Toga Documentation

Release 0.3.1

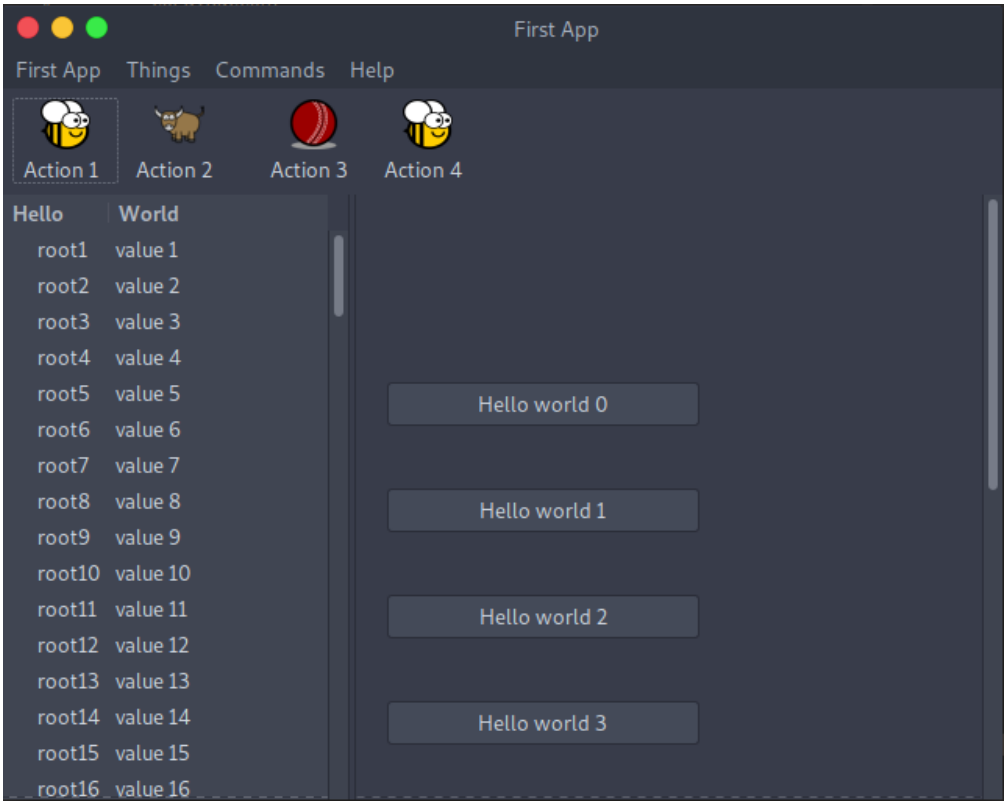
Russell Keith-Magee

Apr 12, 2023

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 macOS, Windows, Linux (GTK), Android, iOS, and for single-page web apps.



CHAPTER

ONE

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

TWO

COMMUNITY

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

- @beeware@fosstodon.org on Mastodon
- Discord
- The Toga Github Discussions forum

We foster a welcoming and respectful community as described in our BeeWare Community Code of Conduct.

2.1 Tutorials

2.1.1 Your first Toga app

Note: Toga is a work in progress, and may not be consistent across all platforms.

Please check the Tutorial Issues label on GitHub to see what's currently broken.

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.

Set up your development environment

Make sure you installed the Toga prerequisites, such as Python 3 and the other libraries. Then create a working directory for your code and change to it.

If Python 3 is *not* installed, you can do so via the official installer, or via pyenv, as described in the environment page.

The recommended way of setting up your development environment for Toga is to install a virtual environment, install the required dependencies and start coding. To set up a virtual environment, run:

macOS

```
$ python3 -m venv venv
$ source venv/bin/activate
```

Linux

```
$ python3 -m venv venv
$ source venv/bin/activate
```

Windows

```
C:\...>py -m venv venv
C:\...>venv\Scripts\activate.bat
```

Your prompt should now have a `(venv)` prefix in front of it.

Next, install Toga into your virtual environment:

macOS

```
(venv) $ python -m pip install toga
```

Linux

Before you install toga, you'll need to install some system packages. These instructions are different on almost every version of Linux; here are some of the common alternatives:

```
# Ubuntu 18.04+ / Debian 10+
(venv) $ sudo apt-get update
(venv) $ sudo apt-get install python3-dev python3-cairo-dev python3-gi-cairo␣
↪libgirepository1.0-dev libcairo2-dev libpango1.0-dev gir1.2-webkit2-4.0 pkg-config

# Fedora
(venv) $ sudo dnf install pkg-config python3-devel gobject-introspection-devel cairo-
↪devel cairo-gobject-devel pango-devel webkitgtk3

# Arch / Manjaro
(venv) $ sudo pacman -Syu git pkgconf cairo python-cairo pango gobject-introspection␣
↪gobject-introspection-runtime python-gobject webkit2gtk

# FreeBSD
(venv) $ sudo pkg update
(venv) $ sudo pkg install gtk3 pango gobject-introspection cairo webkit2-gtk3
```

If you're not using one of these, you'll need to work out how to install the developer libraries for `python3`, `cairo`, `pango`, and `gobject-introspection` (and please let us know so we can improve this documentation!)

Then, install toga:

```
(venv) $ python -m pip install toga
```

Windows

```
(venv) C:\...>python -m pip install toga
```

If you get other errors, please check that you followed the prerequisite instructions.

After a successful installation of Toga you are ready to get coding.

Write the app

Create a new file called `helloworld.py` and add the following code for the "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.padding = 50
    button.style.flex = 1
    box.add(button)

    return box


def main():
    return toga.App("First App", "org.beeware.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, which is 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")
```

When the app gets instantiated (in *main()*, discussed below), Toga will create a window with a menu. We need to provide a method that tells Toga what content to display in the window. The method can be named anything, it just needs to accept 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. By default, Toga uses a style algorithm called `Pack`, which is a bit like "CSS-lite". We can set style properties of the button:

```
button.style.padding = 50
```

What we've done here is say that the button will have a padding of 50 pixels on all sides. If we wanted to define padding of 20 pixels on top of the button, we could have defined `padding_top = 20`, or we could have specified the `padding = (20, 50, 50, 50)`.

Now we will make the button take up all the available width:

```
button.style.flex = 1
```

The `flex` attribute specifies how an element is sized with respect to other elements along its direction. The default direction is row (horizontal) and since the button is the only element here, it will take up the whole width. Check out style docs for more information on how to use the `flex` attribute.

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

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

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 space 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 method 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.beeware.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

The app acts as a Python module, which means you need to run it in a different manner than running a regular Python script: You need to specify the `-m` flag and *not* include the `.py` extension for the script name.

Here is the command to run for your platform from your working directory:

macOS

```
(venv) $ python -m helloworld
```

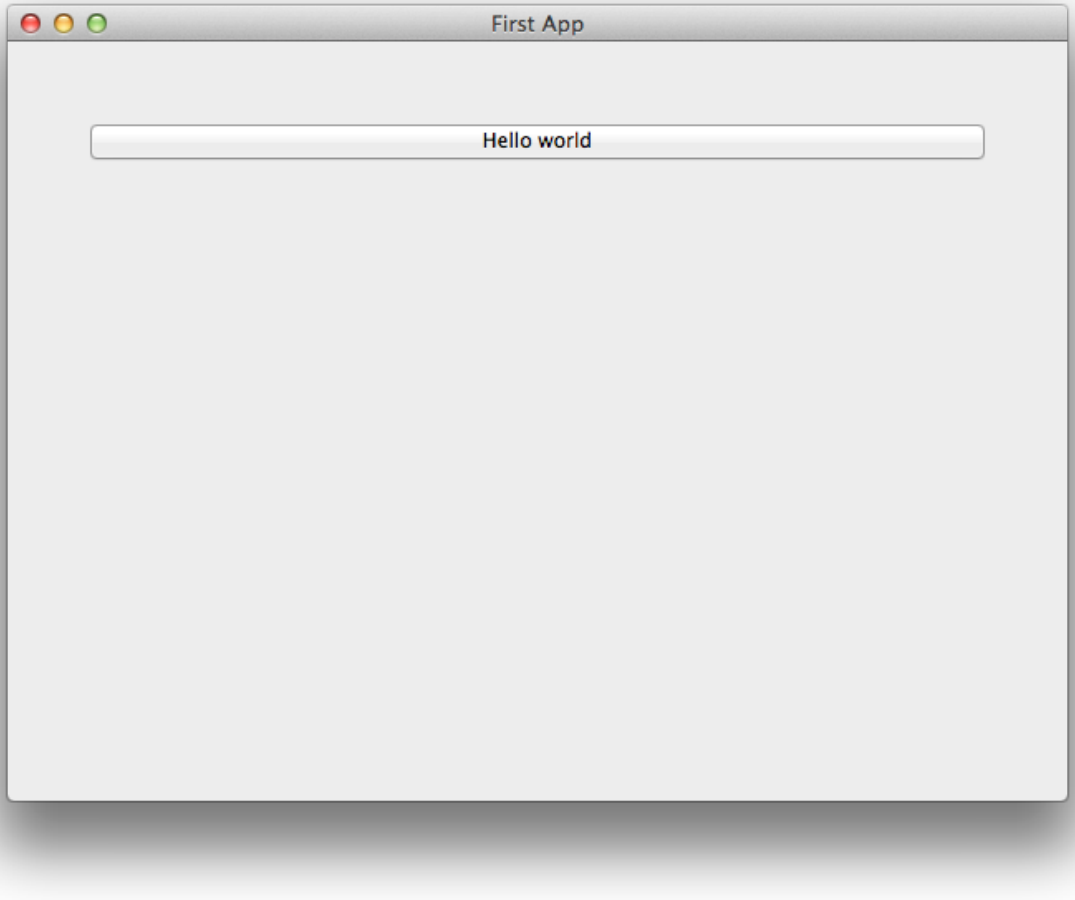
Linux

```
(venv) $ python -m helloworld
```

Windows

```
(venv) C:\...>python -m helloworld
```

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).

Troubleshooting issues

Occasionally you might run into issues running Toga on your computer.

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

```
$ python -m 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 as directed at the top of this guide.

Note: Minimum versions

Toga has some minimum requirements:

- If you're on macOS, you need to be on 10.10 (Yosemite) or newer.
- If you're on Linux (or another Unix-based operating system), you need to have GTK+ 3.10 or newer. This is the version that ships starting with Ubuntu 14.04 and Fedora 20.
- If you're on Windows, you need to have Windows 10 or newer.

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:

```
(venv) $ 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()
```

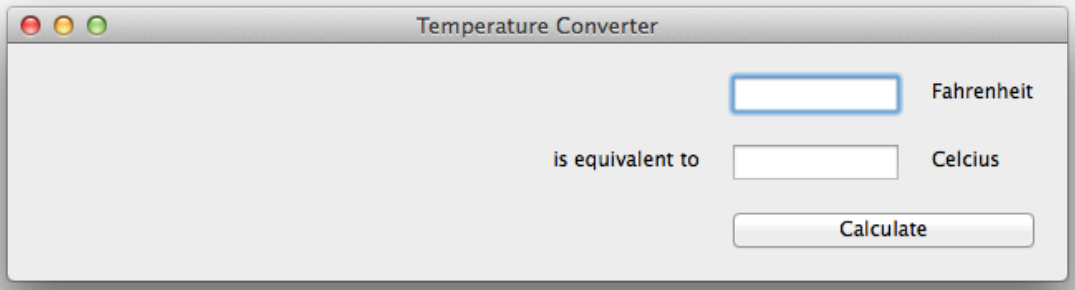
Toga apps must be executed as modules - hence the `-m` flag.

2.1.2 A slightly less toy example

Note: Toga is a work in progress, and may not be consistent across all platforms.

Please check the Tutorial Issues label on GitHub to see what's currently broken.

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
from toga.style.pack import COLUMN, LEFT, RIGHT, ROW, Pack


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", style=Pack(text_align=LEFT))
    f_label = toga.Label("Fahrenheit", style=Pack(text_align=LEFT))
    join_label = toga.Label("is equivalent to", style=Pack(text_align=RIGHT))

    def calculate(widget):
        try:
            c_input.value = (float(f_input.value) - 32.0) * 5.0 / 9.0
        except ValueError:
            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.update(direction=COLUMN, padding=10)
```

(continues on next page)

(continued from previous page)

```
    f_box.style.update(direction=ROW, padding=5)
    c_box.style.update(direction=ROW, padding=5)

    c_input.style.update(flex=1)
    f_input.style.update(flex=1, padding_left=160)
    c_label.style.update(width=100, padding_left=10)
    f_label.style.update(width=100, padding_left=10)
    join_label.style.update(width=150, padding_right=10)

    button.style.update(padding=15)

    return box


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


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

This example shows off some more features of Toga's Pack style engine. 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...

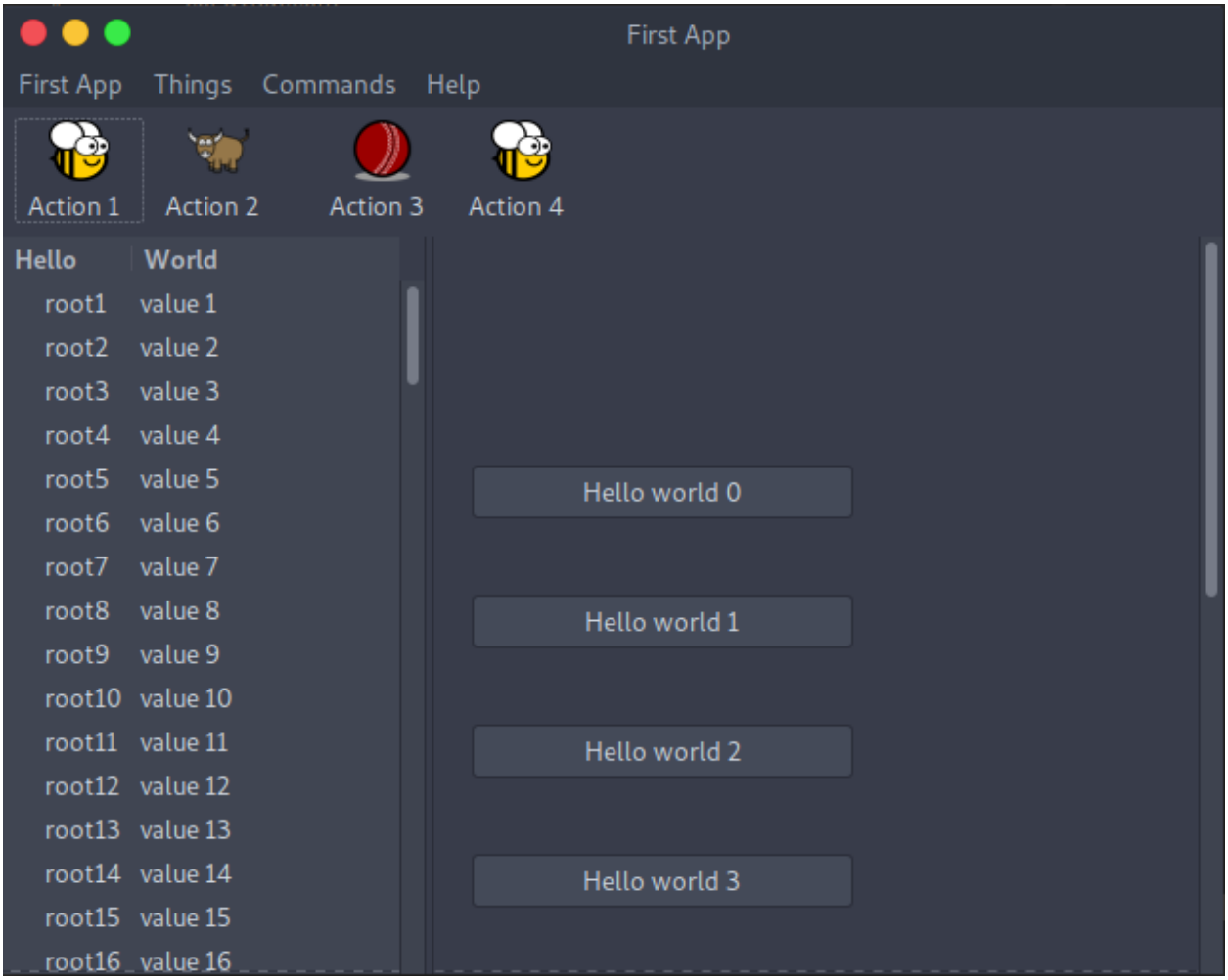
Note: Toga is a work in progress, and may not be consistent across all platforms.

Please check the Tutorial Issues label on GitHub to see what's currently broken.

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's Pack style engine borrows heavily from 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 toga.style.pack import COLUMN, Pack


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")
```

(continues on next page)

(continued from previous page)

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


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


def action5(widget):
    print("action 5")


def action6(widget):
    print("action 6")


def build(app):
    brutus_icon = "icons/brutus"
    cricket_icon = "icons/cricket-72.png"

    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=Pack(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=Pack(width=200, padding=20),
            )
        )

    right_container = toga.ScrollContainer(horizontal=False)

    right_container.content = right_content

    split = toga.SplitContainer()

    # The content of the split container can be specified as a simple list:
    #    split.content = [left_container, right_container]
    # but you can also specify "weight" with each content item, which will
    # set an initial size of the columns to make a "heavy" column wider than
    # a narrower one. In this example, the right container will be twice
    # as wide as the left one.
    split.content = [(left_container, 1), (right_container, 2)]

    # Create a "Things" menu group to contain some of the commands.
    # No explicit ordering is provided on the group, so it will appear
```

(continues on next page)

(continued from previous page)

```
    # after application-level menus, but *before* the Command group.
    # Items in the Things group are not explicitly ordered either, so they
    # will default to alphabetical ordering within the group.
    things = toga.Group("Things")
    cmd0 = toga.Command(
        action0,
        text="Action 0",
        tooltip="Perform action 0",
        icon=brutus_icon,
        group=things,
    )
    cmd1 = toga.Command(
        action1,
        text="Action 1",
        tooltip="Perform action 1",
        icon=brutus_icon,
        group=things,
    )
    cmd2 = toga.Command(
        action2,
        text="Action 2",
        tooltip="Perform action 2",
        icon=toga.Icon.TOGA_ICON,
        group=things,
    )

    # Commands without an explicit group end up in the "Commands" group.
    # The items have an explicit ordering that overrides the default
    # alphabetical ordering
    cmd3 = toga.Command(
        action3,
        text="Action 3",
        tooltip="Perform action 3",
        shortcut=toga.Key.MOD_1 + "k",
        icon=cricket_icon,
        order=3,
    )

    # Define a submenu inside the Commands group.
    # The submenu group has an order that places it in the parent menu.
    # The items have an explicit ordering that overrides the default
    # alphabetical ordering.
    sub_menu = toga.Group("Sub Menu", parent=toga.Group.COMMANDS, order=2)
    cmd5 = toga.Command(
        action5, text="Action 5", tooltip="Perform action 5", order=2, group=sub_menu
    )
    cmd6 = toga.Command(
        action6, text="Action 6", tooltip="Perform action 6", order=1, group=sub_menu
    )

    def action4(widget):
        print("CALLING Action 4")
```

(continues on next page)

(continued from previous page)

```
        cmd3.enabled = not cmd3.enabled

    cmd4 = toga.Command(
        action4, text="Action 4", tooltip="Perform action 4", icon=brutus_icon, order=1
    )

    # The order in which commands are added to the app or the toolbar won't
    # alter anything. Ordering is defined by the command definitions.
    app.commands.add(cmd1, cmd0, cmd6, cmd4, cmd5, cmd3)
    app.main_window.toolbar.add(cmd1, cmd3, cmd2, cmd4)

    return split


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


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

In order to render the icons, you will need to move the icons folder into the same directory as your app file.

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.

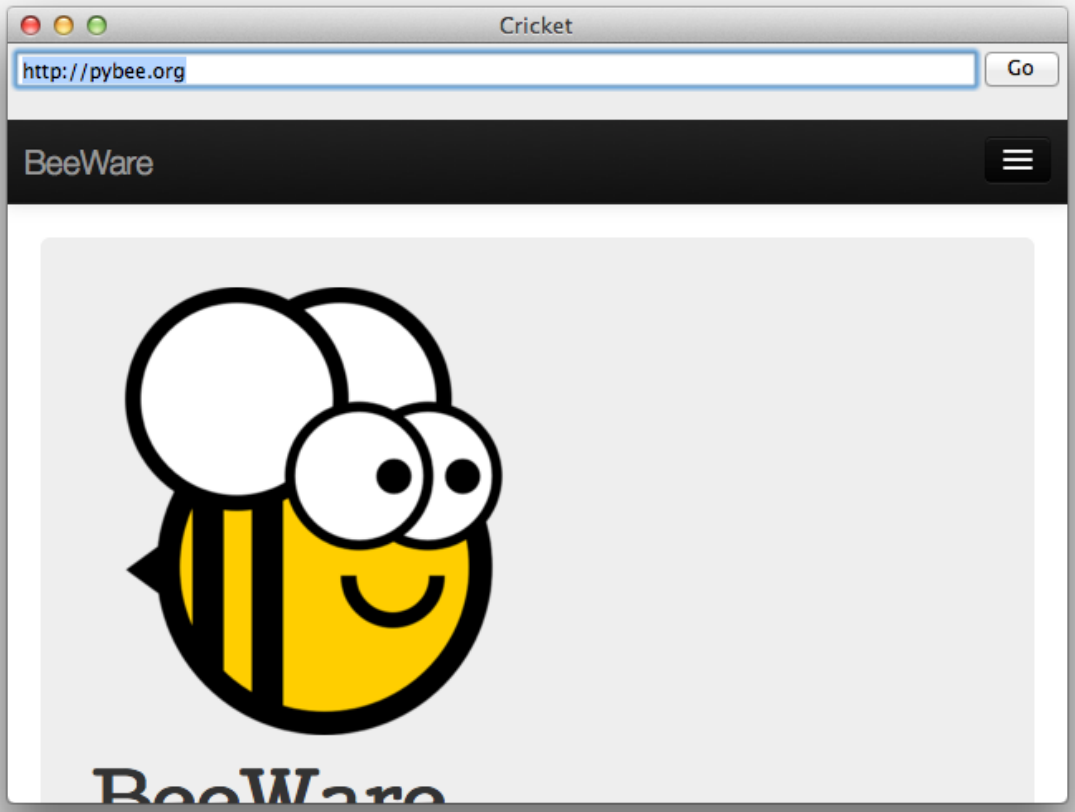
2.1.4 Let's build a browser!

Note: Toga is a work in progress, and may not be consistent across all platforms.

Please check the Tutorial Issues label on GitHub to see what's currently broken.

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:

