Toga Documentation

Release 0.3.0.dev1

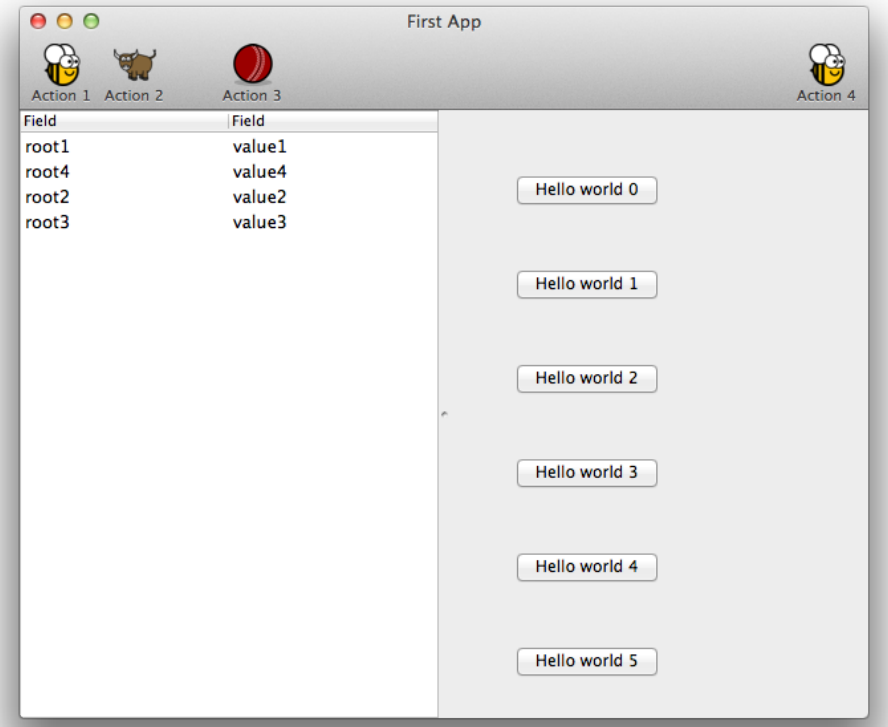
Russell Keith-Magee

Nov 14, 2017

Contents

Toga is a Python native, OS native, cross platform GUI toolkit. Toga consists of a library of base components with a shared interface to simplify platform-agnostic GUI development.

Toga is available on Mac OS, Windows, Linux (GTK), and mobile platforms such as Android and iOS.



CHAPTER 1

Table of contents

1.1 Tutorial

Get started with a hands-on introduction to Toga for beginners

1.2 How-to guides

Guides and recipes for common problems and tasks

1.3 Background

Explanation and discussion of key topics and concepts

1.4 Reference

Technical reference - commands, modules, classes, methods

CHAPTER 2

Community

Toga is part of the BeeWare suite. You can talk to the community through:

- @pybeeware on Twitter
- pybee/general on Gitter

2.1 Tutorials

2.1.1 Your first Toga app

In this example, we're going to build a desktop app with a single button, that prints to the console when you press the button.

Here's a complete code listing for our "Hello world" app:

```
import toga


def button_handler(widget):
    print("hello")


def build(app):
    box = toga.Box()

    button = toga.Button('Hello world', on_press=button_handler)
    button.style.set(margin=50)
    box.add(button)

    return box

def main():
    return toga.App('First App', 'org.pybee.helloworld', startup=build)
```

```
if __name__ == '__main__':
    main().main_loop()
```

Let's walk through this one line at a time.

The code starts with imports. First, we import toga:

```
import toga
```

Then, we set up a handler - a wrapper around behavior that we want to activate when the button is pressed. A handler is just a function. The function takes the widget that was activated as the first argument; depending on the type of event that is being handled, other arguments may also be provided. In the case of a simple button press, however, there are no extra arguments:

```
def button_handler(widget):
    print("hello")
```

By creating an app, we're declaring that we want to have a main window, with a main menu. However, Toga doesn't know what content we want in that main window. The next step is to define a method that describes the UI that we want our app to have. This method is a callable that accepts an app instance:

```
def build(app):
```

We want to put a button in the window. However, unless we want the button to fill the entire app window, we can't just put the button into the app window. Instead, we need create a box, and put the button in the box.

A box is an object that can be used to hold multiple widgets, and to define padding around widgets. So, we define a box:

```
box = toga.Box()
```

We can then define a button. When we create the button, we can set the button text, and we also set the behavior that we want to invoke when the button is pressed, referencing the handler that we defined earlier:

```
button = toga.Button('Hello world', on_press=button_handler)
```

Now we have to define how the button will appear in the window. Toga uses a CSS-based layout scheme, so we can apply CSS styles to each widget:

```
button.style.set(margin=50)
```

Each widget is a "block" in CSS terms, what we've done here is say that the button with have a margin of 50 pixels on each side. If we wanted to define a margin of 20 pixels on top of the button, we could have defined `margin_top=20`, or we could have specified the `margin=(20, 50, 50, 50)`.

The next step is to add the button to the box:

```
box.add(button)
```

The button will, by default, stretch to the size of the box it is placed in. The outer box is also a block, which will stretch to the size of box it is placed in - which, in our case, is the window itself. The button has a default height, defined by the way that the underlying platform draws buttons). As a result, this means we'll see a single button in the app window that stretches to the width of the screen, but has a 50 pixel margin surrounding it.

Now we've set up the box, we return the outer box that holds all the UI content. This box will be the content of the app's main window:

```
return box
```

Lastly, we instantiate the app itself. The app is a high level container representing the executable. The app has a name, and a unique identifier. The identifier is used when registering any app-specific system resources. By convention, the identifier is a "reversed domain name". The app also accepts our callable defining the main window contents. We wrap this creation process into a method called *main*, which returns a new instance of our application:

```
def main():
    return toga.App('First App', 'org.pybee.helloworld', startup=build)
```

The entry point for the project then needs to instantiate this entry point, and start the main app loop. The call to *main_loop()* is a blocking call; it won't return until you quit the main app:

```
if __name__ == '__main__':
    main().main_loop()
```

And that's it! Save this script as `helloworld.py`, and you're ready to go.

Running the app

Before you run the app, you'll need to install toga. Although you *can* install toga by just running:

```
$ pip install toga
```

We strongly suggest that you **don't** do this. We'd suggest creating a virtual environment first, and installing toga in that virtual environment.

Note: Minimum versions

Toga has some minimum requirements:

- If you're on OS X, you need to be on 10.7 (Lion) or newer.
- If you're on Linux, you need to have GTK+ 3.4 or later. This is the version that ships starting with Ubuntu 12.04 and Fedora 17.
- If you want to use the WebView widget, you'll also need to have WebKit, plus the GI bindings to WebKit installed.
  - For Ubuntu that's provided by the `libwebkitgtk-3.0-0` and `gir1.2-webkit-3.0` packages.
  - For Fedora it's all provided in the `webkitgtk3` package.

If these requirements aren't met, Toga either won't work at all, or won't have full functionality.

