anygui

Reference Manual

Magnus Lie Hetland

Anygui 0.1.1 **February 8th, 2002**

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://www.anygui.org`. 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*.

To get the latest *Anygui* distribution, or to get in touch with the developers, please visit the *Anygui* website: `http://www.anygui.org`.

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.

1.2 Warning

The *Anygui* API is currently very much in flux as the *Anygui* team keeps experimenting with it. Because of that, incompatibilities may occur between releases. The current release (0.1.1) should be regarded as a prototype.

1.3 Tutorial

There is also a short tutorial available, which is included in the installation (`doc/tutorial.txt`) and is available from the website (`http://www.anygui.org`).

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*. in *Mac OS*, 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-0.1.1.tar.gz
```

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

```
foo:~/python$ zcat anygui-0.1.1.tar.gz | tar xvf
```

Another possibility is:

```
foo:~/python$ gunzip anygui-0.1.1.tar.gz
foo:~/python$ tar -xvf anygui-0.1.1.tar
```

No matter which version you choose, you should end up with a directory named `anygui-0.1.1`.

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 don't have access to this directory (e.g. because *Python* was installed by a sysadmin, and you don't have root access) you can install it somewhere else 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 is 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 `anygui` package (found in `anygui-0.1.1/lib`) there, or possibly place a symlink in that directory to the `anygui` package.

2.3 Making Sure You Have a Usable GUI Package

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

```
>>> from anygui import *
>>> backend()
```

The `backend` function will return the name of the backend in use. If it is neither `'curses'` nor `'text'` you should be all set for making GUI programs with *Anygui*. (The `'curses'` and `'text'` backends use plain text to emulate graphical interfaces on platforms that don't have them.) *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
PyQt       (qtgui)     http://www.thekompany.com/projects/pykde
Curses     (cursesgui) -- used when no GUI package is available
Plain text (textgui)   -- used if curses is not available
```

Add `gui` to name returned by the `backend` function to get the full name of the backend module (in the `anygui.backends` package). For instance, the `msw` backend is found in `anygui.backends.mswgui` module.

In general, if you end up with a text-based solution, `cursesgui` will be preferred over `textgui`if your *Python*-installation has a wrorking `curses` module. The exception is if you are using *Anygui* in the interactive interpreter, in which `textgui` will be preferred, to avoid interfering with the terminal and locking up the interpreter prompt. (If you'd like to, for some reason, you can override this behaviour with the environment variable `ANYGUI_FORCE_CURSES`; see the API Reference below.)

BeOS Note: The BeOS backend (`beosgui`) is currently not fully functional, but is included nonetheless.

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*.

Note: In *Mac OS* 9, *Anygui* (using *Tkinter*) works with with *Python Classic* and recent versions of *Python Carbon*, but older versions have problems with *Tkinter*.

3 Using Anygui

Note: For some examples of working *Anygui* code, see the `test` and `demo` directories of the distribution. Remember that the test scripts are written to test certain features of *Anygui*, not to represent recommended coding practices.

Using *Anygui* is simple; it's simply a matter of importing the classes and functions you need from the `anygui` module, e.g.:

```
from anygui import *
```

After doing this you must create an `Application` object, at least one `Window`, and probably a few components such as `Buttons` and `TextFields`. The `Windows` are added to the `Application` (through its `add` method), and the various components are added to the `Window`. When you have done this, you call the `run` method of your `Application` instance.

```
# Make components here
win = Window()
# Add components to the Window
app = Application()
app.add(win)
app.run()
```

3.1 Avoiding Namespace Pollution

Importing everything from *Anygui* (as in `from anygui import *`) is fine for small programs, where you're certain that there will be no name clashes. You may also simply import the names you need:

```
from anygui import Application, Window
```

The preferred way to use modules like this is usually to avoid cluttering your namespace, by using simply `import anygui`. However, if you are going to create a lot of widgets, the `anygui` prefix may be cumbersome. Therefore, I suggest renaming it to `gui`, either with a simple assignment...

```
import anygui; gui = anygui
```

... or, in recent versions of *Python*:

```
import anygui as gui
```

Then you can instantiate widgets like this:

```
win = gui.Window()
```

The examples in this documentation use the starred import, for simplicity.

3.2 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.

Note: Compiling `jar` files of *Anygui* programs with *Jython* may not work in the current version.