```
import toga
from toga.style.pack import CENTER, COLUMN, ROW, Pack


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

        self.webview = toga.WebView(
            on_webview_load=self.on_webview_loaded, style=Pack(flex=1)
        )
        self.url_input = toga.TextInput(
            value="https://beeware.org/", style=Pack(flex=1)
        )

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

(continues on next page)

(continued from previous page)

```
                        toga.Button(
                            "Go",
                            on_press=self.load_page,
                            style=Pack(width=50, padding_left=5),
                        ),
                    ],
                    style=Pack(
                        direction=ROW,
                        alignment=CENTER,
                        padding=5,
                    ),
                ),
                self.webview,
            ],
            style=Pack(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

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


def main():
    return Graze("Graze", "org.beeware.graze")


if __name__ == "__main__":
    main().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 Let's draw on a canvas!

Note: Toga is a work in progress, and may not be consistent across all platforms.

Please check the Tutorial Issues label on GitHub to see what's currently broken.

One of the main capabilities needed to create many types of GUI applications is the ability to draw and manipulate lines, shapes, text, and other graphics. To do this in Toga, we use the Canvas Widget.

Utilizing the Canvas is as easy as determining the drawing operations you want to perform and then creating a new

Canvas. All drawing objects that are created with one of the drawing operations are returned so that they can be modified or removed.

1. We first define the drawing operations we want to perform in a new function:

```
def draw_eyes(self):
    with self.canvas.fill(color=WHITE) as eye_whites:
        eye_whites.arc(58, 92, 15)
        eye_whites.arc(88, 92, 15, math.pi, 3 * math.pi)
```

Notice that we also created and used a new fill context called eye_whites. The "with" keyword that is used for the fill operation causes everything draw using the context to be filled with a color. In this example we filled two circular eyes with the color white.

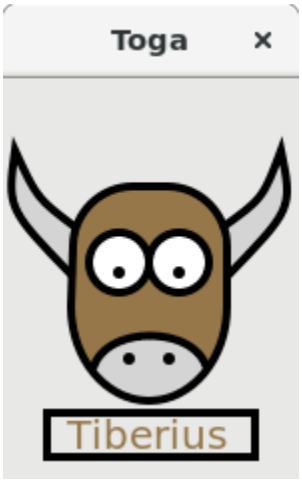
2. Next we create a new Canvas:

```
self.canvas = toga.Canvas(style=Pack(flex=1))
```

That's all there is to! In this example we also add our canvas to the MainWindow through use of the Box Widget:

```
box = toga.Box(children=[self.canvas])
self.main_window.content = box
```

You'll also notice in the full example below that the drawing operations utilize contexts in addition to fill including context, closed_path, and stroke. This reduces the repetition of commands as well as groups drawing operations so that they can be modified together.



Here's the source code

```
import math

import toga
from toga.colors import WHITE, rgb
from toga.fonts import SANS_SERIF
from toga.style import Pack


class StartApp(toga.App):
    def startup(self):
```

(continues on next page)

(continued from previous page)

```
        # Main window of the application with title and size
        self.main_window = toga.MainWindow(title=self.name, size=(148, 250))

        # Create canvas and draw tiberius on it
        self.canvas = toga.Canvas(style=Pack(flex=1))
        box = toga.Box(children=[self.canvas])

        # Add the content on the main window
        self.main_window.content = box

        self.draw_tiberius()

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

    def fill_head(self):
        with self.canvas.fill(color=rgb(149, 119, 73)) as head_filler:
            head_filler.move_to(112, 103)
            head_filler.line_to(112, 113)
            head_filler.ellipse(73, 114, 39, 47, 0, 0, math.pi)
            head_filler.line_to(35, 84)
            head_filler.arc(65, 84, 30, math.pi, 3 * math.pi / 2)
            head_filler.arc(82, 84, 30, 3 * math.pi / 2, 2 * math.pi)

    def stroke_head(self):
        with self.canvas.stroke(line_width=4.0) as head_stroker:
            with head_stroker.closed_path(112, 103) as closed_head:
                closed_head.line_to(112, 113)
                closed_head.ellipse(73, 114, 39, 47, 0, 0, math.pi)
                closed_head.line_to(35, 84)
                closed_head.arc(65, 84, 30, math.pi, 3 * math.pi / 2)
                closed_head.arc(82, 84, 30, 3 * math.pi / 2, 2 * math.pi)

    def draw_eyes(self):
        with self.canvas.fill(color=WHITE) as eye_whites:
            eye_whites.arc(58, 92, 15)
            eye_whites.arc(88, 92, 15, math.pi, 3 * math.pi)
        with self.canvas.stroke(line_width=4.0) as eye_outline:
            eye_outline.arc(58, 92, 15)
            eye_outline.arc(88, 92, 15, math.pi, 3 * math.pi)
        with self.canvas.fill() as eye_pupils:
            eye_pupils.arc(58, 97, 3)
            eye_pupils.arc(88, 97, 3)

    def draw_horns(self):
        with self.canvas.context() as r_horn:
            with r_horn.fill(color=rgb(212, 212, 212)) as r_horn_filler:
                r_horn_filler.move_to(112, 99)
                r_horn_filler.quadratic_curve_to(145, 65, 139, 36)
                r_horn_filler.quadratic_curve_to(130, 60, 109, 75)
            with r_horn.stroke(line_width=4.0) as r_horn_stroker:
                r_horn_stroker.move_to(112, 99)
```

(continues on next page)

(continued from previous page)

```
                r_horn_stroker.quadratic_curve_to(145, 65, 139, 36)
                r_horn_stroker.quadratic_curve_to(130, 60, 109, 75)
        with self.canvas.context() as l_horn:
            with l_horn.fill(color=rgb(212, 212, 212)) as l_horn_filler:
                l_horn_filler.move_to(35, 99)
                l_horn_filler.quadratic_curve_to(2, 65, 6, 36)
                l_horn_filler.quadratic_curve_to(17, 60, 37, 75)
            with l_horn.stroke(line_width=4.0) as l_horn_stroker:
                l_horn_stroker.move_to(35, 99)
                l_horn_stroker.quadratic_curve_to(2, 65, 6, 36)
                l_horn_stroker.quadratic_curve_to(17, 60, 37, 75)

    def draw_nostrils(self):
        with self.canvas.fill(color=rgb(212, 212, 212)) as nose_filler:
            nose_filler.move_to(45, 145)
            nose_filler.bezier_curve_to(51, 123, 96, 123, 102, 145)
            nose_filler.ellipse(73, 114, 39, 47, 0, math.pi / 4, 3 * math.pi / 4)
        with self.canvas.fill() as nostril_filler:
            nostril_filler.arc(63, 140, 3)
            nostril_filler.arc(83, 140, 3)
        with self.canvas.stroke(line_width=4.0) as nose_stroker:
            nose_stroker.move_to(45, 145)
            nose_stroker.bezier_curve_to(51, 123, 96, 123, 102, 145)

    def draw_text(self):
        x = 32
        y = 185
        font = toga.Font(family=SANS_SERIF, size=20)
        width, height = self.canvas.measure_text("Tiberius", font, tight=True)
        with self.canvas.stroke(line_width=4.0) as rect_stroker:
            rect_stroker.rect(x - 2, y - height + 2, width, height + 2)
        with self.canvas.fill(color=rgb(149, 119, 73)) as text_filler:
            text_filler.write_text("Tiberius", x, y, font)

    def draw_tiberius(self):
        self.fill_head()
        self.draw_eyes()
        self.draw_horns()
        self.draw_nostrils()
        self.stroke_head()
        self.draw_text()


def main():
    return StartApp("Tutorial 4", "org.beeware.helloworld")


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

In this example, we see a new Toga widget - *Canvas*.

2.1.6 Tutorial 0 - your first Toga app

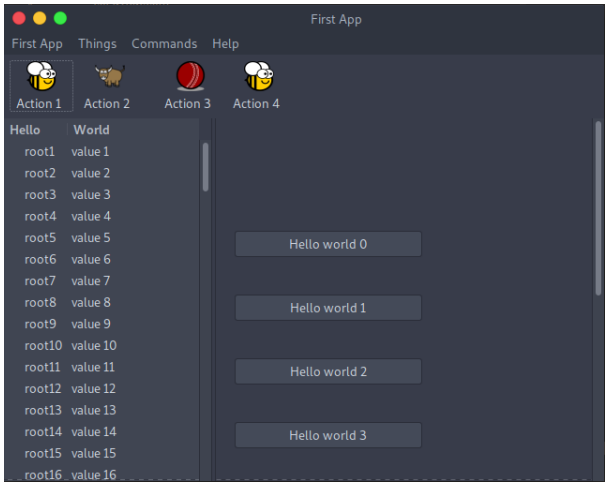
In *Your first Toga app*, you will discover how to create a basic app and have a simple `toga.widgets.button.Button` widget to click.

2.1.7 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.widgets.textinput.TextInput` widget and control layout.

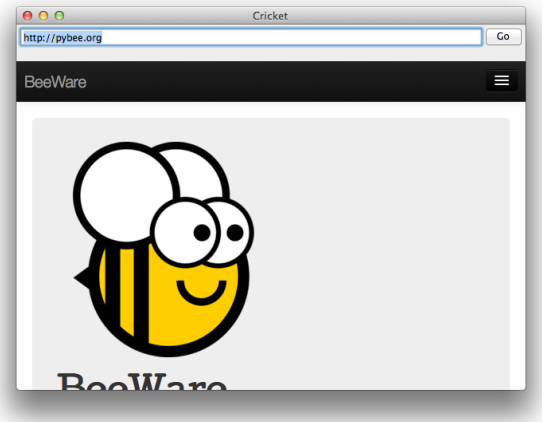
2.1.8 Tutorial 2 - you put the box inside another box...

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



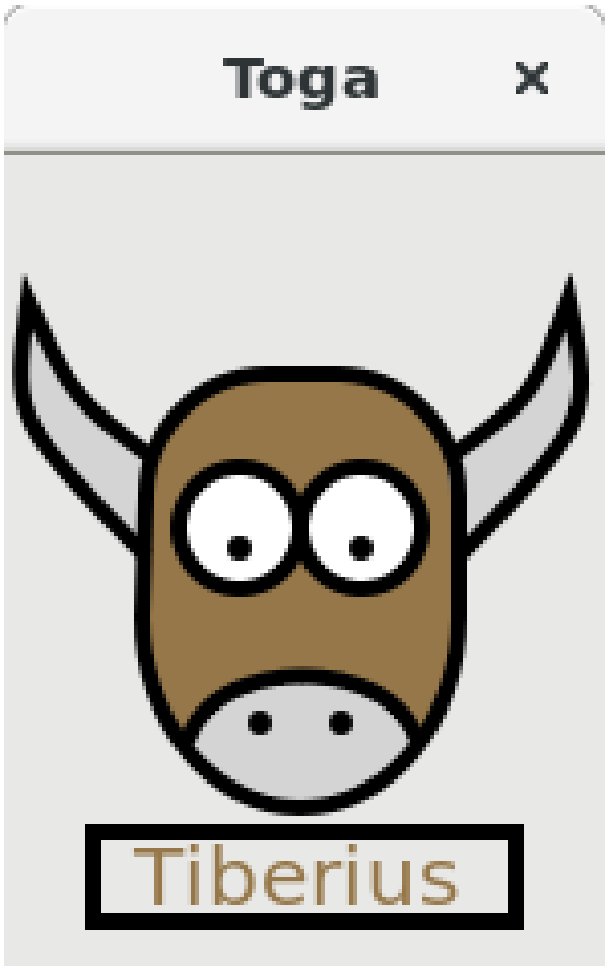
2.1.9 Tutorial 3 - let's build a browser!

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



2.1.10 Tutorial 4 - let's draw on a canvas!

In *Let's draw on a canvas!*, you will discover how to use the `toga.widgets.canvas.Canvas` widget to draw lines and shapes on a canvas.



2.2 How-to Guides

How-to guides are recipes that take the user through steps in key subjects. They are more advanced than tutorials and assume a lot more about what the user already knows than tutorials do, and unlike documents in the tutorial they can stand alone.

2.2.1 How to get started

Quickstart

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

```
$ python -m 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 code 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. You may also find this presentation by BeeWare team member Dan Yeaw helpful. This talk gives an architectural overview of Toga, as well as providing a guide to the process of adding new widgets.

Set up your development environment

First thing is to ensure that you have Python 3 and pip installed. To do this run the following commands:

macOS

```
$ python3 --version
$ pip3 --version
```

Linux

```
$ python3 --version
$ pip3 --version
```

Windows

```
C:\...>python3 --version
C:\...>pip3 --version
```

The recommended way of setting up your development environment for Toga is to install a virtual environment, install the required dependencies and start coding. To set up a virtual environment, run:

macOS

```
$ python3 -m venv venv
$ source venv/bin/activate
```

Linux

```
$ python3 -m venv venv
$ source venv/bin/activate
```

Windows

```
C:\...>python3 -m venv venv
C:\...>venv/Scripts/activate
```

Your prompt should now have a `(venv)` prefix in front of it.

Next, install any additional dependencies for your operating system:

macOS

No additional dependencies

Linux

```
# Ubuntu 18.04+, Debian 10+
(venv) $ sudo apt-get update
(venv) $ sudo apt-get install python3-dev libgirepository1.0-dev libcairo2-dev libpango1.
↪0-dev libwebkit2gtk-4.0-37 gir1.2-webkit2-4.0

# Fedora
(venv) $ sudo dnf install pkg-config python3-devel gobject-introspection-devel cairo-
↪devel cairo-gobject-devel pango-devel webkitgtk3

# Arch / Manjaro
(venv) $ sudo pacman -Syu git pkgconf cairo python-cairo pango gobject-introspection␣
↪gobject-introspection-runtime python-gobject webkit2gtk

# FreeBSD
(venv) $ sudo pkg update
(venv) $ sudo pkg install gtk3 pango gobject-introspection cairo webkit2-gtk3
```

Windows

No additional dependencies

Next, go to the Toga page on GitHub, and fork the repository into your own account, and then clone a copy of that repository onto your computer by clicking on "Clone or Download". If you have the GitHub desktop application installed on your computer, you can select "Open in Desktop"; otherwise, copy the URL provided, and use it to clone using the command line:

macOS

Fork the Toga repository, and then:

```
(venv) $ git clone https://github.com/<your username>/toga.git
```

(substituting your GitHub username)

Linux

Fork the Toga repository, and then:

```
(venv) $ git clone https://github.com/<your username>/toga.git
```

(substituting your GitHub username)

Windows

Fork the Toga repository, and then:

```
(venv) C:\...>git clone https://github.com/<your username>/toga.git
```

(substituting your GitHub username)

Now that you have the source code, you can 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:

macOS

```
(venv) $ cd toga
(venv) $ pip install -e ./core[dev]
(venv) $ pip install -e ./dummy
(venv) $ pip install -e ./cocoa
```

Linux

```
(venv) $ cd toga
(venv) $ pip install -e ./core[dev]
(venv) $ pip install -e ./dummy
(venv) $ pip install -e ./gtk
```

Windows

```
(venv) C:\...>cd toga
(venv) C:\...>pip install -e ./core[dev]
(venv) C:\...>pip install -e ./dummy
(venv) C:\...>pip install -e ./winforms
```

This project uses a tool called Pre-Commit to identify simple issues and standardize code formatting. It does this by installing a git hook that automatically runs a series of code linters prior to finalizing any git commit. To enable pre-commit, run:

macOS

```
(venv) $ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

Linux

```
(venv) $ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

Windows

```
(venv) C:\...>pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

When you commit any change, pre-commit will run automatically. If there are any issues found with the commit, this will cause your commit to fail. Where possible, pre-commit will make the changes needed to correct the problems it has found:

macOS

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
black....................................................................Failed
- hook id: black
- files were modified by this hook

reformatted some/interesting_file.py

All done!
1 file reformatted.

flake8...................................................................Passed
check toml...........................................(no files to check)Skipped
check yaml...........................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
fix end of files.........................................................Passed
trim trailing whitespace.................................................Passed
isort....................................................................Passed
pyupgrade................................................................Passed
docformatter.............................................................Passed
```

Linux

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
black....................................................................Failed
- hook id: black
- files were modified by this hook

reformatted some/interesting_file.py

All done!
1 file reformatted.

flake8...................................................................Passed
check toml...........................................(no files to check)Skipped
check yaml...........................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
fix end of files.........................................................Passed
trim trailing whitespace.................................................Passed
isort....................................................................Passed
pyupgrade................................................................Passed
docformatter.............................................................Passed
```

Windows

```
(venv) C:\...>git add some/interesting_file.py
(venv) C:\...>git commit -m "Minor change"
black....................................................................Failed
- hook id: black
- files were modified by this hook
```

(continues on next page)

(continued from previous page)

```
reformatted some\interesting_file.py

All done!
1 file reformatted.

flake8...................................................................Passed
check toml.....................................(no files to check)Skipped
check yaml.....................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
fix end of files.........................................................Passed
trim trailing whitespace.................................................Passed
isort....................................................................Passed
pyupgrade................................................................Passed
docformatter.............................................................Passed
```

You can then re-add any files that were modified as a result of the pre-commit checks, and re-commit the change.

macOS

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
black....................................................................Passed
flake8...................................................................Passed
check toml.....................................(no files to check)Skipped
check yaml.....................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
fix end of files.........................................................Passed
trim trailing whitespace.................................................Passed
isort....................................................................Passed
pyupgrade................................................................Passed
docformatter.............................................................Passed
[bugfix e3e0f73] Minor change
1 file changed, 4 insertions(+), 2 deletions(-)
```

Linux

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
black....................................................................Passed
flake8...................................................................Passed
check toml.....................................(no files to check)Skipped
check yaml.....................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
fix end of files.........................................................Passed
trim trailing whitespace.................................................Passed
isort....................................................................Passed
pyupgrade................................................................Passed
docformatter.............................................................Passed
[bugfix e3e0f73] Minor change
```

(continues on next page)

(continued from previous page)

```
1 file changed, 4 insertions(+), 2 deletions(-)
```

Windows

```
(venv) C:\...>git add some\interesting_file.py
(venv) C:\...>git commit -m "Minor change"
black....................................................................Passed
flake8...................................................................Passed
check toml.....................................(no files to check)Skipped
check yaml.....................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
fix end of files.........................................................Passed
trim trailing whitespace.................................................Passed
isort....................................................................Passed
pyupgrade................................................................Passed
docformatter.............................................................Passed
```

Now you are ready to start hacking on Toga!

What should I do?

Start by running the core test suite. Toga uses tox to manage the testing process. To run the core test suite:

macOS

```
(venv) $ tox -e py-core
```

Linux

```
(venv) $ tox -e py-core
```

Windows

```
(venv) C:\...>tox -e py-core
```

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!

Although the tests should all pass, the test suite itself is still 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.

At the end of the test output there should be a report of the coverage data that was gathered:

```
Name                                  Stmts   Miss  Cover   Missing
-------------------------------------------------------------------
toga/__init__.py                         29      0   100%
toga/app.py                              50      0   100%
...
```

(continues on next page)

(continued from previous page)

```
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 *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:

```
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.

Add change information for release notes

Before you submit this change as a pull request, there's one more thing required. Toga uses towncrier to automate building release notes. To support this, every pull request needs to have a corresponding file in the `changes/` directory that provides a short description of the change implemented by the pull request.

This description should be a high level summary of the change from the perspective of the user, not a deep technical description or implementation detail. It is distinct from a commit message - a commit message describes what has been done so that future developers can follow the reasoning for a change; the change note is a "user facing" description. For example, if you fix a bug caused by date handling, the commit message might read:

> Modified date validation to accept US-style MM-DD-YYYY format.

The corresponding change note would read something like:

> Date widgets can now accept US-style MM-DD-YYYY format.

See News Fragments for more details on the types of news fragments you can add. You can also see existing examples of news fragments in the `changes/` folder. Name the file using the number of the issue that your pull request is addressing. When there isn't an existing issue, you can create the pull request in two passes: First submit it without a change note - this will fail, but will also assign a pull request number. You can then push an update to the pull request, adding the change note with the assigned number.

Once you've written your code, test, and change note, you can submit your changes as a pull request. One of the core team will review your work, and give feedback. If any changes are requested, you can make those changes, and update

your pull request; eventually, the pull request will be accepted and merged. Congratulations, you're a contributor to Toga!

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.

One example of the type of consistency we're looking for is described in this ticket.

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!

