Toga Documentation

Release 0.3.0.dev18

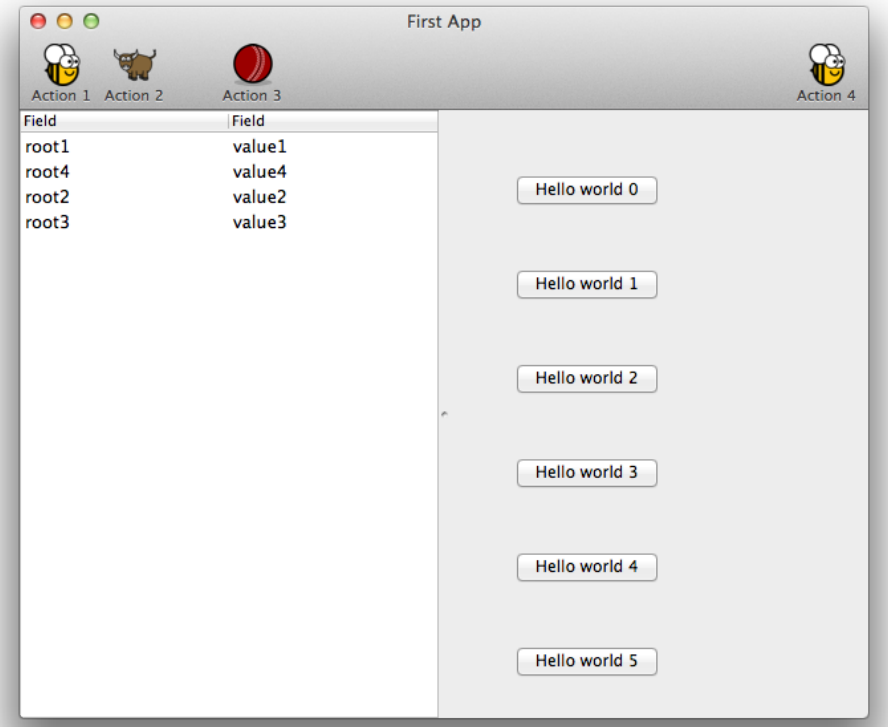
Russell Keith-Magee

Feb 11, 2020

Contents

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

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



CHAPTER 1

Table of contents

1.1 Tutorial

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

1.2 How-to guides

Guides and recipes for common problems and tasks

1.3 Background

Explanation and discussion of key topics and concepts

1.4 Reference

Technical reference - commands, modules, classes, methods

CHAPTER 2

Community

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

- @pybeeware on Twitter
- beeware/general on Gitter

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

Open a command prompt on your computer and make sure that you can successfully run the `python3` command. 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) $ pip install --pre 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, Debian 9
(venv) $ sudo apt-get update
(venv) $ sudo apt-get install python3-dev libgirepository1.0-dev libcairo2-dev␣
↪libpango1.0-dev libwebkitgtk-3.0-0 gir1.2-webkit-3.0

# Debian 10
# has webkit2-4.0
# libwebkitgtk version seems very specific, but that is what it currently is @␣
↪20190825
(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
```

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) $ pip install --pre toga
```

Windows

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

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
```

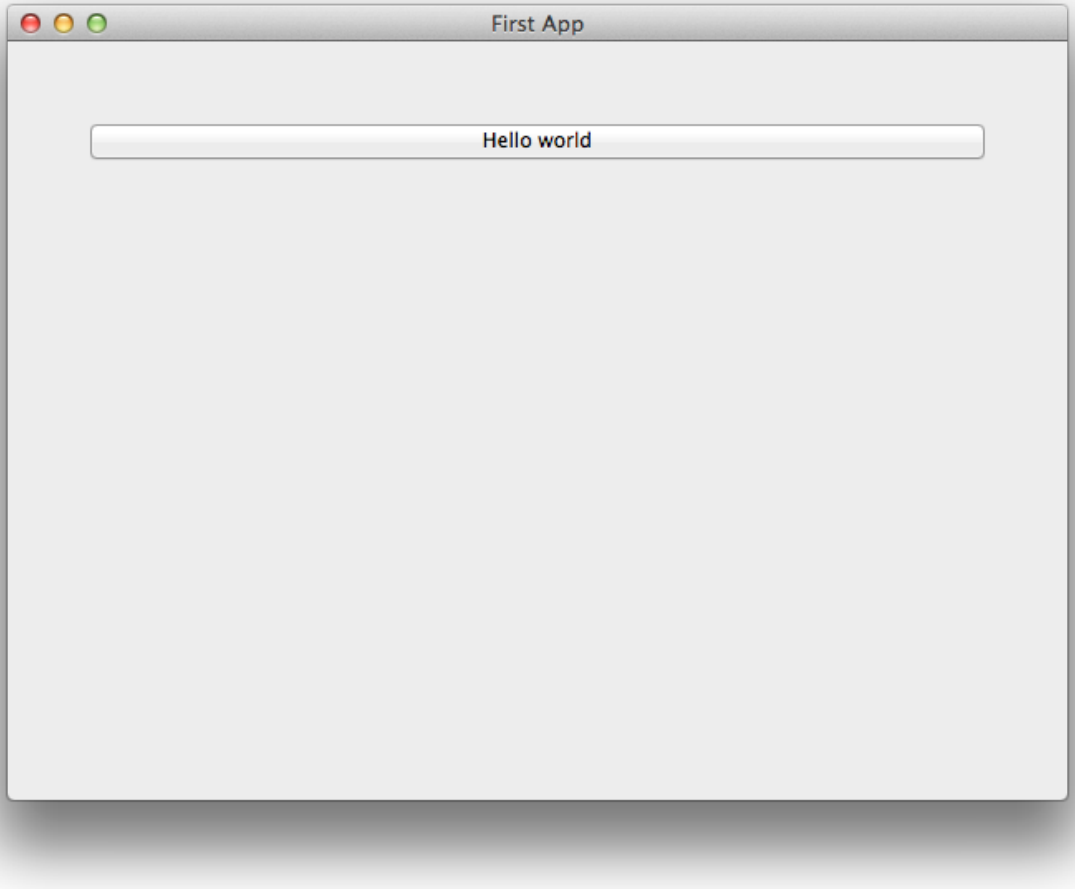
Linux

```
(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:

```
$ pip install --pre 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 OS X, you need to be on 10.7 (Lion) or newer.
- If you're on Linux, you need to have GTK+ 3.4 or later. This is the version that ships starting with Ubuntu 12.04 and Fedora 17.

If 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()
```

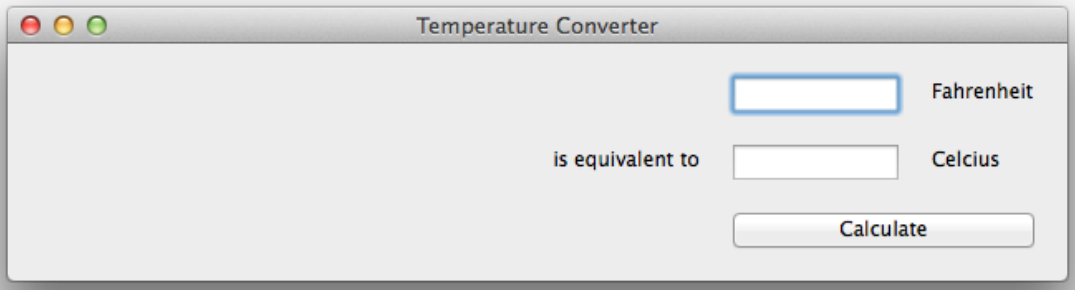
Toga apps must be executed as modules - hence the `-m` flag.

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 *


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:
            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_top=10)
    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, flex=1)

    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...

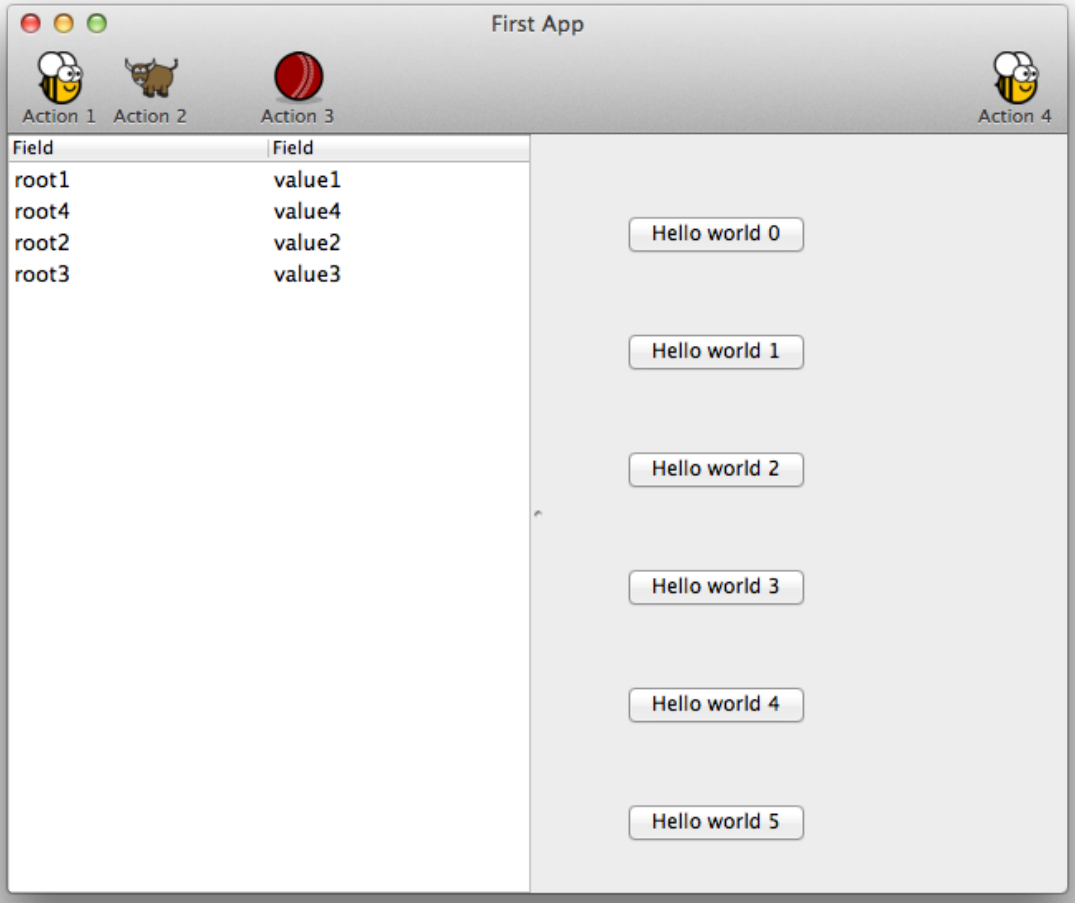
Note: Toga is a work in progress, and may not be consistent across all platforms.