Once you've got toga installed, you can run your script:

```
$ python -m helloworld
```

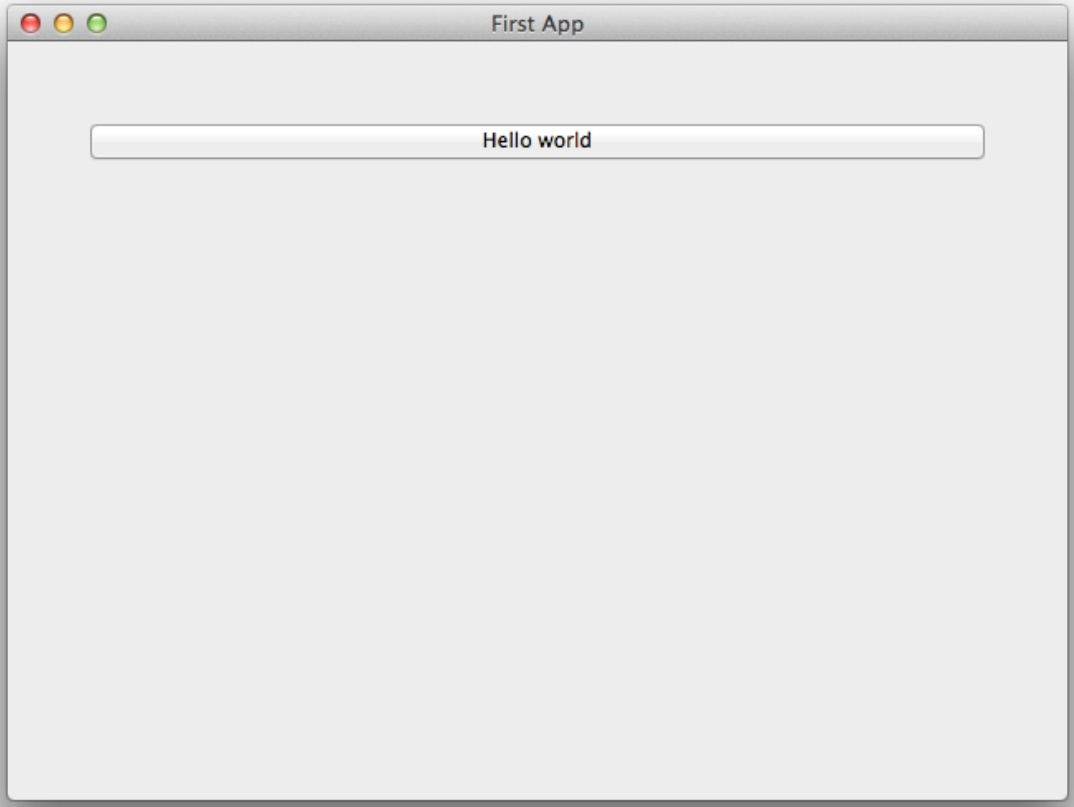
Note: `python -m helloworld` vs `python helloworld.py`

Note the `-m` flag and absence of the `.py` extension in this command line. If you run `python helloworld.py`, you may see some errors like:

```
NotImplementedError: Application does not define open_document()
```

Toga apps must be executed as modules - hence the `-m` flag.

This should pop up a window with a button:



If you click on the button, you should see messages appear in the console. Even though we didn't define anything about menus, the app will have default menu entries to quit the app, and an About page. The keyboard bindings to quit the app, plus the "close" button on the window will also work as expected. The app will have a default Toga icon (a picture of Tiberius the yak).

2.1.2 A slightly less toy example

Most applications require a little more than a button on a page. Lets build a slightly more complex example - a Fahrenheit to Celsius converter:

Here's the source code:

```
import toga

def build(app):
    c_box = toga.Box()
    f_box = toga.Box()
    box = toga.Box()

    c_input = toga.TextInput(readonly=True)
    f_input = toga.TextInput()

    c_label = toga.Label('Celsius', alignment=toga.LEFT_ALIGNED)
    f_label = toga.Label('Fahrenheit', alignment=toga.LEFT_ALIGNED)
    join_label = toga.Label('is equivalent to', alignment=toga.RIGHT_ALIGNED)

    def calculate(widget):
        try:
            c_input.value = (float(f_input.value) - 32.0) * 5.0 / 9.0
        except:
            c_input.value = '???'

    button = toga.Button('Calculate', on_press=calculate)

    f_box.add(f_input)
    f_box.add(f_label)

    c_box.add(join_label)
    c_box.add(c_input)
    c_box.add(c_label)

    box.add(f_box)
    box.add(c_box)
    box.add(button)

    box.style.set(flex_direction='column', padding_top=10)
    f_box.style.set(flex_direction='row', margin=5)
    c_box.style.set(flex_direction='row', margin=5)

    c_input.style.set(flex=1)
    f_input.style.set(flex=1, margin_left=160)
    c_label.style.set(width=100, margin_left=10)
```

```
    f_label.style.set(width=100, margin_left=10)
    join_label.style.set(width=150, margin_right=10)

    button.style.set(margin=15)

    return box

def main():
    return toga.App('Temperature Converter', 'org.pybee.f_to_c', startup=build)

if __name__ == '__main__':
    main().main_loop()
```

This example shows off the use of Flexbox in Toga's CSS styling. Flexbox is a new layout scheme that is part of the CSS3 specification that corrects the problems with the older box layout scheme in CSS2. Flexbox is not yet universally available in all web browsers, but that doesn't matter for Toga - Toga provides an implementation of the Flexbox layout scheme. CSS-tricks provides a good tutorial on Flexbox if you've never come across it before.

In this example app, we've set up an outer box that stacks vertically; inside that box, we've put 2 horizontal boxes and a button.

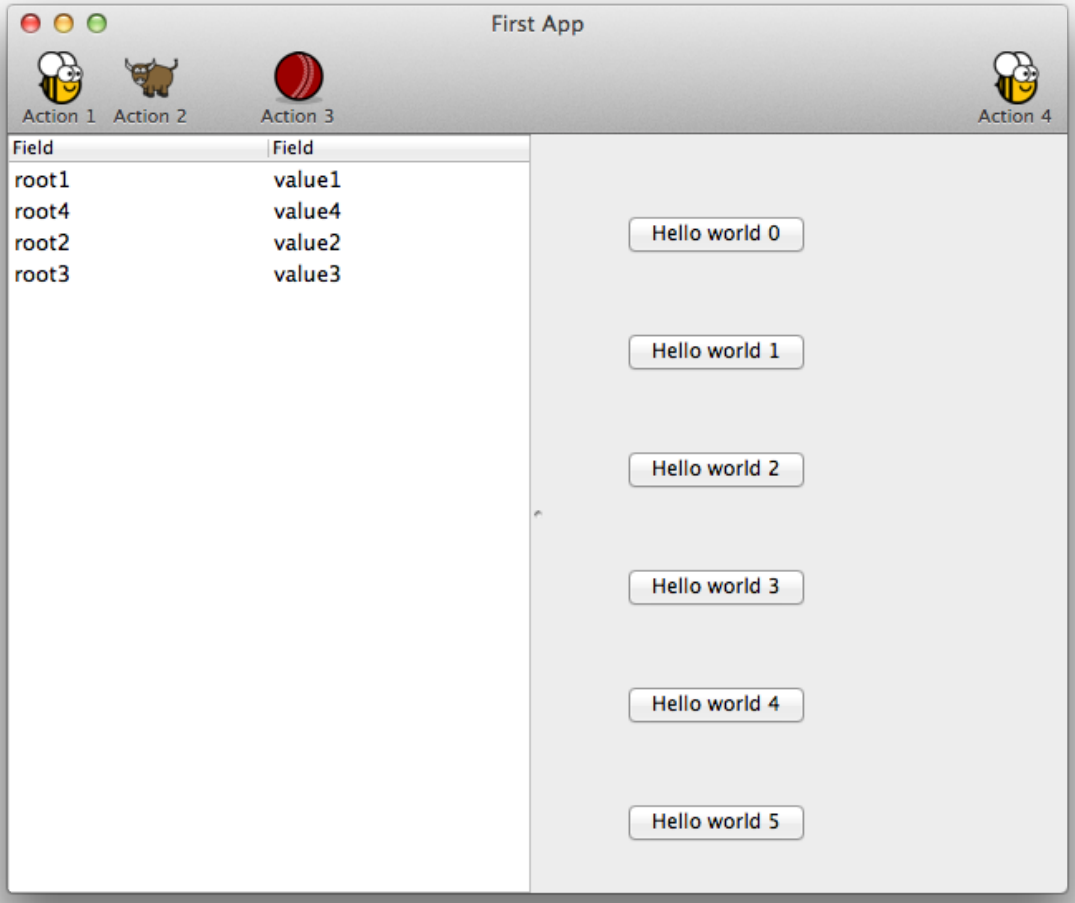
Since there's no width styling on the horizontal boxes, they'll try to fit the widgets they contain into the available space. The `TextInput` widgets have a style of `flex=1`, but the `Label` widgets have a fixed width; as a result, the `TextInput` widgets will be stretched to fit the available horizontal space. The margin and padding terms then ensure that the widgets will be aligned vertically and horizontally.

