

Keyword	Default	Description
`private`	`0`	If non-zero, the text will not be displayed. This is useful for password fields.
`validate`	`None`	A `Validation` object to be called when validating the field. See section 3.
`startdisabled`	`0`	If non-zero, then the field is initially disabled.
`entrywidth`	`30`	The desired input width, in charcters.

class `FileField`(*key, prompt*[, ***keywords*])

This is the same as `EntryField`, except that a "browse" button appears to the right of the entry area, allowing the user to browse the filesystem and select an existing file.

class `DirField`(*key, prompt*[, ***keywords*])

This is the same as `FileField`, except that the user must pick a directory. New in version 1.4.

class `CBField`(*key, prompt*[, ***keywords*])

This object creates a checkbox on the sheet, with `prompt` as the text. The *key* and *prompt* parameters are as described in the `Entryfield` class, above. The returned value will be the text string `''0''` or `''1''`.

The following keyword parameters can also be given:

Keyword	Default	Description
`enables`	`None`	A comma separated list of `keys` that identify the fields controlled by this button. The fields in the list are enabled, or disabled, according to whether this field is checked or not.

class `RBField`(*key, choices*[, ***keywords*])

The *key* parameter is as described in the `Entryfield` class, above. The *choices* parameter is an array of text strings. They will be displayed one per line, with the user allowed to select one. The returned value will be the numeric index of the value chosen, as a string.

class `DDField`(*key, prompt, choices*[, ***keywords*])

This is like a `RBField`, except that it is presented with the specified *prompt* on the left and a pull-down list of the *choices* on the right. The returned value will be the numeric index of the value chosen, as a string. New in version 1.4.

2.1 Font Modifiers

The `wizard` module uses Python dictionaries to control fonts, using the `tkFont` module to create the fonts it needs. The following table lists the dictionary entries that can be used to modify the default fonts chosen by the `wizard` module. Note that the keys are text strings.

Key	Description
family	A text string naming the font family.
size	An integer specifying the ont size in points; use a negative number to specify size in pixels.
weight	The font thickness; use `tkFont.NORMAL` (generally the default) or `tkFont.BOLD`.
slant	The font slant; use `tkFont.NORMAL` (the default) or `tkFont.ITALIC`.
underline	If non-zero, text is underlined; the default is `0` except for the sheet titles.
overwrite	If non-zero, text is overstruck; the default is `0`.

2.2 Sheet Changes

When a `Wizard` object is created, a callback object can be given that will be invoked every time the wizard displays a different sheet. This object must implement the following method:

`sheetchange`(*dict, sheetnum*)
The *dict* parameter is a dictionary with all the current field values. The *sheetnum* parameter is the new current sheet number. The return value is ignored. In particlar, it is impossible to prevent moving to the new sheet — see the description of validators in section 3.

2.3 Example

The following example constructs a three-sheet wizard. Note that it also uses validators which are described in section 3

```
license='''This is your license agreement...
'''

MySheets = (
    LicenseSheet('License', license, font={'family':'fixed'}, mustread=1),
    Sheet('Account information', (
        EntryField('adminname', 'Administrator name',
            validate=Nonblank('administrator')),
        EntryField('adminpass1', 'Administrator password',
            private=1, validate=PassConfirm('administrator', 'adminpass2')),
        EntryField('adminpass2', 'Repeat password',
            private=1),
    )),
    Sheet('Network information', (
        EntryField('hostname', 'Name of the host',
            validate=Nonblank('hostname')),
        FileField('path', 'Hosts file'),
        SpacerField(),
        LabelField('Network protocol'),
        RBField('protocol', ('TCP/IP', 'DECnet') ),
        SpacerField(),
        CBField('localserver', 'Start local server', enables='portnum'),
        EntryField('portnum', 'Server port#',
            validate=InactivePort('port'))
    )),
)
```

3 Validation

Most fields accept a `validate` parameter, which takes an object responsible for validating the user's input. A validator must implement the `validate` function. If the user's input is invalid, it should raise the `InvalidEntry` exception.

class `Validator` (*foo*)
The `Validator` class may be used as a base class for any validators.

`validate` (*dict, field*)
This method validates the user's input. The *dict* parameter will be a dictionary containing all the current values, and while the *field* is the key for the field being validated. (Remember that values in the dictionary are text strings.) If the user's input is not valid, this method should raise an `InvalidEntry` (or subtype) exception. The return value is ignored.

exception `InvalidEntry`
Exception raised when the input is invalid. The constructor takes a text string (stored as the `text` attribute) which will be displayed to the user.