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 Pack, COLUMN


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 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()

    split.content = [left_container, right_container]

    things = toga.Group('Things')

    cmd0 = toga.Command(
        action0,
        label='Action 0',
        tooltip='Perform action 0',
        icon=brutus_icon,
        group=things
    )
    cmd1 = toga.Command(
        action1,
        label='Action 1',
        tooltip='Perform action 1',
        icon=brutus_icon,
        group=things
```

(continues on next page)

(continued from previous page)

```
    )
    cmd2 = toga.Command(
        action2,
        label='Action 2',
        tooltip='Perform action 2',
        icon=toga.Icon.TOGA_ICON,
        group=things
    )
    cmd3 = toga.Command(
        action3,
        label='Action 3',
        tooltip='Perform action 3',
        shortcut=toga.Key.MOD_1 + 'k',
        icon=cricket_icon
    )

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

    cmd4 = toga.Command(
        action4,
        label='Action 4',
        tooltip='Perform action 4',
        icon=brutus_icon
    )

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

    return split


def main():
    return toga.App('First App', 'org.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 Pack, ROW, CENTER, COLUMN


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

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

        box = toga.Box(
            children=[
                toga.Box(
```

(continues on next page)

(continued from previous page)

```
                    children=[
                        self.url_input,
                        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 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.

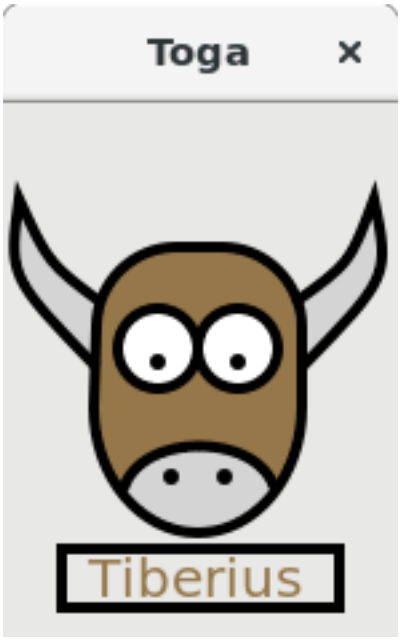
2. Next we create a new Canvas:

```
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):
        # Main window of the application with title and size
```

(continues on next page)

(continued from previous page)

```
        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])
        self.draw_tiberius()

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

        # 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)
                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)
```

(continues on next page)

(continued from previous page)

```
            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 = font.measure('Tiberius', 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

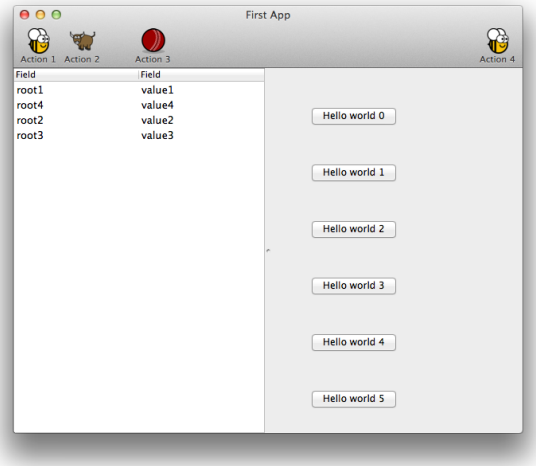
In *Your first Toga app*, you will discover how to create a basic app and have a simple `toga.interface.widgets.button.Button` widget to click.

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

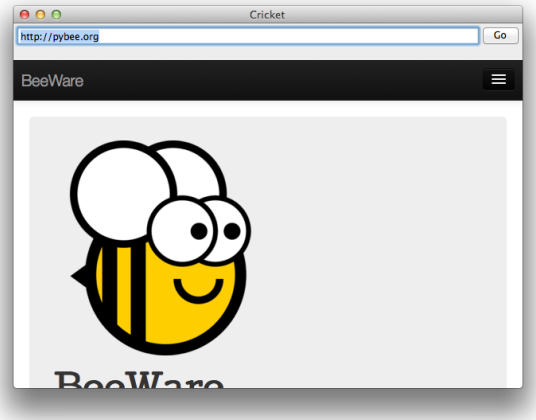
2.1.8 Tutorial 2 - you put the box inside another box...

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



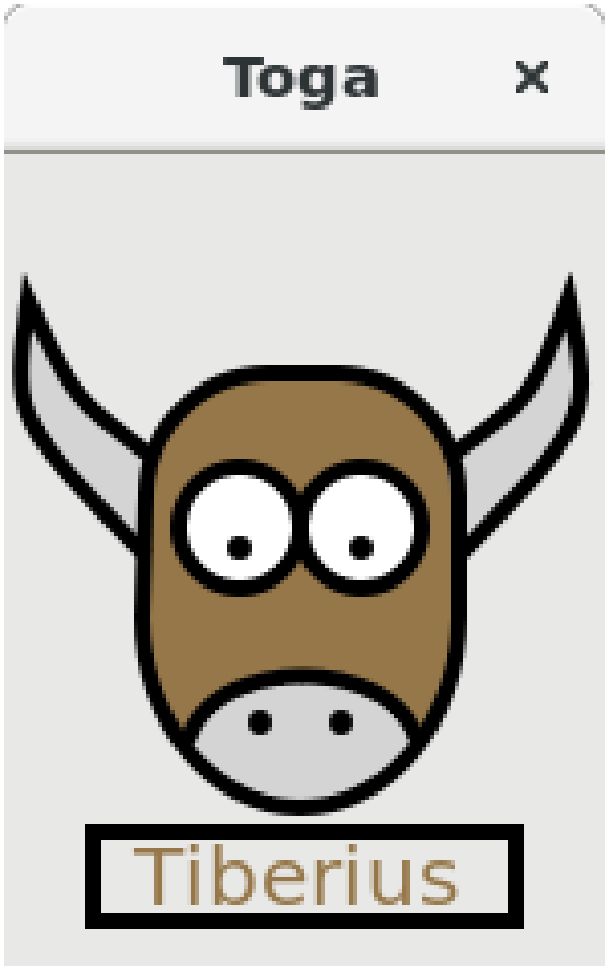
2.1.9 Tutorial 3 - let's build a browser!

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


Toga
Tiberius

2.2 How-to Guides

2.2.1 How to get started

Quickstart

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

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

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

There is a known issue with the current build on some Mac OS distributions. If you are running Mac OS Sierra or higher, use the following installation command instead:

$ pip install toga-demo –pre

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

2.2.2 How to contribute to Toga

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

Set up your development environment

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, Debian 9
(venv) $ sudo apt-get update
(venv) $ sudo apt-get install python3-dev libgirepository1.0-dev libcairo2-dev␣
↪libpango1.0-dev libwebkitgtk-3.0-0 gir1.2-webkit-3.0

# Debian 10
# has webkit2-4.0
# libwebkitgtk version seems very specific, but that is what it currently is @␣
↪20190825
(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
```

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

Linux

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

Windows

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

You can then run the core test suite:

macOS

```
(venv) $ cd src/core
(venv) $ python setup.py test
...
----------------------------------------------------------------------
Ran 181 tests in 0.343s

OK (skipped=1)
```

Linux

```
(venv) $ cd src/core
(venv) $ python setup.py test
...
----------------------------------------------------------------------
Ran 181 tests in 0.343s

OK (skipped=1)
```

Windows

```
(venv) C:\...>cd src/core
(venv) C:\...>python setup.py test
...
----------------------------------------------------------------------
Ran 181 tests in 0.343s

OK (skipped=1)
```

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

Now you are ready to start hacking on Toga!

What should I do?

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

Install coverage, and then re-run the test suite – this time, in a slightly different way so that we can gather some data about the test run. Then we can ask coverage to generate a report of the data that was gathered:

macOS

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

Linux

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

Windows

```
(venv) C:\...>pip install coverage
(venv) C:\...>coverage run setup.py test
(venv) C:\...>coverage report -m --include=toga/*
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 *src/core/tests*, you should find a test file that matches the name of the file that has insufficient coverage. If you don't, it's possible the entire test file is missing - so you'll have to create it!

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

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

macOS

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

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

Linux

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

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

Windows

```
(venv) C:\...>coverage run setup.py test
running test
...
----------------------------------------------------------------------
Ran 101 tests in 0.343s

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

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

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

How does this all work?

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

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

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

One error you might see...

When you're running these tests - especially when you submit your PR, and the tests run on our continous integration (CI) server - it's possible you might get and error that reads:

```
ModuleNotFoundError: No module named 'toga_gtk'.
```

If this happens, you've found an bug in the way the widget you're testing has been constructed.

The Core API is designed to be platform independent. When a widget is created, it calls upon a "factory" to instantiate the underlying platform-dependent implementation. When a Toga application starts running, it will try to guess the right factory to use based on the environment where the code is running. So, if you run your code on a Mac, it will use the Cocoa factory; if you're on a Linux box, it will use the GTK factory.

However, when writing tests, we want to use the "dummy" factory. The Dummy factory isn't the "native" platform anywhere - it's just a placeholder. As a result, the dummy factory won't be used unless you specifically request it - which means every widget has to honor that request.

Most Toga widgets create their platform-specific implementation when they are created. As a result, most Toga widgets should accept a `factory` argument - and that factory should be used to instantiate any widget implementations or sub-widgets.

However, *some* widgets - like Icon - are "late loaded" - the implementation isn't created until the widget is actually *used*. Late loaded widgets don't accept a `factory` when they're created - but they *do* have an *_impl()* method that accepts a factory.

If these factory arguments aren't being passed around correctly, then a test suite will attempt to create a widget, but will fall back to the platform- default factory, rather than the "dummy" factory. If you've installed the appropriate platform default backend, you won't (necessarily) get an error, but your tests won't use the dummy backend. On our CI server, we deliberately don't install a platform backend so we can find these errors.

If you get the `ModuleNotFoundError`, you need to audit the code to find out where a widget is being created without a factory being specified.

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.

