anygui
Reference Manual

Magnus Lie Hetland

Anygui 0.1a3 **October 23, 2001**

Contents

This manual describes the package *Anygui*, a generic GUI module for *Python*. The latest version of this manual and the software distribution is available from `http://anygui.sf.net`. More information about *Python* can be found at `http://www.python.org`.

1 Introduction

The *Python* standard library currently does not contain any platform-independent GUI packages. It is the goal of the *Anygui* project to change this situation. There are many such packages available, but none has been defined as *standard*, so when writing GUI programs for *Python*, you cannot assume that your user has the right package installed.

The problem is that declaring a GUI package as standard would be quite controversial. There are some packages that are quite commonly available, such as *Tkinter*; but it would not be practical to require all installations to include it, nor would it be desirable to require all *Python* GUI programs to use it, since there are many programmers who prefer other packages.

Anygui tries to solve this problem in a manner similar to the standard `anydbm` package. There is no need to choose *one* package at the expense of all others. Instead, *Anygui* gives generic access to several popular packages through a simple API, which makes it possible to write GUI applications that work with all these packages. Thus, one gets a platform-independent GUI module which is written entirely in *Python*.

1.1 Design Goals

A. *Anygui* should be an easy to use GUI package which may be used to create simple graphical programs, or which may serve as the basis for more complex application frameworks.

B. *Anygui* should be a pure *Python* package which serves as a front-end for as many as possible of the GUI packages available for *Python*, in a transparent manner.

C. *Anygui* should include functionality needed to perform most GUI tasks, but should remain as simple and basic as possible.

2 Installation

The *Anygui* package comes in the form of a `gzip` compressed `tar` archive. To install it you will first have to uncompress the archive. On *Windows* this can be done with *WinZip*. On the Mac, you can use *StuffIt Expander*. In *Unix*, first move to a directory where you'd like to put *Anygui*, and then do something like the following:

```
foo:~/python$ tar xzvf anygui-0.1a3.tar.gz
```

If your version of tar doesn't support the `z` switch, you can do something like this:

```
foo:~/python$ zcat anygui-0.1a3.tar.gz | tar xvf
```

Another possibility is:

```
foo:~/python$ gunzip anygui-0.1a3.tar.gz
```

No matter which version you choose, you should end up with a directory named `anygui-0.1a3`.

2.1 Running setup.py

The simple way of installing *Anygui* is to use the installation script that's included in the distribution. This requires *Distutils* (`http://www.python.org/sigs/distutils-sig`), which is included in *Python* distributions from version 2.0. To install the *Anygui* package in the default location, simply run the setup script with the `install` command:

```
foo:~$ python setup.py install
```

This will install *Anygui* in your standard *Python* directory structure. If you haven't installed *Python* yourself, you'll either have to have root access, or install it somewhere else. You can give a prefix with the `--prefix` option:

```
foo:~$ python setup.py install --prefix=${HOME}/python
```

2.2 Doing it Manually

Since *Anygui* consists of only *Python* code, nothing needs to be compiled. And the only thing needed to install *Python* code is to ensure that the packages and modules are found by your *Python* interpreter. This may be as simple as including the `lib` directory of the *Anygui* distribution in your `PYTHONPATH` environment variable. In `bash` (`http://www.gnu.org/manual/bash/`), you could do something like this:

```
foo:~$ export PYTHONPATH=$PYTHONPATH:/path/to/anygui/lib
```

To make this permanent, you should put it in your `.bash_profile` file, or something equivalent. If you don't want to mess around with this, and already have a standard directory where you place your *Python* modules, you can simply copy (or move) the `lib/anygui` there, or possibly place a symlink in that directory to.

2.3 Making Sure You Have a GUI Backend

Once you have *Anygui* installed, you'll want to make sure you have a usable GUI backend. This is easy to check: Simply start an interactive *Python* interpreter and try to execute the following:

```
>>> from anygui import Application
>>> Application()
```

If this works, you should be all set for making GUI programs with *Anygui*. If it raises an exception, however, you will have to install a GUI package. *Anygui* currently supports the following packages:

```
PythonWin   (mswgui)    http://starship.python.net/crew/mhammond/win32
Tkinter     (tkgui)     http://www.python.org/topics/tkinter
wxPython    (wxgui)     http://www.wxpython.org
Java Swing  (javagui)   http://www.jython.org
PyGTK       (gtkgui)    http://www.daa.com.au/~james/pygtk
Bethon      (beosgui)   http://www.bebits.com/app/1564
```

Of these, *Tkinter* is compiled in by default in the *MS Windows* distribution of *Python* (available from `http://www.python.org`), *PythonWin* (as well as *Tkinter*) is included in the *ActiveState* distribution, *ActivePython* (available from `http://www.activestate.com`), and *Java Swing* is automatically available in *Jython*, the *Java* implementation of *Python*.