2.1.3 You put the box inside another box...

If you've done any GUI programming before, you will know that one of the biggest problems that any widget toolkit solves is how to put widgets on the screen in the right place. Different widget toolkits use different approaches - constraints, packing models, and grid-based models are all common. Toga uses an approach that is new for widget toolkits, but well proven in computing: Cascading Style Sheets (CSS).

If you've done any design for the web, you will have come across CSS before as the mechanism that you use to lay out HTML on a web page. Although this is the reason CSS was developed, CSS itself is a general set of rules for laying out any "boxes" that are structured in a tree-like hierarchy. GUI widgets are an example of one such structure.

To see how this works in practice, lets look at a more complex example, involving layouts, scrollers, and containers inside other containers:



Here's the source code

```
import toga
from colosseum import CSS


def button_handler(widget):
    print('button handler')
    for i in range(0, 10):
        print("hello", i)
        yield 1
    print("done", i)


def action0(widget):
    print("action 0")


def action1(widget):
    print("action 1")
```

```
def action2(widget):
    print("action 2")


def action3(widget):
    print("action 3")


def build(app):

    data = [
        ('root%s' % i, 'value %s' % i)
        for i in range(1, 100)
    ]

    left_container = toga.Table(headings=['Hello', 'World'], data=data)

    right_content = toga.Box(
        style=CSS(flex_direction='column', padding_top=50)
    )

    for b in range(0, 10):
        right_content.add(
            toga.Button(
                'Hello world %s' % b,
                on_press=button_handler,
                style=CSS(width=200, margin=20)
            )
        )

    right_container = toga.ScrollContainer(horizontal=False)

    right_container.content = right_content

    split = toga.SplitContainer()

    split.content = [left_container, right_container]

    things = toga.Group('Things')

    cmd0 = toga.Command(
        action1,
        label='Action 0',
        tooltip='Perform action 0',
        icon='icons/brutus.icns',
        group=things
    )
    cmd1 = toga.Command(
        action1,
        label='Action 1',
        tooltip='Perform action 1',
        icon='icons/brutus.icns',
        group=things
    )
    cmd2 = toga.Command(
        action2,
        label='Action 2',
```

```
        tooltip='Perform action 2',
        icon=toga.Icon.TIBERIUS_ICON,
        group=things
    )
    cmd3 = toga.Command(
        action3,
        label='Action 3',
        tooltip='Perform action 3',
        shortcut='k',
        icon='icons/cricket-72.png'
    )

    def action4(widget):
        print ("CALLING Action 4")
        cmd3.enabled = not cmd3.enabled

    cmd4 = toga.Command(
        action4,
        label='Action 4',
        tooltip='Perform action 4',
        icon='icons/brutus.icns'
    )

    app.commands.add(cmd1, cmd3, cmd4, cmd0)
    app.main_window.toolbar.add(cmd1, cmd2, cmd3, cmd4)

    return split


def main():
    return toga.App('First App', 'org.pybee.helloworld', startup=build)


if __name__ == '__main__':
    main().main_loop()
```

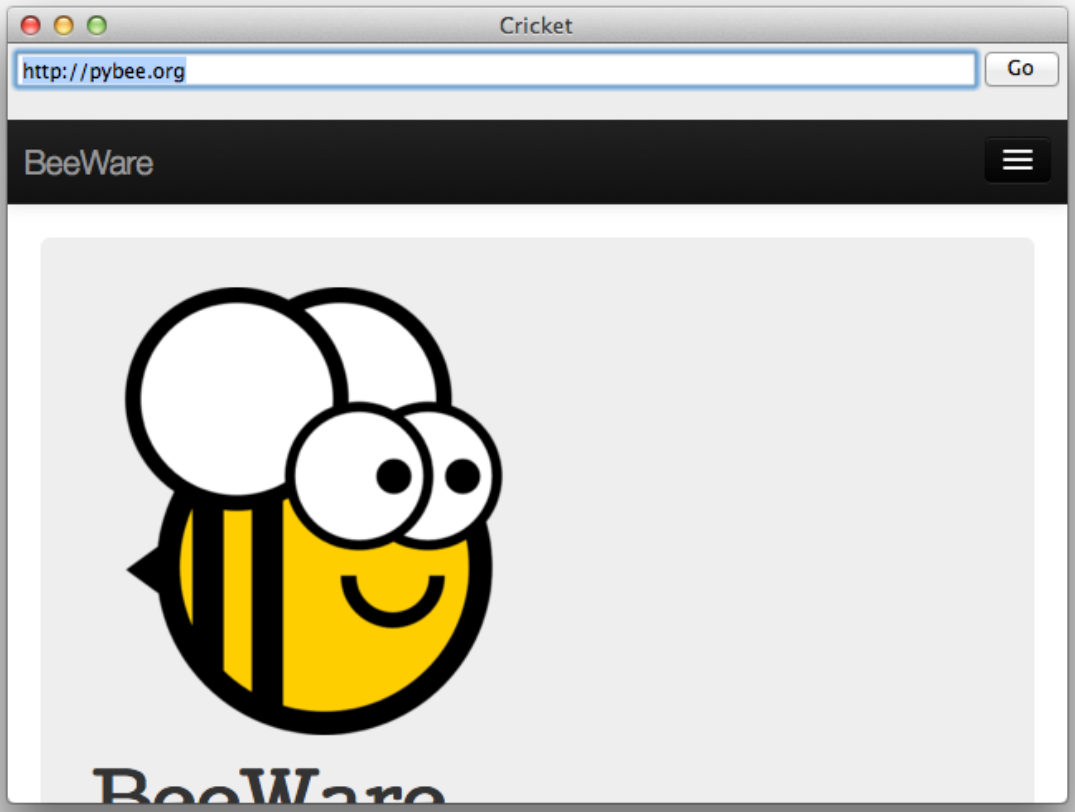
Here are the `Icons`

In this example, we see a couple of new Toga widgets - `Table`, `SplitContainer`, and `ScrollContainer`. You can also see that CSS styles can be added in the widget constructor. Lastly, you can see that windows can have toolbars.

2.1.4 Let's build a browser!

Although it's possible to build complex GUI layouts, you can get a lot of functionality with very little code, utilizing the rich components that are native on modern platforms.

So - let's build a tool that lets our pet yak graze the web - a primitive web browser, in less than 40 lines of code!