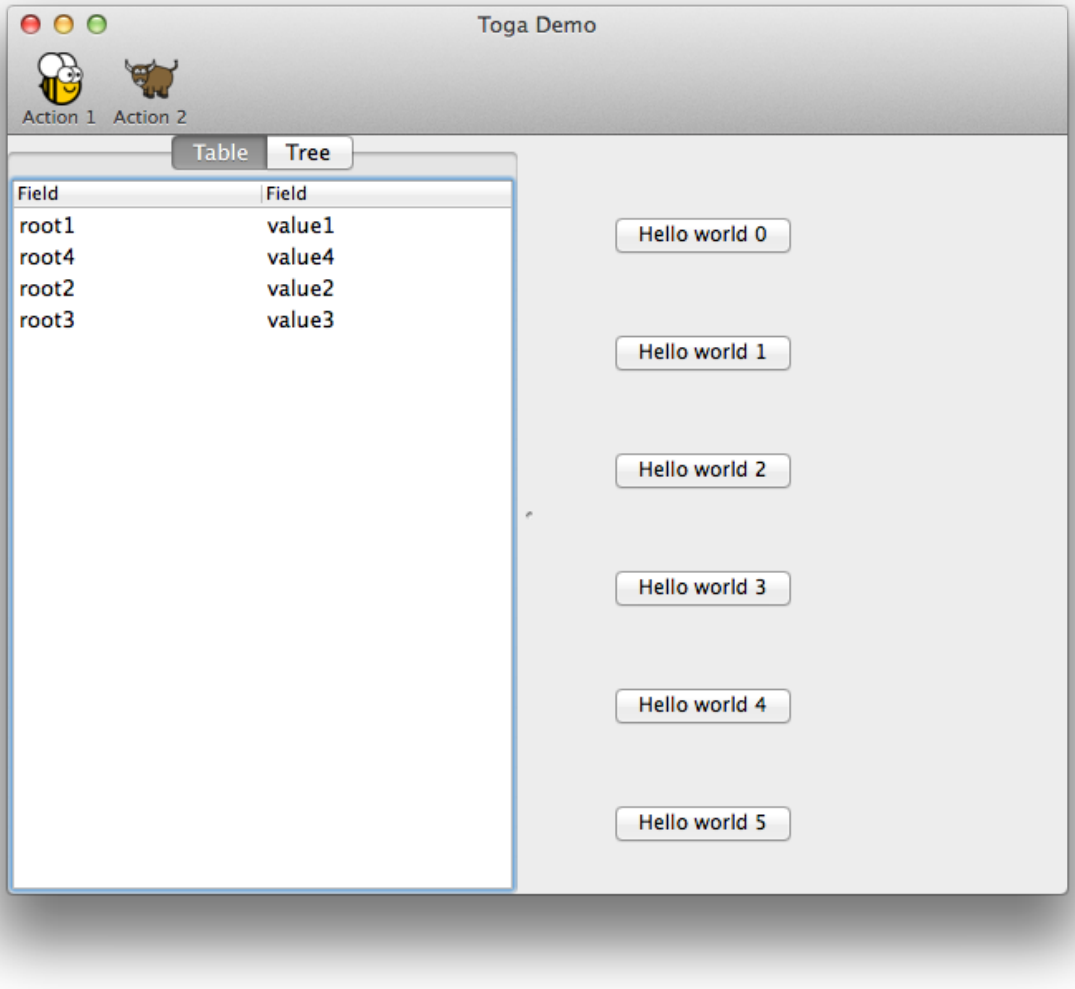
2.3 Reference

2.3.1 Toga supported platforms

Official platform support

Desktop platforms

macOS

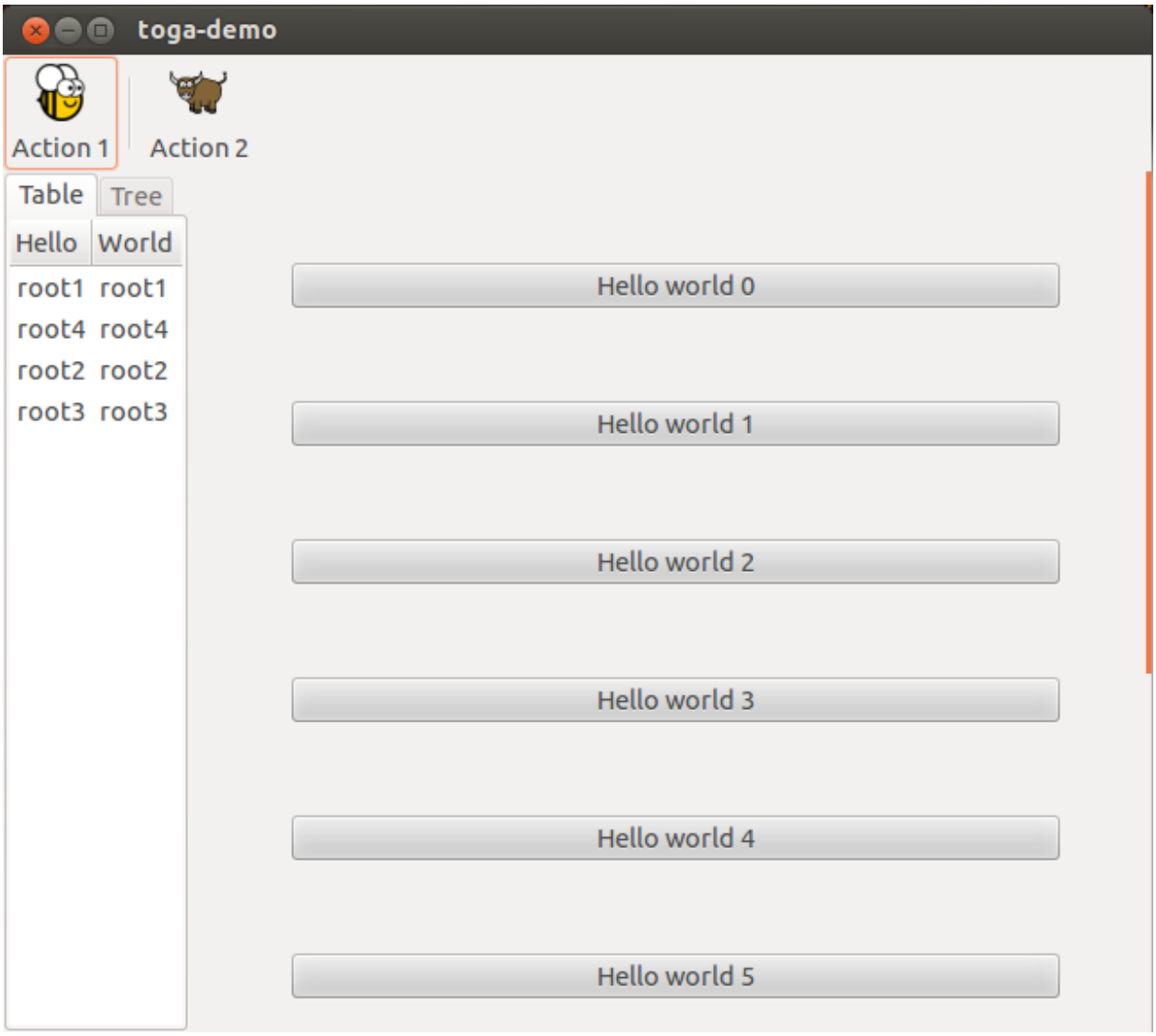


The backend for macOS is named toga-cocoa. It supports macOS 10.7 (Lion) and later. It is installed automatically on macOS machines (machines that report `sys.platform == 'darwin'`), or can be manually installed by invoking:

```
$ 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.

Winforms

The backend for Windows is named toga-winforms. It supports Windows XP or later with .NET 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
```

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

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 VOC to compile Python code to Java class files for execution on Android devices.

Web platforms

Django

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

```
$ pip install toga-django
```

The Django backend is currently proof-of-concept only. Most widgets have not been implemented. It uses Batavia 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:

- Other Python web frameworks (e.g., Flask, Pyramid)

- 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 Twitter or Gitter.

Unofficial platform support

At present, there are no known unofficial platform backends.

2.3.2 Toga widgets by platforms

Core Components

Component	Description	macOS	GTK+	Windows	iOS	Android	Django
App	The application itself	✓	✓	✓	✓	✓	✓
Window	Window object	✓	✓	✓	✓	✓	✓
MainWindow	Main window of the application	✓	✓	✓	✓	✓	✓
ActivityIndicator	A spinning activity animation	✓					

General Widgets

Component	Description	macOS	GTK+	Windows	iOS	Android	Django
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	✓	✓	✓		✓	✓

Continued on next page

Table 2 – continued from previous page

Component	Description	macOS	GTK+	Windows	iOS	Android	Django
Widget	The base widget	✓	✓	✓	✓		✓

Layout Widgets

Component	Description	macOS	GTK+	Windows	iOS	Android	Django
Box	Container for components	✓	✓	✓	✓	✓	✓
ScrollContainer	Scrollable Container	✓	✓	✓	✓		
SplitContainer	Split Container	✓	✓	✓			
OptionContainer	Option Container	✓	✓	✓			

Resources

Component	Description	macOS	GTK+	Windows	iOS	Android	Django
Font	Fonts	✓	✓		✓		
Command	Command	✓	✓				
Group	Command group	✓	✓	✓	✓	✓	✓
Icon	An icon for buttons, menus, etc	✓	✓	✓			
Image	An 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

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓	✓	✓

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*, *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 initialised. 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(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 autogenerated 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 `toga.App`.
- **factory** – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

add_background_task(*handler*)

app = None

app_id
The identifier for the app.

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

Returns The identifier as a `str`.

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

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

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

Returns The author of the app, as a `str`.

current_window
Return the currently active content window

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.

formal_name
The formal name of the app.

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

hide_cursor()
Hide cursor from view.

home_page
The URL of a web page for the app.

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

icon
The Icon for the app.

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

id
The DOM identifier for the app.

This id can be used to target CSS directives.

Returns A DOM identifier for the app.

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.

main_window
The main windows for the app.

Returns The main Window of the app.

module_name
The module name for the app

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

name
The formal name of the app.

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

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

version
The version number of the app.

Returns The version numberof the app, as a `str`.

MainWindow

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓	✓	✓

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)*, *factory=None*)

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

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

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

close()

confirm_dialog(*title*, *message*)
Opens a dialog with a 'Cancel' and 'OK' button.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – The dialog message to display.