Advanced Mode

If you've got expertise in a particular platform (for example, if you've got experience writing iOS apps), or you'd *like* to have that experience, you might want to look into a more advanced problem. Here are some suggestions:

- **Implement a platform native widget** If the core library already specifies an interface, implement that interface; if no interface exists, propose an interface design, and implement it for at least one platform.
- **Add a new feature to an existing widget API** Can you think of a feature than an existing widget should have? Propose a new API for that widget, and provide a sample implementation.
- **Improve platform specific testing** The tests that have been described in this document are all platform independent. They use the dummy backend to validate that data is being passed around correctly, but they don't validate that on a given platform, widgets behave they way they should. If I put a button on a Toga app, is that button displayed? Is it in the right place? Does it respond to mouse clicks? Ideally, we'd have automated tests to validate these properties. However, automated tests of GUI operations can be difficult to set up. If you've got experience with automated GUI testing, we'd love to hear your suggestions.
- **Improve the testing API for application writers** The dummy backend exists to validate that Toga's internal API works as expected. However, we would like it to be a useful resource for *application* authors as well. Testing GUI applications is a difficult task; a Dummy backend would potentially allow an end user to write an application, and validate behavior by testing the properties of the Dummy. Think of it as a GUI mock - but one that is baked into Toga as a framework. See if you can write a GUI app of your own, and write a test suite that uses the Dummy backend to validate the behavior of that app.

2.2.3 Contributing to the documentation

Here are some tips for working on this documentation. You're welcome to add more and help us out!

First of all, you should check the Restructured Text (reST) and Sphinx CheatSheet to learn how to write your `.rst` file.

Create a `.rst` file

Look at the structure and choose the best category to put your `.rst` file. Make sure that it is referenced in the index of the corresponding category, so it will show on in the documentation. If you have no idea how to do this, study the other index files for clues.

Build documentation locally

To build the documentation locally, *set up a development environment*.

You'll also need to install the Enchant spell checking library.

macOS

Enchant can be installed using Homebrew:

```
(venv) $ brew install enchant
```

If you're on an M1 machine, you'll also need to manually set the location of the Enchant library:

```
(venv) $ export PYENCHANT_LIBRARY_PATH=/opt/homebrew/lib/libenchant-2.2.dylib
```

Linux

Enchant can be installed as a system package:

Ubuntu 20.04+ / Debian 10+

```
$ sudo apt-get update
$ sudo apt-get install enchant-2
```

Fedora

```
$ sudo dnf install enchant
```

Arch, Manjaro

```
$ sudo pacman -Syu enchant
```

Windows

Enchant is installed automatically when you set up your development environment.

Once your development environment is set up, run:

macOS

```
(venv) $ tox -e docs
```

Linux

```
(venv) $ tox -e docs
```

Windows

```
(venv) C:\...>tox -e docs
```

The output of the file should be in the `docs/_build/html` folder. If there are any markup problems, they'll raise an error.

Documentation linting

Before committing and pushing documentation updates, run linting for the documentation:

macOS

```
(venv) $ tox -e docs-lint
```

Linux

```
(venv) $ tox -e docs-lint
```

Windows

```
(venv) C:\...>tox -e docs-lint
```

This will validate the documentation does not contain:

- invalid syntax and markup
- dead hyperlinks
- misspelled words

If a valid spelling of a word is identified as misspelled, then add the word to the list in `docs/spelling_wordlist`. This will add the word to the spellchecker's dictionary.

If you get an error related to SSL certificate verification:

```
Exception occurred:
  File "/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/ssl.py", line␣
↪1342, in do_handshake
    self._sslobj.do_handshake()
ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify␣
↪failed: unable to get local issuer certificate (_ssl.c:1007)
```

The root certificate on your machine is out of date. You can correct this by installing the Python package *certifi*, and using that package to provide your SSL root certificate:

macOS

```
(venv) $ python -m pip install certifi
(venv) $ export SSL_CERT_FILE=$(python -m certifi)
```

Linux

```
(venv) $ python -m pip install certifi
(venv) $ export SSL_CERT_FILE=$(python -m certifi)
```

Windows

```
(venv) C:\...>python -m pip install certifi
(venv) C:\...>FOR /f "delims=" %i IN ('python -m certifi') DO SET SSL_CERT_FILE=%i
```

Rebuilding all documentation

To force a rebuild for all of the documentation:

macOS

```
(venv) $ tox -e docs-all
```

Linux

```
(venv) $ tox -e docs-all
```

Windows

```
(venv) C:\...>tox -e docs-all
```

The documentation should be fully rebuilt in the `docs/_build/html` folder. If there are any markup problems, they'll raise an error.

2.2.4 Internal How-to guides

These guides are for the maintainers of the Toga project, documenting internal project procedures.

How to cut a Toga release

The release infrastructure for Toga is semi-automated, using GitHub Actions to formally publish releases.

This guide assumes that you have an `upstream` remote configured on your local clone of the Toga repository, pointing at the official repository. If all you have is a checkout of a personal fork of the Toga repository, you can configure that checkout by running:

```
$ git remote add upstream https://github.com/beeware/toga.git
```

The procedure for cutting a new release is as follows:

1. Check the contents of the upstream repository's main branch:

   ```
   $ git fetch upstream
   $ git checkout --detach upstream/main
   ```

   Check that the HEAD of release now matches upstream/main.

2. Ensure that the release notes are up to date. Run:

   ```
   $ tox -e towncrier -- --draft
   ```

   to review the release notes that will be included, and then:

```
$ tox -e towncrier
```

to generate the updated release notes.

3. Tag the release, and push the branch and tag upstream:

```
$ git tag v1.2.3
$ git push upstream HEAD:main
$ git push upstream v1.2.3
```

4. Pushing the tag will start a workflow to create a draft release on GitHub. You can follow the progress of the workflow on GitHub; once the workflow completes, there should be a new draft release, and entries on the TestPyPI server for toga-core, toga-cocoa, etc.

   Confirm that this action successfully completes. If it fails, there's a couple of possible causes:

   a. The final upload to TestPyPI failed. TestPyPI doesn't have the same service monitoring as PyPI-proper, so it sometimes has problems. However, it's not critical to the release process.

   b. Something else fails in the build process. If the problem can be fixed without a code change to the Toga repository (e.g., a transient problem with build machines not being available), you can re-run the action that failed through the GitHub Actions GUI. If the fix requires a code change, delete the old tag, make the code change, and re-tag the release.

5. Download the "packages" artifact from the GitHub workflow, and use its wheels to build some apps and perform any pre-release testing that may be appropriate.

6. Log into ReadTheDocs, visit the Versions tab, and activate the new version. Ensure that the build completes; if there's a problem, you may need to correct the build configuration, roll back and re-tag the release.

7. Edit the GitHub release. Add release notes (you can use the text generated by towncrier). Check the pre-release checkbox if necessary.

8. Double check everything, then click Publish. This will trigger a publication workflow on GitHub.

9. Wait for the packages to appear on PyPI (toga-core, toga-cocoa, etc.).

Congratulations, you've just published a release!

Once the release has successfully appeared on PyPI or TestPyPI, it cannot be changed. If you spot a problem after that point, you'll need to restart with a new version number.

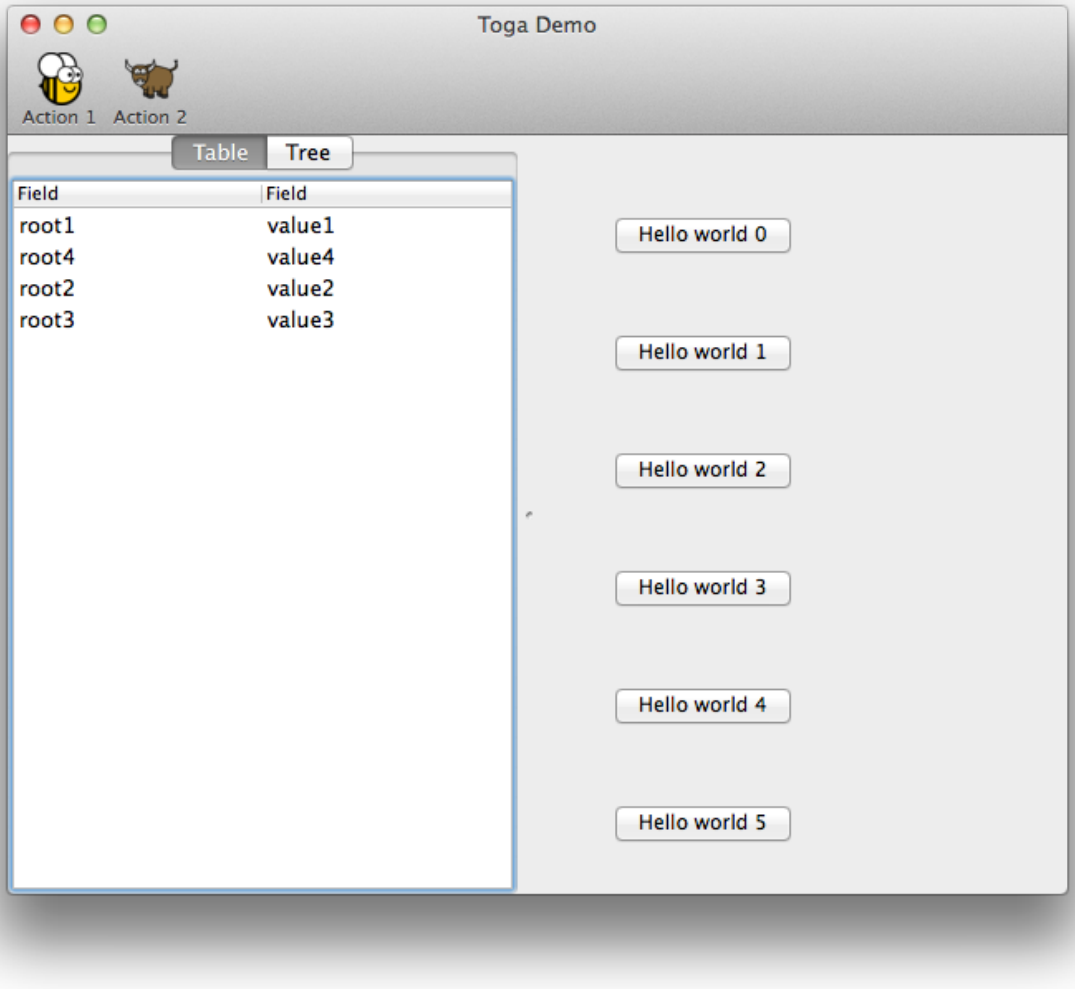
2.3 Reference

2.3.1 Toga supported platforms

Official platform support

Desktop platforms

macOS

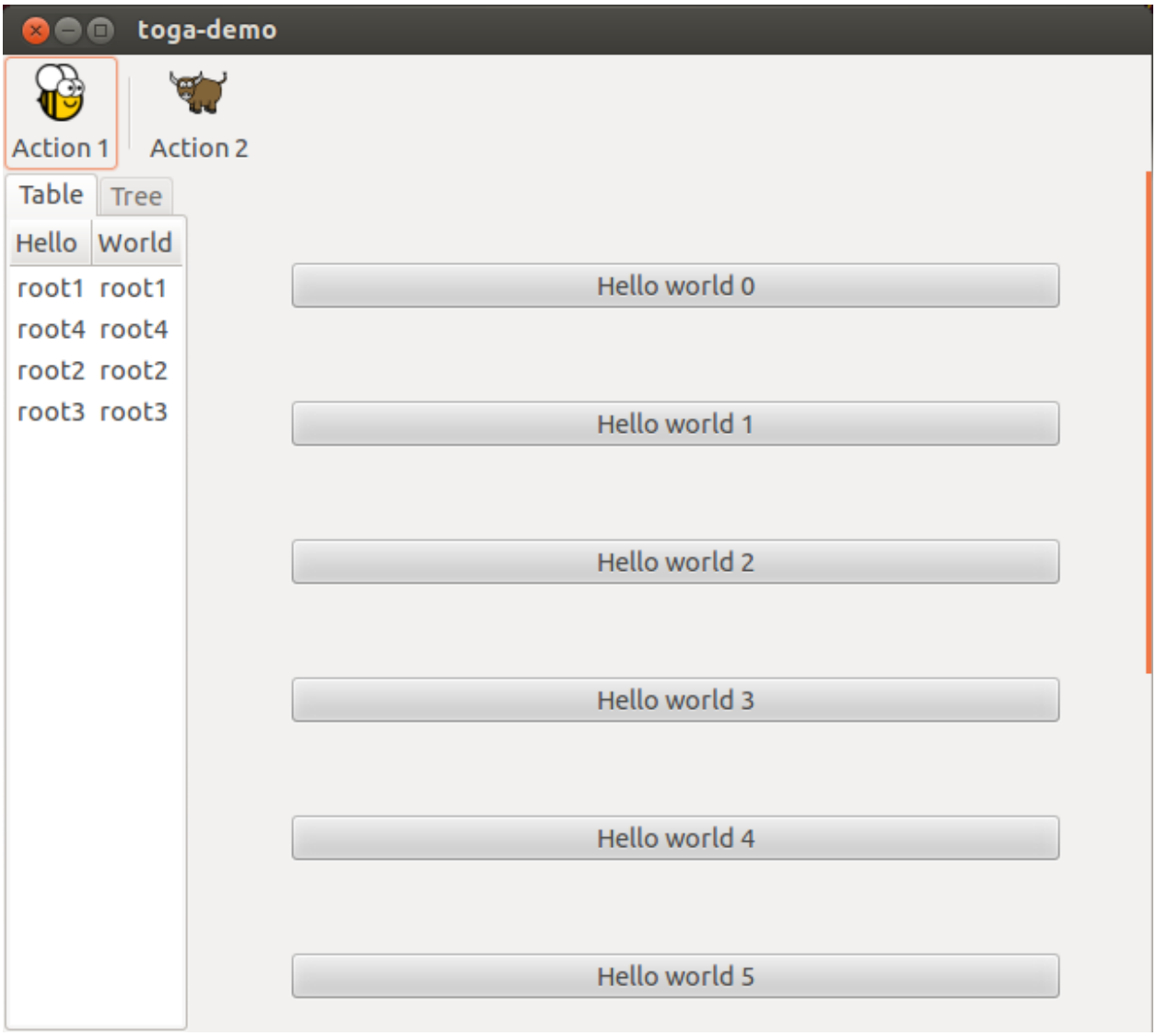


The backend for macOS is named toga-cocoa. It supports macOS 10.10 (Yosemite) and later. It is installed automatically on macOS machines (machines that report `sys.platform == 'darwin'`), or can be manually installed by invoking:

```
$ pip install toga-cocoa
```

The macOS backend has seen the most development to date. It uses Rubicon to provide a bridge to native macOS libraries.

Linux

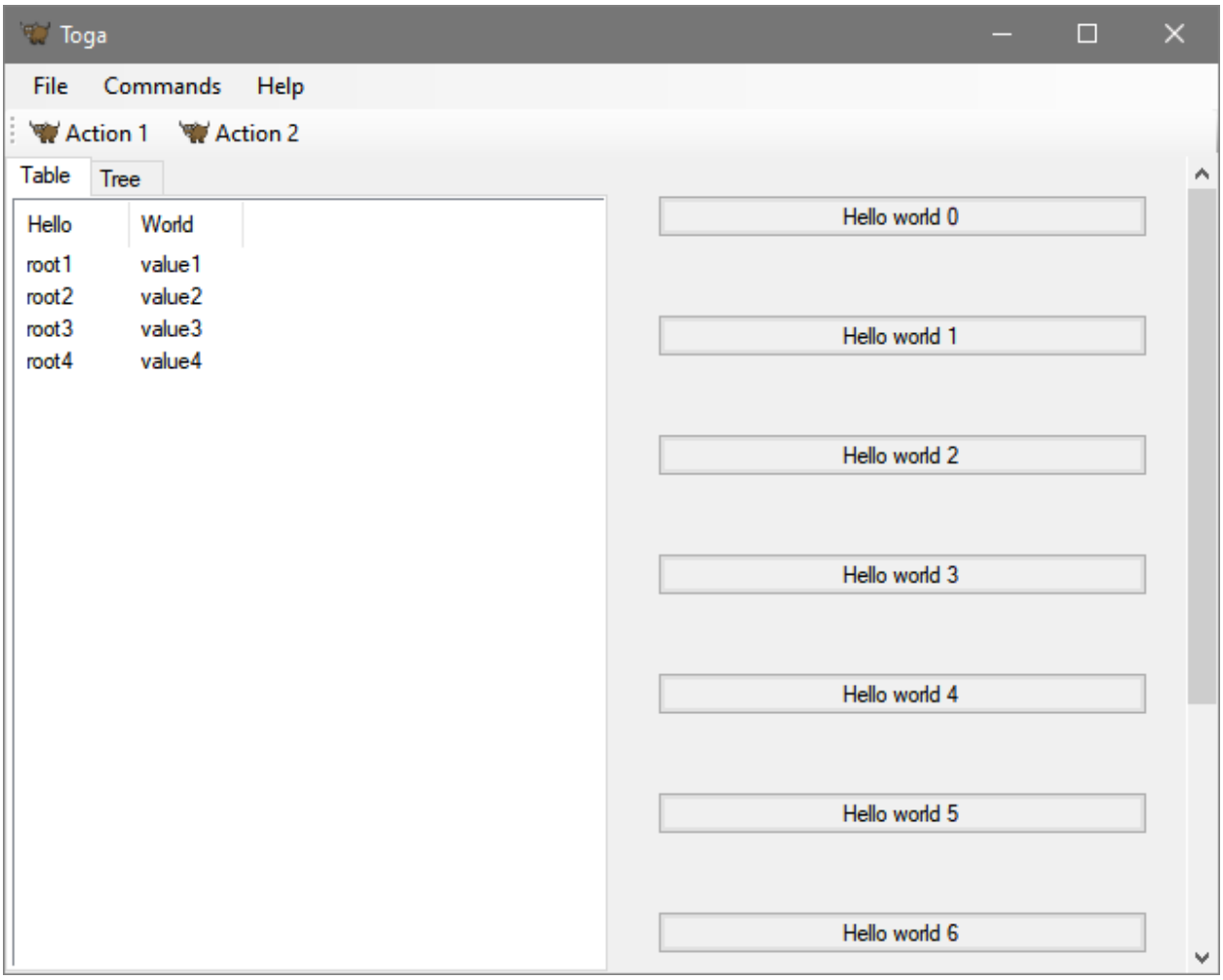


The backend for Linux platforms is named toga-gtk. It supports GTK 3.4 and later. It is installed automatically on Linux machines (machines that report `sys.platform == 'linux'`), or can be manually installed by invoking:

```
$ pip install toga-gtk
```

The GTK backend is reasonably well developed, but currently has some known issues with widget layout. It uses the native GObject Python bindings.

Windows



The backend for Windows is named toga-winforms. It supports Windows 10 with .NET 4 installed. It is installed automatically on Windows machines (machines that report `sys.platform == 'win32'`), or can be manually installed by invoking:

```
$ pip install toga-winforms
```

It uses Python.net. Unfortunately, python.net has not been packaged for Python 3.9 or higher, so you'll need to use Python 3.8 or earlier in your app.

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. It uses Rubicon to provide a bridge to native macOS libraries.

Android

The backend for Android is named toga-android. It can be manually installed by invoking:

```
$ pip install toga-android
```

The android backend is currently proof-of-concept only. Most widgets have not been implemented. It uses Chaquopy to provide a way to access the Android Java libraries and implement Java interfaces in Python.

Web

The Web backend is named toga-web. It can be manually installed by invoking:

```
$ pip install toga-web
```

The Web backend is currently proof-of-concept only. Most widgets have not been implemented. It uses PyScript to run Python code in the browser.

The Dummy platform

Toga also provides a Dummy platform - this is a backend that implements the full interface required by a platform backend, but does not display any widgets visually. It is intended for use in tests, and provides an API that can be used to verify widget operation.

Planned platform support

Eventually, the Toga project would like to provide support for the following platforms:

- UWP (Native Windows 8 and Windows mobile)
- Qt (for KDE based desktops)
- tvOS (for AppleTV devices)
- watchOS (for AppleWatch devices)
- Curses (for console)

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

Unofficial platform support

At present, there are no known unofficial platform backends.

2.3.2 Toga APIs by platform

Key

Partly supported: functionality or testing is incomplete
Fully supported

Core Components

Component	macOS	GTK	Windows	iOS	Android	Web
App						
Window						
MainWindow						

General Widgets

Component	macOS	GTK	Windows	iOS	Android	Web
ActivityIndicator						
Button						
Canvas						
DatePicker						
DetailedList						
Divider						
ImageView						
Label						
MultilineTextInput						
NumberInput						
PasswordInput						
ProgressBar						
Selection						
Slider						
Switch						
Table						
TextInput						
TimePicker						
Tree						
WebView						
Widget						

Layout Widgets

Component	macOS	GTK	Windows	iOS	Android	Web
Box						
ScrollContainer						
SplitContainer						
OptionContainer						

Resources

Component	macOS	GTK	Windows	iOS	Android	Web
Font						
Command						
Group						
Icon						
Image						

2.3.3 API Reference

Core application components

Component	Description
Application	The application itself
Window	Window object
MainWindow	Main Window

General widgets

Component	Description
ActivityIndicator	A (spinning) activity indicator
Button	Basic clickable Button
Canvas	Area you can draw on
DetailedList	A list of complex content
Divider	A horizontal or vertical line
ImageView	Image Viewer
Label	Text label
MultilineTextInput	Multi-line Text Input field
NumberInput	Number Input field
PasswordInput	A text input that hides it's input
ProgressBar	Progress Bar
Selection	Selection
Slider	Slider
Switch	Switch
Table	Table of data
TextInput	Text Input field
Tree	Tree of data
WebView	A panel for displaying HTML
Widget	The base widget

Layout widgets

Usage	Description
Box	Container for components
ScrollContainer	Scrollable Container
SplitContainer	Split Container
OptionContainer	Option Container

Resources

Component	Description
Font	Fonts
Command	Command
Group	Command group
Icon	An icon for buttons, menus, etc
Image	An image

Application

Table 5: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The app is the main entry point and container for the Toga GUI.