Here's the source code:

```
#!/usr/bin/env python

import toga
from colosseum import CSS


class Graze(toga.App):
    def startup(self):
        self.main_window = toga.MainWindow(self.name)
        self.main_window.app = self

        self.webview = toga.WebView(style=CSS(flex=1))
        self.url_input = toga.TextInput(
            initial='https://github.com/',
            style=CSS(flex=1, margin=5)
        )

        box = toga.Box(
            children = [
                toga.Box(
                    children = [
                        self.url_input,
```

```
                        toga.Button('Go', on_press=self.load_page,␣
↪style=CSS(width=50)),
                    ],
                    style=CSS(
                        flex_direction='row'
                    )
                ),
                self.webview,
            ],
            style=CSS(
                flex_direction='column'
            )
        )

        self.main_window.content = box
        self.webview.url = self.url_input.value

        # Show the main window
        self.main_window.show()

    def load_page(self, widget):
        self.webview.url = self.url_input.value

if __name__ == '__main__':
    app = Graze('Graze', 'org.pybee.graze')

    app.main_loop()
```

In this example, you can see an application being developed as a class, rather than as a build method. You can also see boxes defined in a declarative manner - if you don't need to retain a reference to a particular widget, you can define a widget inline, and pass it as an argument to a box, and it will become a child of that box.

2.1.5 Tutorial 0 - your first Toga app

In *Your first Toga app*, you will discover how to create a basic app and have a simple `toga.interface.widgets.button.Button` widget to click.

2.1.6 Tutorial 1 - a slightly less toy example

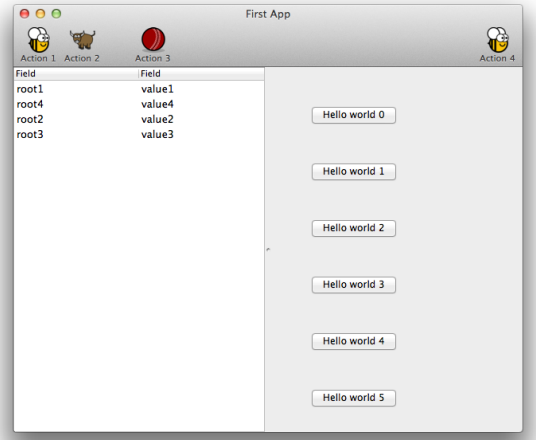
In *A slightly less toy example*, you will discover how to capture basic user input using the `toga.interface.widgets.textinput.TextInput` widget and control layout.

2.1.7 Tutorial 2 - you put the box inside another box...

In *You put the box inside another box...*, you will discover how to use the `toga.interface.widgets.splitcontainer.SplitContainer` widget to display some components, a toolbar and a table.

2.1.8 Tutorial 3 - let's build a browser!

In *Let's build a browser!*, you will discover how to use the `toga.interface.widgets.webview.WebView` widget to display a simple browser.


First App
Action 1
Action 2
Action 3
Action 4
Field
Field
root1
value1
root4
value4
root2
value2
root3
value3
Hello world 0
Hello world 1
Hello world 2
Hello world 3
Hello world 4
Hello world 5


Cricket
http://pybee.org
Go
BeeWare

2.2 How-to Guides

2.2.1 How to get started

Quickstart

Create a new virtualenv. In your virtualenv, install Toga, and then run it:

```
$ pip install toga-demo
$ toga-demo
```

This will pop up a GUI window showing the full range of widgets available to an application using Toga.

Have fun, and see the *Reference* to learn more about what's going on.

2.2.2 How to contribute to Toga

If you experience problems with Toga, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

Set up your development environment

The recommended way of setting up your development environment for Toga is to install a virtual environment, install the required dependencies and start coding. Assuming that you are using `virtualenvwrapper`, you only have to run:

```
$ git clone git@github.com:pybee/toga.git
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ cd toga
```

Then, install Toga into your development environment. The Toga source repository contains multiple packages. Since we're installing from source, we can't rely on pip to install the packages in dependency order. Therefore, we have to manually install each package in a specific order. We start with the core packages:

```
(venv) $ pip install -e src/core
(venv) $ pip install -e src/dummy
```

Then, we can install the code for the specific platform we want to use. For example, if we're on a Mac, you'd run:

```
(venv) $ pip install -e src/cocoa
```

If you were on a Linux box, you'd run:

```
(venv) $ pip install -e src/gtk
```

And so on.

You can then run the core test suite:

```
(venv) $ cd src/core
(venv) $ python setup.py test
...
----------------------------------------------------------------------
```

```
Ran 100 tests in 0.343s

OK (skipped=1)
```

You should get some output indicating that tests have been run. You shouldn't ever get any FAIL or ERROR test results. We run our full test suite before merging every patch. If that process discovers any problems, we don't merge the patch. If you do find a test error or failure, either there's something odd in your test environment, or you've found an edge case that we haven't seen before - either way, let us know!

Now you are ready to start hacking on Toga!

What should I do?

The src/core package of toga has a test suite, but that test suite is incomplete. There are many aspects of the Toga Core API that aren't currently tested (or aren't tested thoroughly). To work out what *isn't* tested, we're going to use a tool called coverage. Coverage allows you to check which lines of code have (and haven't) been executed - which then gives you an idea of what code has (and hasn't) been tested.

Install coverage, and then re-run the test suite – this time, in a slightly different way so that we can gather some data about the test run:

```
(venv) $ pip install coverage
(venv) $ coverage run setup.py test
```

Then, generate a report for the coverage data you just gathered:

```
(venv) $ coverage report -m --include=toga/*
Name                                    Stmts   Miss  Cover   Missing
---------------------------------------------------------------------
toga/__init__.py                           29      0   100%
toga/app.py                                50      0   100%
...
toga/window.py                             79     18    77%   58, 75, 87, 92, 104, 141,␣
↪155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257
---------------------------------------------------------------------
TOTAL                                    1034    258    75%
```

What does this all mean? Well, the "Cover" column tells you what proportion of lines in a given file were executed during the test run. In this run, every line of `toga/app.py` was executed; but only 77% of lines in `toga/window.py` were executed. Which lines were missed? They're listed in the next column: lines 58, 75, 87, and so on weren't executed.

That's what you have to fix - ideally, every single line in every single file will have 100% coverage. If you look in *src/core/tests*, you should find a test file that matches the name of the file that has insufficient coverage. If you don't, it's possible the entire test file is missing - so you'll have to create it!

Your task: create a test that improves coverage - even by one more line.

Once you've written a test, re-run the test suite to generate fresh coverage data. Let's say we added a test for line 58 of `toga/window.py` - we'd expect to see something like:

```
(venv) $ coverage run setup.py test
running test
...
----------------------------------------------------------------------
Ran 101 tests in 0.343s
```

```
OK (skipped=1)
(venv) $ coverage report -m --include=toga/*
Name                                  Stmts   Miss  Cover   Missing
-------------------------------------------------------------------
toga/__init__.py                         29      0   100%
toga/app.py                              50      0   100%
...
toga/window.py                           79     17    78%   75, 87, 92, 104, 141, 155,␣
↪164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257
-------------------------------------------------------------------
TOTAL                                  1034    257    75%
```

That is, one more test has been executed, resulting in one less missing line in the coverage results.

Submit a pull request for your work, and you're done! Congratulations, you're a contributor to Toga!

How does this all work?

Since you're writing tests for a GUI toolkit, you might be wondering why you haven't seen a GUI yet. The Toga Core package contains the API definitions for the Toga widget kit. This is completely platform agnostic - it just provides an interface, and defers actually drawing anything on the screen to the platform backends.

When you run the test suite, the test runner uses a "dummy" backend - a platform backend that *implements* the full API, but doesn't actually *do* anything (i.e., when you say display a button, it creates an object, but doesn't actually display a button).

In this way, it's possible to for the Toga Core tests to exercise every API entry point in the Toga Core package, verify that data is stored correctly on the interface layer, and sent through to the right endpoints in the Dummy backend. If the *dummy* backend is invoked correcty, then any other backend will be handled correctly, too.

It's not just about coverage!

