Tingbot GUI Documentation

Release 0.9.3

Phil Underwood

Oct 08, 2017

Contents

Contents:

CHAPTER 1

Getting Started

Let's build a simple application to get started - a colour[1] picker. Any colour can be described with three numbers - red, green and blue[2] , so we're going to have three sliders, and a button to display the chosen colour.

Setup

To use this software in a tingapp we have to tell it that the library is required. You will need to create `requirements.txt` if it does not already exist. Add the following to this file to ensure that the tingbot_gui library is loaded:

```
tingbot-gui>=0.9
```

Alternatively the library can be installed by `pip install tingbot-gui`

Next lets create `main.py` and import all the relevant libraries:

```
from tingbot import screen, run
import tingbot_gui as gui

#all the rest of the code goes here

def loop():
    pass

screen.fill("black")
gui.show_all()
run(loop)
```

[1] I'm British, so I use the english spelling of *colour*. However, historically the majority of software was written in the USA, so in software, the standard is to spell it *color*.

[2] Actually there are several ways of specifying a colour, many of which are better than simple red green and blue. However, red green and blue is simplest, so we'll stick with that for this example. See the Wikipedia entry on color spaces for more detail than you can possibly want.

This imports the `tingbot_gui` library and renames it `gui` to simplify the typing. We create an (empty) run loop, and set the whole system going. The second to last line ensures that all of the widgets added are shown. All the rest of the code below should be inserted just below the line that states `#all the rest of the code goes here`

Add some sliders

Lets add some sliders to allow the user to select the individual colour elements. We'll put these on the right hand side of the screen

```
red = gui.Slider((190,20),(25,150),align="top",max_val=255)
green = gui.Slider((240,20),(25,150),align="top",max_val=255)
blue = gui.Slider((290,20),(25,150),align="top",max_val=255)
```

This adds three sliders, one for each colour. We set their position with first pair of numbers, and the size within the second pair. The `align="top"` means that the red slider is positioned with the middle of its top border at (180,20).

Add a button

Now we need to make something happen. Let's add a button, and also create a callback to display the chosen color when it is pressed

```
def display():
    color = (int(red.value),int(green.value),int(blue.value))
    screen.rectangle((20,20),(100,100),color,align="topleft")

button = gui.Button((240,200),(80,30),align="top",label="Display",callback=display)
```

This creates a function `display` which creates a color with values determined by the value of each of the sliders. We use `int` to ensure that the values are integers. `screen.rectangle` draws the selected colour on the screen. The last line creates a button labelled "Display", and tells it to call the `display` function when it is pressed. Note that there is no pair of brackets in the reference to `display`. This means that the *function itself* is passed to the Button. If we put `callback=display()` then the result of the function would be passed (which would be None). At this point you will be able to get your colour picker up and running, but it's not very pretty...

Add some labels

It's not obvious which slider is red, green or blue, so lets add some labels next

```
gui.StaticText((190,0),(50,20),align="top",label="Red")
gui.StaticText((240,0),(50,20),align="top",label="Green")
gui.StaticText((290,0),(50,20),align="top",label="Blue")
```

Again we have used the "top" alignment - this allows us to make sure that all the labels are correctly centered. StaticText will automatically centre its label unless you tell it otherwise. Note that we have not assigned these labels to variables. Because we are not going to do anything more with these labels, we can just declare them.

Add some clever labels with lambda

Lets finally add some numbers to each slider to reflect it's current value.

```
red_label = gui.StaticText((190,180),(50,20),label="0")
green_label = gui.StaticText((240,180),(50,20),label="0")
blue_label = gui.StaticText((290,180),(50,20),label="0")

def update_label(label,value):
    label.label = str(int(value))

red.callback = lambda x: update_label(red_label,x)
green.callback = lambda x: update_label(green_label,x)
blue.callback = lambda x: update_label(blue_label,x)
```

First of all we create some more labels - `red_label`, `green_label` and `blue_label`, and we next take a function `update_label` that takes a label and a value and sets that label to display that value as an integer.

Finally we use a special keyword `lambda`. This creates a temporary function, as if we had written

```
def temp_func(x):
    return update_label(red_label,x)

red.callback = temp_func
```

See the section on *Callbacks* for more on how to use callbacks and lambda.

Add an alert

Lets add a little pop-up notice with the Web RGB code when we display our colour. Lets change the display function.

```
def display():
    color = (int(red.value),int(green.value),int(blue.value))
    screen.rectangle((20,20),(100,100),color,align="topleft")
    gui.message_box(message="RGB code is #%02X%02X%02X" % color)
```

This will bring up a window on top of the screen to tell you what the Web RGB code is. This uses the convenience function *`message_box()`* to display the RGB code. The code will stop here until the user presses on "Ok".

CHAPTER 2

Widgets

There are several different elements that can be used in an interface, known as widgets

class **`Widget`** (*xy*, *size*, *align = "center"*, *parent = None*)

This is the base class for all other widgets, but should not be directly used. All other widgets will have the methods listed below. You can make your own widgets by sub-classing this one. You will need to override the draw method, and possibly the on_touch method. All of the screen drawing methods (`fill`, `rectangle`, `image` and `line`) are also available within this class. See the tingbot-python reference for these methods.

Parameters

- **`xy`** – position that the widget will be drawn
- **`size`** – size of the widget
- **`align`** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **`parent`** (`Container`) – container for this widget. If None, widget will be placed directly on the main screen
- **`style`** (`Style`) – *style* for this widget. If None, the widget will have the default style

Attributes

- *visible* – True if the widget is to be displayed. Widget will be hidden if false
- *surface* – A pygame surface that corresponds to the widgets area - use this in the draw method

`on_touch` (*self*, *xy*, *action*)

Override this method for any widgets that respond to touch events

Parameters

- **`xy`** – position of the touch
- **`action`** – one of "up", "move", "down", "drag", "drag_up". The first touch is recorded as "down". If the touch moves, this is passed as a "move" and when the touch finishes an "up" action is passed. If the widget is within a *`ScrollArea`* then as the touch moves the ScrollArea may start moving it's viewable area - this is passed as a "drag" (and finishes

with a "drag_up"). Widgets may wish to ignore "drag" and "drag_up" events as the user likely wanted to interact with the ScrollArea rather than the specific widget.

update (*self*, *upwards=True*, *downwards=False*)
Call this method to redraw the widget. The widget will only be drawn if visible. It is usually wise to call this function after you have altered a widget.