Note that the namespace handling still works just fine:

```
import anygui.backends.tkgui as gui
```

3.3 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.add(win)
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`:

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

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

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

3.4 The set Method and the Options Class

If you want to change some attributes of a widget, you can either just set them directly, or (if you'd like to set several at once), use the `set` method, just like the constructor:

```
w.set(title='Hello, again', size=(300,200))
```

Supplying the same attributes with the same values to a lot of widgets (if you are making several buttons with the same size, for instance) can be a bit impractical (you'll learn more about buttons in a little while):

```
bt1 = Button(left=10, width=50, height=30, text='Button 1')
bt2 = Button(left=10, width=50, height=30, text='Button 2')
bt3 = Button(left=10, width=50, height=30, text='Button 3')
```

To deal with this, the widget constructors (and the `set` method) can take `Options` objects as positional parameters:

```
opt = Options(left=10, width=50, height=30)
bt1 = Button(opt, text='Button 1')
bt2 = Button(opt, text='Button 2')
bt3 = Button(opt, text='Button 3')
```

As you can see, this saves quite a lot of typing. You can use as many `Options` arguments as you like.

3.5 The modify Method

Just like `set` can be used to set the attributes of a `Component`, the `modify` method can be used to *modify* them, without rebinding them to another value. To show the difference, consider the following example (where `foo` is an attribute that does nothing special):

```
>>> from anygui import *
>>> btn = Button()
>>> some_list = [1, 2, 3]
```

```
>>> btn.foo = some_list
>>> btn.modify(foo=[4, 5, 6])
>>> btn.foo
[4, 5, 6]
>>> some_list
[4, 5, 6]
>>> btn.set(foo=[7, 8, 9])
>>> btn.foo
[7, 8, 9]
>>> some_list
[4, 5, 6]
```

As you can see, using `modify` modifies the list, while `set` replaces it. The `modify` method is used for (among other things) implementing Model-View-Controller systems. (More about that later.)

The `modify` method works as follows: If there is a specific internal method for modifying an attribute, that is called. Otherwise, the supplied value will be assigned to `self.name[:]` (where `name` is the attribute in question). If that doesn't work (a `TypeError` exception is raised), the value will be assigned to `self.name.value`. If that doesn't work either, the attribute will be rebound to the new value, with the same result as using `set`. So, to avoid any in-place modification, all you need to do is use immutable values:

```
>>> from anygui import *
>>> btn = Button()
>>> some_list = [1, 2, 3]
>>> btn.foo = tuple(some_list)
>>> btn.modify(foo=[4, 5, 6])
>>> btn.foo
[4, 5, 6]
>>> some_list
[1, 2, 3]
```

3.6 The refresh Method

The `modify` method is used to modify attributes in-place, e.g. to keep them in sync with a widget. This is done automatically when you change a widget through the graphical interface. In a way, the `refresh` method works the other way: If you modify an attribute, you can call the `refresh` method to keep the widget's appearance in sync with its state. When you assign to an attribute, `refresh` is called automatically; you only have to call it yourself if you have an attribute which is a mutable object, and you modify that object.

For more info about the use of `refresh`, see the section "About Models, Views, and Controllers", below.

3.7 Adding a Label

Simple 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)
```

3.8 Layout: Placing Widgets in a Frame

This section goves a simple example of positioning `Components`; for more information about the *Anygui* layout mechanism, please refer to the API Reference (below).

```
win.add(lab, position=(10,10))
win.add(lab, left=10, top=10)
win.add(lab, top=10, right=10)
win.add(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`.)

Just like in component constructors, you can use `Options` objects in the `add` method, after the component to be added:

```
win.add(lab, opt, left=10)
```

3.8.1 Placing More Than One Widget

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

```
win.add((lab1, lab2), position=(10,10),
        direction='right', space=10)
```

Note: Remember to enclose your components in a sequence (such as a tuple or a list), since `add` allows you to use more positional arguments, but will treat them differently. If you want to use `Options` objects, place them outside (after) the sequence. For more information see the section about the `Frame` class in the API Reference below.

3.9 Buttons and Event Handling

`Buttons` (as most components) work more or less the same way as `Labels`. You can set their `size`, their `position`, their `text`, etc. and then `add` them to a `Frame` (such as a `Window`). The thing that makes them interesting is that they emit *events*. Each time the user clicks a button, it sends out a `click` event. You can catch these events by linking your button to one or more *event handlers*. It's really simple:

```
btn = Button(text='Greet Environment')
def greeting(**args):
    print 'Hello, World!'
link(btn, greeting)
```

The event handling is taken care of by the call to `link`. An event handler may receive several keyword arguments, and if you're not particularly interested in any of them, simply use something like `**args` above. (For more information about this, see the section about global functions in the API Reference below.)