Although improving test coverage is the goal, the task ahead of you isn't *just* about increasing numerical coverage. Part of the task is to audit the code as you go. You could write a comprehensive set of tests for a concrete life jacket... but a concrete life jacket would still be useless for the purpose it was intended!

As you develop tests and improve coverage, you should be checking that the core module is internally **consistent** as well. If you notice any method names that aren't internally consistent (e.g., something called `on_select` in one module, but called `on_selected` in another), or where the data isn't being handled consistently (one widget updates then refreshes, but another widget refreshes then updates), flag it and bring it to our attention by raising a ticket. Or, if you're confident that you know what needs to be done, create a pull request that fixes the problem you've found.

What next?

Rinse and repeat! Having improved coverage by one line, go back and do it again for *another* coverage line!

If you're feeling particularly adventurous, you could start looking at a specific platform backend. The Toga Dummy API defines the API that a backend needs to implement; so find a platform backend of interest to you (e.g., cocoa if you're on macOS), and look for a widget that isn't implemented (a missing file in the `widgets` directory for that platform, or an API *on* a widget that isn't implemented (these will be flagged by raising `NotImplementedError()`). Dig into the documentation for native widgets for that platform (e.g., the Apple Cocoa documentation), and work out how to map native widget capabilities to the Toga API. You may find it helpful to look at existing widgets to work out what is needed.

Most importantly - have fun!

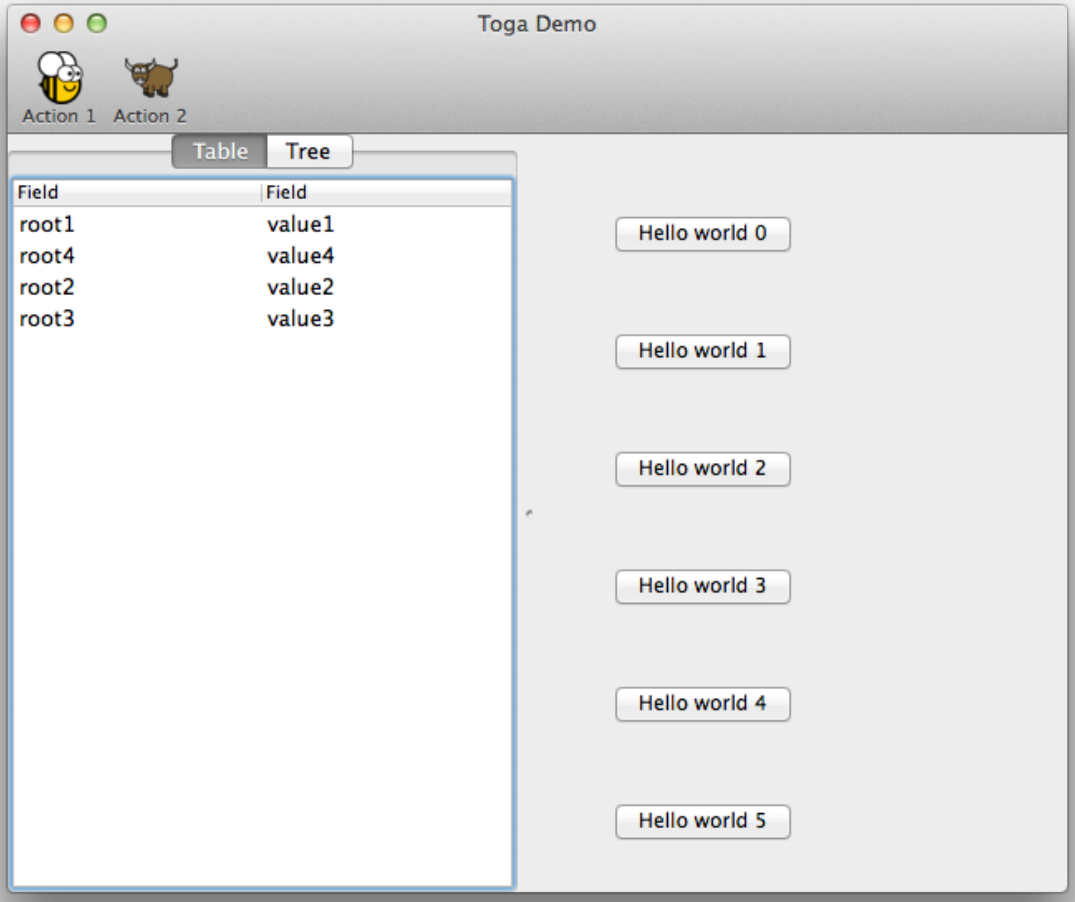
2.3 Reference

2.3.1 Toga supported platforms

Official platform support

Desktop platforms

OS X

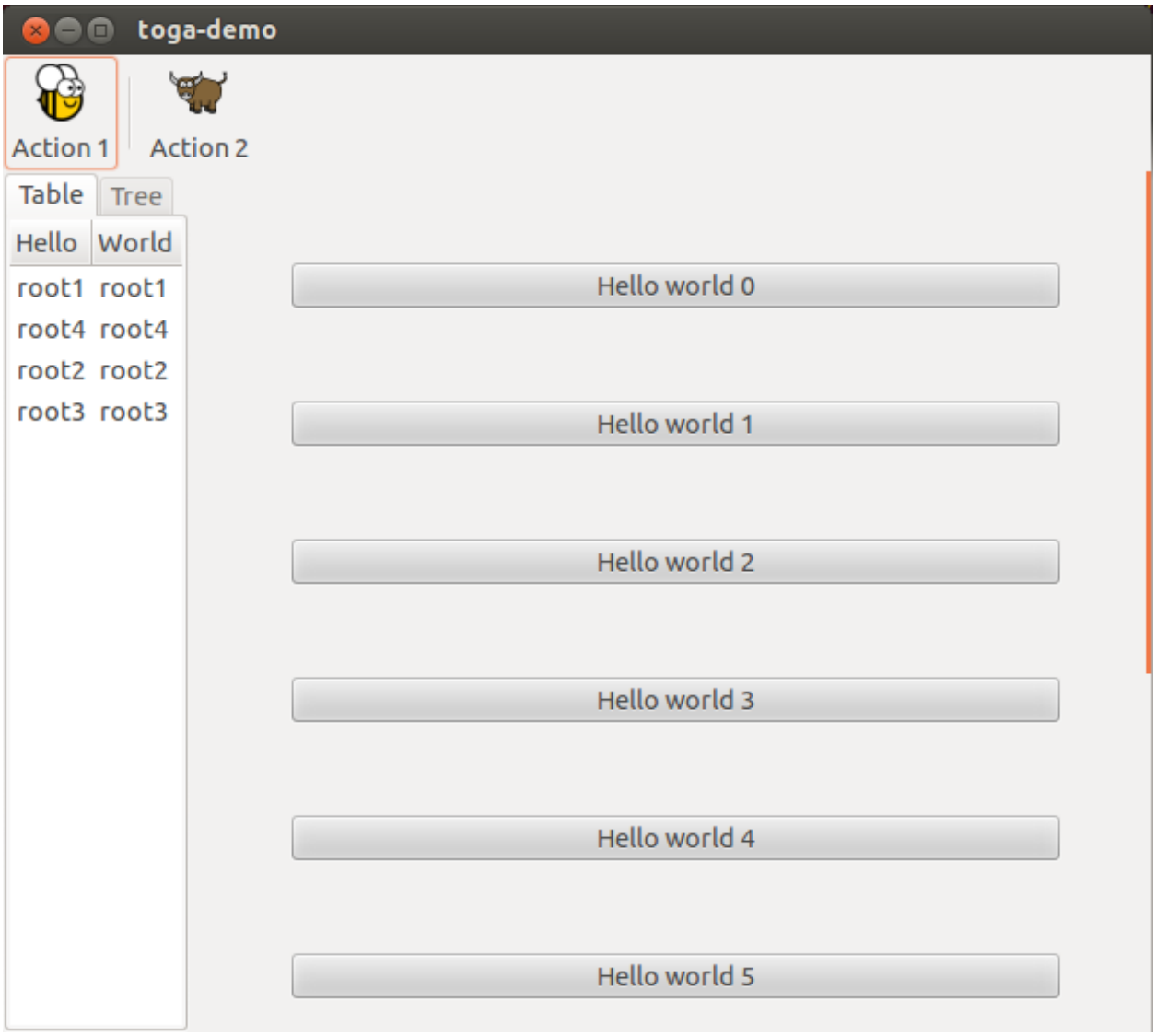


The backend for OS X is named toga-cocoa. It supports OS X 10.7 (Lion) and later. It is installed automatically on OS X machines (machines that report `sys.platform == 'darwin'`), or can be manually installed by invoking:

```
$ pip install toga[cocoa]
```

The OS X backend has seen the most development to date.