Parameters

- **upwards** – set to True to ask any parents (and their parents) to redraw themselves
- **downwards** – set to True to make any children redraw themselves

draw (*self*)
Called when the widget needs to draw itself. Override this method for all derived widgets

text (*self*, *string*, *xy=None*, *size=None*, *color='grey'*, *align='center'*, *font=None*, *font_size=32*, *antialias=None*)
Draw some text on to the widget. If the text will not fit on the widget or in size if specified, then try using a smaller font to see if that will fit, minimum 3/4 specified font size. If the text will still not fit, then truncate the text and add an ellipsis (...).

Parameters

- **string** – text to displayed
- **xy** – location to display the text within the widget. If none will be aligned within the widget according to align
- **size** – size for the text to fit within. If None will be fitted within whole widget
- **color** – color for the text. Can be either a text string e.g. "blue" or a RGB e.g (0,0,255)
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **font** – font to use (or the default font if None)
- **font_size** (*int*) – size of font to use
- **antialias** (*bool*) – whether to antialias the text

Returns a list of the x offsets after each letter

class Button (*xy*, *size*, *align="center"*, *parent=None*, *style=None*, *label="OK"*, *callback=None*, *long_click_callback*)
Base: *Widget*

A simple button control

Parameters

- **xy** – position that the button will be drawn
- **size** – size of the button
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **parent** (Container) – container for this button. If None, button will be placed directly on the main screen
- **style** (Style) – *style* for this button. If None, the button will have the default style
- **label** – Text to be displayed on the button. If starts with `image:` then the rest of the string specifies an image file to be displayed on the button.
- **callback** (*callable*) – function to call when the button is pressed. It should not take any arguments

- **long_click_callback** (*callable*) – function to call when the button has been pressed for more than 1.5 seconds. It should not take any arguments

Attributes

- *label* – Text to be displayed on the button. If starts with `image:` then the rest of the string specifies an image file to be displayed on the button. Two image files can be specified by separating them with a |. The second image will be used when the button is pressed
- *callback* – Function to be called when button is clicked. No arguments passed.
- *long_click_callback* – Function to be called when button is pressed for more than 1.5 seconds. No arguments passed. See *Callbacks* for more information

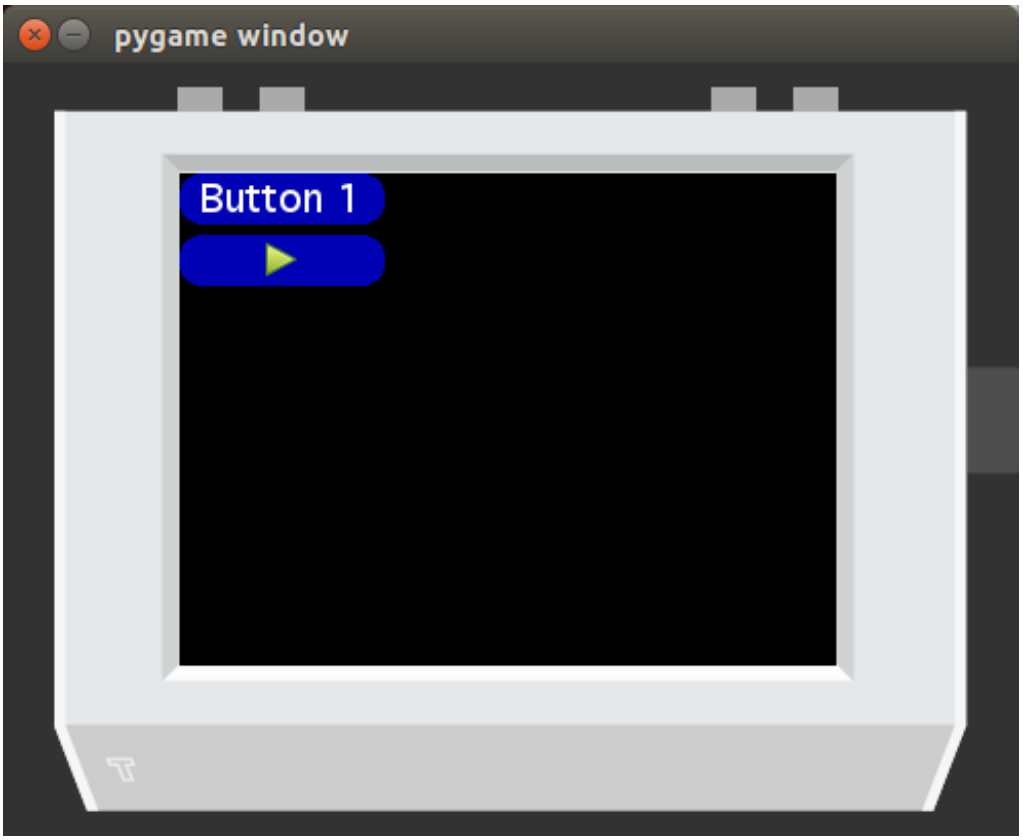
Style Attributes

- *bg_color* – background color
- *button_color* – color of this button when not pressed
- *button_pressed_color* – color to use when button pressed
- *button_rounding* – rounding in pixels of button corners. use 0 for square corners
- *button_text_color* – color to use for text
- *button_text_font* – font to use (default)
- *button_text_font_size* – font size to use
- *button_cancel_on_leave* – if True (default), cancel a button press if the touch leaves the button before release

Example

```
def cb(text):
    print text

button1 = gui.Button((0,0),(100,25),align="topleft",label="Button 1",
             callback = lambda: cb("Button 1"),
             long_click_callback = lambda: cb("Button 1(long"))
button2 = gui.Button((0,30),(100,25),align="topleft",label=
↪"image:player_play.png|player_play_pressed.png",
                     callback = lambda: cb("Button 2(image)"))
```



class `ToggleButton`(*xy*, *size*, *align="center"*, *parent=None*, *style=None*, *label="OK"*, *callback=None*)
Base: `Widget`