Usage

The app class is used by instantiating with a name, namespace and callback to a startup delegate which takes 1 argument of the app instance.

To start a UI loop, call *app.main_loop()*

```
import toga


def build(app):
    # build UI
    pass


if __name__ == '__main__':
    app = toga.App('First App', 'org.beeware.helloworld', startup=build)
    app.main_loop()
```

Alternatively, you can subclass App and implement the startup method

```
import toga


class MyApp(toga.App):
    def startup(self):
        # build UI
        pass


if __name__ == '__main__':
    app = MyApp('First App', 'org.beeware.helloworld')
    app.main_loop()
```

Reference

class toga.app.**App**(*formal_name=None*, *app_id=None*, *app_name=None*, *id=None*, *icon=None*, *author=None*, *version=None*, *home_page=None*, *description=None*, *startup=None*, *windows=None*, *on_exit=None*, *factory=None*)

The App is the top level of any GUI program. It is the manager of all the other bits of the GUI app: the main window and events that window generates like user input.

When you create an App you need to provide it a name, an id for uniqueness (by convention, the identifier is a reversed domain name.) and an optional startup function which should run once the App has initialized. The startup function typically constructs some initial user interface.

If the name and app_id are *not* provided, the application will attempt to find application metadata. This process will determine the module in which the App class is defined, and look for a `.dist-info` file matching that name.

Once the app is created you should invoke the main_loop() method, which will hand over execution of your program to Toga to make the App interface do its thing.

The absolute minimum App would be:

```
>>> app = toga.App(formal_name='Empty App', app_id='org.beeware.empty')
>>> app.main_loop()
```

Parameters

- **formal_name** – The formal name of the application. Will be derived from packaging metadata if not provided.
- **app_id** – The unique application identifier. This will usually be a reversed domain name, e.g. `org.beeware.myapp`. Will be derived from packaging metadata if not provided.
- **app_name** – The name of the Python module containing the app. Will be derived from the module defining the instance of the App class if not provided.
- **id** – The DOM identifier for the app (optional)
- **icon** – Identifier for the application's icon.
- **author** – The person or organization to be credited as the author of the application. Will be derived from application metadata if not provided.
- **version** – The version number of the app. Will be derived from packaging metadata if not provided.
- **home_page** – A URL for a home page for the app. Used in auto-generated help menu items. Will be derived from packaging metadata if not provided.
- **description** – A brief (one line) description of the app. Will be derived from packaging metadata if not provided.
- **startup** – The callback method before starting the app, typically to add the components. Must be a `callable` that expects a single argument of *App*.
- **windows** – An iterable with objects of *Window* that will be the app's secondary windows.

about()

Display the About dialog for the app.

Default implementation shows a platform-appropriate about dialog using app metadata. Override if you want to display a custom About dialog.

add_background_task(*handler*)

Schedule a task to run in the background.

Schedules a coroutine or a generator to run in the background. Control will be returned to the event loop during await or yield statements, respectively. Use this to run background tasks without blocking the GUI. If a regular callable is passed, it will be called as is and will block the GUI until the call returns.

Parameters
handler – A coroutine, generator or callable.

app = None

property app_id

The identifier for the app.

This is a reversed domain name, often used for targeting resources, etc.

Returns
The identifier as a str.

property app_name

The machine-readable, PEP508-compliant name of the app.

Returns
The machine-readable app name, as a str.

property author

The author of the app. This may be an organization name.

Returns
The author of the app, as a str.

property current_window

Return the currently active content window.

property description

A brief description of the app.

Returns
A brief description of the app, as a str.

exit()

Quit the application gracefully.

exit_full_screen()

Exit full screen mode.

property formal_name

The formal name of the app.

Returns
The formal name of the app, as a str.

hide_cursor()

Hide cursor from view.

property home_page

The URL of a web page for the app.

Returns
The URL of the app's home page, as a str.

property icon

The Icon for the app.

Returns
A `toga.Icon` instance for the app's icon.

property id

The DOM identifier for the app.

This id can be used to target CSS directives.

Returns
A DOM identifier for the app.

property is_full_screen

Is the app currently in full screen mode?

main_loop()

Invoke the application to handle user input.

This method typically only returns once the application is exiting.

property main_window

The main window for the app.

Returns
The main Window of the app.

property module_name

The module name for the app.

Returns
The module name for the app, as a `str`.

property name

The formal name of the app.

Returns
The formal name of the app, as a `str`.

property on_exit

The handler to invoke before the application exits.

Returns
The function `callable` that is called on application exit.

set_full_screen(**windows*)

Make one or more windows full screen.

Full screen is not the same as "maximized"; full screen mode is when all window borders and other chrome is no longer visible.

Parameters
windows – The list of windows to go full screen, in order of allocation to screens. If the number of windows exceeds the number of available displays, those windows will not be visible. If no windows are specified, the app will exit full screen mode.

show_cursor()

Show cursor.

startup()

Create and show the main window for the application.

property version

The version number of the app.

Returns
The version number of the app, as a `str`.

visit_homepage()

Open the application's homepage in the default browser.

If the application metadata doesn't define a homepage, this is a no-op.

property widgets

The widgets collection of the entire app.

Can be used to lookup widgets over the entire app through widget id or manually iterating through it. Example: `app.widgets["my_id"]`

Returns
The reference to the widgets collection of the entire app.

MainWindow

Table 6: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

A window for displaying components to the user

Usage

A MainWindow is used for desktop applications, where components need to be shown within a window-manager. Windows can be configured on instantiation and support displaying multiple widgets, toolbars and resizing.

```
import toga

window = toga.MainWindow('id-window', title='This is a window!')
window.show()
```

Reference

class toga.app.**MainWindow**(*id=None*, *title=None*, *position=(100, 100)*, *size=(640, 480)*, *toolbar=None*, *resizeable=True*, *minimizable=True*, *factory=None*, *on_close=None*)

property on_close

The handler to invoke before the window is closed.

Returns
The function `callable` that is called before the window is closed.

Window

Table 7: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

A window for displaying components to the user

Usage

The window class is used for desktop applications, where components need to be shown within a window-manager. Windows can be configured on instantiation and support displaying multiple widgets, toolbars and resizing.

```
import toga


class ExampleWindow(toga.App):
    def startup(self):
        self.label = toga.Label('Hello World')
        outer_box = toga.Box(
            children=[self.label]
        )
        self.window = toga.Window()
        self.window.content = outer_box

        self.window.show()


def main():
    return ExampleWindow('Window', 'org.beeware.window')


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

Reference

class toga.window.**Window**(*id=None*, *title=None*, *position=(100, 100)*, *size=(640, 480)*, *toolbar=None*, *resizeable=True*, *closeable=True*, *minimizable=True*, *factory=None*, *on_close=None*)

The top level container of an application.

Parameters

- **id** (*str*) – The ID of the window (optional).
- **title** (*str*) – Title for the window (optional).
- **position** (tuple of (int, int)) – Position of the window, as x,y coordinates.

- **size** (tuple of (int, int)) – Size of the window, as (width, height) sizes, in pixels.
- **toolbar** (list of *Widget*) – A list of widgets to add to a toolbar
- **resizeable** (*bool*) – Toggle if the window is resizable by the user, defaults to *True*.
- **closeable** (*bool*) – Toggle if the window is closable by the user, defaults to *True*.
- **minimizable** (*bool*) – Toggle if the window is minimizable by the user, defaults to *True*.
- **on_close** – A callback to invoke when the user makes a request to close the window.

property app

Instance of the *toga.app.App* that this window belongs to.

Returns
The app that it belongs to *toga.app.App*.

Raises
Exception – If the window already is associated with another app.

close()

confirm_dialog(*title*, *message*, *on_result=None*)

Ask the user to confirm if they wish to proceed with an action.

Presents as a dialog with 'Cancel' and 'OK' buttons (or whatever labels are appropriate on the current platform)

Parameters

- **title** – The title of the dialog window.
- **message** – A message describing the action to be confirmed.
- **on_result** – A callback that will be invoked when the user selects an option on the dialog.

Returns
An awaitable Dialog object. The Dialog object returns `True` when the 'OK' button was pressed, `False` when the 'CANCEL' button was pressed.

property content

Content of the window. On setting, the content is added to the same app as the window and to the same app.

Returns
A *Widget*

error_dialog(*title*, *message*, *on_result=None*)

Ask the user to acknowledge an error state.

Presents as an error dialog with a 'OK' button to close the dialog.

Parameters

- **title** – The title of the dialog window.
- **message** – The error message to display.
- **on_result** – A callback that will be invoked when the user selects an option on the dialog.

Returns
An awaitable Dialog object. The Dialog object returns `None` after the user pressed the 'OK' button.

property full_screen

hide()

Hide window, if shown.

property id

The DOM identifier for the window. This id can be used to target CSS directives.

Returns
The identifier as a str.

info_dialog(*title*, *message*, *on_result=None*)

Ask the user to acknowledge some information.

Presents as a dialog with a single 'OK' button to close the dialog.

Parameters

- **title** – The title of the dialog window.
- **message** – The message to display.
- **on_result** – A callback that will be invoked when the user selects an option on the dialog.

Returns
An awaitable Dialog object. The Dialog object returns None after the user pressed the 'OK' button.

property on_close

The handler to invoke before the window is closed.

Returns
The function callable that is called before the window is closed.

open_file_dialog(*title*, *initial_directory=None*, *file_types=None*, *multiselect=False*, *on_result=None*)

Ask the user to select a file (or files) to open.

Presents the user a system-native "Open file" dialog.

Parameters

- **title** – The title of the dialog window
- **initial_directory** – The initial folder in which to open the dialog. If None, use the default location provided by the operating system (which will often be "last used location")
- **file_types** – A list of strings with the allowed file extensions.
- **multiselect** – If True, the user will be able to select multiple files; if False, the selection will be restricted to a single file/
- **on_result** – A callback that will be invoked when the user selects an option on the dialog.

Returns
An awaitable Dialog object. The Dialog object returns a list of Path objects if multiselect is True, or a single Path otherwise. Returns None if the open operation is cancelled by the user.

property position

Position of the window, as x, y.

Returns
A tuple of (int, int) int the from (x, y).

`question_dialog`(*title*, *message*, *on_result=None*)

Ask the user a yes/no question.

Presents as a dialog with a 'YES' and 'NO' button.

Parameters

- `title` – The title of the dialog window.
- `message` – The question to be answered.
- `on_result` – A callback that will be invoked when the user selects an option on the dialog.

Returns

An awaitable Dialog object. The Dialog object returns `True` when the 'YES' button was pressed, `False` when the 'NO' button was pressed.

`save_file_dialog`(*title*, *suggested_filename*, *file_types=None*, *on_result=None*)

Prompt the user for a location to save a file.

Presents the user a system-native "Save file" dialog.

This opens a native dialog where the user can select a place to save a file. It is possible to suggest a filename and force the user to use a specific file extension. If no path is returned (e.g. dialog is canceled), a ValueError is raised.

Parameters

- `title` – The title of the dialog window
- `suggested_filename` – A default filename
- `file_types` – A list of strings with the allowed file extensions.
- `on_result` – A callback that will be invoked when the user selects an option on the dialog.

Returns

An awaitable Dialog object. The Dialog object returns a path object for the selected file location, or `None` if the user cancelled the save operation.

`select_folder_dialog`(*title*, *initial_directory=None*, *multiselect=False*, *on_result=None*)

Ask the user to select a directory/folder (or folders) to open.

Presents the user a system-native "Open folder" dialog.

Parameters

- `title` – The title of the dialog window
- `initial_directory` – The initial folder in which to open the dialog. If `None`, use the default location provided by the operating system (which will often be "last used location")
- `multiselect` – If True, the user will be able to select multiple files; if False, the selection will be restricted to a single file/
- `on_result` – A callback that will be invoked when the user selects an option on the dialog.

Returns

An awaitable Dialog object. The Dialog object returns a list of `Path` objects if multiselect is `True`, or a single `Path` otherwise. Returns `None` if the open operation is cancelled by the user.

`show`()

Show window, if hidden.

property size

Size of the window, as width, height.

Returns
A `tuple` of (`int`, `int`) where the first value is the width and the second it the height of the window.

stack_trace_dialog(*title*, *message*, *content*, *retry=False*, *on_result=None*)

Open a dialog that allows to display a large text body, such as a stack trace.

Parameters

- **title** – The title of the dialog window.
- **message** – Contextual information about the source of the stack trace.
- **content** – The stack trace, pre-formatted as a multi-line string.
- **retry** – A boolean; if True, the user will be given a "Retry" and "Quit" option; if False, a single option to acknowledge the error will be displayed.
- **on_result** – A callback that will be invoked when the user selects an option on the dialog.

Returns
An awaitable Dialog object. If retry is enabled, the Dialog object returns `True` if the user selected retry, and `False` otherwise; if retry is not enabled, the dialog object returns `None`.

property title

Title of the window. If no title is given it defaults to "Toga".

Returns
The current title of the window as a `str`.

property toolbar

Toolbar for the window.

Returns
A `list` of *Widget*

property visible

Containers

Box

A generic container for other widgets. Used to construct layouts.

Table 8: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

An empty Box can be constructed without any children, with children added to the box after construction:

```
import toga

box = toga.Box()

label1 = toga.Label('Hello')
label2 = toga.Label('World')

box.add(label1)
box.add(label2)
```

Alternatively, children can be specified at the time the box is constructed:

```
import toga

label1 = toga.Label('Hello')
label2 = toga.Label('World')

box = toga.Box(children=[label1, label2])
```

In most apps, a layout is constructed by building a tree of boxes inside boxes, with concrete widgets (such as *Label* or *Button*) forming the leaf nodes of the tree. Style directives can be applied to enforce padding around the outside of the box, direction of child stacking inside the box, and background color of the box.

Reference

class toga.widgets.box.**Box**(*id=None*, *style=None*, *children=None*)

Create a new Box container widget.

Inherits from *Widget*.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.
- **children** – An optional list of children for to add to the Box.

property enabled

Is the widget currently enabled? i.e., can the user interact with the widget?

Box widgets cannot be disabled; this property will always return True; any attempt to modify it will be ignored.

focus()

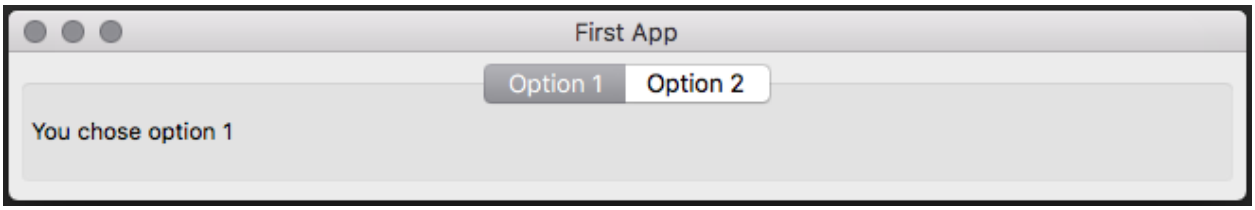
No-op; Box cannot accept input focus

Option Container

Table 9: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The Option Container widget is a user-selection control for choosing from a pre-configured list of controls, like a tab view.



Usage

```
import toga

container = toga.OptionContainer()

table = toga.Table(['Hello', 'World'])
tree = toga.Tree(['Navigate'])

container.add('Table', table)
container.add('Tree', tree)
```

Reference

class toga.widgets.optioncontainer.**OptionContainer**(*id=None*, *style=None*, *content=None*, *on_select=None*, *factory=None*)

The option container widget.

Parameters

- **id** (*str*) – An identifier for this widget.
- **style** (Style) – an optional style object. If no style is provided then a new one will be created for the widget.
- **content** (list of tuple (str, *Widget*)) – Each tuple in the list is composed of a title for the option and the widget tree that is displayed in the option.

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.

- **style** – A style object. If no style is provided, a default style will be applied to the widget.

exception OptionException

add(*text=NOT_PROVIDED*, *widget=NOT_PROVIDED*, *label=None*)

Add a new option to the option container.

Parameters

- **text** (*str*) – The text for the option.
- **widget** (*Widget*) – The widget to add to the option.

property app

The App to which this widget belongs.

When setting the app for a widget, all children of this widget will be recursively assigned to the same app.

Raises `ValueError` if the widget is already associated with another app.

property content

The sub layouts of the *OptionContainer*.

Returns

A OptionList `list` of `OptionItem`. Each element of the list is a sub layout of the *OptionContainer*

Raises

ValueError – If the list is less than two elements long.

property current_tab

insert(*index*, *text*, *widget*)

Insert a new option at the specified index.

Parameters

- **index** (*int*) – Index for the option.
- **text** (*str*) – The text for the option.
- **widget** (*Widget*) – The widget to add to the option.

property on_select

The callback function that is invoked when one of the options is selected.

Returns

(`Callable`) The callback function.

refresh_sublayouts()

Refresh the layout and appearance of this widget.

remove(*index*)

Remove the provided widgets as children of this node.

Any nominated child widget that is not a child of this widget will not have any change in parentage.

Raises `ValueError` if this widget cannot have children.

Parameters

children – The child nodes to remove.

property window

The window to which this widget belongs.

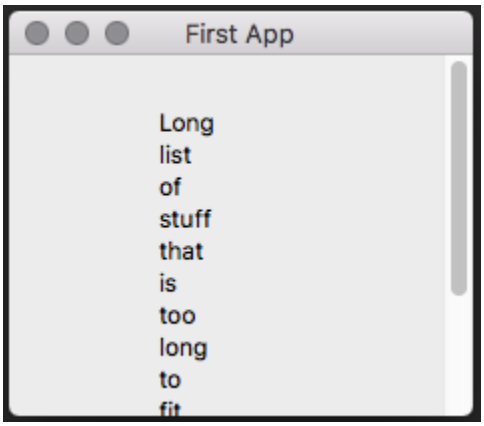
When setting the window for a widget, all children of this widget will be recursively assigned to the same window.

Scroll Container

Table 10: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The Scroll Container is similar to the `iframe` or scrollable `div` element in HTML, it contains an object with its own scrollable selection.



Usage

```
import toga

content = toga.WebView()

container = toga.ScrollContainer(content=content)
```

Scroll settings

Horizontal or vertical scroll can be set via the initializer or using the property.

```
import toga

content = toga.WebView()

container = toga.ScrollContainer(content=content, horizontal=False)

container.vertical = False
```

Reference

class toga.widgets.scrollcontainer.**ScrollContainer**(*id=None*, *style=None*, *horizontal=True*, *vertical=True*, *on_scroll=None*, *content=None*, *factory=None*)

Instantiate a new instance of the scrollable container widget.

Parameters

- **id** (*str*) – An identifier for this widget.
- **style** (Style) – An optional style object. If no style is provided then a new one will be created for the widget.
- **horizontal** (*bool*) – If True enable horizontal scroll bar.
- **vertical** (*bool*) – If True enable vertical scroll bar.
- **content** (*Widget*) – The content of the scroll window.

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

MIN_HEIGHT = 100

MIN_WIDTH = 100

property app

The App to which this widget belongs.

When setting the app for a widget, all children of this widget will be recursively assigned to the same app.

Raises ValueError if the widget is already associated with another app.

property content

Content of the scroll container.

Returns
The content of the widget (*Widget*).

property horizontal

Shows whether horizontal scrolling is enabled.

Returns
(bool) True if enabled, False if disabled.

property horizontal_position

property on_scroll

refresh_sublayouts()

Refresh the layout and appearance of this widget.

property vertical

Shows whether vertical scrolling is enabled.

Returns
(bool) True if enabled, False if disabled.

property vertical_position

property window

The window to which this widget belongs.

When setting the window for a widget, all children of this widget will be recursively assigned to the same window.

Split Container

Table 11: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The split container is a container with a movable split and the option for 2 or 3 elements.

Usage

```
import toga

split = toga.SplitContainer()
left_container = toga.Box()
right_container = toga.ScrollContainer()

split.content = [left_container, right_container]
```

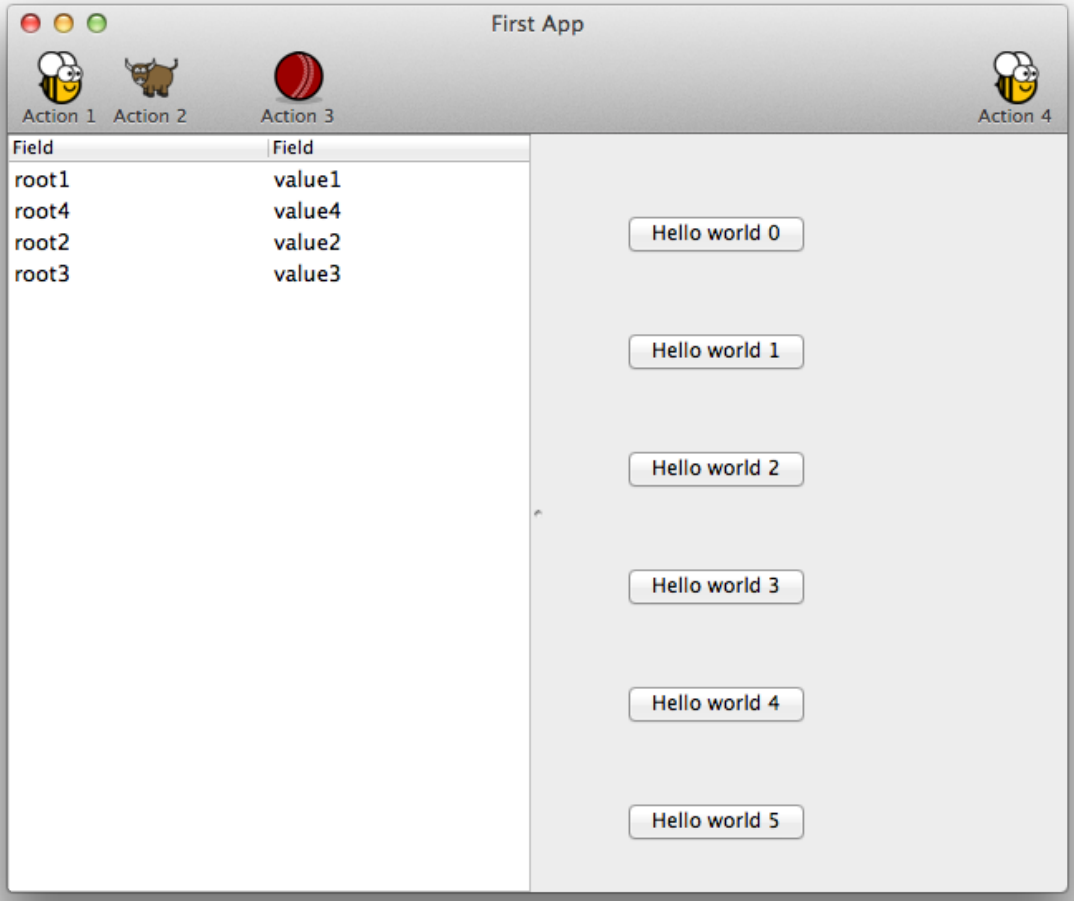
Setting split direction

Split direction is set on instantiation using the *direction* keyword argument. Direction is vertical by default.