In the future *Anygui* should hopefully work in almost any *Python* installation, even those which do not have a specific GUI package installed, either by using Dynamic HTML together with the standard `webbrowser` module, or by providing some simple text interface which is logically equivalent to the GUI version.

If you don't want to bother with all these backends, you may not have to. Just try to use anygui and see if it works. If you have a usable backend, *Anygui* will automatically detect it and use it.

3 Using Anygui

Note: For some examples of working *Anygui* code, see the `test` directory of the distribution.

Using *Anygui* is simple; it's simply a matter of importing the widgets (GUI elements) you need from the `anygui` module, e.g.:

```
from anygui import *
```

After doing this you must first create an `Application` object; then you may instantiate the widgets and combine them in various ways. When you're satisfied, you call the `run` method of your `Application` instance.

```
app = Application()
# Make widgets here
app.run()
```

3.1 Importing the Backends Directly

If you wish to import a backend directly (and "hardwire it" into your program), you may do so. For instance, if you wanted to use the *wxPython* backend, `wxgui`, you'd replace

```
from anygui import *
```

with

```
from anygui.backends.wxgui import *
```

This way you may use *Anygui* in standalone executables built with tools like `py2exe` (`http://starship.python.net/crew/theller/py2exe/`) or the McMillan installer (`http://www.mcmillan-inc.com/install1.html`), or with `jythonc` with the `--deep` option or equivalent.

3.2 Creating a Window

One of the most important classes in *Anygui* is `Window`. Without a `Window` you have no GUI; all the other widgets are added to `Windows`. Knowing this, we may suspect that the following is a minimal *Anygui* program (and we would be right):

```
from anygui import *
app = Application()
win = Window()
app.run()
```

The problem with this code is that the window would be invisible. At the time of creation, windows are hidden, so that we may add widgets to them etc. before showing them to the user. To show a window we simply call its `show` method or set its `visible` property to true (e.g. `1`). `w.show()` is actually just a shortcut for `w.visible = 1`. (Similarly, `w.visible = 0` can be written `w.hide()`.) So, our program becomes:

```
from anygui import *
app = Application()
win = Window()
win.show()
app.run()
```

If you don't mind having your window popping up when it's created (this can be problematic if you create your window after starting your application, because it may be visible before your finished adding other widgets to it), you can simply specify that it should be visible to begin with:

```
from anygui import *
app = Application()
win = Window(visible=1)
app.run()
```

This example gives us a rather uninteresting default window. You may customise it by setting some of its properties, like `title` and `size` (`import` and `show` removed in the interest of brevity):

```
w = Window(visible=1)
w.title = 'Hello, world!'
w.size = (200, 100)
```

If we want to, we can supply all the widget properties as keyword arguments to the constructor:

```
w = Window(title='Hello, world!', size=(200,100), visible=1)
```

3.3 Baby Steps: Adding a Label

Simple (multiline) labels are created with the `Label` class:

```
lab = Label(text='Hello, again!', position=(10,10))
```

Here we have specified a position just for fun; we don't really have to. If we add the label to our window, we'll see that it's placed with its left topmost corner at the point `(10,10)`:

```
w.add(lab)
```

3.4 Placing Widgets in a Frame

We don't have to specify the widget's position like we did with our label. Instead, we can use the `place` method of `Window`. (Actually, it's a method of a more generic class, `Frame`, which isn't currently available to the user of *Anygui*, but will be in future releases.)

The `place` method allows us to specify several constraints (in the form of keyword arguments) that will affect where the widget is placed. Fore detailed information about this, see the API reference. Here are some examples, using only `Labels`:

```
win.place(lab, position=(10,10))
win.place(lab, left=10, top=10)
win.place(lab, top=10, right=10)
win.place(lab, position=(10,10), right=10, hstretch=1)
```

In the last example `hstretch` is a Boolean value indicating whether the widget should be stretched horizontally (to maintain the other specifications) when the containing `Frame` is resized. (The vertical version is `vstretch`.)

Note: `add` and `place` may be merged in a future release, making `add` unavailable.

3.4.1 Placing More Than One Widget

The `place` method can also position a *list* of widgets. The first widget will be placed as before, while the subsequent ones will be placed either to the right or to the left (according to the `direction` argument), at a given distance (`space`):

```
win.place([lab1, lab2], position=(10,10),
          direction='right', space=10)
```

Note: When using `direction` and `space` like this, you should explicitly set the size of your widgets, otherwise they will most likely overlap. This will hopefully be fixed (with a decent system of defaults) in future releases.

3.5 Adding a Button

`Buttons` are quite similar to `Labels`, except that they may perform an *action* when clicked. (They also look different, of course.)

```
def callback(): print 'Hello, world!'
btn = Button(text='Press me', action=callback)
```

When this button is pressed, it will execute the function stored in its `action` property (if there is one).

Note: This system of *callbacks* in the various widgets is all the event handling which is available in the current version of *Anygui* (0.1a3). In future releases a more thorough system will be implemented.