A button which can be in an on or off state

Parameters

- **`xy`** – position that the button will be drawn
- **`size`** – size of the button
- **`align`** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **`parent`** (`Container`) – container for this button. If None, button will be placed directly on the main screen
- **`style`** (`Style`) – *style* for this button. If None, the button will have the default style
- **`label`** – Text to be displayed on the button. If starts with `image:` then the rest of the string specifies an image file to be displayed on the button.
- **`callback`** (*`callable`*) – function to call when the button is pressed. It should accept a single boolean value

Attributes

- *label* – Text to be displayed on the button. If starts with `image:` then the rest of the string specifies an image file to be displayed on the button.
- *pressed* – Current state of the button. True if pressed, False if not
- *callback* – Function to be called when button is clicked. A boolean value is passed which is the current state of the button. See *Callbacks* for more information

Style Attributes

- *bg_color* – background color
- *button_color* – color of this button when not pressed
- *button_pressed_color* – color to use when button pressed
- *button_rounding* – rounding in pixels of button corners. use 0 for square corners
- *button_text_color* – color to use for text
- *button_text_font* – font to use (default)
- *button_text_font_size* – font size to use
- *button_cancel_on_leave* – if True (default), cancel a button press if the touch leaves the button before release

Example

```
def cb(text,value):
    print text,value

button2 = gui.ToggleButton((0,30),(100,25),align="topleft",label=
↪"Toggle",
                           callback = lambda x: cb("Toggle Button",
↪x))
```

class **StaticText** (*xy*, *size*, *align="center"*, *parent=None*, *style=None*, *label=""*, *text_align="center"*)

Base: *Widget*

A static text control

Parameters

- **xy** – position that the text widget will be drawn
- **size** – size of the area for text
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **parent** (Container) – container for this text. If None, text will be placed directly on the main screen
- **style** (Style) – *style* for this text. If None, the text will have the default style
- **label** – Text to display
- **text_align** – alignment of text within the widget

Attributes

- *label* – text
- *text_align* – alignment of the text

Style Attributes

- *bg_color* – background color
- *statictext_color* – color to use for text
- *statictext_font* – font to use (default)
- *statictext_font_size* – font size to use

Example

Listing 2.1: Create a static text widget with a dark red background

```
text = gui.StaticText((0,220),(320,20),align="topleft",
                      label="Static Text"
                      style=gui.Style(bg_color=(30,0,0)))
```

class **Slider**(*xy*, *size*, *align = "center"*, *parent = None*, *style = None*, *max_val=1.0*, *min_val=0.0*, *step = None*, *callback=None*)

Base: *Widget*

A sliding control to allow selection from a range of values

Parameters

- **xy** – position that the slider will be drawn
- **size** – size of the slider
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **parent** (Container) – container for this slider. If None, slider will be placed directly on the main screen
- **style** (Style) – *style* for this slider. If None, the slider will have the default style
- **max_val** (*float*) – maximum value for the slider
- **min_val** (*float*) – minimum value for the slider
- **step** – amount to jump by when clicked outside the slider handle. Defaults to one tenth of `max_val-min_val`
- **callback** (*callable*) – function called when the slider is moved. Passed a float which is the sliders new value
- **release_callback** (*callable*) – function called when the slider is released. Passed a float which is the sliders latest value

Attributes

- *value* – Current value of the slider
- *callback* – Function to be called when the slider is moved. A single float is passed.
- *release_callback* – Function to be called when the slider has finished moving. A single float is passed. See *Callbacks* for more information

Style Attributes

- *bg_color* – background color
- *slider_line_color* – color of the line
- *slider_handle_color* – color of the handle

Example

Listing 2.2: Create a horizontal slider with a range of 40-100

```
def cb(text,value):
    print text,value

gui.Slider((0,0),(200,30),align="topleft",
           max_val=100, min_val=40, step=10,
           callback = lambda x: cb("Slider H",x))
```

class **DropDown** (*xy*, *size*, *align="center"*, *parent=None*, *style=None*, *values=None*, *callback=None*)
Base: *Widget*

A widget that displays its current value, and shows a pop-up menu when clicked, allowing the useer to select a new value from a preset list

Parameters

- **xy** – position that the checkbox will be drawn
- **size** – size of the checkbox
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **parent** (*Container*) – container for this checkbox. If None, checkbox will be placed directly on the main screen
- **style** (*Style*) – *style* for this checkbox. If None, the checkbox will have the default style
- **values** – a list of (label,data), one for each menu item. Alternatively [label1,label2,label3] can be used
- **callback** (*callable*) – callback is a function to be called when the selected item is changed. It is passed two arguments, label and data.

Attributes

- *values* – a list of (label,data), one for each menu item
- *selected* – currently selected menu item as a tuple (label,data)
- *callback* – callback is a function to be called when the selected item is changed. It is passed two arguments, label and data. The label is the new label for the control and data is any associated data (if no data was passed in the constructor, then data will be None). See *Callbacks* for more information

Style Attributes

- *bg_color* – background color
- *button_color* – color of this button when not pressed
- *button_pressed_color* – color to use when button pressed
- *button_rounding* – rounding in pixels of button corners. use 0 for square corners
- *button_text_color* – color to use for text
- *button_text_font* – font to use (default)
- *button_text_font_size* – font size to use
- *button_cancel_on_leave* – if True (default), cancel a button press if the touch leaves the button before release
- *popup_bg_color* – color for the background of the popup
- *popupmenu_button_size* – default size for the menu items
- *popupmenu_button_class* – set this to define a custom button class (for example if you want to add icons etc to it). This class must have the same *__init__* parameters as *Button*.