Returns Returns *True* when the 'OK' button was pressed, *False* when the 'CANCEL' button was pressed.

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 `toga.Widget`

error_dialog(*title*, *message*)
Opens a error dialog with a 'OK' button to close the dialog.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – The dialog message to display.

Returns Returns *None* after the user pressed the 'OK' button.

full_screen

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*)
Opens a info dialog with a 'OK' button to close the dialog.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – The dialog message to display.

Returns Returns *None* after the user pressed the 'OK' button.

on_close()

open_file_dialog(*title*, *initial_directory=None*, *file_types=None*, *multiselect=False*)
This opens a native dialog where the user can select the file to open. It is possible to set the initial folder and only show files with specified file extensions. If no path is returned (eg. dialog is canceled), a ValueError is raised. :param title: The title of the dialog window. :type title: str :param initial_directory: Initial folder displayed in the dialog. :type initial_directory: str :param file_types: A list of strings with the allowed file extensions. :param multiselect: Value showing whether a user can select multiple files.

Returns The absolute path(str) to the selected file or a list(str) if multiselect

position
Position of the window, as x, y

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

question_dialog(*title*, *message*)
Opens a dialog with a 'YES' and 'NO' button.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – The dialog message to display.

Returns Returns *True* when the 'YES' button was pressed, *False* when the 'NO' button was pressed.

save_file_dialog(*title*, *suggested_filename*, *file_types=None*)
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 (eg. dialog is canceled), a ValueError is raised.

Parameters

- **title** (*str*) – The title of the dialog window.
- **suggested_filename** (*str*) – The automatically filled in filename.
- **file_types** – A list of strings with the allowed file extensions.

Returns The absolute path(str) to the selected location.

select_folder_dialog(*title*, *initial_directory=None*, *multiselect=False*)
This opens a native dialog where the user can select a folder. It is possible to set the initial folder. If no path is returned (eg. dialog is canceled), a ValueError is raised. :param title: The title of the dialog window. :type title: str :param initial_directory: Initial folder displayed in the dialog. :type initial_directory: str :param multiselect: Value showing whether a user can select multiple files. :type multiselect: bool

Returns The absolute path(str) to the selected file or None.

show()
Show window, if hidden

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*)
Calling this function opens a dialog that allows to display a large text body in a scrollable fashion.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – The dialog message to display.
- **content** (*str*) –
- **retry** (*bool*) –

Returns Returns *None* after the user pressed the 'OK' button.

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

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

toolbar
Toolbar for the window.

Returns A `list` of `toga.Widget`

Window

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓	✓	✓

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*)

The top level container of a 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 toga.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*.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional; normally not needed)

app

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

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

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

close()

confirm_dialog(*title*, *message*)
Opens a dialog with a 'Cancel' and 'OK' button.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – The dialog message to display.

Returns Returns *True* when the 'OK' button was pressed, *False* when the 'CANCEL' button was pressed.

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 `toga.Widget`

error_dialog(*title*, *message*)
Opens a error dialog with a 'OK' button to close the dialog.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – The dialog message to display.

Returns Returns *None* after the user pressed the 'OK' button.

full_screen

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*)
Opens a info dialog with a 'OK' button to close the dialog.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – The dialog message to display.

Returns Returns *None* after the user pressed the 'OK' button.

on_close()

open_file_dialog(*title*, *initial_directory=None*, *file_types=None*, *multiselect=False*)
This opens a native dialog where the user can select the file to open. It is possible to set the initial folder and only show files with specified file extensions. If no path is returned (eg. dialog is canceled), a ValueError is raised. :param title: The title of the dialog window. :type title: str :param initial_directory: Initial folder displayed in the dialog. :type initial_directory: str :param file_types: A list of strings with the allowed file extensions. :param multiselect: Value showing whether a user can select multiple files.

Returns The absolute path(str) to the selected file or a list(str) if multiselect

position
Position of the window, as x, y

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

question_dialog(*title*, *message*)
Opens a dialog with a 'YES' and 'NO' button.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – The dialog message to display.

Returns Returns *True* when the 'YES' button was pressed, *False* when the 'NO' button was pressed.

save_file_dialog(*title*, *suggested_filename*, *file_types=None*)
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 (eg. dialog is canceled), a ValueError is raised.

Parameters

- **title** (*str*) – The title of the dialog window.
- **suggested_filename** (*str*) – The automatically filled in filename.
- **file_types** – A list of strings with the allowed file extensions.

Returns The absolute path(str) to the selected location.

select_folder_dialog(*title*, *initial_directory=None*, *multiselect=False*)
This opens a native dialog where the user can select a folder. It is possible to set the initial folder. If no path is returned (eg. dialog is canceled), a ValueError is raised. :param title: The title of the dialog window. :type title: str :param initial_directory: Initial folder displayed in the dialog. :type initial_directory: str :param multiselect: Value showing whether a user can select multiple files. :type multiselect: bool

Returns The absolute path(str) to the selected file or None.

show()
Show window, if hidden

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*)
Calling this function opens a dialog that allows to display a large text body in a scrollable fashion.

Parameters

- **title** (*str*) – The title of the dialog window.
- **message** (*str*) – The dialog message to display.
- **content** (*str*) –
- **retry** (*bool*) –

Returns Returns *None* after the user pressed the 'OK' button.

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

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

toolbar
Toolbar for the window.

Returns A `list` of `toga.Widget`

Containers

Box

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓	✓	✓

The box is a generic container for widgets, allowing you to construct layouts.

Usage

A box can be instantiated with no children and the children added later:

```
import toga

box = toga.Box('box1')

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

To create boxes within boxes, use the children argument:

```
import toga

box_a = toga.Box('box_a')
box_b = toga.Box('box_b)

box = toga.Box('box', children=[box_a, box_b])
```

Box Styling

Styling of boxes can be done during instantiation of the Box:

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

box = toga.Box(id='box', style=Pack(direction=COLUMN, padding_top=10))
```

Styles can be also be updated on an existing instance:

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

box = toga.Box(id='box', style=Pack(direction=COLUMN))
```

(continues on next page)

(continued from previous page)

```
box.style.update(padding_top=10)
```

Reference

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

This is a Widget that contains other widgets, but has no rendering or interaction of its own.

Parameters

- **id** (*str*) – An identifier for this widget.
- **(** (*style*) – class:colosseum.CSSNode‘): An optional style object. If no style is provided then a new one will be created for the widget.
- **children** (list of toga.Widget) – An optional list of child Widgets that will be in this box.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

enabled

id

The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a str.

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()

root
The root of the tree containing this node.

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

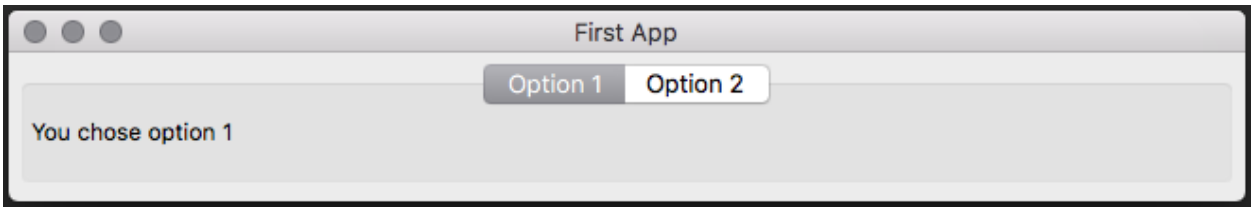
window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Option Container

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓			

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, toga.Widget)) – Each tuple in the list is composed of a title for the option and the widget tree that is displayed in the option.

add(*label*, *widget*)
Add a new option to the option container.

Parameters

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

app
The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

content
The sub layouts of the *SplitContainer*.

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

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

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a str.

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

Returns (callable) The callback function.

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()
Refresh the layout and appearance of this widget.

root
The root of the tree containing this node.

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

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Scroll Container

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓		

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)
```

(continues on next page)

(continued from previous page)

```
container.vertical = False
```

Reference

class toga.widgets.scrollcontainer.**ScrollContainer**(*id=None*, *style=None*, *horizontal=True*, *vertical=True*, *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** (toga.Widget) – The content of the scroll window.
- **(** (*factory*) – module:): A provided factory module will be used to create the implementation of the ScrollContainer.

MIN_HEIGHT = 100

MIN_WIDTH = 100

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

content

Content of the scroll container.

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

enabled

horizontal

Shows whether horizontal scrolling is enabled.

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

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

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 this widget.

refresh_sublayouts()
Refresh the layout and appearance of this widget.

root
The root of the tree containing this node.

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

vertical
Shows whether vertical scrolling is enabled.

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

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Split Container

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓			

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

Usage

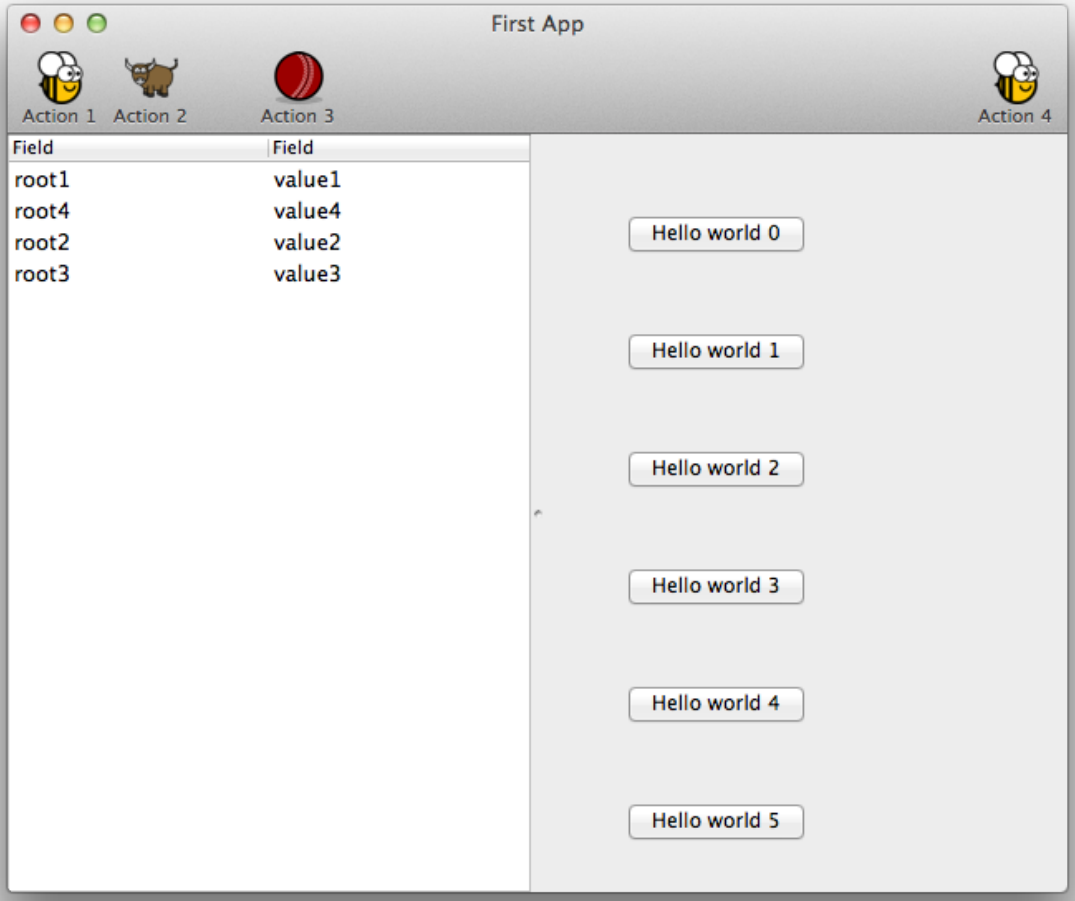
```
import toga

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.


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