3.10 About Models, Views, and Controllers

The *Anygui* MVC mechanism (based on the `refresh` method and the `Assignee` protocol) is described in the API Reference below. Here is a short overview on how to use it.

A *model* is an object that can be modified, and that can notify other objects, called `views`, when it has been modified. A *controller* is an object that can modify the model, in particular as a direct response to a user action (such as clicking the mouse or typing some text). In *Anygui*, `Components` double as both views (showing a model's state to the user) and controllers (letting the user modify the model). Even though *Anygui* supports using models this way, you can also create complete application without using them.

Models are in general instances of some subclass of the `Model` class, although they don't have to be; see the API Reference below for a description of how they work. (The `Model` class is currently internal to the `Anygui` package, but it can be found int he `anygui.Models` module.) The `Models` that are included in *Anygui* are:

```
BooleanModel     -- represents a Boolean value
ListModel        -- behaves like a list
NumberModel      -- represents a numerical value
TextModel        -- acts like a mutable string
```

These all have a `value` attribute which may be used to change their internal value. They also support other operations, such as indexing and slicing etc. for `ListModel`. These are very easy to use: Just assign them to an attribute of a `Component`:

```
# You'll learn about CheckBoxes in a minute
cbx = CheckBox(text='Simple model test')
state = BooleanModel(value=1)
cbx.on = state
```

Now, if you change `state` (e.g. with the statement `state.value=0`) this will automatically be reflected in the `CheckBox` (which will be acting like a view). If the user clicks the `CheckBox`, the model will be changed.

To keep a view up-to-date manually you can call its `refresh` method. This can be useful if you use a simple (non-`Model`) mutable value such as a list in an attribute:

```
btn = Button()
rect = [0, 0, 10, 10]
btn.geometry = rect
rect[3] = 20
btn.refresh()
```

After modifying `rect`, the button will not have changed, since it can't detect the change by itself. (That's only possible when you use a real model.) Therefore, you call `btn.refresh` to tell it to update itself.

If you assign a value to an attribute, the `refresh` method will be called automatically, so another way of doing the same thing is:

```
btn = Button()
rect = [0, 0, 10, 10]
btn.geometry = rect
rect[3] = 20
btn.geometry = rect
```

Warning: Because of the controller behaviour of `Components`, if the `Button` is resized, `rect` will be modified. If you don't want this behaviour, use a tuple instead of a list, since tuples can't be modified.

If you want another object to monitor a model, you can simply use the `link` method, since all models generate an event (of the type `default`) when they are modified.

Example:

```
from anygui import *
>>> mdl = BooleanModel()
>>> mdl.value = 1
>>> def model_changed(**kw):
>>>     print 'The model has changed!'

>>> link(mdl, model_changed)
>>> mdl.value = 0
The model has changed
```

```
>>> mdl.value = 0
The model has changed
```

Note the last two lines: We haven't really changed the model, but the event handler is called nonetheless. If you want to know whether the model really changed, you must retain a copy of its state, and compare the new value.

3.11 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 link an event handler to it.

Whether a `CheckBox` is currently on or off is indicated by its `on` attribute.

3.12 RadioButtons and RadioGroups

`RadioButtons` are toggle buttons, just like `CheckBoxes`. The main differences are that they look slightly different, and that they should 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.

`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)
```

Note: The behaviour of a `RadioButton` when it does not belong to a `RadioGroup` is not defined by the *Anygui* API, and may vary across backend. Basically, a `RadioButton` without a `RadioGroup` is meaningless; use a `CheckBox` instead.

`RadioGroups` also support an `add` method, as all other *Anygui* container-like objects:

add(*button*)

Adds the *button* to the group, including setting *button*`.group` to the group. As with the other `add` methods, the argument may be either a single object, or a sequence of objects.

3.13 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 stored in its attribute `items`, a sequence of arbitrary objects. The text displayed in the widget will be the result of applying the built-in *Python* function `str` to each object.

```
lbx = ListBox()
lbx.items = '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, a `select` event is generated, which is the default event type for a `ListBox`. This means that you can either do

```
link(lbx, 'select', handler)
```

or

```
link(lbx, handler)
```

with the same result. (This is similar to the `click` event, which is default for `Buttons`; for more information, see the API Reference below.)