Example

Listing 2.3: Create a dropdown menu with three options, one with associated data, the other two without

```
def cb(label, data):
    print "Dropdown selected: ", label, data

dropdown1 = gui.DropDown((0,60),(100,25),align="topleft",
                         parent = button_panel.scrolled_area,
                         values = ("one",("two","data for item two"),
↪"three"),
                         callback = cb)
```

class CheckBox(*xy*, *size*, *align="center"*, *parent=None*, *style=None*, *label="OK"*, *callback=None*)

Base: *Widget*

A checkbox control

Parameters

- **xy** – position that the checkbox will be drawn
- **size** – size of the checkbox
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **parent** (Container) – container for this checkbox. If None, checkbox will be placed directly on the main screen
- **style** (Style) – *style* for this checkbox. If None, the checkbox will have the default style
- **label** – Text to display
- **callback** (*callable*) – function to call when the button is pressed. Is passed True if checkbox ticked, False otherwise

Attributes

- *label* – Text to be displayed.
- *value* – Current status of the checkbox - True for checked, False for unchecked
- *callback* – Function to be called when the checkbox is clicked. Is passed True if checkbox ticked, False otherwise See *Callbacks* for more information

Style Attributes

- *bg_color* – background color
- *checkbox_color* – color of the checkbox
- *checkbox_text_color* – color to use for text
- *checkbox_text_font* – font to use (default)
- *checkbox_text_font_size* – font size to use

Example

Listing 2.4: Create a checkbox control

```
def cb(label, data):
    print label, data

gui.CheckBox((0,0),(100,25), align="topleft",
```

```
            label="Checkbox",
            callback=lambda x:cb("Checkbox",x))
```

Radio Buttons

Radio buttons are similar to checkboxes, but only one in a group can be selected at any one time. As they need to be part of a group, a *RadioButton* cannot exist by itself - it needs to be part of a *RadioGroup*.

Listing 2.5: Example: create a set of radiobuttons

```
group = gui.RadioGroup()
radio1 = gui.RadioButton((100,80),(200,20),label="Radio 1",value=1,group=group)
radio2 = gui.RadioButton((100,110),(200,20),label="Radio 2",value=2,group=group)
radio3 = gui.RadioButton((100,140),(200,20),label="Radio 3",value=3,group=group)
```

class RadioGroup(*callback = None*)

Base: object

A group of RadioButtons

Parameters **callback** (*callable*) – function to call when one of the radio buttons is pressed. Will be passed two arguments - first is the buttons label, second is it's value See *Callbacks* for more information

Attributes

- *selected* – Currently selected RadioButton

class RadioButton(*xy*, *size*, *align="center"*, *parent=None*, *style=None*, *label=""*, *value=None*, *group=None*, *callback=None*)

Base: *Widget*

A radio button control

Parameters

- **xy** – position that the radio button will be drawn
- **size** – size of the radio button
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **parent** (Container) – container for this radio button. If None, radio button will be placed directly on the main screen
- **style** (Style) – *style* for this radio button. If None, the radio button will have the default style
- **label** – Text to display
- **value** – Value for this RadioButton, set to label if not specified
- **group** (RadioGroup) – RadioGroup that this Button will be part of.
- **callback** (*callable*) – function to call when the button is pressed. It should not take any arguments

Attributes

- *label* – text to displayed
- *value* – data associated with this radio button

- *pressed* – whether this radio button is pressed or not
- *callback* – function to call when the radio button is pressed. It should not take any arguments See *Callbacks* for more information

Style Attributes

- *bg_color* – background color
- *radiobutton_color* – color of the RadioButton
- *radiobutton_text_color* – color to use for text
- *radiobutton_text_font* – font to use (default)
- *radiobutton_text_font_size* – font size to use

CHAPTER 3

Input

There are two widgets that can be used to enter text, and one dialog

class **Keyboard**(*label*, *text=""*, *style=None*, *callback=None*)

Base: *Dialog*

This dialog box allows you to enter text using an on-screen keyboard. It has lower case, upper case, number and symbols screen. There is a button to add emojis, but it is not currently functional

Parameters

- **label** – Text to display at the top of the dialog e.g. "Username"
- **text** – Text to put in the box for editing e.g. "JohnSmith4001"
- **style** (*Style*) – *style* for this keyboard. If None, the keyboard will have the default style
- **callback** (*callable*) – Function to call when the keyboard is exited. It will be passed a single variable which is the completed string (or None if cancel is pressed)

Attributes

- *callback* – Function to call when the keyboard is exited. It will be passed a single variable which is the completed string (or None if cancel is pressed). See *Callbacks* for more information

Style Attributes

- *bg_color* – background color
- *textentry_bg_color* – color of text entry box
- *textentry_text_color* – color to use for text
- *textentry_text_font* – font to use (default)
- *textentry_text_font_size* – font size to use
- *button_color* – color of the buttons when not pressed

- *button_inverting* – if True, the buttons will use bg_color for text and button_text_color for background when pressed. If False, will use button_pressed_color as background color when pressed
- *button_pressed_color* – color to use when buttons pressed
- *button_rounding* – rounding in pixels of button corners when button_inverting is False. use 0 for square corners
- *button_text_color* – color to use for text
- *button_text_font* – font to use (default)
- *statictext_color* – color to use for text at top of screen (i.e. Cancel, Title, Ok)
- *statictext_font* – font to use (default)
- *statictext_font_size* – font size to use

Example

```
def cb(text):
    print text

Keyboard("Text", "Happy World!", callback=cb)
```

class **TextEntry** (*xy*, *size*, *align="center"*, *parent=None*, *style=None*, *label=""*, *text=""*, *callback=None*)

Base: *Button*

A Text entry widget - shows as a box with text inside it. Clicking on the box will bring up a keyboard to enter new text.

Parameters

- **xy** – position that the TextEntry will be drawn
- **size** – size of the TextEntry
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **parent** (Container) – container for this TextEntry. If None, TextEntry will be placed directly on the main screen
- **style** (Style) – *style* for this TextEntry. If None, the TextEntry will have the default style
- **label** – title for the Keyboard when it is shown
- **string** – text to display in the TextEntry
- **callback** (*callable*) – Function to call when the string is changed. It will be passed a single variable which is the new string.

Attributes

- *label* – title for the Keyboard when it is shown
- *string* – current text of the TextEntry
- *callback* – Function to call when the string is changed. It will be passed a single variable which is the new string. See *Callbacks* for more information

Style Attributes

- *bg_color* – background color
- *textentry_bg_color* – color of TextEntry box
- *textentry_text_color* – color to use for text
- *textentry_text_font* – font to use (default)
- *textentry_text_font_size* – font size to use

class PasswordEntry(*xy*, *size*, *align="center"*, *parent=None*, *style=None*, *label=""*, *text=""*, *callback=None*)

Base: *TextEntry*

A Password entry widget - shows as a box with a series of dots (•) inside it. Clicking on the box will bring up a keyboard to enter new text (but the current text will not be displayed). Ideal for where you don't want a passerby to see any passwords.