A number of utility validators are provided in the `wizard` module:

class `PassConfirm` (*what, confirm*)
Validate that a field and its "confirmation" match. The *what* parameter should be a very short description (usually one word) displayed to the user. The *confirm* is the dictionary key that has the confirming ("other") entry.

class `Nonblank` (*what*)
Validate that a field is not blank. The *what* parameter is described in the `PassConfirm` class, above.

class `PositiveNumber` (*what*)
Validate that a field is a positive number. The *what* parameter is described in the `PassConfirm` class, above.

class `InactivePort` (*what*)
Validate that a field specifies a TCP port that is not used on the local host. This is a subtype of `PositiveNumber`.

3.1 Example

Here is the source for the `PassConfirm` class, used in the previous example:

```
class PassConfirm(Validator):
    def __init__(self, what, confirm):
        self.what = what
        self.confirm = confirm
    def validate(self, dict, field):
        if len(dict[field]) == 0:
            raise InvalidEntry, 'Must provide %s password.' % (self.what,)
        if dict[field] != dict[self.confirm]:
            raise InvalidEntry, 'Mismatched %s passwords.' % (self.what,)
```

4 Help

A `wizard` object can provide context-sensitive help. To do this, the object must be provided when the wizard is created, and it must implement the following two methods:

`help` (*root, key*)

This method should display the help appropriate for the indicated *key*. The *root* parameter is the display root provided when the wizard was created. It can (and often will) be `None`.

`sheethelp`(*root, sheetnum*)
This method is similar, but should display general help for sheet number `sheetnum`.

The `wizard` module provides a utility class that implements pop-up help, `HelpPopup`.

class `HelpPopup`()
This class provides unformatted help text in a popup dialog text box with a scrollbar and a `Dismiss` button. It implements `sheethelp` as a call to `help` with the key specified as `sheetN`, where `N` is the sheet number.

To use this class, create your own class derived from `HelpPopup` and implement the following methods:

`start`(*key*)
This method should do any preparations necessary to retrieve the help text. The return value is ignored. If it raises an exception, then a message saying no help is available will be displayed.

`eof`(*key*)
This method should return zero as long as there is more text to display. This method is called *before* each call to `get`.

`get`(*key*)
This method should return (portions of) the help text to display.

4.1 Example

Here is a class that stores each help in its own file.

```
import os
class MyHelp(HelpPopup):
    def __init__(self, helpdir):
        HelpPopup.__init__(self)
        self.filename = helpdir
    def start(self, key):
        # caller catches exception and prints default.
        self.infile = open(os.path.join(self.helpdir, key), "r")
    def get(key):
        # read all the text at once and return it.
        text = self.infile.read()
        self.infile.close()
        return text
    def eof(key):
        return self.infile.closed
```

5 More Examples

This example uses the code and data from the previous examples, to make a complete wizard:

```
from wizard import *

helpobj = MyHelp('/usr/local/share/helpdir')
w = Wizard(MySheets, 'Installation', None,
        whenvalidate=perpage, help=helpobj)
w.set_defaults({'adminname': 'root',
        'path': '/etc/hosts',
        'protocol': '0',
        'localserver': '1'})
answers = w.run()
if answers == None:
    print 'Cancelled.'
    sys.exit(1)
```

This example puts a cycling series of images to the left of the wizard:

```
from wizard import *
from Tkinter import *

class Cycler:
    def __init__(self, sheets, label):
        self.label = label
        self.images = [ ]
        try:
            Idefault = PhotoImage('default.gif')
        except:
            Idefault = None
        for i in range(len(sheets)):
            try:
                self.images.append(PhotoImage(file="pic%d.gif" % (i,)))
            except:
                self.images.append(Idefault)
    def goto(self, dict, sheetnum):
        self.label['image'] = self.images[sheetnum]

# Create a display root, the left-side frame holds the images
# and the right-side frame holds the wizard.
title = 'Installation'
root = Tk()
root.title(title)
root.iconname(title)
fLeft = Frame(root)
imageholder = Label(fLeft)
imageholder.grid()
fLeft.grid(row=0, column=0)
fRight = Frame(root)
fRight.grid(row=0, column=1)

w = Wizard(MySheets, title, fRight,
    sheetchange=Cycler(MySheets, imageholder))
answers = w.run()
if answers == None:
    print 'Cancelled.'
    sys.exit(1)
```