3.14 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 send an `enterkey` event (which is the `TextField`'s default event type).

The text in a text component is stored in its `text` property (a string or equivalent), and the current selection is stored in its `selection` property (a tuple of two integer indices).

3.15 Making Your Own Components and LayoutManagers

Currently, you can create your own components by combining others in `Frames`, and wrapping the whole thing up as a class. One of the main reasons for doing this would be to emulate a feature (such as a tabbed pane) available in some backends, but not in others. One could then actually *use* the native version in the backends where it is available (such as `wx`, in this case), and use the "emulation" in the others. There is some limited support for this in the `backend` function (which will allow you to check whether you are currently using the correct backend), but in the future, a more complete API will be developed for this, allowing you access to the coolest features of your favorite GUI package, while staying "package independent".

You can already create your own layout managers, by properly supporting the methods `add`, `remove`, and `resized`. The simplest way of doing this is to subclass `LayoutManager`, which gives you the `add` and `remove` methods for free. You can then concentrate on the method `resized` which takes two positional arguments, `dw`, and `dh` (change in width and change in height) and is responsible for changing the geometries of all the components in the `Frame` the `LayoutManager` is managing. (This frame is available through the private attribute `self._container`.)

To get more control over things, you should probably also override the two internal methods `add_components` and `remove_component`:

`add_components`(self, **items*, ***kws*)

Should add all the components in *items*, and associate them with the options in `kws`, for later resizing.

`remove_component`(self, *item*)

Should remove the given item.

4 API Reference

The following reference describes the full official API of the current version (0.1.1) of *Anygui*.

4.1 Environment Variables

Some environment variables affect the behaviour of the *Anygui* package. These must be set in the environment of the program using *Anygui*. They may either be set permanently through normal operating system channels (check your OS documentation for this), or possibly just set temporarily when running your program. In *Unix* shells like *bash*, you can set the variables on the command line before your comand, like this:

```
foo:~$ ANYGUI_SOMEVAR='some value' python someprogram.py
```

where `ANYGUI_SOMEVAR` is some environment variable used by *Anygui*.

Since *Jython* doesn't support OS environment variables, you'll have to supply them with the command-line switch `-D`:

```
foo:~$ jython -DANYGUI_SOMEVAR='some value' someprogram.py
```

You can also set these environment variables in your own program, by using code like the following before you import *Anygui*:

```
import os
os.environ['ANYGUI_SOMEVAR'] = 'some value'
```

This will probably not work well in *Jython*, though.

The environment variables used by *Anygui* are:

`ANYGUI_WISHLIST`: A whitespace separated list of backend names in the order you wish for *Anygui* to try to use them. The backends are identified with a short prefix such as `wx` for `wxgui`, or `tk` for `tkgui`. For a full list of available backends, see the section "Making Sure You Have a GUI Backend" above. Only the backends in this list will be tried; if you don't set `ANYGUI_WISHLIST`, then the following is the default:

```
'msw gtk java wx tk beos qt curses text'
```

If you insert an asterisk in the wishlist, it will be interpreted as "the rest of the backends, in default order". So, for instance,

```
ANYGUI_WISHLIST='tk wx * text curses'
```

is equivalent to

```
ANYGUI_WISHLIST='tk wx msq gtk java beos qt text curses'
```

Example:

```
foo:~$ ANYGUI_WISHLIST='tk wx qt' python someprogram.py
```

`ANYGUI_DEBUG`: When *Anygui* tries to import a backend, it hides all exceptions, assuming they are caused by the fact that a given backend doesn't work in your installation (because you don't have it installed or something similar). However, at times this may not be the reason; it may simple be that a given backend contains a bug. To track down the bug, set the `ANYGUI_WISHLIST` to some true (in a *Python* sense) value. (If the value supplied can be converted to an integer, it will. Otherwise, it will be treated as a string.) This will make *Anygui* print out the stack traces from each backend it tries to import.

There is one exception to this rule: If the true value supplied is the name of one of the backends (such as `tk` or `curses`) only the traceback caused by importing that backend will be shown. This can be useful to make the output somewhat less verbose.

Example:

```
foo:~$ ANYGUI_DEBUG=1 python someprogram.py
```

`ANYGUI_ALTERNATE_BORDER`: This Boolean variable affects `cursesgui`, making it use the same border-drawing characters as `textgui` (`'+'`, `'-'`, and `'`'—). This may be useful if your terminal can't show the special `curses` box-drawing characters properly.