Parameters

- **xy** – position that the TextEntry will be drawn
- **size** – size of the TextEntry
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **parent** (Container) – container for this TextEntry. If None, TextEntry will be placed directly on the main screen
- **style** (Style) – *style* for this TextEntry. If None, the TextEntry will have the default style
- **label** – title for the Keyboard when it is shown
- **string** – text to display in the TextEntry
- **callback** (*callable*) – Function to call when the string is changed. It will be passed a single variable which is the new string.

Attributes

- *label* – title for the Keyboard when it is shown
- *string* – current text of the TextEntry
- *callback* – Function to call when the string is changed. It will be passed a single variable which is the new string. See *Callbacks* for more information

Style Attributes

- *bg_color* – background color
- *textentry_bg_color* – color of TextEntry box
- *textentry_text_color* – color to use for text
- *textentry_text_font* – font to use (default)
- *textentry_text_font_size* – font size to use

CHAPTER 4

Containers

Containers can be used to group widgets together. ScrollAreas can be used to access more widgets than can fit on the screen otherwise.

class Container
Base: *Widget*

A base class for ScrollAreas and Panels

add_child(*self*, *widget*)

Parameters **widget** (Widget) – The widget to be added to this container

Adds a widget to this container. This should rarely be called as the widget will call this itself on initiation

remove_child(*self*, *widget*)

Parameters **widget** (Widget) – The widget to be added to this container

Remove a widget from this container

remove_all(*self*)
Removes all widgets from the container

class Panel(*xy*, *size*, *align="center"*, *parent=None*, *style=None*)
Base: *Container*

Panel class, allows you to collect together various widgets and turn on or off as needed

Parameters

- **xy** – position that the widget will be drawn
- **size** – size of the widget
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **parent** (Container) – container for this widget. If None, widget will be placed directly on the main screen
- **style** (Style) – *style* for this widget. If None, the widget will have the default style

class ScrollArea (*xy*, *size*, *align="center"*, *parent=None*, *style = None*, *canvas_size=None*)
Base: *Container*

ScrollArea gives a viewing area into another, usually larger area. This allows the user to access more widgets than will fit on the display. Scrollbars will be added to the bottom or right edges as needed. The ScrollArea can be moved by dragging within its area. A fast drag will initiate a "flick" where the ScrollArea carries on scrolling after the drag finishes. It will gradually slow and stop, unless it runs out of room.

Parameters

- **xy** – position that the widget will be drawn
- **size** – size of the widget
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **parent** (Container) – container for this widget. If None, widget will be placed directly on the main screen
- **style** (Style) – *style* for this widget. If None, the widget will have the default style
- **canvas_size** – size of the scrollable area (required)

Attributes

- *scrolled_area* – Use this as the parent for any widgets you wish to place within this container

Style Attributes

- *scrollbar_width* – width of the scrollbars
- *slider_line_color* – color of the line
- *scrollarea_flick_decay* – speed that a flick decreases by in pixels per second per second (default 600)
- *scrollarea_min_flick_speed* – speed below which a flick stops in pixels per second (default 60)
- *scrollarea_flick_threshold* – speed required to generate a flick in pixels per second (default 100)
- *slider_handle_color* – color of the handle

resize_canvas (*self*, *canvas_size*)

Parameters **canvas_size** – size of the scrollable area

Resize the scrollable area. Will raise an error if the given size is smaller than required to display all the widgets

Example

Listing 4.1: Create a scrolled area with a size of (500,500)

```
    scroller = gui.ScrollArea((100,0),(135,220),align="topleft",
↪canvas_size=(500,500))
```

class RootWidget
Base: *Container*

There is only ever one RootWidget and it is generated automatically. All widgets that do not explicitly have a parent set are children of this widget.

get_root_widget()
Returns the RootWidget

show_all()
Tells the RootWidget to display all it's children. It's generally useful to call this function immediately before calling the main run loop - this ensures that all the children you have added are drawn on the screen when the program starts.

class NoteBook(*pairs*)
Base: object

A NoteBook allows you to control a set of Panels with a set of ToggleButtons. Set all of the panels to cover the same area, and pressing each button will make the associated panel visible, and hide the others. ScrollAreas can also be used.

Parameters **pairs** – a list of the form [(button1,panel1), (button2,panel2) ...]. panel1,panel2 etc should all occupy the same screen real estate, whereas button1,button2 should be in distinct locations.

Listing 4.2: Example: create three panels and buttons and use them to create a NoteBook

```
but1 = gui.ToggleButton((30,30),(60,60),label="1")
but2 = gui.ToggleButton((30,100),(60,60),label="2")
but3 = gui.ToggleButton((30,170),(60,60),label="3")
panel1 = gui.Panel((0,70),(320,170))
panel2 = gui.Panel((0,70),(320,170))
panel3 = gui.Panel((0,70),(320,170))

nb = NoteBook([(but1,panel1), (but2,panel2), (but3,panel3)])
```

CHAPTER 5

Dialog windows

Dialog windows are modal - this means that only the specified window is active while it is open This is particularly useful for alert boxes and also pop-up menus

class **Dialog**(*xy=None*, *size=None*, *align="center"*, *style=None*, *buttons=None*, *message=""*, *cancellable=True*, *callback=None*, *transition="popup"*)

Base: *Container*

Dialog is a base class you can use to create your own dialogs. Call close to make the dialog disappear. Place widgets with self.panel as the parent, not self

Parameters

- **xy** – position that the dialog will be drawn
- **size** – size of the dialog
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **style** (*Style*) – *style* for this dialog. If None, the dialog will have the default style
- **cancellable** (*bool*) – If true, then the dialog can be closed by clicking outside its area. Any callback specified will be passed None as it's argument
- **callback** (*callable*) – A callback that will be called when the dialog is closed, one argument is passed, which is whatever the close method is called with.
- **transition** –
  - "popup" – the dialog appears on screen as specified by xy, size and align.
  - "slide_left" – the dialog slides in to the left, width as per size, xy and align ignored
  - "slide_right" – the dialog slides in to the right, width as per size, xy and align ignored
  - "slide_left" – the dialog slides in downwards, height as per size, xy and align ignored
  - "slide_left" – the dialog slides in upwards, height as per size, xy and align ignored

Attributes

- *cancellable* – If true, then the dialog can be closed by clicking outside its area. Any callback specified will be passed None as it's argument
- *callback* – A callback that will be called when the dialog is closed, one argument is passed, which is whatever the close method is called with. See *Callbacks* for more
- *panel* – A panel to place widgets on - this allows the dialog to implement the sliding operation

Style Attributes

- *scrollbar_width* – width of the scrollbars
- *slider_line_color* – color of the line
- *slider_handle_color* – color of the handle

close(*self*, *ret_value=None*)

Parameters **ret_value** – value to be returned to the callback function

Close this modal window and return ret_value to the callback function

run(*self*)

Runs the dialog in blocking mode - i.e. execution of other code will stop until the dialog has been closed. Scheduled events via *once* and *every* will continue to run.