```
import toga

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

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

Reference

class toga.widgets.splitcontainer.**SplitContainer**(*id=None*, *style=None*, *direction=True*, *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 toga.Widget) – The list of components to be split.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

HORIZONTAL = False

VERTICAL = True

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

content

The sub layouts of the *SplitContainer*.

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

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

direction
The direction of the split

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

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

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()
Refresh the layout and appearance of this widget.

root
The root of the tree containing this node.

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

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Resources

Font

macOS	GTK+	Windows	iOS	Android	Django
✓	✓		✓		

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*)

bold()
Generate a bold version of this font

italic()
Generate an italic version of this font

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

normal_style()
Generate a normal style version of this font

normal_variant()
Generate a normal variant of this font

normal_weight()
Generate a normal weight version of this font

oblique()
Generate an oblique version of this font

small_caps()
Generate a small-caps variant of this font

Command

macOS	GTK+	Windows	iOS	Android	Django
✓	✓				

Usage

Reference

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

Parameters

- **action** – a function to invoke when the command is activated.
- **label** – a name 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 label within its section.

bind(*factory*)

enabled

icon
The Icon for the app.

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

Group

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓	✓	✓

Usage

Reference

class `toga.command.`**Group**(*label*, *order=None*)

Parameters

- **label** –
- **order** –

```
APP = <toga.command.Group object>
COMMANDS = <toga.command.Group object>
EDIT = <toga.command.Group object>
FILE = <toga.command.Group object>
HELP = <toga.command.Group object>
VIEW = <toga.command.Group object>
WINDOW = <toga.command.Group object>
```

Icon

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓			

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.icons.Icon object>

TOGA_ICON = <toga.icons.Icon object>

bind(*factory*)

Bind the Icon to a factory.

Creates the underlying platform implemenation of the Icon. If the image cannot be found, it will fall back to the default icon.

Parameters **factory** – The platform factory to bind to.

Returns The platform implementation

Image

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓		

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*)

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.

bind(*factory*)

Bind the Image to a factory.

Creates the underlying platform implemenation of the Image. Raises FileNotFoundError if the path is a non-existent local file.

Parameters **factory** – The platform factory to bind to.

Returns The platform implementation

Widgets

Activity Indicator

macOS	GTK+	Windows	iOS	Android	Django
✓					

The activity indicator is a (spinning) animation for showing progress in an indeterminate task.

Usage

```
import toga

spinner = toga.ActivityIndicator()

# make widget visible and start animation
spinner.start()
```

By default, the activity indicator is hidden when it is not running and will become visible when calling `start`. This can be changed by setting the `hide_when_stopped` proprerty to `False`.

Reference

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

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.
- **running** (*bool*) – Set the inital running mode. Defaults to False
- **hide_when_stopped** (*bool*) – Hide the indicator when not running. Defaults to True.
- **factory** (`module`) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

add(**children*)
Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app
The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The `toga.App` to which this widget belongs.

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

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.

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.

enabled

hide_when_stopped
Hide this activity indicator when stopped.

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

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

Returns True if this activity indicator is running False otherwise

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()

root
The root of the tree containing this node.

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

start()
Start this acivity indicator.

stop()
Stop this acivity indicator (if not already stopped).

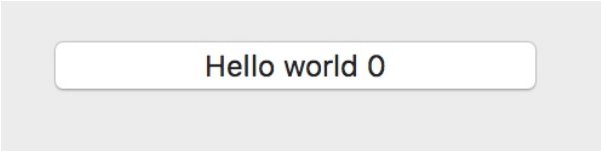
window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Button

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓	✓	✓

The button is a clickable node that fires a callback method when pressed or clicked.



Usage

The most basic button has a text label and a callback method for when it is pressed. The callback expects 1 argument, the instance of the button firing the event.

```
import toga

def my_callback(button):
    # handle event
    pass

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

Reference

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

A clickable button widget.

Parameters

- **label** (*str*) – Text to be shown on the button.
- **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_press** (callable) – Function to execute when pressed.
- **enabled** (*bool*) – Whether or not interaction with the button is possible, defaults to *True*.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The `toga.App` to which this widget belongs.

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

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.

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.

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

label
The button label as a `str`

Type Returns

on_press
The handler to invoke when the button is pressed.

Returns The function `callable` that is called on button press.

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()

root
The root of the tree containing this node.

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

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Canvas

macOS	GTK+	Windows	iOS	Android	Django
✓	✓		✓		

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*, *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) – Function to call when resized.

- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

add(**children*)
Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app
The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

arc(*x*, *y*, *radius*, *startangle=0.0*, *endangle=6.283185307179586*, *anticlockwise=False*)
Constructs and returns a *Arc*.

Parameters

- **x** (*float*) – The x coordinate of the arc's center.
- **y** (*float*) – The y coordinate of the arc's center.
- **radius** (*float*) – The arc's radius.
- **startangle** (*float, optional*) – The angle (in radians) at which the arc starts, measured clockwise from the positive x axis, default 0.0.
- **endangle** (*float, optional*) – The angle (in radians) at which the arc ends, measured clockwise from the positive x axis, default 2*pi.
- **anticlockwise** (*bool, optional*) – 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.

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.

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()
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=6.283185307179586*, *anticlockwise=False*)
Constructs and returns a *Ellipse*.

Parameters

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

Returns *Ellipse* object.

enabled

fill(*color='black'*, *fill_rule='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 'even-odd' 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.

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

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* <*NewPath*> object.

on_resize
The handler to invoke when the canvas is resized.

Returns The function `callable` that is called on canvas resize.

parent
The parent of this node.

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

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.

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

refresh_sublayouts()

remove(*drawing_object*)
Remove a drawing object

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

reset_transform()
Constructs and returns a *ResetTransform*.

Returns *ResetTransform* object.

root
The root of the tree containing this node.

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

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.

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.

translate(*tx*, *ty*)
Constructs and returns a *Translate*.

Parameters

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

Returns *Translate* object.

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

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** (*string*) – 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.Font`, optional) – The font to write with.

Returns *WriteText* object.

Lower-Level Classes

class `toga.widgets.canvas.`**Arc** (*x*, *y*, *radius*, *startangle=0.0*, *endangle=6.283185307179586*, *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** (*float*) – The x coordinate of the arc's center.
- **y** (*float*) – The y coordinate of the arc's center.
- **radius** (*float*) – The arc's radius.
- **startangle** (*float, optional*) – The angle (in radians) at which the arc starts, measured clockwise from the positive x axis, default 0.0.
- **endangle** (*float, optional*) – The angle (in radians) at which the arc ends, measured clockwise from the positive x axis, default 2*pi.
- **anticlockwise** (*bool, optional*) – 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=6.283185307179586*, *anticlockwise=False*)

Constructs and returns a *Arc*.

Parameters

- **x** (*float*) – The x coordinate of the arc's center.
- **y** (*float*) – The y coordinate of the arc's center.
- **radius** (*float*) – The arc's radius.
- **startangle** (*float, optional*) – The angle (in radians) at which the arc starts, measured clockwise from the positive x axis, default 0.0.
- **endangle** (*float, optional*) – The angle (in radians) at which the arc ends, measured clockwise from the positive x axis, default 2*pi.
- **anticlockwise** (*bool, optional*) – 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.

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=6.283185307179586*, *anticlockwise=False*)
Constructs and returns a *Ellipse*.

Parameters

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

Returns *Ellipse* object.

fill(*color='black'*, *fill_rule='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 'even-odd' 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* <*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** (*string*) – 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.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=6.283185307179586*, *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** (*float*) – The x axis of the coordinate for the ellipse's center.
- **y** (*float*) – The y axis of the coordinate for the ellipse's center.
- **radiusx** (*float*) – The ellipse's major-axis radius.
- **radiusy** (*float*) – The ellipse's minor-axis radius.
- **rotation** (*float, optional*) – The rotation for this ellipse, expressed in radians, default 0.0.
- **startangle** (*float, optional*) – The starting point in radians, measured from the x axis, from which it will be drawn, default 0.0.
- **endangle** (*float, optional*) – The end ellipse's angle in radians to which it will be drawn, default 2*pi.
- **anticlockwise** (*bool, optional*) – If true, draws the ellipse anticlockwise (counter-clockwise) instead of clockwise, default false.

class toga.widgets.canvas.**Fill**(*color='black'*, *fill_rule='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.

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** (*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*) – 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.

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** (*string*) – 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.Font, optional) – The font to write with.

DetailedList

macOS	GTK+	Windows	iOS	Android	Django
✓			✓		

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. A updated function can be invoked by pulling the list down.

Parameters

- **id** (*str*) – An identifier for this widget.
- **data** (list of *str*) – List of strings which to display on the widget.
- **on_delete** (callable) – Function that is invoked on row deletion.
- **on_refresh** (callable) – Function that is invoked on user initialised 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.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

Examples

```
>>> import toga
>>> def selection_handler(widget, selection):
>>>     print('Row {} of widget {} was selected.'.format(selection, widget))
>>>
>>> dlist = toga.DetailedList(data=['Item 0', 'Item 1', 'Item 2'], on_
↪select=selection_handler)
```

MIN_HEIGHT = 100

MIN_WIDTH = 100

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

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

Returns Returns a (`ListSource`).

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

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.

on_refresh
The function to be invoked on user initialised refresh.

Type Returns

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

Returns The function to be invoked on selecting a row.

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()

root
The root of the tree containing this node.

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

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

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Canvas

macOS	GTK+	Windows	iOS	Android	Django
✓					

The divider is used to visually separate sections of a user layout with a line.

Usage

Simple usage to separate two labels in a column:

```
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=0*, *factory=None*)
A visual divider line.