```
import toga

split = toga.SplitContainer(direction=toga.SplitContainer.HORIZONTAL)
left_container = toga.Box()
right_container = toga.ScrollContainer()

split.content = [left_container, right_container]
```


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

Reference

class toga.widgets.splitcontainer.**SplitContainer**(*id=None*, *style=None*, *direction=VERTICAL*, *content=None*, *factory=None*)

A SplitContainer displays two widgets vertically or horizontally next to each other with a movable divider.

Parameters

- **id** (*str*) – An identifier for this widget.
- **style** (Style) – An optional style object. If no style is provided then a new one will be created for the widget.
- **direction** – The direction for the container split, either *SplitContainer.HORIZONTAL* or *SplitContainer.VERTICAL*
- **content** (list of *Widget*) – The list of components to be split or tuples of components to be split and adjusting parameters in the following order: widget (*Widget*): The widget that will be added. weight (float): Specifying the weighted splits. flex (boolean): Should the content expand when the widget is resized. (optional)

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

HORIZONTAL = False

VERTICAL = True

property app

The App to which this widget belongs.

When setting the app for a widget, all children of this widget will be recursively assigned to the same app.

Raises ValueError if the widget is already associated with another app.

property content

The sub layouts of the *SplitContainer*.

Returns
A list of *Widget*. Each element of the list is a sub layout of the *SplitContainer*

Raises
ValueError – If the list is less than two elements long.

property direction

The direction of the split.

Returns
True if *True* for vertical, *False* for horizontal.

refresh_sublayouts()

Refresh the layout and appearance of this widget.

property window

The window to which this widget belongs.

When setting the window for a widget, all children of this widget will be recursively assigned to the same window.

Resources

Font

Table 12: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The font class is used for abstracting the platforms implementation of fonts.

Reference

class toga.fonts.**Font**(*family*, *size*, *style=NORMAL*, *variant=NORMAL*, *weight=NORMAL*)

bind(*factory=None*)

measure(*text*, *dpi*, *tight=False*)

static register(*family*, *path*, *weight=NORMAL*, *style=NORMAL*, *variant=NORMAL*)

Registers a file-based font with its family name, style, variant and weight. When invalid values for style, variant or weight are passed, `NORMAL` will be used.

When a font file includes multiple font weight/style/etc., each variant must be registered separately:

```
# Register a simple regular font
Font.register("Font Awesome 5 Free Solid", "resources/Font Awesome 5 Free-Solid-
↪900.otf")

# Register a regular and bold font, contained in separate font files
Font.register("Roboto", "resources/Roboto-Regular.ttf")
Font.register("Roboto", "resources/Roboto-Bold.ttf", weight=Font.BOLD)

# Register a single font file that contains both a regular and bold weight
Font.register("Bahnschrift", "resources/Bahnschrift.ttf")
Font.register("Bahnschrift", "resources/Bahnschrift.ttf", weight=Font.BOLD)
```

Parameters

- **family** – The font family name. This is the name that can be referenced in style definitions.
- **path** – The path to the font file.
- **weight** – The font weight. Default value is `NORMAL`.
- **style** – The font style. Default value is `NORMAL`.
- **variant** – The font variant. Default value is `NORMAL`.

static registered_font_key(*family*, *weight*, *style*, *variant*)

Creates a key for storing a registered font in the font cache.

If weight, style or variant contain an invalid value, NORMAL is used instead.

Parameters

- **family** – The font family name. This is the name that can be referenced in style definitions.
- **weight** – The font weight. Default value is NORMAL.
- **style** – The font style. Default value is NORMAL.
- **variant** – The font variant. Default value is NORMAL.

Returns

The font key

Command

Table 13: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

Reference

class toga.command.**Command**(*action*, *text=NOT_PROVIDED*, *shortcut=None*, *tooltip=None*, *icon=None*, *group=None*, *section=None*, *order=None*, *enabled=True*, *factory=None*, *label=None*)

Parameters

- **action** – a function to invoke when the command is activated.
- **text** – caption for the command.
- **shortcut** – (optional) a key combination that can be used to invoke the command.
- **tooltip** – (optional) a short description for what the command will do.
- **icon** – (optional) a path to an icon resource to decorate the command.
- **group** – (optional) a Group object describing a collection of similar commands. If no group is specified, a default "Command" group will be used.
- **section** – (optional) an integer providing a sub-grouping. If no section is specified, the command will be allocated to section 0 within the group.
- **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 text within its section.
- **enabled** – whether to enable the command or not.

bind(*factory=None*)

property enabled

property icon

The Icon for the app.

Returns
A `toga.Icon` instance for the app's icon.

property key

A unique tuple describing the path to this command

property label

Command text.

DEPRECATED: renamed as text

Returns
The command text as a `str`

Group

Table 14: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

Reference

class `toga.command.`**Group**(*text=NOT_PROVIDED*, *order=None*, *section=None*, *parent=None*, *label=None*)

Parameters

- **text** –
- **order** –
- **parent** –

APP = <Group text=* order=0 parent=None>

COMMANDS = <Group text=Commands order=30 parent=None>

EDIT = <Group text=Edit order=10 parent=None>

FILE = <Group text=File order=1 parent=None>

HELP = <Group text=Help order=100 parent=None>

VIEW = <Group text=View order=20 parent=None>

WINDOW = <Group text=Window order=90 parent=None>

is_child_of(*parent*)

`is_parent_of`(*child*)

`property key`

A unique tuple describing the path to this group

`property label`

Group text.

DEPRECATED: renamed as text

Returns
The button text as a `str`

`property parent`

`property path`

A list containing the chain of groups that contain this group

`property root`

Icon

Table 15: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

An icon is a small, square image, used to decorate buttons and menu items.

A Toga icon is a **late bound** resource - that is, it can be constructed without an implementation. When it is assigned to an app, command, or other role where an icon is required, it is bound to a factory, at which time the implementation is created.

The filename specified for an icon is interpreted as a path relative to the module that defines your Toga application. The only exception to this is a system icon, which is relative to the toga core module itself.

An icon is **guaranteed** to have an implementation. If you specify a filename that cannot be found, Toga will output a warning to the console, and load a default icon.

When an icon file is specified, you can optionally omit the extension. If an extension is provided, that literal file will be loaded. If the platform backend cannot support icons of the format specified, the default icon will be used. If an extension is *not* provided, Toga will look for a file with the one of the platform's allowed extensions.

Reference

class toga.icons.**Icon**(*path*, *system=False*)

A representation of an Icon image.

Icon is a deferred resource - it's impl isn't available until it the icon is assigned to perform a role in an app. At the point at which the Icon is used, the Icon is bound to a factory, and the implementation is created.

Parameters

- **path** – The path to the icon file, relative to the application's module directory.
- **system** – Is this a system resource? Set to `True` if the icon is one of the Toga-provided icons. Default is False.

DEFAULT_ICON

TOGA_ICON

bind(*factory=None*)

Image

Table 16: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

An image is graphical content of arbitrary size.

A Toga icon is a **late bound** resource - that is, it can be constructed without an implementation. When it is assigned to an ImageView, or other role where an Image is required, it is bound to a factory, at which time the implementation is created.

The path specified for an Image can be:

1. A path relative to the module that defines your Toga application.
2. An absolute filesystem path
3. A URL. The content of the URL will be loaded in the background.

If the path specified does not exist, or cannot be loaded, a `FileNotFoundError` will be raised.

Usage

Reference

class toga.images.**Image**(*path=None*, *, *data=None*)

A representation of graphical content.

Parameters

- **path** – Path to the image. Allowed values can be local file (relative or absolute path) or URL (HTTP or HTTPS). Relative paths will be interpreted relative to the application module directory.

- **data** – A bytes object with the contents of an image in a supported format.

bind(*factory=None*)

save(*path*)

Save image to given path.

Parameters

path – Path where to save the image.

Widgets

Activity Indicator

A small animated indicator showing activity on a task of indeterminate length, usually rendered as a "spinner" animation.

Table 17: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

```
import toga

indicator = toga.ActivityIndicator()

# Start the animation
indicator.start()

# Stop the animation
indicator.stop()
```

Notes

- The ActivityIndicator will always take up a fixed amount of physical space in a layout. However, the widget will not be visible when it is in a "stopped" state.

Reference

class toga.widgets.activityindicator.**ActivityIndicator**(*id=None*, *style=None*, *running=False*)

Create a new ActivityIndicator widget.

Inherits from *Widget*.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.
- **running** – Describes whether the indicator is running at the time it is created.

property enabled

Is the widget currently enabled? i.e., can the user interact with the widget?

ActivityIndicator widgets cannot be disabled; this property will always return True; any attempt to modify it will be ignored.

focus()

No-op; ActivityIndicator cannot accept input focus

property is_running

Determine if the activity indicator is currently running.

Use `start()` and `stop()` to change the running state.

True if this activity indicator is running; False otherwise.

start()

Start the activity indicator.

If the activity indicator is already started, this is a no-op.

stop()

Stop the activity indicator.

If the activity indicator is already stopped, this is a no-op.

Button

A widget that can be pressed or clicked to cause an action in an application.

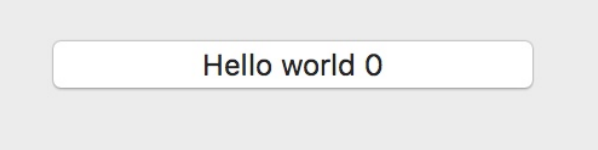


Table 18: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

A button has a text label. A handler can be associated with button press events.

```
import toga

def my_callback(button):
    # handle event
    pass

button = toga.Button("Click me", on_press=my_callback)
```

Notes

- A background color of `TRANSPARENT` will be treated as a reset of the button to the default system color.
- On macOS, the button text color cannot be set directly; any *color* style directive will be ignored. The text color is automatically selected by the platform to contrast with the background color of the button.

Reference

class toga.widgets.button.**Button**(*text*, *id=None*, *style=None*, *on_press=None*, *enabled=True*)

Create a new button widget.

Inherits from *Widget*.

Parameters

- **text** – The text to display on the button.
- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.
- **on_press** – A handler that will be invoked when the button is pressed.
- **enabled** – Is the button enabled (i.e., can it be pressed?). Optional; by default, buttons are created in an enabled state.

property on_press

The handler to invoke when the button is pressed.

property text

The text displayed on the button.

`None`, and the Unicode codepoint U+200B (ZERO WIDTH SPACE), will be interpreted and returned as an empty string. Any other object will be converted to a string using `str()`.

Only one line of text can be displayed. Any content after the first newline will be ignored.

Canvas

Table 19: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The canvas is used for creating a blank widget that you can draw on.

Usage

Simple usage to draw a black circle on the screen using the arc drawing object:

```
import toga
canvas = toga.Canvas(style=Pack(flex=1))
box = toga.Box(children=[canvas])
with canvas.fill() as fill:
    fill.arc(50, 50, 15)
```

More advanced usage for something like a vector drawing app where you would want to modify the parameters of the drawing objects. Here we draw a black circle and black rectangle. We then change the size of the circle, move the rectangle, and finally delete the rectangle.

```
import toga
canvas = toga.Canvas(style=Pack(flex=1))
box = toga.Box(children=[canvas])
with canvas.fill() as fill:
    arc1 = fill.arc(x=50, y=50, radius=15)
    rect1 = fill.rect(x=50, y=50, width=15, height=15)

arc1.x, arc1.y, arc1.radius = (25, 25, 5)
rect1.x = 75
fill.remove(rect1)
```

Use of drawing contexts, for example with a platformer game. Here you would want to modify the x/y coordinate of a drawing context that draws each character on the canvas. First, we create a hero context. Next, we create a black circle and a black outlined rectangle for the hero's body. Finally, we move the hero by 10 on the x-axis.

```
import toga
canvas = toga.Canvas(style=Pack(flex=1))
box = toga.Box(children=[canvas])
with canvas.context() as hero:
    with hero.fill() as body:
        body.arc(50, 50, 15)
    with hero.stroke() as outline:
        outline.rect(50, 50, 15, 15)