Returns whatever *self.close* was called with

class MessageBox(*xy=None*, *size=None*, *align="center"*, *style=None*, *buttons=None*, *message=""*, *cancellable=True*, *callback=None*)

Base: *Dialog*

A MessageBox allows you to alert the user to simple events, and also ask simple Yes/No type questions

Parameters

- **xy** – position that the dialog will be drawn (in the centre if None)
- **size** – size of the dialog, (280x200 if None)
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **style** (*Style*) – *style* for this MessageBox. If None, the MessageBox will have the default style
- **buttons** – A list of labels for buttons to be shown. Three can be displayed with the default window size and button size. Defaults to ["OK"] if None
- **message** (*string*) – Text to display in the MessageBox
- **cancellable** (*bool*) – If true, then the MessageBox can be closed by clicking outside its area. Any callback specified will be passed None as it's argument
- **callback** (*callable*) – A callback that will be called when the MessageBox is closed, one argument is passed, which is the label of the button that was pressed (or None if cancelled)

Attributes

- *cancellable* – If true, then the MessageBox can be closed by clicking outside its area. Any callback specified will be passed None as it's argument
- *callback* – A callback that will be called when the MessageBox is closed, one argument is passed, which is the label of the button that was pressed (or None if cancelled) See *Callbacks* for more

Style Attribute

- *bg_color* – background color
- *button_color* – color of buttons when not pressed
- *button_pressed_color* – color to use when button pressed
- *button_rounding* – rounding in pixels of button corners. use 0 for square corners
- *button_text_color* – color to use for button text
- *button_text_font* – font to use (default font if None)
- *button_text_font_size* – font size to use
- *messagebox_button_size* – size to use for buttons
- *statictext_color* – color to use for message text
- *statictext_font* – font to use (default font if None)
- *statictext_font_size* – font size to use

Example

Listing 5.1: Find out if the user likes cheese

```
def cb(name,value=None):
    print name, value

gui.MessageBox(message="Do you like cheese?",
               buttons=["Yes","No","Maybe"],
               cancellable=False,
               callback = lambda x:cb("Cheese?",x))
```

message_box(*xy=None*, *size=None*, *align="center"*, *style=None*, *buttons=None*, *message=""*, *cancellable=True*)

Utility function. Call this to create a message_box, wait until a button is pressed, and return the value of that button (or None if *cancellable* is true and the user clicks outside the box)

Parameters

- **xy** – position that the dialog will be drawn (in the centre if None)
- **size** – size of the dialog, (280x200 if None)
- **align** – one of topleft, left, bottomleft, top, center, bottom, topright, right, bottomright
- **style** (Style) – *style* for this MessageBox. If None, the MessageBox will have the default style
- **buttons** – A list of labels for buttons to be shown. Three can be displayed with the default window size and button size. Defaults to ["OK"] if None
- **message** (*string*) – Text to display in the MessageBox
- **cancellable** (*bool*) – If true, then the MessageBox can be closed by clicking outside its area. Any callback specified will be passed None as it's argument

Example

Listing 5.2: Find out if the user likes cheese

```
    cheese_preference = gui.message_box(message="Do you like cheese?",
                                        buttons=["Yes","No","Maybe"])
```

class PopupMenu (*xy*, *style=None*, *cancellable=True*, *menu_items=None*, *button_size=None*)

Base: *Dialog*

A PopupMenu (also known as a context menu) allows you to present the user with a menu

Parameters

- **xy** – position for the topleft of the menu. However, the menu may be adjusted so that it fits on the screen. If the menu is so long that it cannot fit on the screen, a scrollbar will be provided
- **style** (*Style*) – *style* for this PopupMenu. If None, the PopupMenu will have the default style
- **cancellable** (*bool*) – If true, then the PopupMenu can be closed by clicking outside its area.
- **menu_items** – is a list of the form [(label,callback)...], one for each entry in the menu. callback takes no arguments and will be called if that menu item selected.
- **button_size** – a size parameter for each button in the popupmenu. If none, button_size will be taken from the style.

Attributes None

Style Attributes

- *bg_color* – background color
- *button_pressed_color* – color to use when menu item pressed
- *button_text_color* – color to use for text
- *button_text_font* – font to use (default)
- *button_text_font_size* – font size to use
- *popup_bg_color* – color for the background of the popup
- *popupmenu_button_size* – default size for the menu items
- *popupmenu_button_class* – set this to define a custom button class (for example if you want to add icons etc to it). This class must have the same *__init__* parameters as *Button*.