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 divider, either `Divider.HORIZONTAL` or `Divider.VERTICAL`. Defaults to *Divider.HORIZONTAL'*

- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

HORIZONTAL = 0

VERTICAL = 1

add(**children*)
Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app
The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

direction
The direction of the split

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

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a str.

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()

root
The root of the tree containing this node.

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

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The toga.Window to which the widget belongs.

Image View

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓		

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** (toga.Image) – The image to display.
- **id** (*str*) – An identifier for this widget.
- **style** (Style) –
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

Todo:

- Finish implementation.

---

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

image

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()

root
The root of the tree containing this node.

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

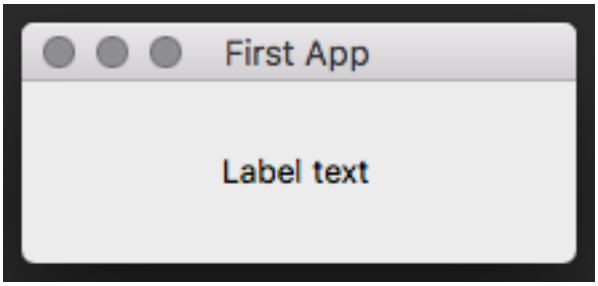
window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Label

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓	✓	

The Label is a text-label for annotating forms or interfaces.



Usage

```
import toga

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

Reference

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

A text label.

Parameters

- **text** (*str*) – Text of the label.
- **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.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional; normally not needed)

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

enabled

id

The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a str.

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()

root

The root of the tree containing this node.

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

text

The text displayed by the label.

Returns The text displayed by the label.

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Multi-line text input

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓		

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*, *initial=None*, *readonly=False*, *placeholder=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.
- **factory** – Optional factory that must be able to return a implementation of a Mulitline-TextInput Widget.
- **initial** (*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.

MIN_HEIGHT = 100

MIN_WIDTH = 100

add(**children*)
Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app
The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The `toga.App` to which this widget belongs.

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

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.

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()
Clears the text from the widget.

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

parent
The parent of this node.

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

placeholder
The placeholder text

Returns The placeholder text as a *str*‘.

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.

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

refresh_sublayouts()

root
The root of the tree containing this node.

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

value
The value of the multi line text input field.

Returns The text of the Widget as a `str`.

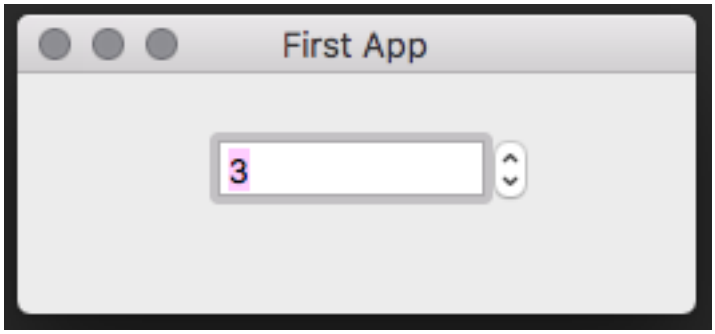
window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Number Input

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓		

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*, *readonly=False*, *on_change=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.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)
- **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.
- **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** –

MIN_WIDTH = 100

add(**children*)
Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app
The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The `toga.App` to which this widget belongs.

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

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.

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.

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

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.

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.

on_change
The handler to invoke when the value changes

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

parent
The parent of this node.

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

readonly
Whether a user can write/change the number input

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

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

refresh_sublayouts()

root
The root of the tree containing this node.

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

step
The step value for the widget

Returns The current step value for the widget.

value
Current value contained by the widget

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

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

PasswordInput

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓	✓	

Usage

Reference

class toga.widgets.passwordinput.**PasswordInput**(*id=None*, *style=None*, *factory=None*, *initial=None*, *placeholder=None*, *readonly=False*)
This widgets behaves like a TextInput but does not reveal what text is entered.

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.
- **factory** (`module`) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)
- **initial** (*str*) – The initial text that is displayed before the user inputs anything.
- **placeholder** (*str*) – The text that is displayed if no input text is present.
- **readonly** (*bool*) – Whether a user can write into the text input, defaults to *False*.

MIN_WIDTH = 100

add(**children*)
Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app
The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The `toga.App` to which this widget belongs.

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

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.

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()
Clears the input field of the widget.

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

parent
The parent of this node.

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

placeholder
The placeholder text is the displayed before the user input something.

Returns The placeholder text (str) of the widget.

readonly
Whether a user can write into the password input

Returns `True` if the user can only read, `False` if the user can read and write into the input.

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

refresh_sublayouts()

root
The root of the tree containing this node.

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

value
The value of the text input field.

Returns The text as a `str` of the password input widget.

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Progress Bar

macOS	GTK+	Windows	iOS	Android	Django
✓	✓		✓		

The progress bar is a simple widget for showing a percentage progress for task completion.

Usage

```
import toga

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

# Update progress
progress.value = 10
```

A progress bar can be in one of four visual states, determined by its `max` properties, and with the `start()` and `stop()` methods. Calling the `start()` method will make the progress bar enter running mode, and calling `stop()` will exit running mode. See the table below:

`max`	`is_running`	Behavior
None	False	disabled
None	True	indeterminate (continuous animation)
number	False	show percentage
number	True	show percentage and busy animation

If a progress bar is indeterminate, it is communicating that it has no exact percentage to report, but that work is still begin done. It may communicate this by continuously pulsing back and forth, for example.

A second type of animation occurs when a percentage is displayed and the application wants to signal that progress is still "busy". Such an animation might involve gradually altering a lighting gradient on the progress bar.

Note: Not every platform may support these animations.

ProgressBar state examples:

```
# use indeterminate mode
progress.max = None
print(progress.is_determinate) #  => False
progress.start()
print(progress.is_running) #  => True

# show percentage and busy animation (if supported)
progress.max = 100
print(progress.is_determinate) #  => True

# signal that no work is begin done with the disabled state
progress.max = None
print(progress.is_determinate) #  => False
progress.stop()
print(progress.is_running) #  => False
```

Reference

class toga.widgets.progressbar.**ProgressBar**(*id=None*, *style=None*, *max=1*, *value=0*, *running=False*, *factory=None*)

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.
- **max** (*float*) – The maximum value of the progressbar.
- **value** (*float*) – To define the current progress of the progressbar.
- **running** (*bool*) – Set the inital running mode.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

MIN_WIDTH = 100

add(**children*)
Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app
The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a str.

is_determinate
Determinate progress bars have a numeric max value (not None).

Returns True if this progress bar is determinate (max is not None) False if max is None

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

Returns True if this progress bar is running False otherwise

max
The maximum value of the progressbar.

Returns The maximum value as a `int` or `float`.

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()

root
The root of the tree containing this node.

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

start()
Starting this progress bar puts it into running mode.

stop()
Stop this progress bar (if not already stopped).

value
The current value as a `int` or `float`.

Type Returns

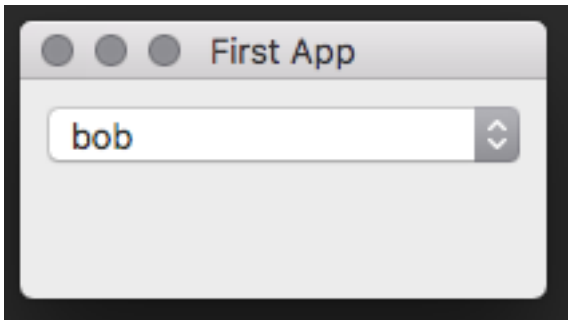
window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Selection

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓		

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.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

MIN_WIDTH = 100

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

enabled

id

The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a str.

items

The list of items.

Returns The list of str of all selectable items.

on_select

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

Return type `callable`

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()

root
The root of the tree containing this node.

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

value
The value of the currently selected item.

Returns The selected item as a `str`.

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Slider

macOS	GTK+	Windows	iOS	Android	Django
✓		✓	✓		

Usage

Reference

class toga.widgets.slider.**Slider**(*id=None*, *style=None*, *default=None*, *range=None*, *on_slide=None*, *enabled=True*, *factory=None*)
Slider widget, displays a range of values

Parameters

- **id** – An identifier for this widget.
- **style** (`Style`) –
- **default** (*float*) – Default value of the slider
- **range** (`tuple`) – Min and max values of the slider in this form (min, max).
- **on_slide** (`callable`) – The function that is executed on_slide.
- **enabled** (*bool*) – Whether user interaction is possible or not.
- **factory** (`module`) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

MIN_WIDTH = 100

add(**children*)
Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app
The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The `toga.App` to which this widget belongs.

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

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.

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.

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

on_slide
The function for when the slider is slided

Returns The `callable` that is executed on slide.

parent
The parent of this node.

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

range
Range composed of min and max slider value.

Returns Returns the range in a `tuple` like this (min, max)

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

refresh_sublayouts()

root
The root of the tree containing this node.

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

value
Current slider value.

Returns The current slider value as a `float`.

Raises `ValueError` – If the new value is not in the range of min and max.

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Switch

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓		

The switch widget is a clickable button with two stable states, True (on, checked) and False (off, unchecked).

Usage