hero.translate(10, 0)
```

Reference

Main Interface

class toga.widgets.canvas.**Canvas**(*id=None*, *style=None*, *on_resize=None*, *on_press=None*, *on_release=None*, *on_drag=None*, *on_alt_press=None*, *on_alt_release=None*, *on_alt_drag=None*, *factory=None*)

Create new canvas.

Parameters

- **id** (*str*) – An identifier for this widget.
- **style** (Style) – An optional style object. If no style is provided then a new one will be created for the widget.
- **on_resize** (Callable) – Handler to invoke when the canvas is resized.
- **on_press** (Callable) – Handler to invoke when the primary (usually the left) button is pressed.
- **on_release** (Callable) – Handler to invoke when the primary (usually the left) button is released.
- **on_drag** (Callable) – Handler to invoke when cursor is dragged with the primary (usually the left) button pressed.
- **on_alt_press** (Callable) – Handler to invoke when the alternate (usually the right) button pressed.
- **on_alt_release** (Callable) – Handler to invoke when the alternate (usually the right) button released
- **on_alt_drag** (Callable) – Handler to invoke when the cursor is dragged with the alternate (usually the right) button pressed.

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

as_image()

measure_text(*text*, *font*, *tight=False*)

property on_alt_drag

Return the handler to invoke when the mouse is dragged while the alternate (usually the right) mouse button is pressed.

Returns

The handler that is invoked when the mouse is dragged with the alternate mouse button pressed.

property on_alt_press

Return the handler to invoke when the alternate (usually the right) mouse button is pressed.

Returns

The handler that is invoked when the alternate mouse button is pressed.

property on_alt_release

Return the handler to invoke when the alternate (usually the right) mouse button is released.

Returns
The handler that is invoked when the alternate mouse button is released.

property on_drag

Return the handler invoked when the mouse is dragged with the primary (usually the left) mouse button is pressed.

Returns
The handler that is invoked when the mouse is dragged with the primary button pressed.

property on_press

Return the handler invoked when the primary (usually the left) mouse button is pressed.

Returns
The handler that is invoked when the primary mouse button is pressed.

property on_release

Return the handler invoked when the primary (usually the left) mouse button is released.

Returns
The handler that is invoked when the primary mouse button is released.

property on_resize

The handler to invoke when the canvas is resized.

Returns
The handler that is invoked on canvas resize.

reset_transform()

Constructs and returns a *ResetTransform*.

Returns
ResetTransform object.

rotate(*radians*)

Constructs and returns a *Rotate*.

Parameters
radians (*float*) – The angle to rotate clockwise in radians.

Returns
Rotate object.

scale(*sx*, *sy*)

Constructs and returns a *Scale*.

Parameters

- **sx** (*float*) – scale factor for the X dimension.
- **sy** (*float*) – scale factor for the Y dimension.

Returns
Scale object.

translate(*tx*, *ty*)

Constructs and returns a *Translate*.

Parameters

- **tx** (*float*) – X value of coordinate.
- **ty** (*float*) – Y value of coordinate.

Returns
Translate object.

Lower-Level Classes

class toga.widgets.canvas.**Arc**(*x*, *y*, *radius*, *startangle=0.0*, *endangle=2 * pi*, *anticlockwise=False*)

A user-created *Arc* drawing object which adds an arc.

The arc is centered at `(x, y)` position with radius `r` starting at `startangle` and ending at `endangle` going in the given direction by anticlockwise (defaulting to clockwise).

Parameters

- **x** – The x coordinate of the arc's center.
- **y** – The y coordinate of the arc's center.
- **radius** – The arc's radius.
- **startangle** – The angle (in radians) at which the arc starts, measured clockwise from the positive x axis, default 0.0.
- **endangle** – The angle (in radians) at which the arc ends, measured clockwise from the positive x axis, default 2*pi.
- **anticlockwise** – If true, causes the arc to be drawn counter-clockwise between the two angles instead of clockwise, default false.

class toga.widgets.canvas.**BezierCurveTo**(*cp1x*, *cp1y*, *cp2x*, *cp2y*, *x*, *y*)

A user-created *BezierCurveTo* drawing object which adds a Bézier curve.

It requires three points. The first two points are control points and the third one is the end point. The starting point is the last point in the current path, which can be changed using move_to() before creating the Bézier curve.

Parameters

- **cp1x** (*float*) – x coordinate for the first control point.
- **cp1y** (*float*) – y coordinate for first control point.
- **cp2x** (*float*) – x coordinate for the second control point.
- **cp2y** (*float*) – y coordinate for the second control point.
- **x** (*float*) – x coordinate for the end point.
- **y** (*float*) – y coordinate for the end point.

class toga.widgets.canvas.**ClosedPath**(*x*, *y*)

A user-created *ClosedPath* drawing object for a closed path context.

Creates a new path and then closes it.

Parameters

- **x** (*float*) – The x axis of the beginning point.
- **y** (*float*) – The y axis of the beginning point.

class toga.widgets.canvas.**Context**(**args*, ***kwargs*)

The user-created *Context* drawing object to populate a drawing with visual context.

The top left corner of the canvas must be painted at the origin of the context and is sized using the rehint() method.

arc(*x*, *y*, *radius*, *startangle=0.0*, *endangle=2 * pi*, *anticlockwise=False*)

Constructs and returns a *Arc*.

Parameters

- **x** – The x coordinate of the arc's center.
- **y** – The y coordinate of the arc's center.
- **radius** – The arc's radius.
- **startangle** – The angle (in radians) at which the arc starts, measured clockwise from the positive x axis, default 0.0.
- **endangle** – The angle (in radians) at which the arc ends, measured clockwise from the positive x axis, default 2*pi.
- **anticlockwise** – If true, causes the arc to be drawn counter-clockwise between the two angles instead of clockwise, default false.

Returns
Arc object.

bezier_curve_to(*cp1x*, *cp1y*, *cp2x*, *cp2y*, *x*, *y*)

Constructs and returns a *BezierCurveTo*.

Parameters

- **cp1x** (*float*) – x coordinate for the first control point.
- **cp1y** (*float*) – y coordinate for first control point.
- **cp2x** (*float*) – x coordinate for the second control point.
- **cp2y** (*float*) – y coordinate for the second control point.
- **x** (*float*) – x coordinate for the end point.
- **y** (*float*) – y coordinate for the end point.

Returns
BezierCurveTo object.

property canvas

The canvas property of the current context.

Returns
The canvas node. Returns self if this node *is* the canvas node.

clear()

Remove all drawing objects.

closed_path(*x*, *y*)

Calls move_to(x,y) and then constructs and yields a *ClosedPath*.

Parameters

- **x** (*float*) – The x axis of the beginning point.
- **y** (*float*) – The y axis of the beginning point.

Yields
ClosedPath object.

context()

Constructs and returns a *Context*.

Makes use of an existing context. The top left corner of the canvas must be painted at the origin of the context and is sized using the rehint() method.

Yields
Context object.

ellipse(*x*, *y*, *radiusx*, *radiusy*, *rotation=0.0*, *startangle=0.0*, *endangle=2 * pi*, *anticlockwise=False*)

Constructs and returns a *Ellipse*.

Parameters

- **x** – The x axis of the coordinate for the ellipse's center.
- **y** – The y axis of the coordinate for the ellipse's center.
- **radiusx** – The ellipse's major-axis radius.
- **radiusy** – The ellipse's minor-axis radius.
- **rotation** – The rotation for this ellipse, expressed in radians, default 0.0.
- **startangle** – The starting point in radians, measured from the x axis, from which it will be drawn, default 0.0.
- **endangle** – The end ellipse's angle in radians to which it will e drawn, default 2*pi.
- **anticlockwise** – If true, draws the ellipse anticlockwise (counter-clockwise) instead of clockwise, default false.

Returns
Ellipse object.

fill(*color=BLACK*, *fill_rule=FillRule.NONZERO*, *preserve=False*)

Constructs and yields a *Fill*.

A drawing operator that fills the current path according to the current fill rule, (each sub-path is implicitly closed before being filled).

Parameters

- **fill_rule** (*str, Optional*) – 'nonzero' is the non-zero winding rule and 'evenodd' is the even-odd winding rule.
- **preserve** (*bool, Optional*) – Preserves the path within the Context.
- **color** (*str, Optional*) – color value in any valid color format, default to black.

Yields
Fill object.

line_to(*x*, *y*)

Constructs and returns a *LineTo*.

Parameters

- **x** (*float*) – The x axis of the coordinate for the end of the line.
- **y** (*float*) – The y axis of the coordinate for the end of the line.

Returns
LineTo object.

move_to(*x*, *y*)

Constructs and returns a *MoveTo*.

Parameters

- **x** (*float*) – The x axis of the point.
- **y** (*float*) – The y axis of the point.

Returns
MoveTo object.

new_path()

Constructs and returns a *NewPath*.

Returns
class: *NewPath* object.

quadratic_curve_to(*cpx*, *cpy*, *x*, *y*)

Constructs and returns a *QuadraticCurveTo*.

Parameters

- **cpx** (*float*) – The x axis of the coordinate for the control point.
- **cpy** (*float*) – The y axis of the coordinate for the control point.
- **x** (*float*) – The x axis of the coordinate for the end point.
- **y** (*float*) – The y axis of the coordinate for the end point.

Returns
QuadraticCurveTo object.

rect(*x*, *y*, *width*, *height*)

Constructs and returns a *Rect*.

Parameters

- **x** (*float*) – x coordinate for the rectangle starting point.
- **y** (*float*) – y coordinate for the rectangle starting point.
- **width** (*float*) – The rectangle's width.
- **height** (*float*) – The rectangle's width.

Returns
Rect object.

redraw()

Force a redraw of the Canvas.

The Canvas will be automatically redrawn after adding or remove a drawing object. If you modify a drawing object, this method is used to force a redraw.

remove(*drawing_object*)

Remove a drawing object.

Parameters
((*drawing_object*) – obj:'Drawing Object'): The drawing object to remove

stroke(*color=BLACK*, *line_width=2.0*, *line_dash=None*)

Constructs and yields a *Stroke*.

Parameters

- **color** (*str, Optional*) – color value in any valid color format, default to black.
- **line_width** (*float, Optional*) – stroke line width, default is 2.0.
- **line_dash** (*array of floats, Optional*) – stroke line dash pattern, default is None.

Yields
Stroke object.

write_text(*text*, *x=0*, *y=0*, *font=None*)

Constructs and returns a *WriteText*.

Writes a given text at the given (x,y) position. If no font is provided, then it will use the font assigned to the Canvas Widget, if it exists, or use the default font if there is no font assigned.

Parameters

- **text** (*str*) – The text to fill.
- **x** (*float, Optional*) – The x coordinate of the text. Default to 0.
- **y** (*float, Optional*) – The y coordinate of the text. Default to 0.
- **font** (*Font*, Optional) – The font to write with.

Returns
WriteText object.

class toga.widgets.canvas.**Ellipse**(*x*, *y*, *radiusx*, *radiusy*, *rotation=0.0*, *startangle=0.0*, *endangle=2 * pi*, *anticlockwise=False*)

A user-created *Ellipse* drawing object which adds an ellipse.

The ellipse is centered at `(x, y)` position with the radii `radiusx` and `radiusy` starting at `startangle` and ending at `endangle` going in the given direction by anticlockwise (defaulting to clockwise).

Parameters

- **x** – The x axis of the coordinate for the ellipse's center.
- **y** – The y axis of the coordinate for the ellipse's center.
- **radiusx** – The ellipse's major-axis radius.
- **radiusy** – The ellipse's minor-axis radius.
- **rotation** – The rotation for this ellipse, expressed in radians, default 0.0.
- **startangle** – The starting point in radians, measured from the x axis, from which it will be drawn, default 0.0.
- **endangle** – The end ellipse's angle in radians to which it will be drawn, default 2*pi.
- **anticlockwise** – If true, draws the ellipse anticlockwise (counter-clockwise) instead of clockwise, default false.

class toga.widgets.canvas.**Fill**(*color=BLACK*, *fill_rule=FillRule.NONZERO*, *preserve=False*)

A user-created *Fill* drawing object for a fill context.

A drawing object that fills the current path according to the current fill rule, (each sub-path is implicitly closed before being filled).

Parameters

- **color** (*str, Optional*) – Color value in any valid color format, default to black.
- **fill_rule** (*str, Optional*) – 'nonzero' if the non-zero winding rule and 'evenodd' if the even-odd winding rule.
- **preserve** (*bool, Optional*) – Preserves the path within the Context.

property color

property fill_rule

class toga.widgets.canvas.**FillRule**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

EVENODD = 0

NONZERO = 1

class toga.widgets.canvas.**LineTo**(*x*, *y*)

A user-created *LineTo* drawing object which draws a line to a point.

Connects the last point in the sub-path to the (x, y) coordinates with a straight line (but does not actually draw it).

Parameters

- **x** (*float*) – The x axis of the coordinate for the end of the line.
- **y** (*float*) – The y axis of the coordinate for the end of the line.

class toga.widgets.canvas.**MoveTo**(*x*, *y*)

A user-created *MoveTo* drawing object which moves the start of the next operation to a point.

Moves the starting point of a new sub-path to the (x, y) coordinates.

Parameters

- **x** (*float*) – The x axis of the point.
- **y** (*float*) – The y axis of the point.

class toga.widgets.canvas.**NewPath**

A user-created *NewPath* to add a new path.

class toga.widgets.canvas.**QuadraticCurveTo**(*cpx*, *cpy*, *x*, *y*)

A user-created *QuadraticCurveTo* drawing object which adds a quadratic curve.

It requires two points. The first point is a control point and the second one is the end point. The starting point is the last point in the current path, which can be changed using `moveTo()` before creating the quadratic Bézier curve.

Parameters

- **cpx** – The x axis of the coordinate for the control point.
- **cpy** – The y axis of the coordinate for the control point.
- **x** – The x axis of the coordinate for the end point.
- **y** – he y axis of the coordinate for the end point.

class toga.widgets.canvas.**Rect**(*x*, *y*, *width*, *height*)

A user-created *Rect* drawing object which adds a rectangle.

The rectangle is at position (x, y) with a size that is determined by width and height. Those four points are connected by straight lines and the sub-path is marked as closed, so that you can fill or stroke this rectangle.

Parameters

- **x** (*float*) – x coordinate for the rectangle starting point.
- **y** (*float*) – y coordinate for the rectangle starting point.
- **width** (*float*) – The rectangle's width.
- **height** (*float*) – The rectangle's width.

class toga.widgets.canvas.**ResetTransform**

A user-created *ResetTransform* to reset the canvas.

Resets the canvas by setting it equal to the canvas with no transformations.

class toga.widgets.canvas.**Rotate**(*radians*)

A user-created *Rotate* to add canvas rotation.

Modifies the canvas by rotating the canvas by angle radians. The rotation center point is always the canvas origin which is in the upper left of the canvas. To change the center point, move the canvas by using the translate() method.

Parameters

radians (*float*) – The angle to rotate clockwise in radians.

class toga.widgets.canvas.**Scale**(*sx*, *sy*)

A user-created *Scale* to add canvas scaling.

Modifies the canvas by scaling the X and Y canvas axes by sx and sy.

Parameters

- **sx** (*float*) – scale factor for the X dimension.
- **sy** (*float*) – scale factor for the Y dimension.

class toga.widgets.canvas.**Stroke**(*color=BLACK*, *line_width=2.0*, *line_dash=None*)

A user-created *Stroke* drawing object for a stroke context.

A drawing operator that strokes the current path according to the current line style settings.

Parameters

- **color** (*str, Optional*) – Color value in any valid color format, default to black.
- **line_width** (*float, Optional*) – Stroke line width, default is 2.0.
- **line_dash** (*array of floats, Optional*) – Stroke line dash pattern, default is None.

property color

class toga.widgets.canvas.**Translate**(*tx*, *ty*)

A user-created *Translate* to translate the canvas.

Modifies the canvas by translating the canvas origin by (tx, ty).

Parameters

- **tx** (*float*) – X value of coordinate.

- **ty** (*float*) – Y value of coordinate.

class toga.widgets.canvas.**WriteText**(*text*, *x*, *y*, *font*)

A user-created *WriteText* to add text.

Writes a given text at the given (x,y) position. If no font is provided, then it will use the font assigned to the Canvas Widget, if it exists, or use the default font if there is no font assigned.

Parameters

- **text** (*str*) – The text to fill.
- **x** (*float, Optional*) – The x coordinate of the text. Default to 0.
- **y** (*float, Optional*) – The y coordinate of the text. Default to 0.
- **font** (*toga.fonts.Font*, Optional) – The font to write with.

DetailedList

Table 20: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

Reference

class toga.widgets.detailedlist.**DetailedList**(*id=None*, *data=None*, *on_delete=None*, *on_refresh=None*, *on_select=None*, *style=None*, *factory=None*)

A widget to hold data in a list form. Rows are selectable and can be deleted. An updated function can be invoked by pulling the list down.

Parameters

- **id** (*str*) – An identifier for this widget.
- **data** (list of *dict*) – List of dictionaries with required 'icon', 'title', and 'subtitle' keys as well as optional custom keys to store additional info like 'pk' for a database primary key (think Django ORM)
- **on_delete** (Callable) – Function that is invoked on row deletion.
- **on_refresh** (Callable) – Function that is invoked on user initialized refresh.
- **on_select** (Callable) – Function that is invoked on row selection.
- **style** (Style) – An optional style object. If no style is provided then a new one will be created for the widget.

Examples

```
>>> import toga
>>> def selection_handler(widget, row):
>>>     print('Row {} of widget {} was selected.'.format(row, widget))
>>>
>>> dlist = toga.DetailedList(
...     data=[
...         {
...             'icon': '',
...             'title': 'John Doe',
...             'subtitle': 'Employee of the Month',
...             'pk': 100
...          }
...      ],
...      on_select=selection_handler
... )
```

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

MIN_HEIGHT = 100

MIN_WIDTH = 100

property data

The data source of the widget. It accepts data in the form of `list` of `dict` or `ListSource`

Returns

Returns a (`ListSource`).

property on_delete

The function invoked on row deletion. The delete handler must accept two arguments. The first is a ref. to the widget and the second the row that is about to be deleted.

Examples

```
>>> def delete_handler(widget, row):
>>>     print('row ', row, 'is going to be deleted from widget', widget)
```

Returns

The function that is invoked when deleting a row.

property on_refresh

Returns: The function to be invoked on user initialized refresh.

property on_select

The handler function must accept two arguments, widget and row.

Returns
The function to be invoked on selecting a row.

`scroll_to_bottom()`

Scroll the view so that the bottom of the list (last row) is visible.

`scroll_to_row(row)`

Scroll the view so that the specified row index is visible.

Parameters
row – The index of the row to make visible. Negative values refer to the nth last row (-1 is the last row, -2 second last, and so on)

`scroll_to_top()`

Scroll the view so that the top of the list (first row) is visible.

`property selection`

The current selection.

A value of None indicates no selection.

Divider

A separator used to visually distinguish two sections of content in a layout.

Table 21: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

To separate two labels stacked vertically with a horizontal line:

```
import toga
from toga.style import Pack, COLUMN

box = toga.Box(
    children=[
        toga.Label("First section"),
        toga.Divider(),
        toga.Label("Second section"),
    ],
    style=Pack(direction=COLUMN, flex=1, padding=10)
)
```

The direction (horizontal or vertical) can be given as an argument. If not specified, it will default to horizontal.

Reference

class toga.widgets.divider.**Divider**(*id=None*, *style=None*, *direction=HORIZONTAL*)

Create a new divider line.

Inherits from *Widget*.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.
- **direction** – The direction in which the divider will be drawn. Defaults to `Divider.HORIZONTAL`

HORIZONTAL = 0

VERTICAL = 1

property direction

The direction in which the visual separator will be drawn.

Returns

`Divider.HORIZONTAL` or `Divider.VERTICAL`

property enabled

Is the widget currently enabled? i.e., can the user interact with the widget?

Divider widgets cannot be disabled; this property will always return True; any attempt to modify it will be ignored.

focus()

No-op; Divider cannot accept input focus

Image View

Table 22: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The Image View is a container for an image to be rendered on the display

Usage

```
import toga

view = toga.ImageView(id='view1', image=my_image)
```

Reference

class toga.widgets.imageview.**ImageView**(*image=None*, *id=None*, *style=None*, *factory=None*)

Parameters

- **image** (*Image*) – The image to display.
- **id** (*str*) – An identifier for this widget.
- **style** (Style) –

Todo:

- Finish implementation.

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

property image

Label

A text label for annotating forms or interfaces.

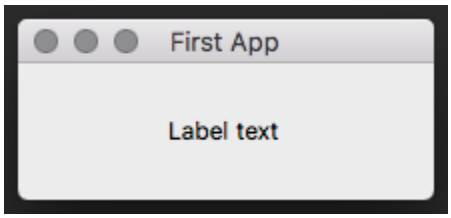


Table 23: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

```
import toga

label = toga.Label("Hello world")
```

Notes

- Winforms does not support an alignment value of `JUSTIFIED`. If this alignment value is used, the label will default to left alignment.

Reference

class toga.widgets.label.**Label**(*text*, *id=None*, *style=None*)

Create a new text label.

Inherits from *Widget*.

Parameters

- **text** – Text of the label.
- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

focus()

No-op; Label cannot accept input focus

property text

The text displayed by the label.

`None`, and the Unicode codepoint U+200B (ZERO WIDTH SPACE), will be interpreted and returned as an empty string. Any other object will be converted to a string using `str()`.

Multi-line text input

Table 24: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The Multi-line text input is similar to the text input but designed for larger inputs, similar to the <textarea> field of HTML.

Usage

```
import toga

textbox = toga.MultilineTextInput(id='view1')
```

Reference

class toga.widgets.multilinetextinput.**MultilineTextInput**(*id=None*, *style=None*, *factory=None*, *value=None*, *readonly=False*, *placeholder=None*, *on_change=None*, *initial=None*)

A multi-line text input widget.

Parameters

- **id** (*str*) – An identifier for this widget.
- **style** (Style) – An optional style object. If no style is provided then a new one will be created for the widget.
- **value** (*str*) – The initial text of the widget.
- **readonly** (*bool*) – Whether a user can write into the text input, defaults to *False*.
- **placeholder** (*str*) – The placeholder text for the widget.
- **on_change** (callable) – The handler to invoke when the text changes.

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

MIN_HEIGHT = 100

MIN_WIDTH = 100

clear()

Clears the text from the widget.

property on_change

The handler to invoke when the value changes.

Returns
The function callable that is called on a content change.

property placeholder

The placeholder text.

Returns
The placeholder text as a *str* `.

property readonly

Whether a user can write into the text input.

Returns
True if the user can only read, *False* if the user can read and write the text.

scroll_to_bottom()

Scroll the view to make the bottom of the text field visible.

scroll_to_top()

Scroll the view to make the top of the text field visible.

property value

The value of the multi line text input field.

Returns

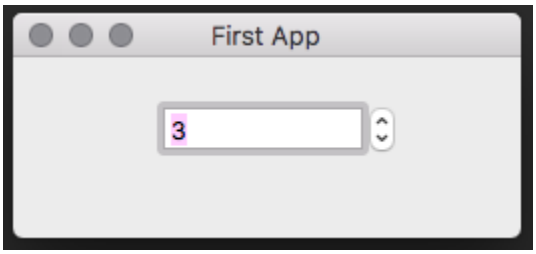
The text of the Widget as a `str`.

Number Input

Table 25: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The Number input is a text input box that is limited to numeric input.



Usage

```
import toga

textbox = toga.NumberInput(min_value=1, max_value=10)
```

Reference

class toga.widgets.numberinput.**NumberInput**(*id=None*, *style=None*, *factory=None*, *step=1*, *min_value=None*, *max_value=None*, *value=None*, *readonly=False*, *on_change=None*, *default=None*)

A *NumberInput* widget specifies a fixed range of possible numbers. The user has two buttons to increment/decrement the value by a step size. Step, min and max can be integers, floats, or Decimals; They can also be specified as strings, which will be converted to Decimals internally. The value of the widget will be evaluated as a Decimal.

Parameters

- **id** (*str*) – An identifier for this widget.
- **style** (`Style`) – an optional style object. If no style is provided then a new one will be created for the widget.
- **step** (*number*) – Step size of the adjustment buttons.

- **min_value** (*number*) – The minimum bound for the widget's value.
- **max_value** (*number*) – The maximum bound for the widget's value.
- **value** (*number*) – Initial value for the widget
- **readonly** (*bool*) – Whether a user can write/change the number input, defaults to *False*.
- **on_change** (`callable`) – The handler to invoke when the value changes.
- ****ex** –

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

property max_value

The maximum bound for the widget's value.

Returns

The maximum bound for the widget's value. If the maximum bound is None, there is no maximum bound.

property min_value

The minimum bound for the widget's value.

Returns

The minimum bound for the widget's value. If the minimum bound is None, there is no minimum bound.

property on_change

The handler to invoke when the value changes.

Returns

The function `callable` that is called on a content change.

property readonly

Whether a user can write/change the number input.

Returns

`True` if only read is possible. `False` if read and write is possible.

property step

The step value for the widget.

Returns

The current step value for the widget.

property value

Current value contained by the widget.

Returns

The current value(int) of the widget. Returns None if the field has no value set.

PasswordInput

Table 26: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

Reference

class toga.widgets.passwordinput.**PasswordInput**(*id=None*, *style=None*, *factory=None*, *value=None*, *placeholder=None*, *readonly=False*, *on_change=None*, *on_gain_focus=None*, *on_lose_focus=None*, *validators=None*, *initial=None*)

This widget behaves like a TextInput, but obscures the text that is entered by the user.

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

Progress Bar

Displays task progress on a horizontal graphical bar. The task being monitored can be of known or indeterminate length.

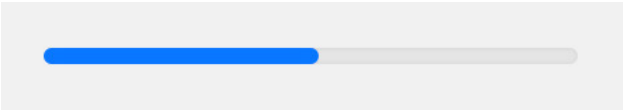


Table 27: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

If a progress bar has a `max` value, it is a *determinate* progress bar. The value of the progress bar can be altered over time, indicating progress on a task. The visual indicator of the progress bar will be filled indicating the proportion of `value` relative to `max`. `max` can be any positive numerical value.

```
import toga

progress = toga.ProgressBar(max=100, value=1)

# Start progress animation
progress.start()

# Update progress to 10%
progress.value = 10

# Stop progress animation
progress.stop()
```

If a progress bar does *not* have a `max` value (i.e., `max == None`), it is an *indeterminate* progress bar. Any change to the value of an indeterminate progress bar will be ignored. When started, an indeterminate progress bar animates as a throbbing or "ping pong" animation.