Example

Listing 5.3: Bring up a Popup prompting to Open or Save

```
    def open_fn():
        print "open"

    def save_fn():
        print "save"

    gui.PopupMenu((160,100), menu_items = [("Open",open_fn), ("Save",
↪save_fn)])
```

CHAPTER 6

Styles

The appearance of many of the gui components (also known as widgets) are highly configurable. There are three ways to alter the style of the interface

1. Use the default style. This will affect every single widget, even ones that have already been created.

   Listing 6.1: Example: alter the background color to blue for every widget

   ```
   style = gui.get_default_style()
   style.bg_color = "blue"
   ```

2. Use a custom style. This allows you to customize individual widgets or groups of widgets

   Listing 6.2: Example: create two buttons with a smaller font size

   ```
   custom_style = gui.Style(button_text_font_size = 12)
   button1 = gui.Button((50,50),(80,80),label="Small text",style=custom_style)
   button2 = gui.Button((150,50),(80,80),label="Small again",style=custom_style)
   button3 = gui.Button((250,50),(80,80),label="OK")
   ```

   Custom made styles will all inherit the default settings, so you only need to specify those items that need to be altered

3. Update a pre-existing style. This allows you to use a pre-existing style and only change those attributes which need to be changed

   Listing 6.3: Example: update a pre-existing style with a new background_color

   ```
   my_new_style = my_old_style.copy(bg_color="teal")
   ```

Styles can be updated dynamically, even after the widget has been created

class **Style**(***kwargs*)

Parameters ****kwargs** – specify style attributes as required

Returns A new Style with attributes set as per kwargs, all others are as per default settings

copy(*self*, ***kwargs*)

Parameters ****kwars** – specify style attributes to update as required

Returns A new Style with attributes set as per kwargs, all others are as per the original style.

get_default_style()

Returns The default style.

CHAPTER 7

Callbacks

Basic usage

Several classes use callbacks to respond to user events. The simplest of these take no arguments

Listing 7.1: Example: respond to a button press

```
def button_callback():
    screen.text("Button pressed")

but = gui.Button((40,40),(80,80),label="Button",callback = button_callback)
```

Notice that `button_callback` has no brackets when passed to the Button. Other callbacks will take a value dependent on the state of the widget. For example, the callback for a slider will pass it's current value as a float

Listing 7.2: Example: display the value of a slider

```
def slider_callback(value):
    screen.rectangle((0,0),(320,200),"black","topleft")
    screen.text("%d" % int(value))

slider = gui.Slider((0,200),(320,20),align="topleft",
                    max_val = 200, callback = slider_callback)
```

Passing extra arguments to callbacks

Sometimes it is useful to pass an extra value to the callback, if you have several widgets, where you want to use the same callback. This can be done using `lambda`.

Listing 7.3: Example: display which button was pressed

```
def button_callback(name):
    screen.rectangle((0,80),(320,240),"black","topleft")
    screen.text("Button %s pressed" % name)

but1 = gui.Button((40,40),(80,80),label="1",callback = lambda : button_callback("1"))
but2 = gui.Button((130,40),(80,80),label="2",callback = lambda : button_callback("2"))
but3 = gui.Button((220,40),(80,80),label="3",callback = lambda : button_callback("3"))
```

If the callback should be passed a value from the widget, then you need to use the form `lambda x:` as below.

Listing 7.4: Example: display which slider has changed

```
def slider_callback(value,name):
    screen.rectangle((0,0),(320,100),"black","topleft")
    screen.text("Slider %s: %d" % (name,int(value)),(160,50))

sld1 = gui.Slider((120,110),(230,20),max_val=200,
                  callback = lambda x: slider_callback(x,"1"))
sld2 = gui.Slider((120,150),(230,20),max_val=200,
                  callback = lambda x: slider_callback(x,"2"))
sld3 = gui.Slider((120,190),(230,20),max_val=200,
                  callback = lambda x: slider_callback(x,"3"))
```

For more information about `lambda` the Mouse vs Python blog is a good summary of the subject.