Linux



The backend for Linux platforms is named toga-gtk. It supports GTK+ 3.4 and later. It is installed automatically on Linux machines (machines that report `sys.platform == 'linux'`), or can be manually installed by invoking:

```
$ pip install toga[gtk]
```

The GTK+ backend is reasonably well developed, but currently has some known issues with widget layout.

Win32

The backend for Windows is named toga-win32. It supports Windows XP or later. It is installed automatically on Windows machines (machines that report `sys.platform == 'win32'`), or can be manually installed by invoking:

```
$ pip install toga[win32]
```

The Windows backend is currently proof-of-concept only. Most widgets have not been implemented.

Mobile platforms

iOS

The backend for iOS is named toga-iOS. It supports iOS 6 or later. It must be manually installed into an iOS Python project (such as one that has been developed using the Python-iOS-template cookiecutter). It can be manually installed by invoking:

```
$ pip install toga[iOS]
```

The iOS backend is currently proof-of-concept only. Most widgets have not been implemented.

Planned platform support

There are plans to provide support for the following platforms:

- Web (using Batavia to run Python on the browser)
- Android
- WinRT (Native Windows 8 and Windows mobile)
- Qt (for KDE based desktops)

If you are interested in these platforms and would like to contribute, please get in touch on Twitter or Gitter.

Unofficial platform support

At present, there are no known unofficial platform backends.

2.3.2 Widget Reference

Core Widgets

Toga includes a set of core widgets, that can be placed with a Box Container.

Component	Usage	Purpose	Class
Application	Documentation	Primary host for UI components	`toga.app.App`
Box	Documentation	Container for components	`toga.widgets.box.Box`
Font	Documentation	Fonts	`toga.font.Font`
Widget	Documentation	Base class for widgets	`toga.widgets.base.Widget`
Window	Documentation	Window object	`toga.window.Window`

General widgets

Component	Usage	Purpose	Class
Button	Documentation	Basic clickable button	`toga.widgets.button.Button`
Canvas	Documentation	Area you can draw on	`toga.widgets.canvas.Canvas`
Image View	Documentation	Image Viewer	`toga.widgets.imageview.ImageView`
Label	Documentation	Text label	`toga.widgets.label.Label`
Multiline Text Input	Documentation	Multi-line Text Input field	`toga.widgets.multilinetextinput.MultilineTextInput`
Number Input	Documentation	Number Input field	`toga.widgets.numberinput.NumberInput`
Option Container	Documentation	Option Container	`toga.widgets.optioncontainer.OptionContainer`
Progress Bar	Documentation	Progress Bar	`toga.widgets.progressbar.ProgressBar`
Selection	Documentation	Selection	`toga.widgets.selection.Selection`
Text Input	Documentation	Text Input field	`toga.widgets.textinput.TextInput`
Table	Documentation	Table of data	`toga.widgets.table.Table`
Tree	Documentation	Tree of data	`toga.widgets.tree.Tree`
Switch	Documentation	Switch	`toga.widgets.switch.Switch`

Layout widgets

Component	Usage	Purpose	Class
Scroll Container	Documentation	Scrollable Container	`toga.widgets.scrollcontainer.ScrollContainer`
Split Container	Documentation	Split Container	`toga.widgets.splitcontainer.SplitContainer`
Option Container	Documentation	Option Container	`toga.widgets.optioncontainer.OptionContainer`
Web View	Documentation	Web View	`toga.widgets.webview.WebView`

2.3.3 Supported widgets by platform

Component	iOS	win32	web	django	cocoa	gtk	android
`toga.interface.app.App`	✓	✓		✓	✓	✓	✓
`toga.interface.widgets.box.Box`	✓	✓		✓	✓	✓	✓
`toga.interface.widgets.button.Button`	✓	✓		✓	✓	✓	✓
`toga.interface.widgets.canvas.Canvas`						✓	
Command					✓	✓	
EXPANDING_SPACER					✓	✓	
`toga.interface.font.Font`	✓				✓	✓	
`toga.interface.widgets.icon.Icon`					✓	✓	
Image					✓		
`toga.interface.widgets.imageview.ImageView`					✓		
`toga.interface.widgets.label.Label`	✓	✓			✓	✓	
`toga.interface.app.MainWindow`	✓			✓	✓	✓	
`toga.interface.widgets.multilinetextinput.MultilineTextInput`		✓			✓		
`toga.interface.widgets.numberinput.NumberInput`		✓			✓	✓	
`toga.interface.widgets.optioncontainer.OptionContainer`					✓	✓	
`toga.interface.widgets.passwordinput.PasswordInput`		✓			✓		
`toga.interface.widgets.progressbar.ProgressBar`					✓		
`toga.interface.widgets.scrollcontainer.ScrollContainer`					✓	✓	
`toga.interface.widgets.selection.Selection`					✓	✓	
SEPARATOR					✓	✓	

2.4 Background

2.4.1 Why Toga?

Toga isn't the world's first widget toolkit - there are dozens of other options. So why build a new one?

Native widgets - not themes

Toga uses native system widgets, not themes. When you see a Toga app running, it doesn't just *look* like a native app - it *is* a native app. Applying an operating system-inspired theme over the top of a generic widget set is an easy way for a developer to achieve a cross-platform goal, but it leaves the end user with the mess.

It's easy to spot apps that have been built using themed widget sets - they're the ones that don't behave quite like any other app. Widgets don't look *quite* right, or there's a menu bar on a window in an OS X app. Themes can get quite close - but there are always tell-tale signs.

On top of that, native widgets are always faster than a themed generic widget. After all, you're using native system capability that has been tuned and optimized, not a drawing engine that's been layered on top of a generic widget.

Abstract the broad concepts

It's not enough to just look like a native app, though - you need to *feel* like a native app as well.

A "Quit" option under a "File" menu makes sense if you're writing a Windows app - but it's completely out of place if you're on OS X - the Quit option should be under the application menu.

And besides - why did the developer have to code the location of a Quit option anyway? Every app in the world has to have a quit option, so why doesn't the widget toolkit provide a quit option pre-installed, out of the box?

Although Toga uses 100% native system widgets, that doesn't mean Toga is just a wrapper around system widgets. Wherever possible, Toga attempts to abstract the broader concepts underpinning the construction of GUI apps, and build an API for *that*. So - every Toga app has the basic set of menu options you'd expect of every app - Quit, About, and so on - all in the places you'd expect to see them in a native app.

When it comes to widgets, sometimes the abstraction is simple - after all, a button is a button, no matter what platform you're on. But other widgets may not be exposed so literally. What the Toga API aims to expose is a set of mechanisms for achieving UI goals, not a literal widget set.