`ANYGUI_SCREENSIZE`: Affects `textgui`. Gives the terminal ("screen") dimensions, in characters. This should be in the format *widthxheight*, e.g. `80x24`. If this environment variable is not supplied, the standard *Unix* variables `COLUMNS` and `LINES` will be used. If neither is provided, the default size `80x23` will be used.

`ANYGUI_FORCE_CURSES`: Normally, `cursesgui` will not be selected if you are in the interactive interpreter. If you want to force the normal selection order (trying to use `cursesgui` before resorting to `textgui`) you can set this variable to a true value. Note that this is not the same as setting `ANYGUI_WISHLIST` to `'curses'`, since that will ignore all other backends.

`ANYGUI_CURSES_NOHELP`: If you don't want the help-screen that appears when an *Anygui* application is started using `cursesgui` (or `textgui`), you can set this variable to a true value.

4.2 Global Functions

application()

Returns the current `Application` object.

backend()

Returns the name (as used in `ANYGUI_WISHLIST`) of the backend currently in use.

Example:

```
if backend() == 'wx':
    some_wx_code()
else:
    some_generic_code()
```

link(*source*, [*event*,] *handler*, *weak*=0, *loop*=0)

Creates a link in the *Anygui* event system, between the `source` (any object) and the `handler` (any callable, or a `(obj,func)` pair, where `func` is an unbound method or function, and `obj` is an object which will be supplied as the first parameter to `func`). Supplying an `event` (a string) will make the link carry only information about events of that type. If no event is supplied, `'default'` will be assumed. Setting `weak` to a true value will use weak references when setting up the link, so that no objects will be "kept alive" by the link.

A send-loop occurs if an object sends an event "to itself" (i.e. it is the `source` argument of a call to `send` which hasn't returned at the point where one of its methods are about to be activated as a handler). The truth value `loop` decides whether this handler will be activated in such a loop. (If `send` was called with `loop=1`, loops will be allowed anyway.)

Note that `source`, `event`, and `handler` are strictly positional parameters, while the others (`weak`, and `loop`) must be supplied as keyword parameters.

Sometimes one might want an event handler that reacts to a specific event from *any* source, or *any* event from a specific source; or even *any* event from *any* source. To do that, simply use the special value `any` as either source, event, or both.

Example:

```
from anygui import *
>>> def monitor_events(event, **kw):
...     print 'An event occurred:', event
...
>>> link(any, any, monitor_events)
>>> btn = Button()
>>> send(btn, 'foobar')
An event occurred: foobar
```

If you use `send(btn, 'click')` in this example, you will get *two* events, since the `Button` will detect the `click` event (which is its default), and issue a `default` event as well.

Note: You need to explicitly supply the event type if you want to respond to any event type; otherwise you will only respond to the `default` type.

Event handlers that react to the same event will be called in the order they were registered (with `link`), subject to the following: (1) All handlers registered with a specific source will be called before handlers with the value `any` as source; (2) all handlers registered with a specific event (including `default`) are called before handlers with the value `any` as event.

For more information on sending events, see `send`, below.

send(*source*, *event*=`'default'`, *loop*=0, ***kwds*)

When this is called, any handlers (callables) linked to the source, but which will not cause a send-loop (unless `loop` is true) will be called with all the keyword arguments provided (except `loop`), in the order in which they were linked. In addition to the supplied keyword arguments, the event framework will add `source`, `event`, and the time (as measured by the standard *Python* function `time.time`) when `send` was called, supplied with the `time` argument.

Note that `source`, and `event`, are strictly positional parameters, while the others (`loop`, and any additional arguments the user might add) must be supplied as keyword parameters.

Example:

```
# Link an event handler to a button, and then manually send a
```

```
# default event from the button. This event would have been
# sent automatically if we clicked the button. Note that we
# only use the arguments we need, and lump the rest in **kw.

def click(source, time, **kw):
    print 'Button %s clicked at %f.' % (source.text, time)

btn = Button(text='Click me')
link(btn, click)

send(btn) # Fake a button click -- will call click()
```

For information about the order in which event handlers are called, see `link`, above.