CHAPTER 8

Full example

Here is a fully worked example including all the widgets available. The top layer is a notebook, with toggle buttons down the left side.

```
#!/usr/bin/env python
from tingbot import screen, run
from tingbot.graphics import _xy_from_align
import tingbot_gui as gui

action_text = gui.StaticText((0,220),(320,20),align="topleft",
                             style=gui.Style(bg_color=(30,0,0)))

def cb(name,value=None):
    text = name + " actioned: " +str(value)
    action_text.label = text
    print text
    action_text.update()

panel_layouts = {'xy':(85,0),
                 'size':(235,220),
                 'align':"topleft"}
labels = ('Basics','Slider','Text','Buttons','Dialogs','Dynamic')
notebook_style = gui.get_default_style().copy()
notebook_style.bg_color = (25,25,25)
notebook_args = {'size':(80,28),'align':"topleft",'style':notebook_style}
notebook_buttons = [gui.ToggleButton((0,30*i),label=l,**notebook_args)
                    for i,l in enumerate(labels)]

#basic control panel
basic_panel = gui.Panel(**panel_layouts)
gui.CheckBox((0,0),(100,25), align="topleft",
             label="Checkbox",
             parent = basic_panel,
             callback=lambda x:cb("Checkbox",x))
radio_group = gui.RadioGroup(callback=cb)
gui.RadioButton((0,30),(100,25),align="topleft",
```

```
                label="Radio 1",
                parent = basic_panel,
                group = radio_group,
                callback=lambda :cb("Radio Button 1"))
gui.RadioButton((0,60),(100,25),align="topleft",
                label="Radio 2",  value="R2 value",
                parent = basic_panel, group = radio_group)
gui.RadioButton((0,90),(100,25),align="topleft",
                label="Radio 3",
                parent = basic_panel, group = radio_group)

#slider panel
slider_panel = gui.Panel(**panel_layouts)
gui.Slider((0,0),(200,30),align="topleft",
           max_val=100,min_val=40, step=15,
           parent=slider_panel,
           callback = lambda x: cb("Slider H",x))
gui.Slider((0,40),(30,180),align="topleft",
           parent=slider_panel,
           callback = lambda x: cb("Slider V",x),
           release_callback = lambda x: cb("Slider V Release",x))

#text panel
text_panel = gui.ScrollArea(canvas_size=(500,500),
                            style=gui.Style(bg_color="navy"),
                            **panel_layouts)
positions = ['topleft', 'left', 'bottomleft',
             'top', 'center', 'bottom',
             'topright', 'right', 'bottomright']
scrollarea = text_panel.scrolled_area
args = [{'xy':_xy_from_align(x,(500,500)),
         'size':(80,50),
         'align':x,
         'text_align':x,
         'label':x} for x in positions]
texts = [gui.StaticText(parent=scrollarea,**a) for a in args]

#button panel
no_cancel_style = gui.Style(button_cancel_on_leave=False)
button_panel = gui.ScrollArea(canvas_size=panel_layouts['size'],**panel_layouts)
button1 = gui.Button((0,0),(100,25),align="topleft",label="Button 1",
                     parent = button_panel.scrolled_area,
                     callback = lambda: cb("Button 1"),
                     long_click_callback = lambda: cb("Button 1(long)"))
button2 = gui.Button((0,30),(100,25),align="topleft",label="image:player_play.png",
                     parent = button_panel.scrolled_area,
                     callback = lambda: cb("Button 2(image)"),
                     style = no_cancel_style)
button3 = gui.ToggleButton((0,60),(100,25),align="topleft",label="Tog But",
                           parent = button_panel.scrolled_area,
                           callback = lambda x: cb("Toggle Button",x))
button4 = gui.ToggleButton((0,90),(100,25),align="topleft",label="Tog But NC",
                           parent = button_panel.scrolled_area,
                           callback = lambda x: cb("Toggle Button (no cancel)",x),
                           style=no_cancel_style)
dropdown1 = gui.DropDown((0,120),(100,25),align="topleft",
                         parent = button_panel.scrolled_area,
                         values = ("DD one",("two","data for item two")),
```

```
                          callback = lambda x,y:cb("DropDown1",(x,y)))
dropdown2 = gui.DropDown((0,150),(100,25),align="topleft",
                          parent = button_panel.scrolled_area,
                          values = range(100),
                          callback = lambda x,y:cb("DropDown2",(x,y)))
dropdown3 = gui.DropDown((0,190),(100,25),align="topleft",
                          parent = button_panel.scrolled_area,
                          values = ("DD 3","two",3,"ridiculously long text here"),
                          callback = lambda x,y:cb("DropDown3",(x,y)))

#dialog panel
class DialogDemo(gui.Dialog):
    def __init__(self,transition):
        if transition in ('slide_left','slide_right'):
            size = (150,240)
        if transition in ('slide_up','slide_down'):
            size = (320,150)
        super(DialogDemo,self).__init__(size=size,transition=transition)
        button = gui.Button(xy=(75,75), size=(50,50), label="ok", callback=self.close,
↪ parent=self.panel)

dialog_panel = gui.Panel(**panel_layouts)

def alert():
    cb("Alert dialog",gui.message_box(message="Alert triggered"))

def question():
    dialog_style = gui.Style(statictext_color="red")
    msg = gui.MessageBox(message="Do you like cheese?\nEven Camembert?",
                          buttons=["Yes","No","Maybe"],
                          cancellable=False,
                          callback = lambda x:cb("Question dialog",x),
                          style=dialog_style)
    msg.run()

def popup(xy = (160,120)):
    gui.popup_menu(xy, menu_items = [("File",lambda: cb("File (Popup)")),
                                     ("Save",lambda: cb("Save (Popup)"))])

def slide_in():
    for x in ['slide_up','slide_left','slide_down','slide_right']:
        cb("Dialog slide",x)
        DialogDemo(x).run()

gui.Button((0,0),(80,25),align="topleft",parent = dialog_panel,
           label="Alert",callback=alert)
gui.Button((0,30),(80,25),align="topleft",parent = dialog_panel,
           label="Question",callback=question)
gui.Button((0,60),(80,25),align="topleft",parent = dialog_panel,
           label="Popup",callback=popup)
gui.Button((0,90),(80,25),align="topleft",parent = dialog_panel,
           label="Popup2",callback=lambda: popup((310,230)))
gui.Button((0,120),(80,25),align="topleft",parent = dialog_panel,
           label="Slide-ins",callback=slide_in)
gui.TextEntry((0,150),(160,25),align="topleft",parent = dialog_panel,
           label="Text Entry",callback=lambda x: cb("TextEntry",x))
gui.PasswordEntry((0,180),(160,25),align="topleft",parent = dialog_panel,
           label="Password",callback=lambda x: cb("PasswordEntry",x))
```

```
148
149 #dynamic list panel
150 dynamic_panel = gui.Panel(**panel_layouts)
151 scroller = gui.ScrollArea((100,0),(135,220),align="topleft",
152                           parent=dynamic_panel,canvas_size=(135,75))
153
154 def make_button(index,pos,text="Dynamic"):
155     label = text + ' ' + str(index)
156     button = gui.PopupButton((0,pos*25),(135,25),align="topleft",
157                              parent = scroller.scrolled_area,
158                              label = label,
159                              callback = lambda: cb(label))
160     return button
161
162 button_list = [make_button(i,i,"Original") for i in range(3)]
163 count = 3
164
165 def add_item():
166     global count
167     cb("Add dynamic item",count)
168     scroller.resize_canvas((135,(len(button_list)+1)*25))
169     button_list.append(make_button(count,len(button_list)))
170     count += 1
171     scroller.update(downwards=True)
172
173 def remove_last_item():
174     cb("Remove dynamic item",'last')
175     if button_list:
176         scroller.scrolled_area.remove_child(button_list.pop())
177         scroller.resize_canvas((135,max(len(button_list)*25,10)))
178         scroller.update(downwards=True)
179
180 def remove_all():
181     cb("Remove dynamic items",'all')
182     scroller.scrolled_area.remove_all()
183     scroller.resize_canvas((135,10))
184     button_list[:] = []
185     scroller.update(downwards=True)
186
187 gui.Button((0,0),(90,25),align="topleft",parent=dynamic_panel,
188            label="Add item",callback=add_item)
189 gui.Button((0,30),(90,25),align="topleft",parent=dynamic_panel,
190            label="Del last",callback=remove_last_item)
191 gui.Button((0,60),(90,25),align="topleft",parent=dynamic_panel,
192            label="Del all",callback=remove_all)
193
194
195 def nb_cb(button,panel):
196     print "Notebook panel changed: " +button.label
197
198 notebook_panels = [basic_panel,slider_panel,text_panel,
199                    button_panel,dialog_panel,dynamic_panel]
200 nb = gui.NoteBook(zip(notebook_buttons,notebook_panels),callback = nb_cb)
201 print "Current notebook tab: " + nb.selected.label
202
203 gui.get_root_widget().fill(notebook_style.bg_color)
204 gui.show_all()
205 def loop():
```

```
206     pass
207
208 run(loop)
```

CHAPTER 9

Graphical user interface

This module provides a graphical user interface for the tingbot. It allows you to include some of the commoner elements (also known as widgets) of your phone or computer interface within your application, rather than having to develop them from scratch.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Index