```
import toga

progress = toga.ProgressBar(max=None)

# Start progress animation
progress.start()

# Stop progress animation
progress.stop()
```

Notes

- The visual appearance of progress bars varies from platform to platform. Toga will try to provide a visual distinction between running and not-running determinate progress bars, but this cannot be guaranteed.

Reference

class toga.widgets.progressbar.**ProgressBar**(*id=None*, *style=None*, *max=1.0*, *value=0.0*, *running=False*)

Create a new Progress Bar widget.

Inherits from *Widget*.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.
- **max** – The value that represents completion of the task. Must be > 0.0; defaults to 1.0. A value of `None` indicates that the task length is indeterminate.

- **value** – The current progress against the maximum value. Must be between 0.0 and `max`; any value outside this range will be clipped. Defaults to 0.0.
- **running** – Describes whether the indicator is running at the time it is created. Default is False.

property enabled

Is the widget currently enabled? i.e., can the user interact with the widget?

ProgressBar widgets cannot be disabled; this property will always return True; any attempt to modify it will be ignored.

property is_determinate

Describe whether the progress bar has a known or indeterminate maximum.

True if the progress bar has determinate length; False otherwise.

property is_running

Describe if the activity indicator is currently running.

Use `start()` and `stop()` to change the running state.

True if this activity indicator is running; False otherwise.

property max

The value indicating completion of the task being monitored.

Must be a number > 0, or `None` for a task of indeterminate length.

start()

Start the progress bar.

If the progress bar is already started, this is a no-op.

stop()

Stop the progress bar.

If the progress bar is already stopped, this is a no-op.

property value

The current value of the progress indicator.

If the progress bar is determinate, the value must be between 0 and `max`. Any value outside this range will be clipped.

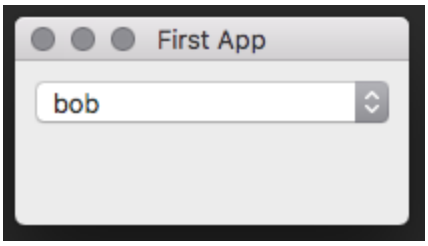
If the progress bar is indeterminate, changes in value will be ignored, and the current value will be returned as `None`.

Selection

Table 28: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The Selection widget is a simple control for allowing the user to choose between a list of string options.



Usage

```
import toga

container = toga.Selection(items=['bob', 'jim', 'lilly'])
```

Reference

class toga.widgets.selection.**Selection**(*id=None*, *style=None*, *items=None*, *on_select=None*, *enabled=True*, *factory=None*)

The Selection widget lets you pick from a defined selection of options.

Parameters

- **id** (*str*) – An identifier for this widget.
- **style** (Style) – An optional style object. If no style is provided then a new one will be created for the widget.
- **items** (list of str) – The items for the selection.

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

property items

The list of items.

Returns
The list of str of all selectable items.

property on_select

The callable function for when a node on the Tree is selected.

Return type
callable

property value

The value of the currently selected item.

Returns
The selected item as a str.

Slider

A widget for selecting a value within a range. The range is shown as a horizontal line, and the selected value is shown as a draggable marker.

Table 29: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

A slider can either be continuous (allowing any value within the range), or discrete (allowing a fixed number of equally-spaced values). For example:

```
import toga

def my_callback(slider):
    print(slider.value)

# Continuous slider, with an event handler.
toga.Slider(range=(-5, 10), value=7, on_change=my_callback)

# Discrete slider, accepting the values [0, 1.5, 3, 4.5, 6, 7.5].
toga.Slider(range=(0, 7.5), tick_count=6)
```

Reference

class toga.widgets.slider.**Slider**(*id=None*, *style=None*, *value=None*, *range=(0, 1)*, *tick_count=None*, *on_change=None*, *on_press=None*, *on_release=None*, *enabled=True*)

Create a new slider widget.

Inherits from *Widget*.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.
- **value** – Initial *value* of the slider. Defaults to the mid-point of the range.
- **range** – Initial *range* range of the slider. Defaults to `(0, 1)`.
- **tick_count** – Initial *tick_count* for the slider. If `None`, the slider will be continuous.
- **on_change** – Initial *on_change* handler.
- **on_press** – Initial *on_press* handler.
- **on_release** – Initial *on_release* handler.

- **enabled** – Whether the user can interact with the widget.

property max: float

Maximum allowed value.

This property is read-only, and depends on the value of *range*.

property min: float

Minimum allowed value.

This property is read-only, and depends on the value of *range*.

property on_change: Callable

Handler to invoke when the value of the slider is changed, either by the user or programmatically.

Setting the widget to its existing value will not call the handler.

property on_press: Callable

Handler to invoke when the user presses the slider before changing it.

property on_release: Callable

Handler to invoke when the user releases the slider after changing it.

property range: Tuple[float]

Range of allowed values, in the form (min, max).

If a range is set which doesn't include the current value, the value will be changed to the min or the max, whichever is closest.

Raises
ValueError – If the min is not strictly less than the max.

property tick_count: int | None

Number of tick marks to display on the slider.

- If this is `None`, the slider will be continuous.
- If this is an `int`, the slider will be discrete, and will have the given number of possible values, equally spaced within the *range*.

Setting this property to an `int` will round the current value to the nearest tick.

Raises
ValueError – If set to a count which is not at least 2 (for the min and max).

Note: On iOS, tick marks are not currently displayed, but discrete mode will otherwise work correctly.

---

property tick_step: float | None

Step between adjacent ticks.

- If the slider is continuous, this property returns `None`
- If the slider is discrete, it returns the difference in value between adjacent ticks.

This property is read-only, and depends on the values of *tick_count* and *range*.

property tick_value: int | None

Value of the slider, measured in ticks.

- If the slider is continuous, this property returns `None`.

- If the slider is discrete, it returns an integer between 1 (representing *min*) and *tick_count* (representing *max*).

Raises
ValueError – If set to anything inconsistent with the rules above.

property value: float

Current value.

If the slider is discrete, setting the value will round it to the nearest tick.

Raises
ValueError – If set to a value which is outside of the *range*.

Switch

A clickable button with two stable states: True (on, checked); and False (off, unchecked). The button has a text label.

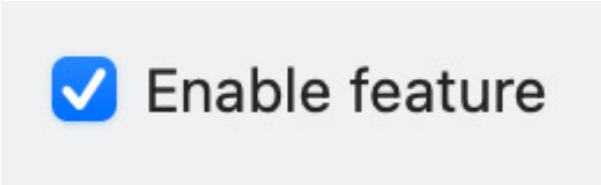


Table 30: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Usage

```
import toga

switch = toga.Switch()

# What is the current state of the switch?
print(f"The switch is {switch.value}")
```

Notes

- The button and the label are considered a single widget for layout purposes.
- The visual appearance of a Switch is not guaranteed. On some platforms, it will render as a checkbox. On others, it will render as a physical "switch" whose position (and color) indicates if the switch is active. When rendered as a checkbox, the label will appear to the right of the checkbox. When rendered as a switch, the label will be left-aligned, and the switch will be right-aligned.
- On macOS, the text color of the label cannot be set directly; any *color* style directive will be ignored.

Reference

class toga.widgets.switch.**Switch**(*text*, *id=None*, *style=None*, *on_change=None*, *value=False*, *enabled=True*)

Create a new Switch widget.

Inherits from *Widget*.

Parameters

- **text** – The text to display beside the switch.
- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.
- **on_change** – A handler that will be invoked when the switch changes value.
- **enabled** – Is the switch enabled (i.e., can it be pressed?). Optional; by default, switches are created in an enabled state.

property on_change

The handler to invoke when the value of the switch changes.

property text

The text label for the Switch.

`None`, and the Unicode codepoint U+200B (ZERO WIDTH SPACE), will be interpreted and returned as an empty string. Any other object will be converted to a string using `str()`.

Only one line of text can be displayed. Any content after the first newline will be ignored.

toggle()

Reverse the current value of the switch.

property value

The current state of the switch, as a Boolean.

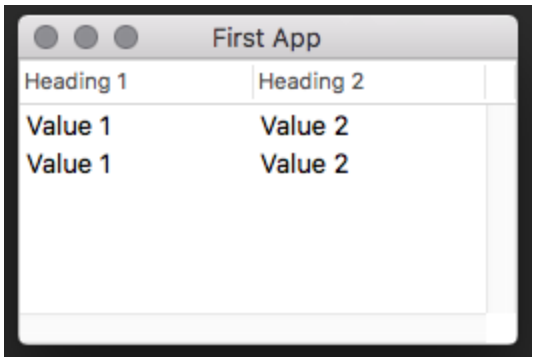
Any non-Boolean value will be converted to a Boolean.

Table

Table 31: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The table widget is a widget for displaying tabular data. It can be instantiated with the list of headings and then data rows can be added.



Usage

```
import toga

table = toga.Table(['Heading 1', 'Heading 2'])

# Append to end of table
table.data.append('Value 1', 'Value 2')

# Insert to row 2
table.data.insert(2, 'Value 1', 'Value 2')
```

Reference

class toga.widgets.table.**Table**(*headings*, *id=None*, *style=None*, *data=None*, *accessors=None*, *multiple_select=False*, *on_select=None*, *on_double_click=None*, *missing_value=None*, *factory=None*)

A Table Widget allows the display of data in the form of columns and rows.

Parameters

- **headings** (list of str) – The list of headings for the table.
- **id** (*str*) – An identifier for this widget.
- **data** (list of tuple) – The data to be displayed on the table.
- **accessors** – A list of methods, same length as headings, that describes how to extract the data value for each column from the row. (Optional)
- **style** (Style) – An optional style object. If no style is provided` then a new one will be created for the widget.
- **on_select** (callable) – A function to be invoked on selecting a row of the table.
- **on_double_click** (callable) – A function to be invoked on double clicking a row of the table.
- **missing_value** (str or None) – value for replacing a missing value in the data source. (Default: None). When 'None', a warning message will be shown.

Examples

```
>>> headings = ['Head 1', 'Head 2', 'Head 3']
>>> data = []
>>> table = Table(headings, data=data)
```

Data can be in several forms. A list of dictionaries, where the keys match the heading names:

```
>>> data = [{'head_1': 'value 1', 'head_2': 'value 2', 'head_3': 'value3'}),
>>>         {'head_1': 'value 1', 'head_2': 'value 2', 'head_3': 'value3'}]
```

A list of lists. These will be mapped to the headings in order:

```
>>> data = [('value 1', 'value 2', 'value3'),
>>>         ('value 1', 'value 2', 'value3')]
```

A list of values. This is only accepted if there is a single heading.

```
>>> data = ['item 1', 'item 2', 'item 3']
```

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

add_column(*heading*, *accessor=None*)

Add a new column to the table.

Parameters

- **heading** (`string`) – title of the column
- **accessor** – accessor of this new column

property data

The data source of the widget. It accepts table data in the form of `list`, `tuple`, or `ListSource`

Returns

Returns a (`ListSource`).

property missing_value

property multiple_select

Does the table allow multiple rows to be selected?

property on_double_click

The callback function that is invoked when a row of the table is double clicked. The provided callback function has to accept two arguments table (*Table*) and row (`Row` or `None`).

The value of a column of row can be accessed with row.accessor_name

Returns

(`callable`) The callback function.

property on_select

The callback function that is invoked when a row of the table is selected. The provided callback function has to accept two arguments table (*Table*) and row (Row or None).

The value of a column of row can be accessed with row.accessor_name

Returns
(callable) The callback function.

remove_column(*column*)

Remove a table column.

Parameters
column (int) – accessor or position (>0)

scroll_to_bottom()

Scroll the view so that the bottom of the list (last row) is visible.

scroll_to_row(*row*)

Scroll the view so that the specified row index is visible.

Parameters
row – The index of the row to make visible. Negative values refer to the nth last row (-1 is the last row, -2 second last, and so on)

scroll_to_top()

Scroll the view so that the top of the list (first row) is visible.

property selection

The current selection of the table.

A value of None indicates no selection. If the table allows multiple selection, returns a list of selected data nodes. Otherwise, returns a single data node.

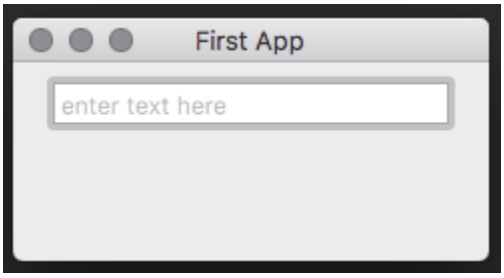
The value of a column of the selection can be accessed with selection.accessor_name (for single selection) and with selection[x].accessor_name (for multiple selection)

Text Input

Table 32: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The text input widget is a simple input field for user entry of text data.



Usage

```
import toga

input = toga.TextInput(placeholder='enter name here')
```

Reference

class toga.widgets.textinput.**TextInput**(*id=None*, *style=None*, *factory=None*, *value=None*, *placeholder=None*, *readonly=False*, *on_change=None*, *on_gain_focus=None*, *on_lose_focus=None*, *validators=None*, *initial=None*)

A widget get user input.

Parameters

- **id** (*str*) – An identifier for this widget.
- **style** (Style) – An optional style object. If no style is provided then a new one will be created for the widget.
- **value** (*str*) – The initial text for the input.
- **placeholder** (*str*) – If no input is present this text is shown.
- **readonly** (*bool*) – Whether a user can write into the text input, defaults to *False*.
- **on_change** (callable) – Method to be called when text is changed in text box
- **validators** (*list*) – list of validators to run on the value of the text box. Should return None is value is valid and an error message if not.
- **on_change** – The handler to invoke when the text changes.
- **on_gain_focus** (callable) – Function to execute when get focused.
- **on_lose_focus** (callable) – Function to execute when lose focus.

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

MIN_WIDTH = 100

clear()

Clears the text of the widget.

property is_valid

property on_change

The handler to invoke when the value changes.

Returns

The function callable that is called on a content change.

property on_gain_focus

The handler to invoke when the widget get focus.

Returns

The function `callable` that is called on widget focus gain.

property on_lose_focus

The handler to invoke when the widget lose focus.

Returns

The function `callable` that is called on widget focus loss.

property placeholder

The placeholder text.

Returns

The placeholder text as a `str`.

property readonly

Whether a user can write into the text input.

Returns

`True` if only read is possible. `False` if read and write is possible.

validate()

property validators

property value

The value of the text input field.

Returns

The current text of the widget as a `str`.

Tree

Table 33: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

The tree widget is still under development.

Usage

```
import toga

tree = toga.Tree(['Navigate'])

tree.insert(None, None, 'root1')

root2 = tree.insert(None, None, 'root2')
```

(continues on next page)

(continued from previous page)

```
tree.insert(root2, None, 'root2.1')
root2_2 = tree.insert(root2, None, 'root2.2')

tree.insert(root2_2, None, 'root2.2.1')
tree.insert(root2_2, None, 'root2.2.2')
tree.insert(root2_2, None, 'root2.2.3')
```

Reference

class toga.widgets.tree.**Tree**(*headings*, *id=None*, *style=None*, *data=None*, *accessors=None*, *multiple_select=False*, *on_select=None*, *on_double_click=None*, *factory=None*)

Tree Widget.

Parameters

- **headings** – The list of headings for the interface.
- **id** – An identifier for this widget.
- **style** – An optional style object. If no style is provided then a new one will be created for the widget.
- **data** – The data to display in the widget. Can be an instance of `TreeSource`, a list, dict or tuple with data to display in the tree widget, or a class instance which implements the interface of `TreeSource`. Entries can be:
  - any Python object `value` with a string representation. This string will be shown in the widget. If `value` has an attribute `icon`, instance of (*Icon*), the icon will be shown in front of the text.
  - a tuple `(icon, value)` where again the string representation of `value` will be used as text.
- **accessors** – Optional; a list of attributes to access the value in the columns. If not given, the headings will be taken.
- **multiple_select** – Boolean; if `True`, allows for the selection of multiple rows. Defaults to `False`.
- **on_select** – A handler to be invoked when the user selects one or multiple rows.
- **on_double_click** – A handler to be invoked when the user double clicks a row.

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

MIN_HEIGHT = 100

MIN_WIDTH = 100

property data

The data source of the tree

Type
returns

property multiple_select

Does the table allow multiple rows to be selected?

property on_double_click

The callable function for when a node on the Tree is selected. The provided callback function has to accept two arguments tree (*Tree*) and node (`Node` or `None`).

Return type
`callable`

property on_select

The callable function for when a node on the Tree is selected. The provided callback function has to accept two arguments tree (*Tree*) and node (`Node` or `None`).

Return type
`callable`

property selection

The current selection of the table.

A value of None indicates no selection. If the tree allows multiple selection, returns a list of selected data nodes. Otherwise, returns a single data node.

WebView

Table 34: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

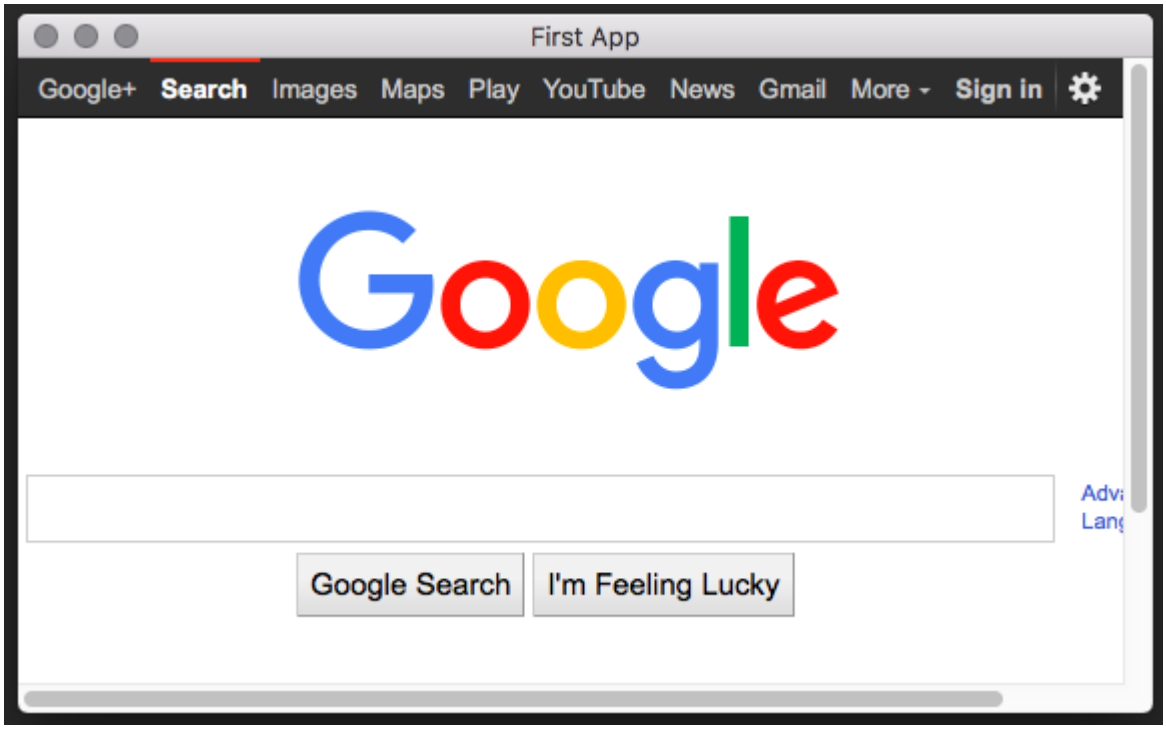
The Web View widget is used for displaying an embedded browser window within an application.

Both sites served by a web server and local content can be displayed. Due to security restrictions in the macOS backend WKWebView, local content on macOS can only be loaded from a single directory, relative to the base URL, and not from an absolute "file://" URL. As a workaround, it is possible to use a lightweight webserver instead.

Usage

```
import toga

web = toga.WebView(url='https://google.com')
```



Debugging

If you need to debug the HTML, JavaScript or CSS content of a view, you may want to use the "inspect element" feature of the WebView. This is not be turned on by default on some platforms. To enable WebView debugging:

- macOS

  Run the following at the terminal:

  ```
  $ defaults write com.example.appname WebKitDeveloperExtras -bool true
  ```

  substituting *com.example.appname* with the bundle ID for your app.

Reference

class toga.widgets.webview.**WebView**(*id=None*, *style=None*, *factory=None*, *url=None*, *user_agent=None*, *on_key_down=None*, *on_webview_load=None*)

A widget to display and open html content.

Parameters

- **id** (str) – An identifier for this widget.
- **style** (Style) – An optional style object. If no style is provided then a new one will be created for the widget.
- **url** (str) – The URL to start with.
- **user_agent** (str) – The user agent for the web view.
- **on_key_down** (callable) – The callback method for when a key is pressed within the web view

- **on_webview_load** (callable) – The callback method for when the webview loads (or reloads).

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

property dom

The current DOM.

Returns
The current DOM

Return type
str

async evaluate_javascript(*javascript*)

Evaluate a JavaScript expression, returning the result.

This is an asynchronous operation. The method will complete when the return value is available.

Parameters
javascript (str) – The javascript expression to evaluate.

invoke_javascript(*javascript*)

Invoke a JavaScript expression.

The result (if any) of the javascript is ignored.

No guarantee is provided that the javascript has completed execution when `invoke()` returns

Parameters
javascript (str) – The javascript expression to evaluate.

property on_key_down

The handler to invoke when the button is pressed.

Returns
The function that is called on button press.

Return type
callable

property on_webview_load

The handler to invoke when the webview finishes loading.

Returns
The function that is called when the webview finishes loading.

Return type
callable

set_content(*root_url*, *content*)

Set the content of the web view.

Parameters

- **root_url** (str) – The URL

- **content** (str) – The new content

property url

The current URL.

Returns
The current URL

Return type
str

property user_agent

The user agent for the web view as a str.

Returns
The user agent

Return type
str

Widget

The abstract base class of all widgets. This class should not be be instantiated directly.

Table 35: Availability (Key)

macOS	GTK	Windows	iOS	Android	Web

Reference

class toga.widgets.base.**Widget**(*id=None*, *style=None*)

Create a base Toga widget.

This is an abstract base class; it cannot be instantiated.

Parameters

- **id** – The ID for the widget.
- **style** – A style object. If no style is provided, a default style will be applied to the widget.

add(**children*)

Add the provided widgets as children of this widget.

If a child widget already has a parent, it will be re-parented as a child of this widget. If the child widget is already a child of this widget, there is no change.

Raises ValueError if this widget cannot have children.

Parameters
children – The widgets to add as children of this widget.

property app

The App to which this widget belongs.

When setting the app for a widget, all children of this widget will be recursively assigned to the same app.

Raises ValueError if the widget is already associated with another app.

property can_have_children

Determine if the node can have children.

This does not resolve whether there actually *are* any children; it only confirms whether children are theoretically allowed.

property children

The children of this node. This *always* returns a list, even if the node is a leaf and cannot have children.

Returns
A list of the children for this widget.

clear()

Clear all children from this node.

Raises
ValueError – If this node is a leaf, and cannot have children.

property enabled

Is the widget currently enabled? i.e., can the user interact with the widget?

focus()

Give this widget the input focus.

This method is a no-op if the widget can't accept focus. The ability of a widget to accept focus is platform-dependent. In general, on desktop platforms you can focus any widget that can accept user input, while on mobile platforms focus is limited to widgets that accept text input (i.e., widgets that cause the virtual keyboard to appear).

property id

The unique identifier for the widget.

insert(*index*, *child*)

Insert a widget as a child of this widget.

If a child widget already has a parent, it will be re-parented as a child of this widget. If the child widget is already a child of this widget, there is no change.

Raises `ValueError` if this node cannot have children.

Parameters

- **index** – The position in the list of children where the new widget should be added.
- **child** – The child to insert as a child of this node.

property parent

The parent of this node.

Returns
The parent of this node. Returns None if this node is the root node.

refresh()

Refresh the layout and appearance of the tree this node is contained in.

refresh_sublayouts()

remove(**children*)

Remove the provided widgets as children of this node.

Any nominated child widget that is not a child of this widget will not have any change in parentage.

Raises `ValueError` if this widget cannot have children.

Parameters
children – The child nodes to remove.

property root

The root of the tree containing this node.

Returns
The root node. Returns self if this node *is* the root node.

property tab_index

The position of the widget in the focus chain for the window.

Note: This is a beta feature. The `tab_index` API may change in future.

property window

The window to which this widget belongs.

When setting the window for a widget, all children of this widget will be recursively assigned to the same window.

2.3.4 Style

The Pack Style Engine

Toga's default style engine, **Pack**, is a layout algorithm based around the idea of packing boxes inside boxes. Each box specifies a direction for its children, and each child specifies how it will consume the available space - either as a specific width, or as a proportion of the available width. Other properties exist to control color, text alignment and so on.

It is similar in some ways to the CSS Flexbox algorithm; but dramatically simplified, as there is no allowance for overflowing boxes.

Note: The string values defined here are the string literals that the Pack algorithm accepts. These values are also pre-defined as Python constants in the `toga.style.pack` module with the same name; however, following Python style, the constants use upper case. For example, the Python constant `toga.style.pack.COLUMN` evaluates as the string literal `"column"`.

Pack style properties

display

Values: `pack` | `none`

Initial value: `pack`

Used to define the how to display the element. A value of `pack` will apply the pack layout algorithm to this node and its descendants. A value of `none` removes the element from the layout entirely. Space will be allocated for the element as if it were there, but the element itself will not be visible.

visibility

Values: hidden | visible

Initial value: visible

Used to define whether the element should be drawn. A value of visible means the element will be displayed. A value of hidden removes the element from view, but allocates space for the element as if it were still in the layout.

Any children of a hidden element are implicitly removed from view.

If a previously hidden element is made visible, any children of the element with a visibility of hidden will remain hidden. Any descendants of the hidden child will also remain hidden, regardless of their visibility.

direction

Values: row | column

Initial value: row

The packing direction for children of the box. A value of column indicates children will be stacked vertically, from top to bottom. A value of row indicates children will be packed horizontally; left-to-right if text_direction is ltr, or right-to-left if text_direction is rtl.

alignment

Values: top | bottom | left | right | center

Initial value: top if direction is row; left if direction is column

The alignment of children relative to the outside of the packed box.

If the box is a column box, only the values left, right and center are honored.

If the box is a row box, only the values top, bottom and center are honored.

If a value is provided, but the value isn't honored, the alignment reverts to the default for the direction.

width

Values: <integer> | none

Initial value: none

Specify a fixed width for the box.

The final width for the box may be larger, if the children of the box cannot fit inside the specified space.

height

Values: <integer>|none

Initial value: none

Specify a fixed height for the box.

The final height for the box may be larger, if the children of the box cannot fit inside the specified space.

flex

Values: <number>

Initial value: 0

A weighting that is used to compare this box with its siblings when allocating remaining space in a box.

Once fixed space allocations have been performed, this box will assume `flex / (sum of all flex for all siblings)` of all remaining available space in the direction of the parent's layout.

padding_top

padding_right

padding_bottom

padding_left

Values: <integer>

Initial value: 0

The amount of space to allocate between the edge of the box, and the edge of content in the box, on the top, right, bottom and left sides, respectively.

padding

Values: <integer> or <tuple> of length 1-4

A shorthand for setting the top, right, bottom and left padding with a single declaration.

If 1 integer is provided, that value will be used as the padding for all sides.

If 2 integers are provided, the first value will be used as the padding for the top and bottom; the second will be used as the value for the left and right.

If 3 integers are provided, the first value will be used as the top padding, the second for the left and right padding, and the third for the bottom padding.

If 4 integers are provided, they will be used as the top, right, bottom and left padding, respectively.

color

Values: <color>

Initial value: System default

Set the foreground color for the object being rendered.

Some objects may not use the value.

background_color

Values: <color> | transparent

Initial value: The platform default background color

Set the background color for the object being rendered.

Some objects may not use the value.

text_align

Values: left | right | center | justify

Initial value: left if text_direction is ltr; right if text_direction is rtl

Defines the alignment of text in the object being rendered.

text_direction

Values: rtl | ltr

Initial value: rtl

Defines the natural direction of horizontal content.

font_family

Values: system | serif | sans-serif | cursive | fantasy | monospace | <string>

Initial value: system

The font family to be used.

A value of system indicates that whatever is a system-appropriate font should be used.

A value of serif, sans-serif, cursive, fantasy, or monospace will use a system defined font that matches the description (e.g.,"Times New Roman" for serif, "Courier New" for monospace).

Otherwise, any font name can be specified. If the font name cannot be resolved, the system font will be used.

font_style

Values: `normal` | `italic` | `oblique`

Initial value: `normal`

The style of the font to be used.

font_variant

Values: `normal` | `small_caps`

Initial value: `normal`

The variant of the font to be used.

font_weight

Values: `normal` | `bold`

Initial value: `normal`

The weight of the font to be used.

font_size

Values: `<integer>`

Initial value: System default

The relationship between Pack and CSS

Pack aims to be a functional subset of CSS. Any Pack layout can be converted into an equivalent CSS layout. After applying this conversion, the CSS layout should be considered a "reference implementation". Any disagreement between the rendering of a converted Pack layout in a browser, and the layout produced by the Toga implementation of Pack should be considered to be either a bug in Toga, or a bug in the mapping.

The mapping that can be used to establish the reference implementation is:

- The reference HTML layout document is rendered in no-quirks mode, with a default CSS stylesheet:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="UTF-8" />
      <title>Pack layout testbed</title>
      <style>
         html, body {
            height: 100%;
         }
         body {
            overflow: hidden;
            display: flex;
```

(continues on next page)

(continued from previous page)

```
            margin: 0;
            white-space: pre;
        }
        div {
            display: flex;
            white-space: pre;
        }
    </style>
  </head>
  <body></body>
</html>
```

- The root element of the Pack layout can be mapped to the <body> element of the HTML reference document. The rendering area of the browser window becomes the view area that Pack will fill.
- All other elements in the DOM tree are mapped to elements.
- The following Pack declarations can be mapped to equivalent CSS declarations:

Pack property	CSS property
alignment: top	align-items: start if direction == row; otherwise ignored.
alignment: bottom	align-items: end if direction == row; otherwise ignored.
alignment: left	align-items: start if direction == column; otherwise ignored.
alignment: right	align-items: end if direction == column; otherwise ignored.
alignment: center	align-items: center
direction: <str>	flex-direction: <str>
display: pack	display: flex
flex: <int>	If direction = row and width is set, or direction = column and height is set, ignore. Otherwise, flex: <int> 0 0.
font_size: <int>	font-size: <int>pt
height: <int>	height: <int>px
padding_top: <int>	margin-top: <int>px
padding_bottom <int>	margin-bottom: <int>px
padding_left: <int>	margin-left: <int>px
padding_right: <int>	margin-right: <int>px
text_direction <str>	direction: <str>
width: <int>	width: <int>px

- All other Pack declarations should be used as-is as CSS declarations, with underscores being converted to dashes (e.g., `background_color` becomes `background-color`).

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 a macOS 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 macOS - 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, macOS 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 neighbor's 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 BeeWare 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 Success Stories

Want to see examples of Toga in use? Here's some:

- Travel Tips is an app in the iOS App Store that uses Toga to describe it's user interface.
- Eddington is a data fitting tool based on *Toga* and *Briefcase*
- taRpnCalcTG is a Toga based calculator for Android, Windows and MacOS which is extensible with Python scripts.
- pyPlayground is a Toga based app for Android and Windows which can be modified to try Toga without additional tool chain.
- taAppLister is a Toga based Android app for listing and exporting all installed apps.

- RemoteCommand is a Toga based app for synchronizing the clipboard between Windows and MacOS.

2.4.4 Release History

0.3.1 (2023-04-12)

Features

- The Button widget now has 100% test coverage, and complete API documentation. (#1761)
- The mapping between Pack layout and HTML/CSS has been formalized. (#1778)
- The Label widget now has 100% test coverage, and complete API documentation. (#1799)
- TextInput now supports focus handlers and changing alignment on GTK. (#1817)
- The ActivityIndicator widget now has 100% test coverage, and complete API documentation. (#1819)
- The Box widget now has 100% test coverage, and complete API documentation. (#1820)
- NumberInput now supports changing alignment on GTK. (#1821)
- The Divider widget now has 100% test coverage, and complete API documentation. (#1823)
- The ProgressBar widget now has 100% test coverage, and complete API documentation. (#1825)
- The Switch widget now has 100% test coverage, and complete API documentation. (#1832)
- Event handlers have been internally modified to simplify their definition and use on backends. (#1833)
- The base Toga Widget now has 100% test coverage, and complete API documentation. (#1834)
- Support for FreeBSD was added. (#1836)
- The Web backend now uses Shoelace to provide web components. (#1838)
- Winforms apps can now go full screen. (#1863)

Bugfixes

- Issues with reducing the size of windows on GTK have been resolved. (#1205)
- iOS now supports newlines in Labels. (#1501)
- The Slider widget now has 100% test coverage, and complete API documentation. (#1708)
- The GTK backend no longer raises a warning about the use of a deprecated `set_wmclass` API. (#1718)
- MultilineTextInput now correctly adapts to Dark Mode on macOS. (#1783)
- The handling of GTK layouts has been modified to reduce the frequency and increase the accuracy of layout results. (#1794)
- The text alignment of MultilineTextInput on Android has been fixed to be TOP aligned. (#1808)
- GTK widgets that involve animation (such as Switch or ProgressBar) are now redrawn correctly. (#1826)

Improved Documentation

- API support tables now distinguish partial vs full support on each platform. (#1762)
- Some missing settings and constant values were added to the documentation of Pack. (#1786)
- Added documentation for `toga.App.widgets`. (#1852)

Misc

- #1750, #1764, #1765, #1766, #1770, #1771, #1777, #1797, #1802, #1813, #1818, #1822, #1829, #1830, #1835, #1839, #1854, #1861

0.3.0 (2023-01-30)

Features

- Widgets now use a three-layered (Interface/Implementation/Native) structure.
- A GUI testing framework was added.
- A simplified "Pack" layout algorithm was added.
- Added a web backend.

Bugfixes

- Too many to count!

0.2.15

- Added more widgets and cross-platform support, especially for GTK+ and Winforms

0.2.14

- Removed use of `namedtuple`

0.2.13

- Various fixes in preparation for PyCon AU demo

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

2.4.5 Road Map

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

Widgets

The core of Toga is its 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 text field that provides options (e.g., text with past choices)
  - Cocoa: `NSComboBox`
  - GTK: `Gtk.ComboBox.new_with_model_and_entry`
  - iOS: ?
  - Winforms: `ComboBox`
  - Android: `Spinner`
- `DateTimeInput` - A widget for selecting a date and a time.
  - Cocoa: `NSDatePicker`
  - GTK: *Gtk.Calendar* + ?
  - iOS: `UIDatePicker`
  - Winforms: `DateTimePicker`
  - Android: ?
- `ColorInput` - A widget for selecting a color
  - Cocoa: `NSColorWell`
  - GTK: `Gtk.ColorButton` or `Gtk.ColorSelection`
  - iOS: ?
  - Winforms: ?
  - Android: ?
- `SearchInput` - A variant of `TextField` that is decorated as a search box.
  - Cocoa: `NSSearchField`
  - GTK: `Gtk.Entry`
  - iOS: `UISearchBar`?
  - Winforms: ?
  - Android: ?

Views

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

- `VideoView` - Display a video
  - Cocoa: `AVPlayerView`
  - GTK: Custom integration with `GStreamer`
  - iOS: `MPMoviePlayerController`
  - Winforms: ?
  - Android: ?
- `PDFView` - Display a PDF document
  - Cocoa: `PDFView`
  - GTK: ?
  - iOS: Integration with QuickLook?
  - Winforms: ?
  - Android: ?
- `MapView` - Display a map
  - Cocoa: `MKMapView`
  - GTK: Probably a `Webkit.WebView` pointing at Google Maps/OpenStreetMap
  - iOS: `MKMapView`
  - Winforms: ?
  - Android: ?

Container widgets

Containers are widgets that can contain other widgets.

- `ButtonContainer` - A layout for a group of radio/checkbox options
  - Cocoa: `NSMatrix`, or `NSView` with pre-set constraints.
  - GTK: `Gtk.ListBox`
  - iOS: ?
  - Winforms: ?
  - Android: ?
- `FormContainer` - A layout for a "key/value" or "label/widget" form
  - Cocoa: `NSForm`, or `NSView` with pre-set constraints.
  - GTK:
  - iOS:
  - Winforms: ?
  - Android: ?

- `NavigationContainer` - A container view that holds a navigable tree of sub-views

  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 macOS System Preferences panel.

  - Cocoa: No native control
  - GTK: No native control; `Gtk.HeaderBar` in 3.10+
  - iOS: `UINavigationBar` + `NavigationController`
  - Winforms: ?
  - Android: ?

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.

- Preferences - Support for saving app preferences, and visualizing them in a platform native way.
- 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 macOS, GTK on Linux, Winforms on Windows, iOS and Android. Proof-of-concept support exists for single page web apps. Support for a more modern Windows API would be desirable.

2.4.6 Architecture

Although Toga presents a single interface to the end user, there are three internal layers that make up every widget. They are:

- The **Interface** layer
- The **Implementation** layer
- The **Native** layer

Interface

The interface layer is the public, documented interface for each widget. Following *Toga's design philosophy*, these widgets reflect high-level design concepts, rather than specific common widgets. It forms the public API for creating apps, windows, widgets, and so on.

The interface layer is responsible for validation of any API inputs, and storage of any persistent values retained by a widget. That storage may be supplemented or replaced by storage on the underlying native widget (or widgets), depending on the capabilities of that widget.

The interface layer is also responsible for storing style and layout-related attributes of the widget.

The interface layer is defined in the `toga-core` module.

Implementation

The implementation layer is the platform-specific representation of each widget. Each platform that Toga supports has its own implementation layer, named after the widget toolkit that the implementation layer is wrapping – `toga-cocoa` for macOS (Cocoa being the name of the underlying macOS widget toolkit); `toga-gtk` for Linux (using the GTK+ toolkit); and so on. The implementation provides a private, internal API that the interface layer can use to create the widgets described by the interface layer.

The API exposed by the implementation layer is different to that exposed by the interface layer and is *not* intended for end-user consumption. It is a utility API, servicing the requirements of the interface layer.

Every widget in the implementation layer corresponds to exactly one widget in the interface layer. However, the reverse will not always be true. Some widgets defined by the interface layer are not available on all platforms.

An interface widget obtains its implementation when it is constructed, using the platform factory. Each platform provides a factory implementation. When a Toga application starts, it guesses its platform based on the value of `sys.platform`, and uses that factory to create implementation-layer widgets.

If you have an interface layer widget, the implementation widget can be obtained using the `_impl` attribute of that widget.

Native

The lowest layer of Toga is the native layer. The native layer represents the widgets required by the widget toolkit of your system. These are accessed using whatever bridging library or Python-native API is available on the implementation platform. This layer is usually provided by system-level APIs, not by Toga itself.

Most implementation widgets will have a single native widget. However, when a platform doesn't expose a single widget that meets the requirements of the Toga interface specification, the implementation layer will use multiple native widgets to provide the required functionality.

In this case, the implementation must provide a single "container" widget that represents the overall geometry of the combined native widgets. This widget is called the "primary" native widget. When there's only one native widget, the native widget is the primary native widget.

If you have an implementation widget, the interface widget can be obtained using the `interface` attribute, and the primary native widget using the `native` attribute.

If you have a native widget, the interface widget can be obtained using the `interface` attribute, and the implementation widget using the `impl` attribute.

An example

Here's how Toga's three-layer API works on the Button widget.

- `toga.Button` is defined in `core/src/toga/widgets/button.py`. This defines the public interface for the Button widget, describing (amongst other things) that there is an `on_click` event handler on a Button. It expects that there will be *an* implementation, but doesn't care which implementation is provided.
- `toga-gtk.widgets.Button` is defined in `gtk/src/toga-gtk/widgets/button.py`. This defines the Button at the implementation layer. It describes how to create a button on GTK, and how to connect the GTK `clicked` signal to the `on_click` Toga handler.
- `Gtk.Button` is the native GTK-Python widget API that implements buttons on GTK.

This three layered approach allows us to change the implementation of `Button` without changing the public API that end-users rely upon. For example, we could switch out `toga-gtk.widgets.Button` with `toga-cocoa.widgets.Button` to provide a macOS implementation of the Button without altering the API that end-users use to construct buttons.

The layered approach is especially useful with more complex widgets. Every platform provides a Button widget, but other widgets are more complex. For example, macOS doesn't provide a native DetailedList view, so it must be constructed out of a scroll view, a table view, and a collection of other pieces. The three layered architecture hides this complexity - the API exposed to developers is a single (interface layer) widget; the complexity of the implementation only matters to the maintainers of Toga.

Lastly, the layered approach provides a testing benefit. In addition to the Cocoa, GTK, and other platform implementations, there is a "dummy" implementation. This implementation satisfies all the API requirements of a Toga implementation layer, but without actually performing any graphical operations. This dummy API can be used to test code using the Toga interface layer.

2.4.7 Understanding widget layout

One of the major tasks of a GUI framework is to determine where each widget will be displayed within the application window. This determination must be made when a window is initially displayed, and every time the window changes size (or, on mobile devices, changes orientation).

Layout in Toga is performed using style engine. Toga provides a *built-in style engine called Pack*; however, other style engines can be used. Every widget keeps a style object, and it is this style object that is used to perform layout operations.

Each widget can also report an "intrinsic" size - this is the size of the widget, as reported by the underlying GUI library. The intrinsic size is a width and height; each dimension can be fixed, or specified as a minimum. For example, a button may have a fixed intrinsic height, but a minimum intrinsic width (indicating that there is a minimum size the button can be, but it can stretch to assume any larger size). This intrinsic size is computed when the widget is first displayed; if fundamental properties of the widget ever change (e.g., changing the text or font size on a button), the widget needs to be rehinted, which re-calculates the intrinsic size, and invalidates any layout.

Widgets are constructed in a tree structure. The widget at the root of the tree is called the *container* widget. Every widget keeps a reference to the container at the root of its widget tree.

When a widget is added to a window, a *Viewport* is created. This viewport connects the widget to the available space provided by the window.

When a window needs to perform a layout, the layout engine asks the style object for the container to lay out its contents with the space that the viewport has available. This will perform whatever calculations are required and apply any position information to the widgets in the widget tree.

Every window has a container and viewport, representing the total viewable area of the window. However, some widgets (called Container widgets) establish sub-containers. When a refresh is requested on a container, any sub-containers will also be refreshed.

2.4.8 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 its 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):
    ...
```

(continues on next page)

(continued from previous page)

```
    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 the 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.

2.4.9 Data Sources

Most widgets in a user interface will need to interact with data - either displaying it, or providing a way to manipulate it.

Well designed GUI applications will maintain a strong separation between the data, and how that data is displayed. This separation allows developers to radically change how data is visualized without changing the underlying interface for interacting with this data.

Toga encourages this separation by using data sources. Instead of directly telling a widget to display a particular value (or collection of values), Toga requires you to define a **data source**, and then tell a widget to display that source.

Built-in data sources

There are three built-in data source types in Toga:

- **Value Sources**: For managing a single value. A `Value` has a single attribute, `value`, which is the value that will be rendered for display purposes.
- **List Sources**: For managing a list of items, each of which has one or more values. List data sources support the data manipulation methods you'd expect of a `list`, and return `Row` objects. The attributes of each `Row` object are the values that should be displayed.

- **Tree Sources**: For managing a hierarchy of items, each of which has one or more values. Tree data sources also behave like a `list`, except that each item returned is a `Node`. The attributes of the `Node` are the values that should be displayed; a `Node` also has children, accessible using the `list` interface on the `Node`.

Listeners

Data sources communicate to widgets (and other data sources) using a listener interface. Once a data source has been created, any other object can register as a listener on that data source. When any significant event occurs to the data source, all listeners will be notified.

Notable events include: * Adding a new item * Removing an existing item * Changing a value on an item * Clearing an entire data source

If any attribute of a `Value`, `Row` or `Node` is modified, the source will generate a change event.

Custom data sources

Although Toga provides built-in data sources, in general, *you shouldn't use them*. Toga's data sources are wrappers around Python's primitive data types - *int*, *str*, *list*, *dict*, and so on. While this is useful for quick demonstrations, or to visualize simple data, more complex applications should define their own data sources.

A custom data source enables you to provide a data manipulation API that makes sense for your application. For example, if you were writing an application to display files on a file system, you shouldn't just build a dictionary of files, and use that to construct a `TreeSource`. Instead, you should write your own `FileSystemSource` that reflects the files on the file system. Your file system data source doesn't need to expose `insert()` or `remove()` methods - because the end user doesn't need an interface to "insert" files into your filesystem. However, you might have a *create_empty_file()* method that creates a new file in the filesystem and adds a representation to the tree.

Custom data sources are also required to emit notifications whenever notable events occur. This allows the widgets rendering the data source to respond to changes in data. If a data source doesn't emit notifications, widgets may not reflect changes in data.

Value sources

A Value source is any object with a "value" attribute.

List sources

List data sources need to provide the following methods:

- `__len__(self)` - returns the number of items in the list
- `__getitem__(self, index)` - returns the item at position `index` of the list.

Each item returned by the List source is required to expose attributes matching the accessors for any widget using the source.

Tree sources

Tree data sources need to provide the following methods:

- `__len__(self)` - returns the number of root nodes in the tree
- `__getitem__(self, index)` - returns the root node at position `index` of the tree.

Each node returned by the Tree source is required to expose attributes matching the accessors for any widget using the source. The node is also required to implement the following methods:

- `__len__(self)` - returns the number of children of the node.
- `__getitem__(self, index)` - returns the child at position `index` of the node.
- `can_have_children(self)` - returns True if the node is allowed to have children. The result of this method does *not* depend on whether the node actually has any children; it only describes whether it is allowed to store children.

PYTHON MODULE INDEX

t

INDEX

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W