Important: Due to the current semantics of the `any` value, using it in `send` may not be a good idea, since the result might not be what you expect. For instance, calling `send(any, any)` will only activate event handlers which have been linked to the value `any` as both source and event, not to "event handlers with any source and any event". This may change in future releases. The current behaviour of `send` with `any` is consistent with `unlink`.

unlink(*source*, [*event*,] *handler*)

Undoes a call to `link` with the same positional arguments. If *handler* has been registered with either *source* or *event* as `any`, that parameter will be irrelevant when deciding whether or not to remove that link. For instance:

```
link(foo, any, bar)
unlink(foo, baz, bar)
```

Here the link created by `link(foo, any, bar)` will be removed by the call to `unlink`.

Note: This behaviour (unlinking handlers registered with the `any` value) may change in future releases.

Default Events: When used without the event argument, both `link` and `send` use an event type called `default`. Most event-generating components have a default event type, such as `click` for `Buttons`. The fact that this event type is default for `Button` means that when a `Button` generates a `click` event it will *also* generate a `default` event. So, if you listen to both `click` events and `default` events from a `Button`, your event handler will always be called twice.

unlinkHandler(*handler*)

Removes a handler completely from the event framework.

unlinkMethods(*obj*)

Unlinks all handlers that are methods of `obj`.

unlinkSource(*source*)

Remove the source (and all handlers linked to it) from the event framework.

4.3 Classes

Base Classes and Common Behaviour

All components are subclasses of corresponding abstract components which implement behaviour common to all the backends. So, for instance, `Button` subclasses `AbstractButton`. These abstract components, again, subclass `AbstractComponent`, which implements behaviour common to all components.

Perhaps the most important behaviour is attribute handling (inherited from the `Attrib` mixin), which means that setting a components attributes may trigger some internal method calls. For instance,

```
win.size = 300, 200
```

will automatically resize the component `win`. Attributes common to all components are:

```
x          -- x-coordinate of upper left corner
y          -- y-coordinate of upper left corner
position   -- equivalent to (x, y)
width      -- component width
height     -- component height
size       -- equivalent to (width, height)
geometry   -- equivalent to (x, y, width, height)
visible    -- whether the component is visible
enabled    -- whether the component is enabled
text       -- text associated with the component
```

These can all be set as keyword arguments to the component constructors. Also, `Options` objects (with similar constructors) can be used as positional arguments in the constructor, with all the `Options`'s attributes being set in the component as well.

Common to `Application`, `Window`, and `Frame` is the `contents` attribute, as well as the `add` and `remove` methods. These will be described with the individual classes below.

All `Attrib` subclasses (including components, `Application`, and `RadioGroup`) share the following methods:

set(**args*, ***kwds*)

Used to set attributes. Works like the `Attrib` constructor, setting attributes, and optionally using `Options` objects.

modify(**args*, ***kwds*)

Works like the `set` method, except that the attributes are modified *in place*. That means the following (for an attribute named `foo`): (1) If there exists an internal method (implemented in *Anygui*) for modifying the attribute inplace (called `_modify_foo`), use that; otherwise (2) try to use slice assignment to change the value (will work for lists and `ListModels` etc.); if that doesn't work, (3) assign to the value's `value` attribute (used to modify `Models`. If neither of these approaches work, simply rebind the attribute (equivalent to using the `set` method).

As with `set` and ordinary attribute assignment, the `refresh` method will automatically be called when you use `modify`.

refresh()

When an attribute of a `Component` (or `Application`, `RadioGroup`, or an instance of another `Attrib` subclass) is assigned a value, the `Component` is automatically updated to reflect its new state. For instance, if you have a `Labellbl`, assigning a value to `lbl.geometry` would immediately change the `Label`'s geometry, and assigning to `lbl.text` would change its text.

This is good enough for most cases, but sometimes an attribute can contain a mutable value, such as a list, and changing that will not update the `Component`. For instance, if you use a list to hold the `items` of a `ListBox`, you could end up in the following situation:

```
lbx = ListBox()
lbx.items = 'first second third'.split()
# More code...
lbx.items.append('fourth')
```

After performing this code, nothing will have happened to the `ListBox`, because it has no way of knowing that the list has changed. To fix that, you can simply call its `refresh` method:

```
lbx.refresh()
```

This method checks whether any attributes have changed, and make sure that the `Component` us up to date.