Python native

Most widget toolkits start their life as a C or C++ layer, which is then wrapped by other languages. As a result, you end up with APIs that taste like C or C++.

Toga has been designed from the ground up to be a Python native widget toolkit. This means the API is able to exploit language level features like generators and context managers in a way that a wrapper around a C library wouldn't be able to (at least, not easily).

This also means supporting Python 3, and 3 only because that's where the future of Python is at.

pip install and nothing more

Toga aims to be no more than a *pip install* away from use. It doesn't require the compilation of C extensions. There's no need to install a binary support library. There's no need to change system paths and environment variables. Just install it, import it, and start writing (or running) code.

Embrace mobile

10 years ago, being a cross-platform widget toolkit meant being available for Windows, OS X and Linux. These days, mobile computing is much more important. But despite this, there aren't many good options for Python programming on mobile platforms, and cross-platform mobile coding is still elusive. Toga aims to correct this.

2.4.2 Why "Toga"? Why the Yak?

So... why the name Toga?

We all know the aphorism that "When in Rome, do as the Romans do."

So - what does a well dressed Roman wear? A toga, of course! And what does a well dressed Python app wear? Toga!

So... why the yak mascot?

It's a reflection of the long running joke about yak shaving in computer programming. The story originally comes from MIT, and is related to a Ren and Stimpy episode; over the years, the story has evolved, and now goes something like this:

> You want to borrow your neighbors hose so you can wash your car. But you remember that last week, you broke their rake, so you need to go to the hardware store to buy a new one. But that means driving to the hardware store, so you have to look for your keys. You eventually find your keys inside a tear in a cushion - but you can't leave the cushion torn, because the dog will destroy the cushion if they find a little tear. The cushion needs a little more stuffing before it can be repaired, but it's a special cushion filled with exotic Tibetan yak hair.
>
> The next thing you know, you're standing on a hillside in Tibet shaving a yak. And all you wanted to do was wash your car.

An easy to use widget toolkit is the yak standing in the way of progress of a number of PyBee projects, and the original creator of Toga has been tinkering with various widget toolkits for over 20 years, so the metaphor seemed appropriate.

2.4.3 Commands, Menus and Toolbars

A GUI requires more than just widgets laid out in a user interface - you'll also want to allow the user to actually *do* something. In Toga, you do this using `Commands`.

A command encapsulates a piece of functionality that the user can invoke - no matter how they invoke it. It doesn't matter if they select a menu item, press a button on a toolbar, or use a key combination - the functionality is wrapped up in a Command.

When a command is added to an application, Toga takes control of ensuring that the command is exposed to the user in a way that they can access it. On desktop platforms, this may result in a command being added to a menu.

You can also choose to add a command (or commands) to a toolbar on a specific window.

Defining Commands

When you specify a `Command`, you provide some additional metadata to help classify and organize the commands in your application:

- An **action** - a function to invoke when the command is activated.
- A **label** - a name for the command to.

- A **tooltip** - a short description of what the command will do
- A **shortcut** - (optional) A key combination that can be used to invoke the command.
- An **icon** - (optional) A path to an icon resource to decorate the command.
- A **group** - (optional) a `Group` object describing a collection of similar commands. If no group is specified, a default "Command" group will be used.
- A **section** - (optional) an integer providing a sub-grouping. If no section is specified, the command will be allocated to section 0 within the group.
- An **order** - (optional) an integer indicating where a command falls within a section. If a `Command` doesn't have an order, it will be sorted alphabetically by label within it's section.

Commands may not use all the metadata - for example, on some platforms, menus will contain icons; on other platforms, they won't. Toga will use the metadata if it is provided, but ignore it (or substitute an appropriate default) if it isn't.

Commands can be enabled and disabled; if you disable a command, it will automatically disable any toolbar or menu item where the command appears.

Groups

Toga provides a number of ready-to-use groups:

- `Group.APP` - Application level control
- `Group.FILE` - File commands
- `Group.EDIT` - Editing commands
- `Group.VIEW` - Commands to alter the appearance of content
- `Group.COMMANDS` - A Default
- `Group.WINDOW` - Commands for managing different windows in the app
- `Group.HELP` - Help content

You can also define custom groups.

Example

The following is an example of using menus and commands:

```
import toga

def callback(sender):
    print("Command activated")

def build(app):
    ...
    stuff_group = Group('Stuff', order=40)

    cmd1 = toga.Command(
        callback,
        label='Example command',
        tooltip='Tells you when it has been activated',
        shortcut='k',
        icon='icons/pretty.png'
```

```
        group=stuff_group,
        section=0
    )
    cmd2 = toga.Command(
        ...
    )
    ...

    app.commands.add(cmd1, cmd4, cmd3)
    app.main_window.toolbar.add(cmd2, cmd3)
```

This code defines a command `cmd1` that will be placed in first section of the "Stuff" group. It can be activated by pressing CTRL-k (or CMD-K on a Mac).

The definitions for `cmd2`, `cmd3`, and `cmd4` have been omitted, but would follow a similar pattern.

It doesn't matter what order you add commands to the app - the group, section and order will be used to put the commands in the right order.

If a command is added to a toolbar, it will automatically be added to the app as well. It isn't possible to have functionality exposed on a toolbar that isn't also exposed by the app. So, `cmd2` will be added to the app, even though it wasn't explicitly added to the app commands.

2.5 About the project

2.5.1 Release History

0.2.12

- Migrated to CSS-based layout, rather than Cassowary/constraint layout.
- Added Windows backend
- Added Django backend
- Added Android backend

0.2.0 - 0.2.11

Internal Development releases.

0.1.2

- Further improvements to multiple-repository packaging strategy.
- Ensure Ctrl-C is honored by apps.
- **Cocoa:** Added runtime warnings when minimum OS X version is not met.

0.1.1

- Refactored code into multiple repositories, so that users of one backend don't have to carry the overhead of other installed platforms

- Corrected a range of bugs, mostly related to problems under Python 3.

0.1.0

Initial public release. Includes:

- A Cocoa (OS X) backend
- A GTK+ backend
- A proof-of-concept Win32 backend
- A proof-of-concept iOS backend

2.5.2 Toga Roadmap

Toga is a new project - we have lots of things that we'd like to do. If you'd like to contribute, providing a patch for one of these features.

Widgets

The core of Toga is it's widget set. Modern GUI apps have lots of native controls that need to be represented. The following widgets have no representation at present, and need to be added.

There's also the task of porting widgets available on one platform to another platform.

Input

Inputs are mechanisms for displaying and editing input provided by the user.