3.6 About Models and Views

Some widgets, such as `TextField` or `CheckBox`, reflect some state in the program, be it a text value or a binary (Boolean) value. These widgets are called *views* since they are ways of viewing the underlying state, which is called a *model*. In the current version of *Anygui* (0.1a3) there are three types of model:

`BooleanModel`: This represents a Boolean value (true or false), and is the underlying model for `CheckBox` and `RadioButton`.

`ListModel`: This represents a sequence, and supports all the normal list methods. It is the underlying model for `ListBox`.

`TextModel`: This represents a mutable string, and also supports standard list methods. (It is, in fact, a subclass of `ListModel`.) It is the underlying model of `TextField` and `TextArea`.

Each widget which functions as a view, has an attribute called `model`, which contains its model. The widget will always reflect the current state of the model, and changes in the widget (such as editin the text of a `TextField`) will automatically be reflected in the model. (Thus, in the terminology of the *Model-View-Controller* paradigm, these views are actually controllers as well.)

The models have an attribute called `value` which is a representation of the model as a primitive value. By assigning to this attribute you will change the model.

Note: The following example is not safe:

```
lb = ListBox()
some_list = [1, 2, 3]
lb.model.value = some_list
some_list[1] = 'foo'
```

Since `some_list` is a normal list, changing it (in the last line) will not afect the `ListBox` in any way. A better version of the last line would be

```
lb.model[1] = 'foo'
```

The `model` property will contain a default ("empty") model when the widget is created, so code like this is possible:

```
lb = ListBox()
lb.model.append('This is the first item')
```

Note that one of the advantages of using models like this is that you can use the models separately from the GUI, not worrying about updating the views etc. In a complex program you probably wouldn't use code like the above where you explicitly refer to the model of a specific widget.

3.7 Using CheckBoxes

A `CheckBox` is a *toggle button*, a button which can be in one of two states, "on" or "off". Except for that, it works more or less like any other button in that you can place it, set its text, and associate a callback with it.

Whether a `CheckBox` is currently on or off is indicated by its model's `value`.

3.8 RadioButtons and RadioGroups

`RadioButtons` are toggle buttons, just like `CheckBoxes`. The main differences are that they look slightly different, and that they may belong to a `RadioGroup`.

A `RadioGroup` is a set of `RadioButtons` where only *one*`RadioButton` is permitted to be "on" at one time. Thus, when one of the buttons in the group is turned on, the others are automatically turned off. This can be useful for selecting among different alternatives, for instance.

`RadioButtons` are added to a `RadioGroup` by setting their `group` property:

```
radiobutton.group = radiogroup
```

This may also be done when constructing the button:

```
grp = RadioGroup()
rbn = RadioButton(group=grp)
```

3.9 ListBox

A `ListBox` is a vertical list of items that can be selected, either by clicking on them, or by moving the selection up and down with the arrow keys. (For the arrow keys to work, you must make sure that the `ListBox` has keyboard focus. In some backends this requires using the `tab` key.)

Note: When using *Anygui* with *Tkinter*, using the arrow keys won't change the selection, only which item is underlined. You'll have to use the arrow keys until the item you want to select is underlined; then select it by pressing the space bar.

A `ListBox`'s items are set by manipulating its model (a `ListModel`). The model can contain any objects, and the text displayed in the widget will be the result of applying the built-in *Python* function `str` to each object.

```
lbx = ListBox()
lbx.model.value = 'This is a test'.split()
```

The currently selected item can be queried or set through the `selection` property (an integer index, counting from zero). Also, when an item is selected, the `ListBox`'s `action` callback is activated (if present).

3.10 TextField and TextArea

Anygui's two text widgets, `TextField` and `TextArea` are quite similar. The difference between them is that `TextField` permits neither newlines or tab characters to be typed, while `TextArea` does. Typing a tab in a `TextField` will simply move the focus to another widget, while pressing the enter key will activate the `TextField`'s `action` callback (if present).

The text in a text component is queried or set through its model's `value` property (a string or equivalent), and the current selection may be queried or set through the `selection` property (a tuple of two integer indices).

4 API Reference

[Under construction]

5 Known Problems

For an overview of known bugs in the current release, see the file `KNOWN_BUGS` found in the distribution.

6 Plans for Future Releases

For an overview of future plans, see the `TODO` file found in the distribution.

7 Frequently Asked Questions

Q: Which version of Python is required for Anygui?

A: Anygui requires *Python* version 2.0 or newer.

8 Contributing

If you want to contribute to the *Anygui* project, we could certainly use your help. First of all, you should visit the *Anygui* web site at `http://anygui.sf.net`, and subscribe to the developer's mailing list (`anygui-devel@lists.sf.net`) and try to familiarise yourself with how the package works behind the scenes. Then, you may either help develop the currently supported GUI packages, or you may start writing a backend of your own. Several backend targets may be found at `http://starbase.neosoft.com/~claird/comp.lang.python/python_GUI.html`.

9 Anygui License