```
import toga

input = toga.Switch()
```

Reference

class `toga.widgets.switch.`**Switch**(*label*, *id=None*, *style=None*, *on_toggle=None*, *is_on=False*, *enabled=True*, *factory=None*)

Switch widget, a clickable button with two stable states, True (on, checked) and False (off, unchecked)

Parameters

- **label** (*str*) – Text to be shown next to the switch.
- **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_toggle** (`callable`) – Function to execute when pressed.
- **is_on** (*bool*) – Current on or off state of the switch.
- **enabled** (*bool*) – Whether or not interaction with the button is possible, defaults to *True*.
- **factory** (`module`) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The `toga.App` to which this widget belongs.

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

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.

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.

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

is_on
Button Off/On state.

Returns `True` if on and `False` if the switch is off.

label
Accompanying text label of the Switch.

Returns The label text of the widget as a `str`.

on_toggle
The callable function for when the switch is pressed

Returns The `callable` on_toggle function.

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()

root
The root of the tree containing this node.

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

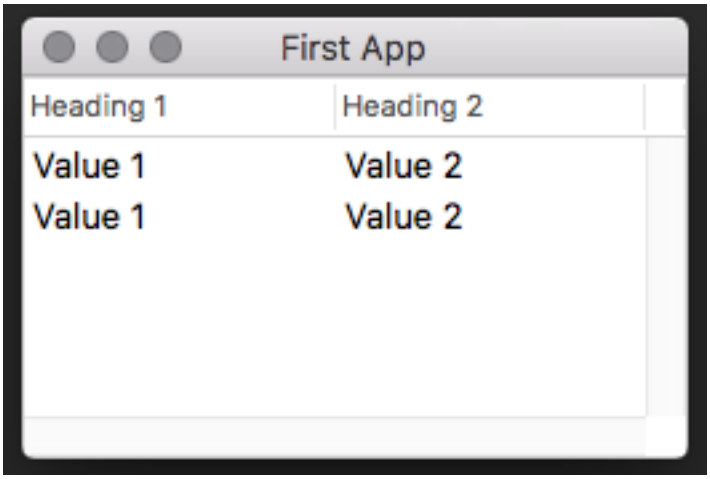
window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Table

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓			

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*, *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.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)

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']

MIN_HEIGHT = 100

MIN_WIDTH = 100

add(**children*)
Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app
The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The `toga.App` to which this widget belongs.

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

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.

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.

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

Returns Returns a (`ListSource`).

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

multiple_select
Does the table allow multiple rows to be selected?

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 (`:obj:Table`) and row (``int` or `None`).

Returns (`callable`) The callback function.

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()

root
The root of the tree containing this node.

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

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

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.

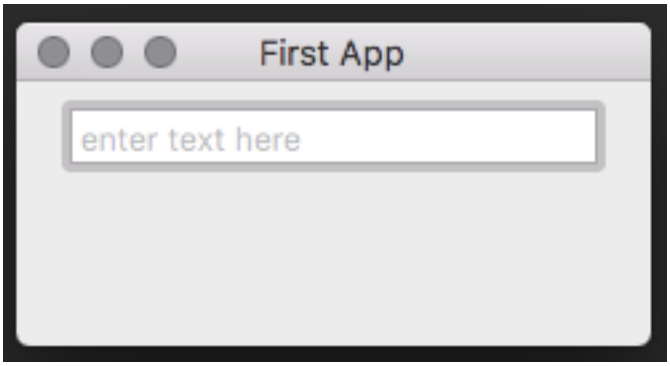
window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Text Input

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓	✓	✓

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*, *initial=None*, *placeholder=None*, *readonly=False*, *on_change=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.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)
- **initial** (*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*.

MIN_WIDTH = 100

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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()

Clears the text of the widget

enabled

id

The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

on_change
The handler to invoke when the value changes

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

parent
The parent of this node.

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

placeholder
The placeholder text.

Returns The placeholder text as a `str`.

readonly
Whether a user can write into the text input

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

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

refresh_sublayouts()

root
The root of the tree containing this node.

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

value
The value of the text input field

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

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Tree

macOS	GTK+	Windows	iOS	Android	Django
✓	✓				

The tree widget is still under development.

Usage

```
import toga

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

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

(continues on next page)

(continued from previous page)

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

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*, *factory=None*)

Tree Widget

Parameters

- **headings** (list of str) – The list of headings for the interface.
- **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.

Kwargs:

data: The data to display in the widget. Can be an instance of toga.sources.TreeSource, a list, dict or tuple with data to display in the tree widget, or a class instance which implements the interface of toga.sources.TreeSource. Entries can be given as follows:

- Any Python object value with a string represntation. This string will be shown in the widget. If value has an attribute icon, instance of (toga.Icon), the icon will be shown in front of the text.
- A tuple (icon, value) where again the string represnation of value will be used as text.

accessors (list of str): Optional: a list of attributes to access the value in the columns. If not given, the headings will be taken.

multiple_select (bool): If True, allows for the selection of multiple rows. Defaults to False.

on_select: A function to be called when the user selects one or multiple rows. factory (module): A python module that is capable to return a

implementation of this class with the same name. (optional & normally not needed)

MIN_HEIGHT = 100

MIN_WIDTH = 100

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

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

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.

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.

data
The data source of the tree :rtype: `dict`

Type returns

enabled

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a `str`.

multiple_select
Does the table allow multiple rows to be selected?

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

Return type `callable`

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()

root
The root of the tree containing this node.

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

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.

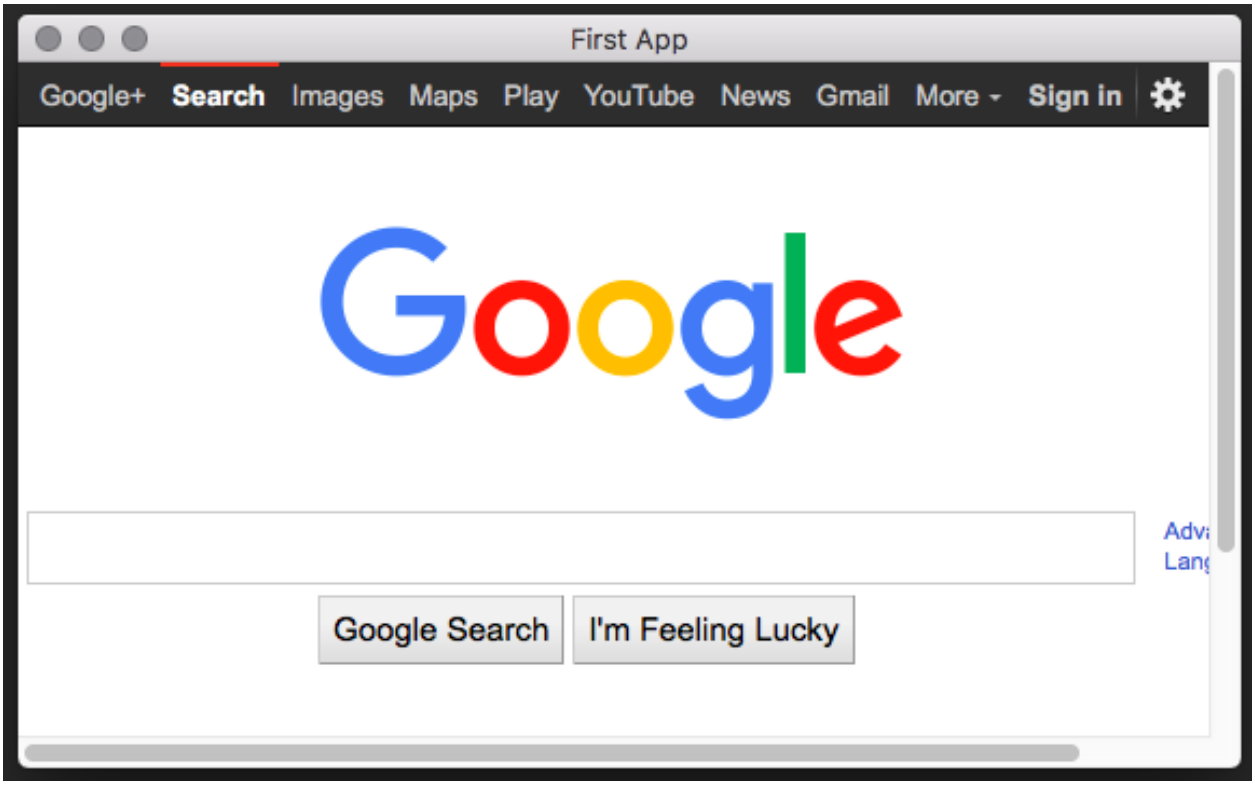
window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