- **ComboBox - A free entry TextField that provides options (e.g., text with** past choices) - Cocoa: NSComboBox - GTK+: Gtk.ComboBox.new_with_model_and_entry - iOS: ?
- **Switch - A native control for enabled/disabled**
  - Cocoa: Done
  - GTK+: Gtk.CheckButton (maybe Gtk.Switch?)
  - iOS: UISwitch
- **DateInput - A widget for selecting a date**
  - Cocoa: NSDatePicker, constrained to DMY
  - GTK+: Gtk.Calendar?
  - iOS: UIDatePicker
- **TimeInput - A widget for selecting a time**
  - Cocoa: NSDatePicker, Constrained to Time
  - GTK+: ?
  - iOS: UIDatePicker
- **DateTimeInput - A widget for selecting a date and a time.**
  - Cocoa: NSDatePicker

  - GTK+: Gtk.Calendar + ?
  - iOS: UIDatePicker
- **MultilineTextInput - A widget for displaying multiline text, optionally** editable. - Cocoa: NSTextView inside an NSScrollView - GTK+: Gtk.TextView? (is there a simpler version than a full text editor?) - iOS: UITextView
- **Selection - A button that allows the user to choose from one of a fixed** number of options - Cocoa: NSPopupButton, with NSMenu for options. - GTK+: Gtk.ComboBox.new_with_model - iOS: UIPickerView
- **ColorInput - A widget for selecting a color**
  - Cocoa: NSColorWell
  - GTK+: Gtk.ColorButton
  - iOS: ?
- **SliderInput (H & V) - A widget for selecting a value from a range.**
  - Cocoa: NSSlider
  - GTK+: Gtk.Scale
  - iOS: UISlider
- **NumberInput - A widget to allow entry of a numerical value, possibly with** helper buttons to make it easy to increase/decrease the value. - Cocoa: NSTextField with NSStepper - GTK+: GTKSpinButton - iOS: UITextField with UIStepper
- **Table: A scrollable display of columns of tabular data**
  - Cocoa: Done
  - GTK+: Gtk.TreeView with a Gtk.ListStore
  - iOS: UITableView
- **Tree: A scrollable display of hierarchical data**
  - Cocoa: Done
  - GTK+: Gtk.TreeView with a Gtk.TreeStore
  - iOS: UITableView with navigation
- **DetailedList: A scrollable list of a single column of detailed data**
  - Cocoa: NSTableView with custom view?
  - iOS: UITableView with navigation
- **SearchInput - A variant of TextField that is decorated as a search box.**
  - Cocoa: NSSearchField
  - GTK+: ?
  - iOS: UISearchBar?

Views

Views are mechanisms for displaying rich content, usually in a read-only manner.

- **Separator - a visual separator; usually a faint line.**

  - Cocoa: NSSeparator
  - GTK+:
  - iOS:

- **ProgressBar - A horizontal bar that displays progress, either progress** against a known value, or indeterminate - Cocoa: Done - GTK+: Gtk.ProgressBar - iOS: UIProgressView
- **ActivityIndicator - A spinner widget showing that something is happening**
  - Cocoa: NSProgressIndicator, Spinning style
  - GTK+: Gtk.Spinner
  - iOS: UIActivityIndicatorView
- **ImageView - Display an graphical image**
  - Cocoa: Done
  - GTK+: Gtk.Image
  - iOS: UIImageView
- **VideoView - Display a video**
  - Cocoa: AVPlayerView
  - GTK+: Custom Integrate with GStreamer
  - iOS: MPMoviePlayerController
- **WebView - Display a web page. Just the web page; no URL chrome, etc.**
  - Cocoa: Done
  - GTK+: Webkit.WebView (via WebkitGtk)
  - iOS: UIWebView
- **PDFView - Display a PDF document**
  - Cocoa: PDFView
  - GTK+: ?
  - iOS: ? Integration with QuickLook?
- **MapView - Display a map**
  - Cocoa: MKMapView
  - GTK+: Probably a Webkit.WebView pointing at Google Maps/OpenStreetMap.org
  - iOS: MKMapView

Container widgets

Containers are widgets that can contain other widgets.

- **Box - A box drawn around a collection of widgets; often has a label**
  - Cocoa: NSBox
  - GTK+:
  - iOS:

- **ButtonContainer - A layout for a group of radio/checkbox options**
  - Cocoa: NSMatrix, or NSView with pre-set constraints.
  - GTK+: ListBox?
  - iOS:
- **ScrollContainer - A container whose internal content can be scrolled.**
  - Cocoa: Done
  - GTK+:
  - iOS: UIScrollView?
- **SplitContainer - An adjustable separator bar between 2+ visible panes of content**
  - Cocoa: Done
  - GTK+:
  - iOS:
- **FormContainer - A layout for a "key/value" or "label/widget" form**
  - Cocoa: NSForm, or NSView with pre-set constraints.
  - GTK+:
  - iOS:
- **OptionContainer - (suggestions for better name welcome) A container view that** holds a small, fixed number of subviews, only one of which is visible at any given time. Generally rendered as something with "lozenge" style buttons over a box. Examples of use: OS X System preference panes that contain multiple options (e.g., Keyboard settings have an option layout for "Keyboard", "Text", "Shortcuts" and "Input sources") - Cocoa: Done - GTK+: GtkNotebook (Maybe GtkStack on 3.10+?) - iOS: ?
- **SectionContainer - (suggestions for better name welcome) A container view that** holds a small number of subviews, only one of which is visible at any given time. Each "section" has a name and icon. Examples of use: top level navigation in Safari's preferences panel. - Cocoa: NSTabView - GTK+: ? - iOS: ?
- **TabContainer - A container view for holding an unknown number of subviews, each** of which is of the same type - e.g., web browser tabs. - Cocoa: ? - GTK+: GtkNotebook - iOS: ?
- **NavigationContainer - A container view that holds a navigable tree of subviews;** essentially a view that has a "back" button to return to the previous view in a hierarchy. Example of use: Top level navigation in the OS X System Preferences panel. - Cocoa: No native control - GTK+: No native control; Gtk.HeaderBar in 3.10+ - iOS: UINavigationBar + NavigationController

Dialogs and windows

GUIs aren't all about widgets - sometimes you need to pop up a dialog to query the user.

- **Info - a modal dialog providing an "OK" option**
  - Cocoa: Done
  - GTK+: Gtk.MessageDialog, type Gtk.MessageType.INFO, buttons Gtk.ButtonsType.OK
  - iOS:
- **Error - a modal dialog showing an error, and a continue option.**
  - Cocoa: Done

  - GTK+: Gtk.MessageDialog, type Gtk.MessageType.ERROR, buttons Gtk.ButtonsType.CANCEL
  - iOS:

- **Question - a modal dialog that asks a Yes/No question**
  - Cocoa: Done
  - GTK+: Gtk.MessageDialog, type Gtk.MessageType.QUESTION, buttons Gtk.ButtonsType.YES_NO
  - iOS:

- **Confirm - a modal dialog confirming "OK" or "cancel"**
  - Cocoa: Done
  - GTK+: Gtk.MessageDialog, type Gtk.MessageType.WARNING, buttons Gtk.ButtonsType.OK_CANCEL
  - iOS:

- **StackTrace - a modal dialog for displaying a long stack trace.**
  - Cocoa: Done
  - GTK+: Custom Gtk.Dialog
  - iOS:

- **File Open - a mechanism for finding and specifying a file on disk.**
  - Cocoa:
  - GTK+: Gtk.FileChooserDialog
  - iOS:

- **File Save - a mechanism for finding and specifying a filename to save to.**
  - Cocoa: Done
  - GTK+:
  - iOS:

Miscellaneous

One of the aims of Toga is to provide a rich, feature-driven approach to app development. This requires the development of APIs to support rich features.

- Long running tasks - GUI toolkits have a common pattern of needing to periodically update a GUI based on some long running background task. They usually accomplish this with some sort of timer-based API to ensure that the main event loop keeps running. Python has a "yield" keyword that can be prepurposed for this.
- Toolbar - support for adding a toolbar to an app definition. Interpretation in mobile will be difficult; maybe some sort of top level action menu available via a slideout tray (e.g., GMail account selection tray)
- Preferences - support for saving app preferences, and visualizing them in a platform native way.
- Easy handling of long running tasks - possibly using generators to yield control back to the event loop.
- Notification when updates are available
- Easy Licensing/registration of apps. Monetization is not a bad thing, and shouldn't be mutually exclusive with open source.

Platforms

Toga currently has good support for Cocoa on OS X, GTK+, and iOS. Proof-of-concept support exists for Windows Win32. Support for a more modern Windows API would be desirable.

In the mobile space, it would be great if Toga supported Android, Windows Phone, or any other phone platform.