Updating Automatically

Updating `Components` explicitly can be useful, but sometimes you would want it to be done for you, automatically, each time you modify an object that is referred to by a `Component` attribute. This can be taken care of by `link` and `send`. If your object uses `send` every time it's modified, and you `link` the object to your `Component`'s `refresh` method, things will happen by themselves:

```
class TriggerList:
    def __init__(self):
        self.list = []
    def append(self, obj):
        self.list.append(obj)
        send(self)
    def __getitem__(self, i):
        return self.list[i]
```

```
lbx = ListBox()
lbx.items = TriggerList()
link(lbx.items, lbx.refresh)
```

Now, if we call `lbx.items.append('fourth')`, `lbx.refresh` will automatically be called. To make your life easier, *Anygui* already contains some classes that send signals whend they are modified; these classes are called `Models`.

Model and Assignee

The *Anygui* models (`BooleanModel`, `ListModel`, `TextModel`, and `NumberModel`) are objects that call `send` (with the `'default'` event) when they are modified.

An `Assignee` (part of the *Anygui* Model-View-Controller mechanism) is an object that supports the methods `assigned` and `removed`. These are automatically called (if present) when the object is assigned to one of the attributes of an `Attrib` object (such as a `Component`). `Models` use this behaviour to automatically call `link` and `unlink`, so when the `Model` is modified, the `refresh` method of the `Attrib` object is called automatically.

All models have a `value` attribute, which contains a "simple" version of its state (such as a number for `NumberModel`, a list for `ListModel`, etc.) Assigning to this attribute is a simple way of modifying the model in place.

class Application

To instantiate `Windows`, you must have an `Application` to manage them. You typically instantiate an application at the beginning of your program:

```
app = Application()
# Build GUI and run application
```

In some cases subclassing `Application` might be a useful way of structuring your program, but it is in no way required.

`Application` has the following methods:

run()

Starts the main event loop of the graphical user interface. Usually called at the end of the program which set up the interface:

```
app = Application()
# Set up interface
app.run()
```

add(*win*)

Adds a `Window` to the `Application`, in the same way `Components` can be added to `Frames` (see below). A `Window` will not be visible until it has been added to the current `Application` object, and that `Application is running`. When constructing new `Windows` after `Application.run` has been called, you should ensure that you `add` your `Window` to your running `Application` after all the `Components` have been added to your `Window`; otherwise, you may see them appearing and moving about as *Anygui* takes care of the layout. (Before `Application.run` is called, this is not an issue, since no `Windows` will be appear before that time.)

The parameter `win` can be either a single `Window`, or a sequence of `Windows`.

remove(*win*)

Removes a `Window` from the application. This will make the `Window` disappear.

contents

A read-only property containing a tuple of the `Windows` the `Application` currently manages.

class Button

A component which, when pressed, generate a `'click'` event, as well as a `'default'` event. Thus, in the following example, both `handler1` and `handler2` will be called when the button is pressed:

```
btn = Button()
def handler1(**kw): print 'Handler 1'
def handler2(**kw): print 'Handler 2'
link(btn, 'click', handler1)
link(btn, handler2)
```

class CheckBox

`CheckBox` is a kind of button, and thus will also generate `'click'` and `'default'` events when clicked. But in addition, each `CheckBox` has a Boolean attribute `on`, which is toggled each time the box is clicked. The state of the `CheckBox` can be altered by assigning to this attribute.

The `on` property will be automatically modified (as per the MVC mechanism) when the user clicks the `CheckBox`. This will also cause the `CheckBox` to `send` a `click` and a `default`event.

The `on` attribute is a useful place to use a `BooleanModel`.

class Frame

`Frame` is a component which can contain other components. Components are added to the `Frame` with the `add` method:

add(*comp*, [*opts*,] ***kwds*)

Adds one or more components. The parameter `comp` may be either a single component, or a sequence of components. In the latter case, all the components will be added.

The `opts` parameter containes an `Options` object (see below) which gives information about how the object should be laid out. These options can be overridden with keyword arguments, and all this information will be passed to the `LayoutManager` (see below) of the `Frame`, if any. This `LayoutManager` is stored in the `layout` property.

remove(*comp*)

Removes a component from the `Frame`.

contents

This is a read-only property which contains the contents (a tuple of `Components`) of the `Frame`.

class Label

A `Label` is a simple component which displays a string of text. (`Label` can only handle one line of text.)

class LayoutManager

A layout manager is responsible for setting the `geometry` properties of a set of components when their parent `Frame` changes shape. The default `LayoutManager` (and the only one supplied with the current release) is the `Placer` (see below).