WebView

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓		✓	✓

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



Usage

```
import toga

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

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.

- **factory** (module) – A python module that is capable to return a implementation of this class with the same name. (optional & normally not needed)
- **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).

MIN_HEIGHT = 100

MIN_WIDTH = 100

add(**children*)
Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app
The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

dom
The current DOM

Returns The current DOM as a str.

enabled

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.

id
The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a str.

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.

on_key_down
The handler to invoke when the button is pressed.

Returns The function `callable` that is called on button press.

on_webview_load
The handler to invoke when the webview finishes loading pressed.

Returns The function `callable` that is called when the webview finished loading.

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()

root
The root of the tree containing this node.

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

set_content(*root_url*, *content*)
Set the content of the web view.

Parameters

- **root_url** (*str*) – The URL.
- **content** (*str*) – The new content.

Returns:

url
The current URL

Returns The current URL as a `str`.

user_agent
The user agent for the web view as a `str`.

Returns The user agent as a `str`.

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

Widget

macOS	GTK+	Windows	iOS	Android	Django
✓	✓	✓	✓		✓

The widget class is a base class for all widgets and not designed to be instantiated directly.

Reference

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

This is the base widget implementation that all widgets in Toga derive from.

It defines the interface for core functionality for children, styling, layout and ownership by specific App and Window.

Apart from the above, this is an abstract implementation which must be made concrete by some platform-specific code for the _apply_layout method.

Parameters

- **id** (*str*) – An identifier for this widget.
- **enabled** (*bool*) – Whether or not interaction with the button is possible, defaults to *True*.
- **style** – An optional style object. If no style is provided then a new one will be created for the widget.
- **factory** (module) – A python module that is capable to return a implementation of this class with the same name (optional & normally not needed).

add(**children*)

Add a node as a child of this one. :param child: A node to add as a child to this node.

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

app

The App to which this widget belongs. On setting the app we also iterate over all children of this widget and set them to the same app.

Returns The toga.App to which this widget belongs.

Raises ValueError – If the widget is already associated with another app.

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.

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.

enabled

id

The node identifier. This id can be used to target styling directives

Returns The widgets identifier as a str.

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()

root

The root of the tree containing this node.

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

window
The Window to which this widget belongs. On setting the window, we automatically update all children of this widget to belong to the same window.

Returns The `toga.Window` to which the widget belongs.

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.

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 descendents. 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: `visible` | `none`

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 `none` removes the element, but still allocates space for the element as if it were in the element tree.

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 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_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

font

A shorthand value

The Pack algorithm

The pack algorithm is applied to the root of a layout tree, with a box specifying the allocated width and allocated height.

1. **Establish the available width**

   If the element has a `width` specified, the available width is set to that width.

   Otherwise, the adjusted view width is set to the view width, less the amount of `padding_left` and `padding_right`. If this results in a value less than 0, the adjusted view width is set to 0.

   If the element has a fixed intrinsic width, the available width is set to the minimum of the adjusted view width and the intrinsic width.

If the element has a minimum intrinsic width, the available width is fixed to the maximum of the adjusted view width and the intrinsic minimum width.

If the element does not have an intrinsic width, the available width is set to the adjusted view width.

2. **Establish the available height**

   If the element has a `height` specified, the available height is set to that height.

   Otherwise, the adjusted view height is set to the view height, less the amount of `padding_top` and `padding_bottom`. If this results in a value less than 0, the adjusted view height is set to 0.

   If the element has a fixed intrinsic height, the available height is set to the minimum of the adjusted view height and the intrinsic height.

   If the element has a minimum intrinsic height, the available height is fixed to the maximum of the adjusted view height and the intrinsic minimum height.

   If the element does not have an intrinsic height, the available height is set to the adjusted view height.

3. **Layout children**

   If the element has no children, the final width of the element is set to the available width, and the final height of the element is set to the available height.

   Otherwise, the element is a parent element, the final width is set to 0, and the children are laid out.

   If the parent element has a `display` value of `row`, it is a **row box**, and child layout occurs as follows:

   1. **Allocated fixed width elements**

      This step is performed on every child, in definition order.

      If the child has:

      - an explicitly specified `width`; or
      - a fixed intrinsic width; or
      - a `flex` value of 0

      then the child is then laid out using a recursive call to this algorithm, using the current available width and available height.

      The child's full width is then evaluated as the content width allocated by the recursive layout call, plus the `padding_left` and `padding_right` of the child. The final width of the parent element is increased by the child's full width; the available width of the parent element is decreased by the child's full width.

   2. **Evaluate flex quantum value**

      The flex total is set to the sum of the `flex` value for every element that *wasnt'* laid out in substep 1.

      If the available width is less than 0, or the flex total is 0, the flex quantum is set to 0. Otherwise, the flex quantum is set to the available width divided by the flex total.

   3. **Evaluate the flexible width elements**

      This step is performed on every child, in definition order.

      If the child was laid out in step 1, no layout is required, and this step can be skipped.

      Otherwise, the child's flex allocation is the product of the flex quantum and the child's `flex` value.

      If the child has a minimum intrinsic width, the child's allocated width is set to the maximum of the flex allocation and the minimum intrinsic width.

      Otherwise, the child's allocated width is set to the flex allocation.

The child is then laid out using a recursive call to this algorithm, using the child's allocated width and the available height.

The child's full width is then evaluated as the content width allocated by the recursive layout call, plus the `padding_left` and `padding_right` of the child. The overall width of the parent element is increased by the child's full width.

4. **Evaluate row height, and set the horizontal position of each element**.

   The current horizontal offset is set to 0, and then this step is performed on every child, in definition order.

   If the `text_direction` of parent element is `ltr`, the left position of the child element is set to the current horizontal offset plus the child's `padding_left`. The current horizontal offset is then increased by the child's content width plus the child's `padding_right`.

   If the `text_direction` of the parent element is `rtl`, the right position of the child element is set to the parent's final width, less the offset, less the child's `padding_right`. The current horizontal offset is then increased by the child's content width plus the child's `padding_left`.

5. **Set the vertical position of each child inside the row**

   This step is performed on every child, in definition order.

   The extra height for a child is defined as the difference between the parent elements final height and the child's full height.

   If the parent element has a `alignment` value of `top`, the vertical position of the child is set to 0, relative to the parent.

   If the parent element has a `alignment` value of `bottom`, the vertical position of the child is set to the extra height, relative to the parent.

   If the parent element has a `alignment` value of `center`, the vertical position of the child is set to 1/2 of the extra height, relative to the parent.

If the parent element has a `display` value of `column`, it is a **column box**, and child layout occurs as follows:

1. **Allocated fixed height elements**

   This step is performed on every child, in definition order.

   If the child has:

   - an explicitly specified `height`; or
   - a fixed intrinsic height; or
   - a `flex` value of 0

   then the child is then laid out using a recursive call to this algorithm, using the current available width and available height.

   The child's full height is then evaluated as the content height allocated by the recursive layout call, plus the `padding_top` and `padding_bottom` of the child. The final height of the parent element is increased by the child's full height; the available height of the parent element is decreased by the child's full height.

2. **Evaluate flex quantum value**

   The flex total is set to the sum of the `flex` value for every element that *wasn't* laid out in substep 1.

   If the available height is less than 0, or the flex total is 0, the flex quantum is set to 0. Otherwise, the flex quantum is set to the available height divided by the flex total.

3. **Evaluate the flexible height elements**

   This step is performed on every child, in definition order.

If the child was laid out in step 1, no layout is required, and this step can be skipped.

Otherwise, the child's flex allocation is the product of the flex quantum and the child's `flex` value.

If the child has a minimum intrinsic height, the child's allocated height is set to the maximum of the flex allocation and the minimum intrinsic height.

Otherwise, the child's allocated height is set to the flex allocation.

The child is then laid out using a recursive call to this algorithm, using the child's allocated height and the available width.

The child's full height is then evaluated as the content height allocated by the recursive layout call, plus the `padding_top` and `padding_bottom` of the child. The overall height of the parent element is increased by the child's full height.

4. **Evaluate column width, and set the vertical position of each element**.

   The current vertical offset is set to 0, and then this step is performed on every child, in definition order.

   The top position of the child element is set to the current vertical offset plus the child's `padding_top`. The current vertical offset is then increased by the child's content height plus the child's `padding_bottom`.

5. **Set the horizontal position of each child inside the column**

   This step is performed on every child, in definition order.

   The extra width for a child is defined as the difference between the parent element's final width and the child's full width.

   If the parent element has a `alignment` value of `left`, the horizontal position of the child is set to 0, relative to the parent.

   If the parent element has a `alignment` value of `right`, the horizontal position of the child is set to the extra width, relative to the parent.

   If the parent element has a `text_align` value of `center`, the horizontal position of the child is set to 1/2 of the extra width, relative to the parent.

2.4 Background

2.4.1 Why Toga?

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

Native widgets - not themes

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

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

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

Abstract the broad concepts

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

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

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

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

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

Python native

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

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

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

pip install and nothing more

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

Embrace mobile

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

2.4.2 Why "Toga"? Why the Yak?

So... why the name Toga?

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

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

So... why the yak mascot?

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

> You want to borrow your 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.

2.4.4 Release History

0.3.0 - In development

- Move to a three-layered Interface/Implementation/Native code structure
- Added a test framework
- Added a simplified "Pack" layout

0.2.15

- Added more widgets and cross-plaform 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 Toga Roadmap

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 TextField that provides options (e.g., text with past choices)
  - Cocoa: NSComboBox
  - GTK+: Gtk.ComboBox.new_with_model_and_entry
  - iOS: ?
  - Winforms: ComboBox
  - Android: Spinner
- DateInput - A widget for selecting a date
  - Cocoa: NSDatePicker, constrained to DMY
  - GTK+: Gtk.Calendar
  - iOS: UIDatePicker
  - Winforms: DateTimePicker
  - Android: ?
- TimeInput - A widget for selecting a time
  - Cocoa: NSDatePicker, Constrained to Time
  - GTK+: Custom Gtk.SpinButton
  - iOS: UIDatePicker
  - Winforms: DateTimePicker
  - Android: ?
- 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: ?
- SliderInput (H & V) - A widget for selecting a value from a range.
  - Cocoa: NSSlider
  - GTK+: Done
  - iOS: UISlider

  - 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.

- Separator - a visual separator; usually a faint line.
  - Cocoa: NSSeparator
  - GTK+: Gtk.Separator
  - iOS:
  - Winforms: ?
  - Android: ?
- ActivityIndicator - A spinner widget showing that something is happening
  - Cocoa: NSProgressIndicator, Spinning style
  - GTK+: Gtk.Spinner
  - iOS: UIActivityIndicatorView
  - Winforms: ?
  - Android: ?
- VideoView - Display a video
  - Cocoa: AVPlayerView
  - GTK+: Custom Integrate 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.org
  - 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: ?
- SectionContainer - (suggestions for better name welcome)

    A container view that holds a small number of subviews, only one of which is visible at any given time. Each "section" has a name and icon. Examples of use: top level navigation in Safari's preferences panel.

    - Cocoa: NSTabView
    - GTK+: ?
    - iOS: ?
    - Winforms: ?
    - Android: ?
- TabContainer - A container view for holding an unknown number of subviews, each of which is of the same type - e.g., web browser tabs.
  - Cocoa: ?
  - GTK+: GtkNotebook
  - iOS: ?
  - Winforms: ?
  - Android: ?
- NavigationContainer - A container view that holds a navigable tree of subviews

Essentially a view that has a "back" button to return to the previous view in a hierarchy. Example of use: Top level navigation in the OS X System Preferences panel.

- Cocoa: No native control
- GTK+: No native control; Gtk.HeaderBar in 3.10+
- iOS: UINavigationBar + NavigationController
- Winforms: ?
- Android: ?

Dialogs and windows

GUIs aren't all about widgets - sometimes you need to pop up a dialog to query the user. Info, Error, Question, Confirm, StackTrace and Save File Dialogs have been implemented.

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

- Long running tasks -

  GUI toolkits have a common pattern of needing to periodically update a GUI based on some long running background task. They usually accomplish this with some sort of timer-based API to ensure that the main event loop keeps running. Python has a "yield" keyword that can be repurposed for this.

- Toolbar -

  Support for adding a toolbar to an app definition. Interpretation in mobile will be difficult; maybe some sort of top level action menu available via a slideout tray (e.g., GMail account selection tray)

- Preferences -

  Support for saving app preferences, and visualizing them in a platform native way.

- Easy handling of long running tasks -

  Possibly using generators to yield control back to the event loop.

- Notification when updates are available
- Easy Licensing/registration of apps -

  Monetization is not a bad thing, and shouldn't be mutually exclusive with open source.

Platforms

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

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

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.

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 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 native widget toolkit. These are accessed using whatever bridging or Python-native API is available on the implementation platform.

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.

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

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W