Note: Although Anygui 0.1 comes only with this layout manager, more will appear in the future.

class ListBox

Shows a list of options, of which one may be selected. The `ListBox` has two special attributes: `items`, a sequence of items to display, and `selection`, the currently selected (as an index in the `items` sequence).

The `selection` property will be automatically modified (as per the MVC mechanism) when the user makes a selection. This will also cause the `ListBox` to `send` a `select` and a `defaultevent`.

class Model

See the section on `Model` and `Assignee` above.

class Options

`Options` is a very simple class. It is simply used to store a bunch of named values; basically a dictionary with a different syntax. (For more information about the *bunch* class, see `http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52308`.)

You can set the attributes of an `Options` object and then supply it as an optional first parameter to the constructors of widgets:

```
opt = Options()
opt.width = 100
opt.height = 50
opt.x = 10
btn = Button(opt, y=10)
lbl = Label(opt, y=70)
```

Here `btn` and `lbl` will have the same `width`, `height`, and `x` attributes, but differing `y` attributes.

You can also set the attributes of an `Options` object through its constructur, just like with components:

```
opt = Options(width=100, height=50, x=10)
```

`Options` objects can also be used when supplying arguments to the `add` method of `Frame`:

```
# Assuming a Placer LayoutManager:
opt = Options(left=10, right=10, hstretch=1)
win.add(lbl, opt, top=10)
win.add(btn, opt, top=(lbl,10))
```

class Placer

A simple but powerful layout manager. When adding components to a `Frame` whose `layout` attribute is set to a `Placer`, you can supply the following keyword arguments:

```
left          -- the Component's left edge
right         -- the Component's right edge
top           -- the Component's top edge
bottom        -- the Component's bottom edge
hmove         -- move horizontally on resize
vmove         -- move vertically on resize
hstretch      -- stretch horizontally on resize
vstretch      -- stretch vertically on resize
direction     -- 'left', 'right', 'up', or 'down'
space         -- spacing between multiple Components
```

The geometry specifiers (`left`, `right`, `top`, and `bottom`) can be set to either `None` (the default; will use the `Component`'s existing coordinates), a distance (from the corresponding `Frame` edge), a `Component` (will align the edge with the opposite edge of the given component), or a tuple (*component*, *distance*) (as with only a `Component`, except that a gap of size *distance* is inserted between the two).

The movement arguments (`hmove` and `vmove`) specify (with a Boolean value) whether the `Component` should be moved (horizontally, vertically, or both) to maintain the given distance to the surrounding `Frame`'s edges; the stretching arguments (`hstretch` and `vstretch`) specify whether the `Component` may be stretched to maintain these distances.

class RadioButton

A `RadioButton` is a toggle button, just like `CheckBox`, with slightly different appearance, and with the difference that it belongs to a `RadioGroup`. Only one `RadioButton` can be active (have its `on` attribute be a true Boolean value)in the `RadioGroup` at one time, so when one is clicked or programmatically turned on, the others are automatically switched off by the `RadioGroup`. Each `RadioButton` also has a `value` attribute, which should be unique within its `RadioGroup`. When one `RadioButton` is active, the `value` attribute of its `RadioGroup` is automatically set to that of the active `RadioButton`. The `RadioGroup` of a `RadioButton` is set by assigning the `RadioGroup` to the `group` attribute of the `RadioButton`. Setting the `value` attribute of the `RadioGroup` will automatically activate the correct `RadioButton`.

class RadioGroup

See `RadioButton` above.

class TextArea

A multiline text-editing `Component`. Its text is stored in the `text` attribute, which will be modified (according to the MVC mechanism) when the component loses focus. It also supports the Boolean `editable` property, which may be used to control whether the user can edit the text area or not.

class TextField

A one-line text-editing `Component`. (See also `TextArea`, above.) If the enter/return key is pressed within a `TextField`, the `TextField` will `send` a `enterkey` event.

class Window

A window, plain and simple. `Window` is a type of `Frame`, so you can add components to it and set its `layout` property etc. To make your window appear, you must remember to add it to your `Application`, just like you add other components to `Frames` and `Windows`:

```
win = Window()
app = Application()
app.add(win)
app.run()
```

`Windows` have a `title` attribute which may be used by the operating system or window manager to identify the window to the user in various ways.

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 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://www.anygui.org`, subscribe to the developer's mailing list (`devel@anygui.org`) and the user's list (`users@anygui.org`), 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 potential backend targets may be found at `http://starbase.neosoft.com/~claird/comp.lang.python/python_GUI.html`.

8 Anygui License